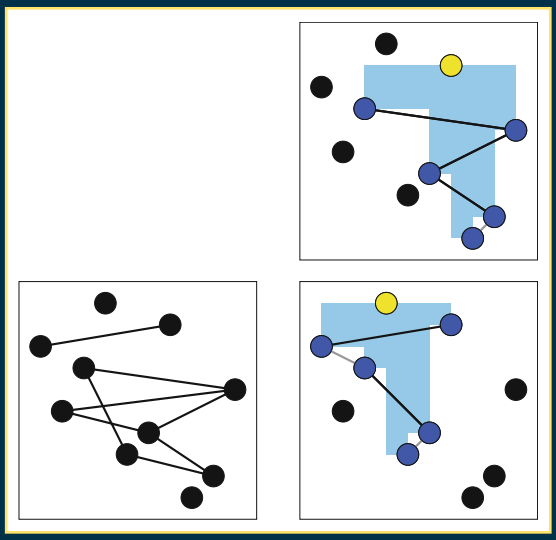
Festschrift

LNCS 8066

Andrej Brodnik
Alejandro López-Ortiz
Venkatesh Raman
Alfredo Viola (Eds.)

Space-Efficient Data Structures, Streams, and Algorithms

Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday



Springer

Lecture Notes in Computer Science 8066

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Andrej Brodnik Alejandro López-Ortiz
Venkatesh Raman Alfredo Viola (Eds.)

Space-Efficient Data Structures, Streams, and Algorithms

Papers in Honor of J. Ian Munro
on the Occasion of His 66th Birthday

Springer

Volume Editors

Andrej Brodnik
University of Ljubljana, Faculty of Computer and Information Science
Ljubljana, Slovenia *and*
University of Primorska, Department of Information Science and Technology
Koper, Slovenia
E-mail: andrej.brodnik@fri.uni-lj.si

Alejandro López-Ortiz
University of Waterloo, Cheriton School of Computer Science
Waterloo, ON, Canada
E-mail: alopez-o@uwaterloo.ca

Venkatesh Raman
The Institute of Mathematical Sciences
Chennai, India
E-mail: vraman@imsc.res.in

Alfredo Viola
Universidad de la República, Facultad de Ingeniería
Montevideo, Uruguay
E-mail: viola@fing.edu.uy

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40272-2 e-ISBN 978-3-642-40273-9
DOI 10.1007/978-3-642-40273-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013944678

CR Subject Classification (1998): F.2, E.1, G.2, H.3, I.2.8, E.5, G.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

J. Ian Munro

Preface

This volume contains research articles and surveys presented at Ianfest-66, a conference on space-efficient data structures, streams, and algorithms held during August 15–16, 2013, at the University of Waterloo, Canada.

The conference was held to celebrate Ian Munro's 66th birthday. Just like Ian's interests, the articles in this volume encompass a spectrum of areas including sorting, searching, selection and several types of, and topics in, data structures including space-efficient ones.

Ian Munro completed his PhD at the University of Toronto, around the time when computer science in general, and analysis of algorithms in particular, was maturing to be a field of research. His PhD thesis resulted in the classic book *The Computational Complexity of Algebraic and Numeric Problems* with his PhD supervisor Allan Borodin. He presented his first paper in STOC 1971, the same year and conference in which Stephen Cook (also from the same university) presented the paper on what we now call "NP-completeness." Knuth's first two volumes of *The Art of Computer Programming* were out, and the most influential third volume was to be released soon after. Hopcroft and Tarjan were developing important graph algorithms (for planarity, biconnected components etc).

Against this backdrop, Ian started making fundamental contributions in sorting, selection and data structures (including optimal binary search trees, heaps and hashing). He steadfastly stayed focused on these subjects, always taking an expansive view, which included text search and data streams at a time when few others were exploring these topics.

While the exact worst case comparison bound to find the median is still open, he closed this problem in 1984 along with his student Walter Cunto for the average case. His seminal work on implicit data structures with his student Hendra Suwanda marked his focus on space-efficient data structures. This was around the time of "megabyte" main memories, so space was becoming cheaper, though, as usual, the input sizes were becoming much larger. He saw early on that these trends will continue making the focus on space-efficiency more, rather than less, important. This trend has continued with the development of personal computing in its many forms and multilevel caches. His unique expertise helped contribute significantly to the Oxford English Dictionary (OED) project at Waterloo, and the founding of the OpenText as a company dealing with text-based algorithms.

His invited contribution at the FSTTCS conference titled *Tables* brought the focus of work on succinct data structures in the years to come. His early work with Mike Paterson on selection is regarded as the first paper in a model that has later been called the "streaming model," a model of intense study in the modern Internet age. In this model, his other paper "Frequency estimation of

internet packet streams with limited space" with Erik Demaine and Alejandro López-Ortiz has received over 300 citations.

In addition to his research, Ian is an inspiring teacher. He has supervised (or co-supervised) over 20 PhD students and about double the number of Master's students. For many years, Ian has been part of the faculty team that coached Canadian high school students for the International Olympiad in Informatics (IOI). He has led the Canadian team and served on the IOI's international scientific committee.

Ian also gets a steady stream of post-doctoral researchers and other visitors from throughout the world. He has served in many program committees of international conferences and in editorial boards of journals, and has given plenary talks at various international conferences. He has held visiting positions at several places including Princeton University, University of Washington, AT&T Bell Labaratories, University of Arizona, University of Warwick and Universitè libre de Bruxelles. Through his students and their students and other collaborators, he has helped establish strong research groups in various parts of the world including Chile, India, South Korea, Uruguay and in many parts of Europe and North America. He also has former students in key positions in leading companies of the world.

His research achievements have been recognized by his election as Fellow of the Royal Society of Canada (2003) and Fellow of the ACM (2008). He was made a University Professor in 2006.

Ian has a great sense of generosity, wit and humor. Ian, his wife, Marilyn, and his children, Alison and Brian, are more than a host to his students and collaborators; they have helped establish a long-lasting friendship with them.

At 66 Ian is going strong, makes extensive research tours, supervises many PhD students and continues to educate and inspire students and researchers. We wish him and his family many more years of fruitful and healthy life.

We thank a number of people that made this volume possible. First and foremost, we thank all the authors who came forward to contribute their articles on a short notice, all anonymous referees, proofreaders, and the speakers at the conference. We thank Marko Grgurovič, Wendy Rush and Jan Vesel for collecting and verifying data about Ian's students and work. We thank Alfred Hofmann, Anna Kramer and Ronan Nugent at Springer for their enthusiastic support and help in producing this Festschrift. We thank Alison Conway at Fields Institute at Toronto for maintaining the conference website and managing registration, and Fields Institute for their generous financial support, and University of Waterloo for their infrastructural and organizational support.

This volume contains surveys on emerging, as well as established, fields in data structures and algorithms, written by leading experts, and we feel that it will become a book to cherish in the years to come.

June 2013

Andrej Brodnik
Alejandro López-Ortiz
Venkatesh Raman
Alfredo Viola

Curriculum Vitae J. Ian Munro

Current Position

University Professor and *Canada Research Chair in Algorithm Design*

Address

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
`https://cs.uwaterloo.ca/~imunro/`

Personal Information

Born: July 10, 1947
Married: to Marilyn Miller
Two children: Alison and Brian

Education

Ph.D., Computer Science, University of Toronto, 1971
M.Sc., Computer Science, University of British Columbia, 1969
B.A. (Hons), Mathematics, University of New Brunswick, 1968

Experience

1971-present: Professor, University of Waterloo, Ontario, Canada

Professional Interests

Data structures, particularly fast and space-efficient structures.
The design, analysis and implementation of algorithms.
Bioinformatics.
Database systems and data warehousing, particularly efficiency issues.

Co-authors

1. Brian Allen
2. Stephen Alstrup
3. Helmut Alt
4. Lars Arge
5. Diego Arroyuelo
6. Jérémy Barbay
7. Michael A. Bender
8. David Benoit
9. Therese Biedl
10. Allan Borodin
11. Prosenjit Bose
12. Gerth Stølting Brodal
13. Andrej Brodnik
14. Jean Cardinal
15. Svante Carlsson
16. Luca Castelli Aleardi
17. Pedro Celis
18. Timothy Chan
19. David R. Clark
20. Francisco Claude
21. Gordon Cormack
22. Joseph C. Culberson
23. Walter Cunto
24. David DeHaan
25. Erik D. Demaine
26. Martin L. Demaine
27. David P. Dobkin
28. Reza Dorrigiv
29. Stephane Durocher
30. Amr Elmasry
31. Martin Farach-Colton
32. Arash Farzan
33. Paolo Ferragina
34. Amos Fiat
35. Faith E. Fich
36. Jeremy T. Fineman
37. Samuel Fiorini
38. Rudolf Fleischer
39. Gianni Franceschini
40. Robert Fraser
41. Michael L. Fredman
42. Travis Gagie
43. W. Morven Gentleman
44. Pedram Ghodsnia
45. Lukasz Golab
46. Mordecai Golin
47. Alexander Golynski
48. Gastón H. Gonnet
49. Roberto Grossi
50. Gwenaël Joret
51. Torben Hagerup
52. Angèle M. Hamel
53. E. R. Hansen
54. Nicholas J. A. Harvey
55. Meng He
56. Bryan Holland-Minkley
57. John Iacono
58. Lars Jacobsen
59. X Richard Ji
60. Raphaël M. Jungers
61. Kanela Kaligosi
62. T. Kameda
63. Johan Karlsson
64. Rolf G. Karlsson
65. Marek Karpinski
66. Paul E. Kearney
67. Graeme Kemkes
68. James A. King
69. Alejandro López-Ortiz
70. Stefan Langerman
71. Per-Åke Larson
72. Anna Lubiw
73. Kurt Mehlhorn
74. Peter Bro Miltersen
75. Pat Morin
76. Moni Naor
77. Yakov Nekrich
78. Patrick K. Nicholson
79. Andreas Nilsson
80. B. John Oommen
81. Mark H. Overmars
82. Linda Pagli
83. Thomas Papadakis
84. Michael S. Paterson
85. Derek Phillips
86. Patricio V. Poblete
87. M. Ziaur Rahman
88. Raúl J. Ramírez

89. Rajeev Raman
90. Venkatesh Raman
91. Theis Rauhe
92. Manuel Rey
93. Edward L. Robertson
94. Doron Rotem
95. Alejandro Salinger
96. Jeffrey S. Salowe
97. Peter Sanders
98. Srinivasa Rao Satti
99. Jeanette P. Schmidt
100. Allen J. Schwenk
101. Alejandro A. Schäffer
102. Robert Sedgewick
103. Alan Siegel
104. Matthew Skala
105. Philip M. Spira
106. Adam J. Storm
107. Hendra Suwanda
108. David J. Taylor
109. Mikkel Thorup
110. Kamran Tirdad
111. Frank Wm. Tompa
112. Troy Vasiga
113. Alfredo Viola
114. Derick Wood
115. Gelin Zhou

Books and Book chapters

[1] Barbay, J., Munro, J.I.: Succinct encoding of permutations: Applications to text indexing. In: Kao, M.Y. (ed.) Encyclopedia of Algorithms, pp. 915–919. Springer (2008)
[2] Borodin, A., Munro, J.I.: The computational complexity of algebraic and numeric problems. American Elsevier, New York (1975)
[3] Munro, J.I., Satti, S.R.: Succinct representation of data structures. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications. Chapman & Hall/Crc Computer and Information Science Series, ch. 37. Chapman & Hall/CRC (2004)

Edited Proceedings

[4] Blum, M., Galil, Z., Ibarra, O.H., Kozen, D., Miller, G.L., Munro, J.I., Ruzzo, W.L. (eds.): SFCS 1983: Proceedings of the 24th Annual Symposium on Foundations of Computer Science, p. iii. IEEE Computer Society, Washington, DC (1983)
[5] Chwa, K.-Y., Munro, J.I. (eds.): COCOON 2004. LNCS, vol. 3106. Springer, Heidelberg (2004)
[6] López-Ortiz, A., Munro, J.I. (eds.): ACM Transactions on Algorithms 2(4), 491 (2006)
[7] Munro, J.I. (ed.): Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14. SIAM (2004)

Journal and Conference Papers

[8] Allen, B., Munro, J.I.: Self-organizing binary search trees. J. ACM 25(4), 526–535 (1978); A preliminary version appeared in SFCS 1976: Proceedings of the 17th Annual Symposium on Foundations of Computer Science, pp. 166–172. IEEE Computer Society, Washington, DC (1976)

[9] Alt, H., Mehlhorn, K., Munro, J.I.: Partial match retrieval in implicit data structures. Inf. Process. Lett. 19(2), 61–65 (1984); A preliminary version appeared in Gruska, J., Chytil, M.P. (eds.): MFCS 1981. LNCS, vol. 118, pp. 156–161. Springer, Heidelberg (1981)

[10] Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Ian Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM J. Comput. 36(6), 1672–1695 (2007)

[11] Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: Cache-oblivious priority queue and graph algorithm applications. In: STOC 2002: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, pp. 268–276. ACM, New York (2002)

[12] Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Ian Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search. Theor. Comput. Sci. 412(32), 4200–4211 (2011); A Preliminary Version Appeared in Dong, Y., Du, D.-Z., Ibarra, O. (eds.): ISAAC 2009. LNCS, vol. 5878, pp. 203–212. Springer, Heidelberg (2009)

[13] Barbay, J., Castelli Aleardi, L., He, M., Munro, J.I.: Succinct representation of labeled graphs. Algorithmica 62(1-2), 224–257 (2012); A Preliminary Version Appeared in Tokuyama, T. (ed.): ISAAC 2007. LNCS, vol. 4835, pp. 316–328. Springer, Heidelberg (2007)

[14] Barbay, J., Golynski, A., Munro, J.I., Satti, S.R.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theor. Comput. Sci. 387(3), 284–297 (2007); A Preliminary Version Appeared in Lewenstein, M., Valiente, G. (eds.): CPM 2006. LNCS, vol. 4009, pp. 24–35. Springer, Heidelberg (2006)

[15] Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Trans. Algorithms 7(4), 1–27 (2011); A Preliminary Version Appeared in SODA 2007: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 680–689. Society for Industrial and Applied Mathematics, Philadelphia (2007)

[16] Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Satti, S.R.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005); A Preliminary Version Appeared in Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.): WADS 1999. LNCS, vol. 1663, pp. 169–180. Springer, Heidelberg (1999)

[17] Biedl, T., Chan, T., Demaine, E.D., Fleischer, R., Golin, M., King, J.A., Munro, J.I.: Fun-sort – or the chaos of unordered binary search. Discrete Appl. Math. 144(3), 231–236 (2004)

[18] Biedl, T., Golynski, A., Hamel, A.M., López-Ortiz, A., Munro, J.I.: Sorting with networks of data structures. Discrete Appl. Math. 158(15), 1579–1586 (2010)
[19] Borodin, A., Munro, J.I.: Evaluating polynomials at many points. Inf. Process. Lett. 1(2), 66–68 (1971)
[20] Bose, P., Brodnik, A., Carlsson, S., Demaine, E.D., Fleischer, R., López-Ortiz, A., Morin, P., Munro, J.I.: Online routing in convex subdivisions. Int. J. Comput. Geometry Appl. 12(4), 283–296 (2002); A Preliminary Version Appeared in Lee, D.T., Teng, S.-H. (eds.): ISAAC 2000. LNCS, vol. 1969, pp. 47–59. Springer, Heidelberg (2000)
[21] Bose, P., Lubiw, A., Munro, J.I.: Efficient visibility queries in simple polygons. Comput. Geom. Theory Appl. 23(3), 313–335 (2002)
[22] Brodal, G.S., Demaine, E.D., Fineman, J.T., Iacono, J., Langerman, S., Munro, J.I.: Cache-oblivious dynamic dictionaries with update/query tradeoffs. In: SODA 2010: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1448–1456. Society for Industrial and Applied Mathematics, Philadelphia (2010)
[23] Brodal, G.S., Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41(4-5), 273–291 (2005)
[24] Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.D.: Resizable arrays in optimal time and space. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 37–48. Springer, Heidelberg (1999)
[25] Brodnik, A., Carlsson, S., Fredman, M.L., Karlsson, J., Munro, J.I.: Worst case constant time priority queue. J. Syst. Softw. 78(3), 249–256 (2005); A Preliminary Version Appeared in SODA 2001: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 523–528. Society for Industrial and Applied Mathematics, Philadelphia (2001)
[26] Brodnik, A., Karlsson, J., Munro, J.I., Nilsson, A.: An O(1) solution to the prefix sum problem on a specialized memory architecture. In: IFIP TCS, pp. 103–114 (2006)
[27] Brodnik, A., Miltersen, P.B., Munro, J.I.: Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In: Rau-Chaplin, A., Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 1997. LNCS, vol. 1272, pp. 426–439. Springer, Heidelberg (1997)
[28] Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM J. Comput. 28(5), 1627–1640 (1999); A Preliminary Version Appeared in van Leeuwen, J. (ed.): ESA 1994. LNCS, vol. 855, pp. 72–81. Springer, Heidelberg (1994)
[29] Brodnik, A., Munro, J.I.: Neighbours on a grid. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 309–320. Springer, Heidelberg (1996)

[30] Cardinal, J., Fiorini, S., Joret, G., Jungers, R.M., Munro, J.I.: An efficient algorithm for partial order production. CoRR abs/0811.2572 (2008); A Preliminary Version Appeared in STOC 2009: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 93–100. ACM, New York (2009)
[31] Cardinal, J., Fiorini, S., Joret, G., Jungers, R.M., Munro, J.I.: An efficient algorithm for partial order production. SIAM J. Comput. 39(7), 2927–2940 (2010)
[32] Cardinal, J., Fiorini, S., Joret, G., Jungers, R.M., Munro, J.I.: Sorting under partial information (without the ellipsoid algorithm). In: STOC 2010: Proceedings of the 42nd ACM Symposium on Theory of Computing, pp. 359–368. ACM, New York (2010)
[33] Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
[34] Celis, P., Larson, P.Å., Munro, J.I.: Robin hood hashing. In: SFCS 1985: Proceedings of the 26th Annual Symposium on Foundations of Computer Science, pp. 281–288. IEEE Computer Society, Washington, DC (1985)
[35] Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA 1996: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 383–391. Society for Industrial and Applied Mathematics, Philadelphia (1996)
[36] Claude, F., Munro, J.I., Nicholson, P.K.: Range queries over untangled chains. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 82–93. Springer, Heidelberg (2010)
[37] Cormack, G., Munro, J.I., Vasiga, T., Kemkes, G.: Structure, scoring and purpose of computing competitions. Informatics in education 5(1), 15–36 (2006)
[38] Culberson, J.C., Munro, J.I.: Explaining the behaviour of binary search trees under prolonged updates: a model and simulations. Comput. J. 32(1), 68–75 (1989)
[39] Culberson, J.C., Munro, J.I.: Analysis of the standard deletion algorithms in exact fit domain binary search trees. Algorithmica 5(3), 295–311 (1990)
[40] Cunto, W., Gonnet, G.H., Munro, J.I., Poblete, P.V.: Fringe analysis for extquick: an in situ distributive external sorting algorithm. Inf. Comput. 92(2), 141–160 (1991)
[41] Cunto, W., Munro, J.I.: Average case selection. J. ACM 36(2), 270–279 (1989); A Preliminary Version Appeared in STOC 1984: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, pp. 369–375. ACM, New York (1984)
[42] Cunto, W., Munro, J.I., Poblete, P.V.: A case study in comparison based complexity: Finding the nearest value(s). In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519, pp. 1–12. Springer, Heidelberg (1991)
[43] Cunto, W., Munro, J.I., Rey, M.: Selecting the median and two quartiles in a set of numbers. Softw. Pract. Exper. 22(6), 439–454 (1992)

[44] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: SODA 2000: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 743–752. Society for Industrial and Applied Mathematics, Philadelphia (2000)

[45] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Experiments on adaptive set intersections for text retrieval systems. In: Buchsbaum, A.L., Snoeyink, J. (eds.) ALENEX 2001. LNCS, vol. 2153, pp. 91–104. Springer, Heidelberg (2001)

[46] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 348–360. Springer, Heidelberg (2002)

[47] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Robot localization without depth perception. In: Penttonen, M., Schmidt, E.M. (eds.) SWAT 2002. LNCS, vol. 2368, pp. 249–259. Springer, Heidelberg (2002)

[48] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Note: on universally easy classes for NP-complete problems. Theor. Comput. Sci. 304(1-3), 471–476 (2003); A Preliminary Version Appeared in SODA 2001: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 910–911. Society for Industrial and Applied Mathematics, Philadelphia (2001)

[49] Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. In: Pandu Rangan, C., Raman, V., Sarukkai, S. (eds.) FST TCS 1999. LNCS, vol. 1738, pp. 84–96. Springer, Heidelberg (1999)

[50] Dobkin, D.P., Munro, J.I.: Time and space bounds for selection problems. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 192–204. Springer, Heidelberg (1978)

[51] Dobkin, D.P., Munro, J.I.: Determining the mode. Theor. Comput. Sci. 12, 255–263 (1980)

[52] Dobkin, D.P., Munro, J.I.: Optimal time minimal space selection algorithms. J. ACM 28(3), 454–461 (1981)

[53] Dobkin, D.P., Munro, J.I.: Efficient uses of the past. J. Algorithms 6(4), 455–465 (1985); A Preliminary Version Appeared in SFCS 1980: Proceedings of the 21st Annual Symposium on Foundations of Computer Science, pp. 200–206. IEEE Computer Society, Washington, DC (1980)

[54] Dorrigiv, R., Durocher, S., Farzan, A., Fraser, R., López-Ortiz, A., Munro, J.I., Salinger, A., Skala, M.: Finding a hausdorff core of a polygon: On convex polygon containment with bounded hausdorff distance. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 218–229. Springer, Heidelberg (2009)

[55] Dorrigiv, R., López-Ortiz, A., Munro, J.I.: List update algorithms for data compression. In: DCC 2008: Proceedings of the Data Compression Conference, p. 512. IEEE Computer Society, Washington, DC (2008)

[56] Dorrigiv, R., López-Ortiz, A., Munro, J.I.: An application of self-organizing data structures to compression. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 137–148. Springer, Heidelberg (2009)

[57] Dorrigiv, R., López-Ortiz, A., Munro, J.I.: On the relative dominance of paging algorithms. Theor. Comput. Sci. 410(38-40), 3694–3701 (2009). A Preliminary Version Appeared in Tokuyama, T. (ed.): ISAAC 2007. LNCS, vol. 4835, pp. 488–499. Springer, Heidelberg (2007)

[58] Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. Inf. Comput. 222, 169–179 (2013); A preliminary version appeared in Aceto, L., Henzinger, M., Sgall, J. (eds.): ICALP 2011, Part I. LNCS, vol. 6755, pp. 244–255. Springer, Heidelberg (2011)

[59] Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 150–159. Springer, Heidelberg (2011)

[60] Farzan, A., Ferragina, P., Franceschini, G., Munro, J.I.: Cache-oblivious comparison-based algorithms on multisets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 305–316. Springer, Heidelberg (2005)

[61] Farzan, A., Munro, J.I.: Succinct representation of finite abelian groups. In: ISSAC 2006: Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation, pp. 87–92. ACM, New York (2006)

[62] Farzan, A., Munro, J.I.: Succinct representations of arbitrary graphs. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 393–404. Springer, Heidelberg (2008)

[63] Farzan, A., Munro, J.I.: A uniform approach towards succinct representation of trees. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 173–184. Springer, Heidelberg (2008)

[64] Farzan, A., Munro, J.I.: Succinct representation of dynamic trees. Theor. Comput. Sci. 412(24), 2668–2678 (2011)

[65] Farzan, A., Munro, J.I.: Dynamic succinct ordered trees. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 439–450. Springer, Heidelberg (2009)

[66] Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica, 1–25 (2012), http://dx.doi.org/10.1007/s00453-012-9664-0

[67] Farzan, A., Munro, J.I., Raman, R.: Succinct indices for range queries with applications to orthogonal range maxima. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 327–338. Springer, Heidelberg (2012)

[68] Fiat, A., Munro, J.I., Naor, M., Schäffer, A.A., Schmidt, J.P., Siegel, A.: An implicit data structure for searching a multikey table in logarithmic time. J. Comput. Syst. Sci. 43(3), 406–424 (1991)

[69] Fich, F.E., Munro, J.I., Poblete, P.V.: Permuting in place. SIAM J. Comput. 24(2), 266–278 (1995); A Preliminary Version Appeared in SFCS 1990: Proceedings of the 31st Annual Symposium on Foundations of Computer Science, vol.1, pp. 372–379, IEEE Computer Society, Washington, DC (1990)
[70] Franceschini, G., Grossi, R., Munro, J.I., Pagli, L.: Implicit B-trees: a new data structure for the dictionary problem. J. Comput. Syst. Sci. 68(4), 788–807 (2004); A Preliminary Version Appeared in FOCS 2002: Proceedings of the 43rd Symposium on Foundations of Computer Science, pp. 145–154. IEEE Computer Society, Washington, DC (2002)
[71] Franceschini, G., Munro, J.I.: Implicit dictionaries with O(1) modifications per update and fast search. In: SODA 2006: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, pp. 404–413. ACM, New York (2006)
[72] Gagie, T., He, M., Munro, J.I., Nicholson, P.K.: Finding frequent elements in compressed 2D arrays and strings. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 295–300. Springer, Heidelberg (2011)
[73] Gentleman, W.M., Munro, J.I.: Designing overlay structures. Softw. Pract. Exper. 7(4), 493–500 (1977)
[74] Ghodsnia, P., Tirdad, K., Munro, J.I., Lóez-Ortiz, A.: A novel approach for leveraging co-occurrence to improve the false positive error in signature files. J. of Discrete Algorithms 18, 63–74 (2013)
[75] Golab, L., DeHaan, D., Demaine, E.D., López-Ortiz, A., Munro, J.I.: Identifying frequent items in sliding windows over on-line packet streams. In: IMC 2003: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, pp. 173–178. ACM, New York (2003)
[76] Golynski, A., Munro, J.I., Satti, S.R.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA 2006: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, pp. 368–373. ACM, New York (2006)
[77] Gonnet, G.H., Larson, P.Å., Munro, J.I., Rotem, D., Taylor, D.J., Tompa, F.W.: Database storage structures research at the University of Waterloo. IEEE Database Eng. Bull. 5(1), 49–52 (1982)
[78] Gonnet, G.H., Munro, J.I.: Efficient ordering of hash tables. SIAM J. Comput. 8(3), 463–478 (1979)
[79] Gonnet, G.H., Munro, J.I.: A linear probing sort and its analysis. In: STOC 1981: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 90–95. ACM, New York (1981)
[80] Gonnet, G.H., Munro, J.I.: The analysis of an improved hashing technique. In: STOC 1977: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, pp. 113–121. ACM, New York (1977)
[81] Gonnet, G.H., Munro, J.I.: The analysis of linear probing sort by the use of a new mathematical transform. J. Algorithms 5(4), 451–470 (1984)

[82] Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. 15(4), 964–971 (1986); A Preliminary Version Appeared in Nielsen, M., Schmidt, E.M. (eds.): ICALP 1982. LNCS, vol. 140, pp. 282–291. Springer, Heidelberg (1982)
[83] Gonnet, G.H., Munro, J.I., Suwanda, H.: Toward self-organizing linear search. In: SFCS 1979: Proceedings of the 20th Annual Symposium on Foundations of Computer Science, pp. 169–174. IEEE Computer Society, Washington, DC (1979)
[84] Gonnet, G.H., Munro, J.I., Suwanda, H.: Exegesis of self-organizing linear search. SIAM J. Comput. 10(3), 613–637 (1981)
[85] Gonnet, G.H., Munro, J.I., Wood, D.: Direct dynamic structures for some line segment problems. Computer Vision, Graphics, and Image Processing 23(2), 178–186 (1983)
[86] Hagerup, T., Mehlhorn, K., Munro, J.I.: Maintaining discrete probability distributions optimally. In: Lingas, A., Carlsson, S., Karlsson, R. (eds.) ICALP 1993. LNCS, vol. 700, pp. 253–264. Springer, Heidelberg (1993)
[87] Harvey, N.J.A., Munro, J.I.: Deterministic skipnet. Inf. Process. Lett. 90(4), 205–208 (2004); A Preliminary Version Appeared in PODC 2003: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 152–152. ACM, New York(2003)
[88] He, M., Munro, J.I.: Succinct representations of dynamic strings. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 334–346. Springer, Heidelberg (2010)
[89] He, M., Munro, J.I.: Space efficient data structures for dynamic orthogonal range counting. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 500–511. Springer, Heidelberg (2011)
[90] He, M., Munro, J.I., Nicholson, P.K.: Dynamic range selection in linear space. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 160–169. Springer, Heidelberg (2011)
[91] He, M., Munro, J.I., Satti, S.R.: A categorization theorem on suffix arrays with applications to space efficient text indexes. In: SODA 2005: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 23–32. Society for Industrial and Applied Mathematics, Philadelphia (2005)
[92] He, M., Munro, J.I., Satti, S.R.: Succinct ordinal trees based on tree covering. ACM Trans. Algorithms 8(4), 1–32 (2012); A Preliminary Version Appeared in Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.): ICALP 2007. LNCS, vol. 4596, pp. 509–520. Springer, Heidelberg (2007)
[93] He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 140–149. Springer, Heidelberg (2011)
[94] He, M., Munro, J.I., Zhou, G.: A framework for succinct labeled ordinal trees over large alphabets. In: Chao, K.-M., Hsu, T.-s., Lee, D.-T. (eds.) ISAAC 2012. LNCS, vol. 7676, pp. 537–547. Springer, Heidelberg (2012)
[95] He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 575–586. Springer, Heidelberg (2012)

[96] Kaligosi, K., Mehlhorn, K., Munro, J.I., Sanders, P.: Towards optimal multiple selection. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 103–114. Springer, Heidelberg (2005)

[97] Kameda, T., Munro, J.I.: A $O(|V| * |E|)$ algorithm for maximum matching of graphs. Computing 12(1), 91–98 (1974)

[98] Karlsson, R.G., Munro, J.I.: Proximity of a grid. In: Mehlhorn, K. (ed.) STACS 1985. LNCS, vol. 182, pp. 187–196. Springer, Heidelberg (1984)

[99] Karlsson, R.G., Munro, J.I., Robertson, E.L.: The nearest neighbor problem on bounded domains. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 318–327. Springer, Heidelberg (1985)

[100] Kearney, P.E., Munro, J.I., Phillips, D.: Efficient generation of uniform samples from phylogenetic trees. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 177–189. Springer, Heidelberg (2003)

[101] Munro, J.I.: Efficient determination of the transitive closure of a directed graph. Inf. Process. Lett. 1(2), 56–58 (1971)

[102] Munro, J.I.: Some results concerning efficient and optimal algorithms. In: STOC 1971: Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 40–44. ACM Press, New York (1971)

[103] Munro, J.I.: Efficient polynomial evaluation. In: Proc. Sixth Annual Princeton Conference on Information Sciences and Systems (1972)

[104] Munro, J.I.: In search of the fastest algorithm. In: AFIPS 1973: Proceedings of the National Computer Conference and Exposition, June 4-8, pp. 453–453. ACM, New York (1973)

[105] Munro, J.I.: The parallel complexity of arithmetic computation. In: Karpinski, M. (ed.) FCT 1977. LNCS, vol. 56, pp. 466–475. Springer, Heidelberg (1977)

[106] Munro, J.I.: Review of "The Complexity of Computing" (Savage, J.E. in 1977). IEEE Transactions on Information Theory 24(3), 401–401 (1978)

[107] Munro, J.I.: A multikey search problem. In: Proceedings of the 17th Allerton Conference on Communication, Control and Computing, pp. 241–244 (1979)

[108] Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. J. Comput. Syst. Sci. 33(1), 66–74 (1986); A Preliminary Version An Implicit Data Structure for the Dictionary Problem that Runs in Polylog Time Appeared in SFCS 1984: Proceedings of the 25th Annual Symposium onFoundations of Computer Science, pp. 369–374. IEEE Computer Society, Washington, DC (1984)

[109] Munro, J.I.: Developing implicit data structures. In: Wiedermann, J., Gruska, J., Rovan, B. (eds.) MFCS 1986. LNCS, vol. 233, pp. 168–176. Springer, Heidelberg (1986)

[110] Munro, J.I.: Searching a two key table under a single key. In: STOC 1987: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, pp. 383–387. ACM, New (1987)

[111] Munro, J.I.: Data manipulations based on orderings. In: Algorithms and Order, pp. 283–306. Springer, Netherlands (1988)
[112] Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
[113] Munro, J.I.: On the competitiveness of linear search. In: Paterson, M. (ed.) ESA 2000. LNCS, vol. 1879, pp. 338–345. Springer, Heidelberg (2000)
[114] Munro, J.I.: Space efficient suffix trees. J. Algorithms 39(2), 205–222 (2001)
[115] Munro, J.I.: Succinct data structures. Electr. Notes Theor. Comput. Sci. 91, 3 (2004)
[116] Munro, J.I.: Lower bounds for succinct data structures. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, p. 3. Springer, Heidelberg (2008)
[117] Munro, J.I.: Reflections on optimal and nearly optimal binary search trees. Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday, pp. 115–120 (2009)
[118] Munro, J.I.: The complexity of partial orders. In: Bader, D.A., Mutzel, P. (eds.) ALENEX, p. 64. SIAM/Omnipress (2012)
[119] Munro, J.I., Borodin, A.: Efficient evaluation of polynomial forms. J. Comput. Syst. Sci. 6(6), 625–638 (1972)
[120] Munro, J.I., Celis, P.: Techniques for collision resolution in hash tables with open addressing. In: ACM 1986: Proceedings of 1986 ACM Fall Joint Computer Conference, pp. 601–610. IEEE Computer Society Press, Los Alamitos (1986)
[121] Munro, J.I., Ji, X.R.: On the pivot strategy of quicksort. In: Canadian Conference on Electrical and Computer Engineering, vol. 1, pp. 302–305. IEEE (1996)
[122] Munro, J.I., Nicholson, P.K.: Succinct posets. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 743–754. Springer, Heidelberg (2012)
[123] Munro, J.I., Overmars, M.H., Wood, D.: Variations on visibility. In: SCG 1987: Proceedings of the Third Annual Symposium on Computational Geometry, pp. 291–299. ACM, New York (1987)
[124] Munro, J.I., Papadakis, T., Sedgewick, R.: Deterministic skip lists. In: SODA 1992: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 367–375. Society for Industrial and Applied Mathematics, Philadelphia (1992)
[125] Munro, J.I., Paterson, M.S.: Optimal algorithms for parallel polynomial evaluation. J. Comput. Syst. Sci. 7(2), 189–198 (1973); A Preliminary Version Appeared in SWAT 1971: Proceedings of the 12th Annual Symposium on Switching and Automata Theory, pp. 132–139. IEEE Computer Society, Washington, DC (1971)
[126] Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theor. Comput. Sci. 12, 315–323 (1980); A Preliminary Version Appeared in SFCS 1978: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, pp. 253–258. IEEE Computer Society, Washington, DC (1978)

[127] Munro, J.I., Poblete, P.V.: Implicit structuring of data. In: Proceedings of the Twelfth Manitoba Conference on Numerical Mathematics and Computing: held at the University of Manitoba, September 30-October 2, vol. 37, p. 73. Utilitas Mathematica Publishing (1983)
[128] Munro, J.I., Poblete, P.V.: A discipline for robustness or storage reduction in binary search trees. In: PODS 1983: Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 70–75. ACM Press, New York (1983)
[129] Munro, J.I., Poblete, P.V.: Probabilistic issues in data structures. In: Computer Science and Statistics: proceedings of the 14th Symposium on the Interface, p. 32. Springer, Heidelberg (1983)
[130] Munro, J.I., Poblete, P.V.: Fault tolerance and storage reduction in binary search trees. Information and Control 62(2/3), 210–218 (1984)
[131] Munro, J.I., Poblete, P.V.: Searchability in merging and implicit data structures. BIT 27(3), 324–329 (1987); A preliminary version appeared in Díaz, J. (ed.): ICALP 1983. LNCS, vol. 154, pp. 527–535. Springer, Heidelberg (1983)
[132] Munro, J.I., Raman, R., Raman, V., Satti, S.R.: Succinct representations of permutations and functions. Theor. Comput. Sci. 438, 74–88 (2012); A Preliminary Version Appeared in Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.): ICALP 2003. LNCS, vol. 2719, pp. 345–356. Springer, Heidelberg (2003)
[133] Munro, J.I., Raman, V.: Fast sorting in-place sorting with $O(n)$ data. In: Biswas, S., Nori, K.V. (eds.) FSTTCS 1991. LNCS, vol. 560, pp. 266–277. Springer, Heidelberg (1991)
[134] Munro, J.I., Raman, V.: Sorting multisets and vectors in-place. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519, pp. 473–480. Springer, Heidelberg (1991)
[135] Munro, J.I., Raman, V.: Sorting with minimum data movement. J. Algorithms 13(3), 374–393 (1992); A Preliminary Version Appeared in Dehne, F., Santoro, N., Sack, J.-R. (eds.): WADS 1989. LNCS, vol. 382, pp. 552–562. Springer, Heidelberg (1989)
[136] Munro, J.I., Raman, V.: Fast stable in-place sorting with $O(n)$ data moves. Algorithmica 16(2), 151–160 (1996)
[137] Munro, J.I., Raman, V.: Selection from read-only memory and sorting with minimum data movement. Theor. Comput. Sci. 165(2), 311–323 (1996); A preliminary version appeared in Shyamasundar, R.K. (ed.): FSTTCS 1992. LNCS, vol. 652, pp. 380–391. Springer, Heidelberg (1992)
[138] Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001); A Preliminary Version Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs Appeared in FOCS 1997: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, p. 118. IEEE Computer Society, Washington, DC (1997)
[139] Munro, J.I., Raman, V., Salowe, J.S.: Stable in situ sorting and minimum data movement. BIT 30(2), 220–234 (1990)

[140] Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. In: Arvind, V., Sarukkai, S. (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 186–197. Springer, Heidelberg (1998)
[141] Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: SODA 2001: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 529–536. Society for Industrial and Applied Mathematics, Philadelphia (2001)
[142] Munro, J.I., Ramirez, R.J.: Technical note - reducing space requirements for shortest path problems. Operations Research 30(5), 1009–1013 (1982)
[143] Munro, J.I., Robertson, E.L.: Parallel algorithms and serial data structures. In: Proceedings of the 17th Allerton Conference on Communication, Control and Computing, pp. 21–26 (1979)
[144] Munro, J.I., Robertson, E.L.: Continual pattern replication. Information and Control 48(3), 211–220 (1981)
[145] Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1006–1015. Springer, Heidelberg (2004)
[146] Munro, J.I., Spira, P.M.: Sorting and searching in multisets. SIAM J. Comput. 5(1), 1–8 (1976)
[147] Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. J. Comput. Syst. Sci. 21(2), 236–250 (1980); A Preliminary Version Appeared in STOC 1979: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, pp. 108–117. ACM, New York(1979)
[148] Oommen, B.J., Hansen, E.R., Munro, J.I.: Deterministic optimal and expedient move-to-rear list organizing strategies. Theor. Comput. Sci. 74(2), 183–197 (1990); A Preliminary Version Deterministic move-to-rear list Organizing Strategies with Optimal and Expedient Properties Appeared in Proceedings of the 25th Allerton Conference on Communication, Control and Computing (1987)
[149] Papadakis, T., Munro, J.I., Poblete, P.V.: Analysis of the expected search cost in skip lists. In: Gilbert, J.R., Karlsson, R. (eds.) SWAT 1990. LNCS, vol. 447, pp. 160–172. Springer, Heidelberg (1990)
[150] Papadakis, T., Munro, J.I., Poblete, P.V.: Average search and update costs in skip lists. BIT 32(2), 316–332 (1992)
[151] Poblete, P.V., Munro, J.I.: The analysis of a fringe heuristic for binary search trees. J. Algorithms 6(3), 336–350 (1985)
[152] Poblete, P.V., Munro, J.I.: Last-come-first-served hashing. J. Algorithms 10(2), 228–248 (1989)
[153] Poblete, P.V., Munro, J.I., Papadakis, T.: The binomial transform and the analysis of skip lists. Theor. Comput. Sci. 352(1), 136–158 (2006); A Preliminary Version The Binomial Transform and its Application to the Analysis of Skip lists appeared in Spirakis, P.G. (ed.): ESA 1995. LNCS, vol. 979, pp. 554–569. Springer, Heidelberg (1995)
[154] Poblete, P.V., Viola, A., Munro, J.I.: Analyzing the LCFS linear probing hashing algorithm with the help of Maple. MAPLETECH 4(1), 8–13 (1997)

[155] Poblete, P.V., Viola, A., Munro, J.I.: The diagonal Poisson transform and its application to the analysis of a hashing scheme. Random Struct. Algorithms 10(1-2), 221–255 (1997); A Preliminary Version The Analysis of a Hashing Schema by the Diagonal Poisson Transform (extended abstract) appeared in van Leeuwen, J. (ed.): ESA 1994. LNCS, vol. 855, pp. 94–405. Springer, Heidelberg (1994)
[156] Rahman, M.Z., Munro, J.I.: Integer representation and counting in the bit probe model. Algorithmica 56(1), 105–127 (2010); A Preliminary Version Appeared in Tokuyama, T. (ed.): ISAAC 2007. LNCS, vol. 4835, pp. 5–16. Springer, Heidelberg (2007)
[157] Ramírez, R.J., Tompa, F.W., Munro, J.I.: Optimum reorganization points for arbitrary database costs. Acta Inf. 18, 17–30 (1982)
[158] Robertson, E.L., Munro, J.I.: Generalized instant insanity and polynomial completeness. In: Proceedings of the 1975 Conference on Information Sciences and Systems: Papers Presented, April 2-4, p. 263. The Johns Hopkins University, Dept. of Electrical Engineering (1975)
[159] Robertson, E.L., Munro, J.I.: NP-completeness, puzzles and games. Utilitas Math. 13, 99–116 (1978)
[160] Schwenk, A.J., Munro, J.I.: How small can the mean shadow of a set be? The American Mathematical Monthly 90(5), 325–329 (1983)
[161] Tirdad, K., Ghodsnia, P., Munro, J.I., López-Ortiz, A.: COCA filters: Co-occurrence aware bloom filters. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 313–325. Springer, Heidelberg (2011)

Technical Reports

[162] Alstrup, S., Bender, M.A., Demaine, E.D., Farach-Colton, M., Munro, J.I., Rauhe, T., Thorup, M.: Efficient tree layout in a multilevel memory hierarchy. CoRR cs.DS/0211010 (2002)
[163] Biedl, T., Demaine, E.D., Demaine, M.L., Fleischer, R., Jacobsen, L., Munro, J.I.: The complexity of clickomania. CoRR cs.CC/0107031 (2001)
[164] Dobkin, D.P., Munro, J.I.: A minimal space selection algorithm that runs in linear time. Tech. Rep. 106, Department of Computer Science, University of Waterloo (1977)
[165] He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. CoRR abs/1104.5517 (2011)
[166] Karpinski, M., Munro, J.I., Nekrich, Y.: Range reporting for moving points on a grid. CoRR abs/1002.3511 (2010)
[167] Munro, J.I.: On random walks in binary trees. Tech. Rep. CS-76-33, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (1976)
[168] Munro, J.I., Poblete, P.V.: A lower bound for determining the median. Tech. Rep. CS-82-21, Faculty of Mathematics, University of Waterloo (1982)

Others

[169] Blum, M., Galil, Z., Ibarra, O.H., Kozen, D., Miller, G.L., Munro, J.I., Ruzzo, W.L.: Foreword. In: SFCS 1983: Proceedings of the 24th Annual Symposium on Foundations of Computer Science, p. iii. IEEE Computer Society, Washington, DC (1983)
[170] Chwa, K.Y., Munro, J.I.: Computing and combinatorics - Preface. Theor. Comput. Sci. 363(1), 1–1 (2006)
[171] López-Ortiz, A., Munro, J.I.: Foreword. ACM Transactions on Algorithms 2(4), 491 (2006)
[172] Munro, J.I.: Some results in the study of algorithms. Ph.D. thesis, University of Toronto (1971)
[173] Munro, J.I., Brodnik, A., Carlsson, S.: Digital memory structure and device, and methods for the management thereof, US Patent App. 09/863,313 (May 24, 2001)
[174] Munro, J.I., Brodnik, A., Carlsson, S.: A digital memory structure and device, and methods for the management thereof, eP Patent 1,141,951 (October 10, 2001)
[175] Munro, J.I., Wagner, D.: Preface. J. Exp. Algorithmics 14, 1:2.1–1:2.1 (2010), http://doi.acm.org/10.1145/1498698.1537596

Ph.D. Student Supervision

[1] Allen, B.: On Binary Search Trees (1977)
[2] Osborn, S.L.: Normal Forms for Relational Data Bases (1978)
[3] Suwanda, H.: Implicit Data Structures for the Dictionary Problem (1980)
[4] Pucci, W.C.: Average Case Selection (1983)
[5] Poblete, P.V.: Formal Techniques for Binary Search Trees (1983)
[6] Karlsson, R.G.: Algorithms in a Restricted Universe (1985)
[7] Celis, P.: Robin Hood Hashiong (1986)
[8] Culberson, J.C.: The Effect of Asymmetric Deletions on Binary Search Trees (1986)
[9] Raman, V.: Sorting In-Place with Minimum Data Movement (1991)
[10] Papadakis, T.: Skip Lists and Probabilistic Analysis of Algorithms (1993)
[11] Brodnik, A.: Searching in Constant Time and Minimum Space Minimae Res Magni Momenti Sunt (1995)
[12] Viola, A.: Analysis of Hashing Algorithms and a New Mathematical Transform (1995)
[13] Clark, D.: Compact PAT Trees (1997)
[14] Zhang, W.: Improving the Performance of Concurrent Sorts in Datsbase Systems (1997)
[15] Demaine, E.D.: Folding and Unfolding (2001)
[16] Golynski, A.: Upper and Lower Bounds for Text Indexing Data Structures (2007)
[17] He, M.: Succinct Indexes (2008)

[18] Skala, M.A.: Aspects of Metric Spaces in Computation (2008)
[19] Farzan, A.: Succinct Representation of Trees and Graphs (2009)
[20] Claude, F.: Space-Efficient Data Structures for Information Retrieval (2013)
[21] Salinger, A.J.: Models for Parallel Computation in Multi-Core, Heterogeneous, and Ultra Wide-Word Architectures

_ * _

[22] Nicholson, P.: Space efficient data structures in Word-RAM and Bitprobe Models (working title) (2013) (expected year of graduation)
[23] G. Zhou.

Professional Service and Honors

Fellow of Royal Society of Canada
ACM Fellow

Table of Contents

The Query Complexity of Finding a Hidden Permutation*

Peyman Afshani[1], Manindra Agrawal[2], Benjamin Doerr[3], Carola Doerr[3,4], Kasper Green Larsen[1], and Kurt Mehlhorn[3]

[1] MADALGO, Department of Computer Science, Aarhus University, Denmark
[2] Indian Institute of Technology Kanpur, India
[3] Max Planck Institute for Informatics, Saarbrücken, Germany
[4] Université Paris Diderot - Paris 7, LIAFA, Paris, France

Abstract. We study the query complexity of determining a hidden permutation. More specifically, we study the problem of learning a secret (z, π) consisting of a binary string z of length n and a permutation π of $[n]$. The secret must be unveiled by asking queries $x \in \{0,1\}^n$, and for each query asked, we are returned the score $f_{z,\pi}(x)$ defined as

$$f_{z,\pi}(x) := \max\{i \in [0..n] \mid \forall j \leq i : z_{\pi(j)} = x_{\pi(j)}\}\,;$$

i.e., the length of the longest common prefix of x and z with respect to π. The goal is to minimize the number of queries asked. We prove matching upper and lower bounds for the deterministic and randomized query complexity of $\Theta(n \log n)$ and $\Theta(n \log \log n)$, respectively.

1 Introduction

Query complexity, also referred to as decision tree complexity, is one of the most basic models of computation. We aim at learning an unknown object (a secret) by asking queries of a certain type. The cost of the computation is the number of queries made until the secret is unveiled. All other computation is free.

Let S_n denote the set of permutations of $[n] := \{1, \ldots, n\}$; let $[0..n] := \{0, 1, \ldots, n\}$. Our problem is that of learning a hidden permutation $\pi \in S_n$ together with a hidden bit-string $z \in \{0,1\}$ through queries of the following type. A query is again a bit-string $x \in \{0,1\}^n$. As answer we receive the length of the longest common prefix of x and z in the order of π, which we denote by

$$f_{z,\pi}(x) := \max\{i \in [0..n] \mid \forall j \leq i : z_{\pi(j)} = x_{\pi(j)}\}\,.$$

We call this problem the HiddenPermutation problem. It is a Mastermind-like problem; however, the secret is now a permutation and a string and not just a string. Figure 1 sketches a gameboard for the HiddenPermutation game.

It is easy to see that $O(n \log n)$ queries suffice deterministically to unveil the secret. Doerr and Winzen [1] showed that randomization allows to do better.

* The full paper is available at `http://eccc.hpi-web.de/report/2012/087/`

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 1–11, 2013.

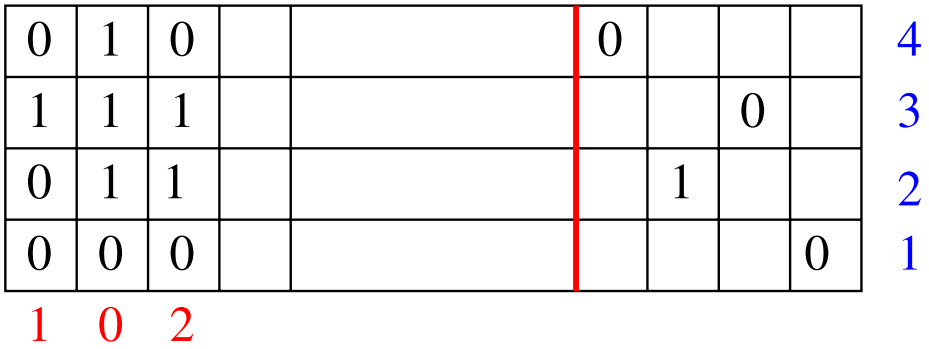


Fig. 1. A gameboard for the HiddenPermutation game for $n = 4$. The first player (codemaker) chooses z and π by placing a string in $\{0,1\}^4$ into the 4×4 grid on the right side, one digit per row and column; here, $z = 0100$ and $\pi(1) = 4$, $\pi(2) = 2$, $\pi(3) = 3$, and $\pi(4) = 1$. The second player (codebreaker) places its queries into the columns on the left side of the board. The score is shown below each column. The computation of the score by the codemaker is simple. She goes through the codematrix column by column and advances as long as the query and the code agrees.

They gave a randomized algorithm with $O(n \log n / \log\log n)$ expected complexity. The information-theoretic lower bound is only $\Theta(n)$ as the answer to each query is a number between zero and n and hence may reveal as many as $\log n$ bits. We show: the deterministic query complexity is $\Theta(n \log n)$, cf. Section 3, and the randomized query complexity is $\Theta(n \log\log n)$, cf. Sections 4 and 5. Both upper bound strategies are efficient, i.e., can be implemented in polynomial time. The lower bound is established by a (standard) adversary argument in the deterministic case and by a potential function argument in the randomized case. The randomized upper and lower bound require a non-trivial argument.

The archetypal guessing game is *Mastermind*. The secret is a string $z \in [k]^n$, and a query is also a string $x \in [k]^n$. The answer to a query is the number $\mathrm{eq}(z,x)$ of positions in which x and z agree and the number $w(z,x)$ of additional colors in x that appear in z (formally, $w(z,x) := \max_{\pi \in S_n} |\{i \in [n] \mid z_i = x_{\pi(i)}\}| - \mathrm{eq}(z,x)$). Some applications were found recently [2,3]. Mastermind has been studied intensively since the sixties [4,5,6,7,8,9] and thus even before it was invented as a board game. In particular, [4,6] show that for all n and $k \leq n^{1-\varepsilon}$, the secret code can be found by asking $\Theta(n \log k / \log n)$ random queries. This can be turned into a deterministic strategy having the same asymptotic complexity. The information-theoretic lower bound of $\Omega(n \log k / \log n)$ shows that this is best possible, and also, that there is no difference between the randomized and deterministic case. Similar situations have been observed for a number of guessing, liar, and pusher-chooser games (see, e.g., [10,11]). Our results show that the situation is different for the HiddenPermutation game. The complexity of Mastermind with n positions and $k = n$ colors is open. The best upper bound is $O(n \log\log n)$, cf. [12], and the best lower bound is the trivial linear one.

2 Preliminaries

For all positive integers $k \in \mathbb{N}$ we define $[k] := \{1, \ldots, k\}$ and $[0..k] := [k] \cup \{0\}$. By e_k^n we denote the kth unit vector $(0, \ldots, 0, 1, 0, \ldots, 0)$ of length n. For a set $I \subseteq [n]$ we define $e_I^n := \sum_{i \in I} e_i^n = \oplus_{i \in I} e_i^n$, where $\oplus$ denotes the bitwise exclusive-or. We say that we *create y from x by flipping I* or that we *create y from x by flipping the entries in position(s) I* if $y = x \oplus e_I^n$. By S_n we denote the set of all permutations of $[n]$. For $r \in \mathbb{R}_{\geq 0}$, let $\lceil r \rceil := \min\{n \in \mathbb{N}_0 \mid n \geq r\}$. and $\lfloor r \rfloor := \max\{n \in \mathbb{N}_0 \mid n \leq r\}$. To increase readability, we sometimes omit the $\lceil \cdot \rceil$ signs; that is, whenever we write r where an integer is required, we implicitly mean $\lceil r \rceil$.

Let $n \in \mathbb{N}$. For $z \in \{0,1\}^n$ and $\pi \in S_n$, define $f_{z,\pi} : \{0,1\}^n \to [0..n]$ as in the introduction. We call z the *target string* and π the *target permutation*. The score of a query x^i is $s^i = f_{z,\pi}(x^i)$. We may stop after t queries x^1 to x^t if there is only a *single* pair $(z, \pi) \in \{0,1\}^n \times S_n$ with $s^i = f_{z,\pi}(x^i)$ for $1 \leq i \leq t$.

A randomized strategy for the HiddenPermutation problem is a tree of outdegree $n+1$ in which a probability distribution over $\{0,1\}^n$ is associated with every node of the tree. The search starts as the root. In any node, the query is selected according to the probability distribution associated with the node, and the search proceeds to the child selected by the score. The complexity of a strategy on input (z, π) is the expected number of queries required to identify the secret, and the randomized query complexity of a strategy is the worst case over all secrets. Deterministic strategies are the special case in which a fixed query is associated with every node. The deterministic (randomized) query complexity of HiddenPermutation is the best possible (expected) complexity.

We remark that knowing z allows us to determine π with $n-1$ queries $z \oplus e_i^n$, $1 \leq i < n$. Observe that $\pi^{-1}(i)$ equals $f_{z,\pi}(z \oplus e_i^n) + 1$. Conversely, knowing the target permutation π we can identify z in a linear number of guesses. The first query is arbitrary. If our current query x has a score of k, we next query the string x' created from x by flipping the entry in position $\pi(k+1)$. Thus, learning one part of the secret is no easier (up to $O(n)$ questions) than learning the full.

A simple information-theoretic argument gives an $\Omega(n)$ lower bound for the deterministic query complexity and, together with Yao's minimax principle [13], also for the randomized complexity. The *search space* has size $2^n n!$, since the unknown secret is an element of $\{0,1\}^n \times S_n$. That is, we need to "learn" $\Omega(n \log n)$ bits of information. Each *score* is a number between 0 and n, i.e., we learn at most $O(\log n)$ bits of information per query, and the $\Omega(n)$ bound follows.

Let $\mathcal{H} := (x^i, s^i)_{i=1}^t$ be a vector of queries $x^i \in \{0,1\}^n$ and scores $s^i \in [0..n]$. We call $\mathcal{H}$ a *guessing history*. A secret (z, π) is *consistent with* $\mathcal{H}$ if $f_{z,\pi}(x^i) = s^i$ for all $i \in [t]$. $\mathcal{H}$ is *feasible* if there exists a secret consistent with it.

An observation crucial in our proofs is the fact that a vector $(V_1, \ldots, V_n)$ of subsets of $[n]$, together with a top score query (x^*, s^*), captures the total knowledge provided by a *guessing history* $\mathcal{H} = (x^i, s^i)_{i=1}^t$ about the set of secrets consistent with $\mathcal{H}$. We will call V_j the *candidate set* for position j; V_j will contain all indices $i \in [n]$ for which the following simple rules (1) to (3) do not rule out that $\pi(j)$ equals i.

Theorem 1. *Let $t \in \mathbb{N}$, and let $\mathcal{H} = (x^i, s^i)_{i=1}^t$ be a guessing history. Construct the candidate sets $V_1, \ldots, V_n \subseteq [n]$ according to the following rules:*

(1) If there are h and ℓ with $j \le s^h \le s^\ell$ and $x_i^h \neq x_i^\ell$, then $i \notin V_j$.

(2) If there are h and ℓ with $s = s^h = s^\ell$ and $x_i^h \neq x_i^\ell$, then $i \notin V_{s+1}$.

(3) If there are h and ℓ with $s^h < s^\ell$ and $x_i^h = x_i^\ell$, then $i \notin V_{s^h+1}$.

(4) If i is not excluded by one of the rules above, then $i \in V_j$.

Furthermore, let $s^* := \max\{s^1, \ldots, s^t\}$ and let $x^* = x^j$ for some j with $s^j = s^*$.

Then a pair (z, π) is consistent with $\mathcal{H}$ if and only if (a) $f_{z,\pi}(x^*) = s^*$ and (b) $\pi(j) \in V_j$ for all $j \in [n]$.

Proof. Let (z, π) satisfy conditions (a) and (b). We show that (z, π) is consistent with $\mathcal{H}$. To this end, let $h \in [t]$, let $x = x^h$, $s = s^h$, and $f := f_{z,\pi}(x)$. We need to show $f = s$.

Assume $f < s$. Then $z_{\pi(f+1)} \neq x_{\pi(f+1)}$. Since $f+1 \le s^*$, this together with (a) implies $x_{\pi(f+1)} \neq x^*_{\pi(f+1)}$. Rule (1) yields $\pi(f+1) \notin V_{f+1}$; a contradiction to (b).

Similarly, if we assume $f > s$, then $x_{\pi(s+1)} = z_{\pi(s+1)}$. We distinguish two cases. If $s < s^*$, then by condition (a) we have $x_{\pi(s+1)} = x^*_{\pi(s+1)}$. By rule (3) this implies $\pi(s+1) \notin V_{s+1}$; a contradiction to (b).

On the other hand, if $s = s^*$, then $x_{\pi(s+1)} = z_{\pi(s+1)} \neq x^*_{\pi(s+1)}$ by (a). Rule (2) implies $\pi(s+1) \notin V_{\pi(s+1)}$, again contradicting (b).

Necessity is trivial. □

We may construct the sets V_j incrementally. The following update rules are direct consequences of Theorem 1. In the beginning, let $V_j := [n]$, $1 \le j \le n$. After the first query, record the first query as x^* and its score as s^*. For all subsequent queries, do the following: Let I be the set of indices in which the current query x and the current best query x^* agree. Let s be the objective value of x and let s^* be the objective value of x^*.

Rule A: If $s < s^*$, then $V_i \leftarrow V_i \cap I$ for $1 \le i \le s$ and $V_{s+1} \leftarrow V_{s+1} \setminus I$.

Rule B: If $s = s^*$, then $V_i \leftarrow V_i \cap I$ for $1 \le i \le s^* + 1$.

Rule C: If $s > s^*$, then $V_i \leftarrow V_i \cap I$ for $1 \le i \le s^*$ and $V_{s^*+1} \leftarrow V_{s^*+1} \setminus I$. We further replace $s^* \leftarrow s$ and $x^* \leftarrow x$.

It is immediate from the update rules that the V_js form a *laminar family*; i.e., for $i < j$ either $V_i \cap V_j = \emptyset$ or $V_i \subseteq V_j$. As a consequence of Theorem 1 we obtain a polynomial time test for the feasibility of histories. It gives additional insight in the meaning of the candidate sets $V_1, \ldots, V_n$.

Theorem 2. *It is decidable in polynomial time whether a guessing history is feasible. Furthermore, we can efficiently compute the number of pairs consistent with it.*

3 Deterministic Complexity

We show that the deterministic query complexity of HiddenPermutation is $\Theta(n \log n)$.

The *upper bound* is achieved by an algorithm that resembles binary search and iteratively identifies $\pi(1), \ldots, \pi(n)$ and the corresponding bit values $z_{\pi(1)}, \ldots, z_{\pi(n)}$: We start by querying the all-zeros string 0^n and the all-ones string 1^n. The scores determine $z_{\pi(1)}$. By flipping a set I containing half of the bit positions in the better (the one achieving largest score) of the two strings, we can determine whether $\pi(1) \in I$ or not. This allows us to find $\pi(1)$ via a binary search strategy in $O(\log n)$ queries. Once $\pi(1)$ and $z_{\pi(1)}$ are known, we iterate this strategy on the remaining bit positions to determine $\pi(2)$ and $z_{\pi(2)}$, and so on, yielding an $O(n \log n)$ query strategy for identifying the secret.

We proceed to the lower bound. The adversary strategy proceeds in rounds. In every round, the adversary reveals the next two values of π and the corresponding bits of z and every algorithm uses $\Omega(\log n)$ queries. We describe the first phase. Let x_1 be the first query. The adversary gives it a score of 1, sets $(x^*, s^*) = (x_1, 1)$ and V_i to $[n]$ for $1 \leq i \leq n$. In the first phase, the adversary will only return scores 0 and 1; observe that according to the rules for the incremental update of the sets V_i, only sets V_1 and V_2 will be modified in the first phase, and all other V_is stay equal to the $[n]$.

Let x be the next query and assume $|V_1| \geq 3$. Let $I = \{i \mid x_i = x_i^*\}$ be the set of positions in which x and x^* agree. If $|I \cap V_1| \geq |V_1|/2$, the adversary returns a score of 1 and replaces V_1 by $V_1 \cap I$ and V_2 by $V_2 \cap I$. Otherwise, the adversary returns a score of 0 and replaces V_1 by $V_1 \setminus I$. In either case, the cardinality of V_1 is at most halved and V_1 stays a subset of V_2. The adversary proceeds as long as $|V_1| \geq 3$ before the query. Then $|V_1| \geq 2$ after the answer by the adversary.

If $|V_1| = 2$ before the query, the adversary starts the next phase by giving x a score of 3. Let $i_1 \in V_1$ and $i_2 \in V_2$ be arbitrary. The adversary commits to $\pi(1) = i_1$, $\pi(2) = i_2$, $z_{i_1} = x^*_{i_1}$, and $z_{i_2} = 1 - x^*_{i_2}$, removes i_1 and i_2 from V_3 to V_n, and sets $(x^*, s^*) = (x, 3)$.

Theorem 3. *The deterministic query complexity of the* HiddenPermutation *problem with n positions is $\Theta(n \log n)$.*

4 The Randomized Strategy

We show that the randomized query complexity is only $O(n \log \log n)$. Our randomized strategy learns an expected number of $\Theta(\log n / \log \log n)$ bits per query, and we have already seen that deterministic strategies can only learn a constant number of bits per query in the worst case. In the language of the candidate sets V_i, we manage to reduce the sizes of many V_is in parallel, that is, we gain information on several $\pi(i)$s despite the seemingly sequential way $f_{z,\pi}$ offers information. The key to this is using partial information given by the V_i (that is, information that does not determine π_i, but only restricts it) to guess with good probability an x with $f_{z,\pi}(x) > s^*$.

Theorem 4. *The randomized query complexity of the* HiddenPermutation *problem with n positions is $O(n \log\log n)$.*

The strategy has two parts. In the main part, we identify the positions $\pi(1), \ldots, \pi(q)$ and the corresponding bit values $z_{\pi(1)}, \ldots, z_{\pi(q)}$ for some $q \in n - \Theta(n/\log n)$ with $O(n \log\log n)$ queries. In the second part, we find the remaining $n-q \in \Theta(n/\log n)$ positions and entries using the binary search algorithm with $O(\log n)$ queries per position. Part 1 is outlined below; the details are given in the full paper.

4.1 The Main Strategy

Here and in the following we denote by s^* the current best score, and by x^* we denote a corresponding query; i.e., $f_{z,\pi}(x^*) = s^*$. For brevity, we write f for $f_{z,\pi}$.

There is a trade-off between learning more information about π by reducing the sets $V_1, \ldots, V_{s^*+1}$ and increasing the score s^*. Our main task is to find the right balance between the two. In our $O(n \log\log n)$ strategy, we partition the sets $V_1, \ldots, V_n$ into several *levels*, each of which has a certain capacity. Depending on the status (i.e., the fill rate) of these levels, either we try to increase s^*, or we aim at reducing the sizes of the candidate sets.

In the beginning, all candidate sets $V_1, \ldots, V_n$ belong to level 0. In the *first step* we aim at moving $V_1, \ldots, V_{\log n}$ to the first level. This is done sequentially. We start by querying $f(x)$ and $f(y)$, where x is arbitrary and $y = x \oplus 1^n$ is the bitwise complement of x. By swapping x and y if needed, we may assume $f(x) = 0 < f(y)$. We now run a randomized binary search for finding $\pi(1)$. We choose uniformly at random a subset $F_1 \subseteq V_1$ ($V_1 = [n]$ in the beginning) of size $|F_1| = |V_1|/2$. We query $f(y')$ where y' is obtained from y by flipping the bits in F_1. If $f(y') > f(x)$, we set $V_1 \leftarrow V_1 \setminus F_1$; we set $V_1 \leftarrow F_1$ otherwise. This ensures $\pi(1) \in V_1$. We stop this binary search once $\pi(2) \notin V_1$ is sufficiently likely; the analysis will show that $\Pr[\pi(2) \in V_1] \le 1/\log^d n$ (and hence $|V_1| \le n/\log^d n$) for some large enough constant d is a good choice.

We now start pounding on V_2. Let $\{x, y\} = \{y, y \oplus 1_{[n]\setminus V_1}\}$. If $\pi(2) \notin V_1$, one of $f(x)$ and $f(y)$ is one and the other is larger than one. Swapping x and y if necessary, we may assume $f(x) = 1 < f(y)$. We now use randomized binary search to reduce the size of V_2 to $n/\log^d n$. The randomized binary search is similar to before. Initially we have $V_2 = [n] \setminus V_1$. At each step we chose a subset $F_2 \subseteq V_2$ of size $|V_2|/2$ and we create y' from y by flipping the bits in positions F_2. If $f(y') = 1$ we update V_2 by F_2 and we update V_2 by $V_2 \setminus F_2$ otherwise. We stop once $|V_2| \le n/\log^d n$.

At this point we have $|V_1|, |V_2| \le n/\log^d n$ and $V_1 \cap V_2 = \emptyset$. We hope that $\pi(3) \notin V_1 \cup V_2$, in which case we move set V_3 from level 0 to level 1 (the case $\pi(3) \in V_1 \cup V_2$ is called a *failure* and needs to be treated separately. In case of a failure we *abort* the first level and we move V_1 and V_2 to the second level by decreasing their sizes to at most $n/\log^{2d} n$, we potentially move them further to the third level, and so on until we finally have $\pi(3) \notin V_1 \cup V_2$, in which case we move V_3 to level 1 as before).

At some point the probability that $\pi(i) \notin V_1 \cup \ldots \cup V_{i-1}$ drops below a certain threshold and we cannot ensure to make progress anymore by simply querying $y \oplus ([n] \backslash (V_1 \cup \ldots \cup V_{i-1}))$. This situation is reached when $i = \log n$ and hence we abandon the previously described strategy once $s^* = \log n$. At this point, we move our focus from increasing the current best score s^* to reducing the size of the candidate sets $V_1, \ldots, V_{s^*}$, thus adding them to the second level. More precisely, we reduce their sizes to at most $n/\log^{2d} n$. This reduction is carried out by `subroutine2`, which we describe in the full paper. It reduces the sizes of the up to $x_{\ell-1}$ candidate sets from some value $\leq n/x_{\ell-1}^d$ to the target size n/x_ℓ^d of level ℓ with an expected number of $O(1)x_{\ell-1}d(\log(x_\ell) - \log(x_{\ell-1}))/\log(x_{\ell-1})$ queries.

Once the sizes $|V_1|, \ldots, |V_{s^*}|$ have been reduced to at most $n/\log^{2d} n$, we move our focus back to increasing s^*. The probability that $\pi(s^*+1) \in V_1 \cup \ldots \cup V_{s^*}$ will now be small enough , and we proceed as before by flipping $[n] \setminus (V_1 \cup \ldots \cup V_{s^*})$ and reducing the size of V_{s^*+1} to $n/\log^d n$. Again we iterate this process until the first level is filled; i.e., until we have $s^* = 2\log n$. As we did with $V_1, \ldots, V_{\log n}$, we reduce the sizes of $V_{\log n+1}, \ldots, V_{2\log n}$ to $n/\log^{2d} n$, thus adding them to the second level. We iterate this process of moving $\log n$ sets from level 0 to level 1 and then moving them to the second level until $\log^2 n$ sets have been added to the second level. At this point the second level has reached its capacity and we proceed by reducing the sizes of $V_1, \ldots, V_{\log^2 n}$ to at most $n/\log^{4d} n$, thus adding them to the *third level.*

In total we have $t = O(\log\log n)$ levels. For $1 \leq i \leq t$, the ith level has a capacity of $x_i := \log^{2^{i-1}} n$ sets, each of which is required to be of size at most n/x_i^d. Once level i has reached its capacity, we reduce the size of the sets on the ith level to at most n/x_{i+1}^d, thus moving them from level i to level $i+1$. When x_t sets $V_i, \ldots, V_{i+x_t}$ have been added to the last level, level t, we finally reduce their sizes to one. This corresponds to determining $\pi(i+j)$ for each $j \in [x_t]$.

This concludes the presentation of the main ideas of the first phase.

5 The Lower Bound

In this section, we prove a tight lower bound for the randomized query complexity of the HiddenPermutation problem. The lower bound is stated in the following:

Theorem 5. *The randomized query complexity of the* HiddenPermutation *problem with n positions is $\Omega(n \log\log n)$.*

To prove a lower bound for randomized query schemes, we appeal to Yao's principle. That is, we first define a hard distribution over the secrets and show that every deterministic query scheme for this hard distribution needs $\Omega(n \log\log n)$ queries in expectation. This part of the proof is done using a potential function argument.

Hard Distribution. Let Π be a permutation drawn uniformly among all the permutations of $[n]$ (in this section, we use capital letters to denote random

variables). Given such a permutation, we let our target string Z be the one satisfying $Z_{\Pi(i)} = (i \bmod 2)$ for $i = 1, \ldots, n$. Since Z is uniquely determined by the permutation Π, we will mostly ignore the role of Z in the rest of this section. Finally, we use $F(x)$ to denote the value of the random variable $f_{Z,\Pi}(x)$ for $x \in \{0,1\}^n$. We will also use the notation $a \equiv b$ to mean that $a \equiv b \bmod 2$.

Deterministic Query Schemes. By fixing the random coins, a randomized solution with expected t queries implies the existence of a deterministic query scheme with expected t queries over our hard distribution. The rest of this section is devoted to lower bounding t for such a deterministic query scheme.

A deterministic query scheme is a decision tree T in which each node v is labeled with a string $x_v \in \{0,1\}^n$. Each node has $n+1$ children, numbered from 0 to n, and the ith child is traversed if $F(x_v) = i$. To guarantee correctness, no two inputs can end up in the same leaf.

For a node v in the decision tree T, we define $\max_v$ as the largest value of F seen along the edges from the root to v. Note that $\max_v$ is not a random variable and in fact, at any node v and for any ancestor u of v, conditioned on the event that the search path reaches v, the value of $F(x_u)$ is equal to the index of the child of u that lies on the path to v. Finally, we define S_v as the subset of inputs (as outlined above) that reach node v.

We use a potential function which measures how much "information" the queries asked have revealed about Π. Our goal is to show that the expected increase in the potential function after asking each query is small. Our potential function depends crucially on the candidate sets. The update rules for the candidate sets are slightly more specific than the ones in Section 2 because we now have a fixed connection between the two parts of the secret. We denote the candidate set for $\pi(i)$ at node v with V_i^v. At the root node r, we have $V_i^r = [n]$ for all i. Let v be a node in the tree and let $w_0, \ldots, w_n$ be its children (w_i is traversed when the score i is returned). Let P_0^v (resp. P_1^v) be the set of positions in x_v that contain 0 (resp. 1). Thus, formally, $P_0^v = \{i \mid x_v[i] = 0\}$ and $P_1^v = \{i \mid x_v[i] = 1\}$.[1] The precise definition of candidate sets is as follows:

$$V_i^{w_j} = \begin{cases} V_i^v \cap P_{i \bmod 2}^v & \text{if } i \leq j\,, \\ V_i^v \cap P_{j \bmod 2}^v & \text{if } i = j+1\,, \\ V_i^v & \text{if } i > j+1. \end{cases}$$

As with the upper bound case, the candidate sets have some very useful properties. These properties are slightly different from the ones observed before, due to the fact that some extra information has been announced to the query algorithm. We say that a candidate set V_i^v is *active (at v)* if the following conditions are met: (i) at some ancestor node u of v, we have $F(x_u) = i - 1$, (ii) at every ancestor node w of u we have $F(x_w) < i - 1$, and (iii) $i < \min\{n/3, \max_v\}$. We call $V_{\max_v+1}^v$ *pseudo-active (at v)*.

For intuition on the requirement $i < n/3$, observe from the following lemma that $V_{\max_v+1}^v$ contains all sets V_i^v for $i \leq \max_v$ and $i \equiv \max_v$. At a high level,

[1] To prevent our notations from becoming too overloaded, here and in the remainder of the section we write $x = (x[1], \ldots, x[n])$ instead of $x = (x_1, \ldots, x_n)$.

this means that the distribution of $\Pi(\max_v+1)$ is not independent of $\Pi(i)$ for $i \equiv \max_v$. The bound $i < n/3$, however, forces the dependence to be rather small (there are not too many such sets). This greatly helps in the potential function analysis.

In the full paper, we show (using similar arguments as for showing Theorem 1) that the candidate sets satisfy the following:

Lemma 1. *The candidate sets have the following properties:*

(i) Two candidate sets V_i^v and V_j^v with $i < j \leq \max_v$ and $i \not\equiv j$ are disjoint.

(ii) An active candidate set V_j^v is disjoint from any candidate set V_i provided $i < j < \max_v$.

(iii) The candidate set V_i^v, $i \leq \max_v$ is contained in the set $V_{\max_v+1}^v$ if $i \equiv \max_v$ and is disjoint from it if $i \not\equiv \max_v$.

(iv) For two candidate sets V_i^v and V_j^v, $i < j$, if $V_i^v \cap V_j^v \neq \emptyset$ then $V_i^v \subset V_j^v$.

5.1 Potential Function

We define the potential of an active candidate set V_i^v as $\log\log(2n/|V_i^v|)$. This is inspired by the upper bound: a potential increase of 1 corresponds to a candidate set advancing one level in the upper bound context (in the beginning, a set V_i^v has size n and thus its potential is 0 while at the end its potential is $\Theta(\log\log n)$. With each level, the quantity n divided by the size of V_i is squared). We define the potential at a node v as

$$\varphi(v) = \log\log \frac{2n}{|V^v_{\max_v+1}| - \mathrm{Con}_v} + \sum_{j \in A_v} \log\log \frac{2n}{|V_j^v|}\,,$$

in which A_v is the set of indices of active candidate sets at v and Con_v is the number of candidate sets contained inside $V^v_{\max_v+1}$. Note that from Lemma 1, it follows that $\mathrm{Con}_v = \lfloor \max_v/2 \rfloor$.

The intuition for including the term Con_v is the same as our requirement $i < n/3$ in the definition of active candidate sets, namely that once Con_v approaches $|V^v_{\max_v+1}|$, the distribution of $\Pi(\max_v+1)$ starts depending heavily on the candidate sets V_i^v for $i \leq \max_v$ and $i \equiv \max_v$. Thus we have in some sense determined $\Pi(\max_v+1)$ already when $|V^v_{\max_v+1}|$ approaches Con_v. Therefore, we have to take this into account in the potential function since otherwise changing $V^v_{\max_v+1}$ from being pseudo-active to being active could give a huge potential increase.

After some lengthy calculations, it is possible to prove the following lemma.

Lemma 2. *Let v be a node in T and let $\boldsymbol{i}_v$ be the random variable giving the value of $F(x_v)$ when $\Pi \in S_v$ and 0 otherwise. Also let $w_0, \ldots, w_n$ denote the children of v, where w_j is the child reached when $F(x_v) = j$. Then, $\mathbb{E}[\varphi(w_{\boldsymbol{i}_v}) - \varphi(v) \mid \Pi \in S_v] = O(1)$.*

5.2 Potential at the End

Intuitively, if the maximum score value increases after a query, it increases, in expectation, only by an additive constant. In fact, in the full paper, we prove

that the probability of increasing the maximum score value by α after one query is $2^{-\Omega(\alpha)}$. Thus, it follows from the definition of the active candidate sets that when the score reaches $n/3$ we expect $\Omega(n)$ active candidate sets. However, by Lemma 1, the active candidate sets are disjoint. This means that a fraction of them (again at least $\Omega(n)$ of them), must be small, or equivalently, their total potential is $\Omega(n \log\log n)$.

Lemma 3. *Let ℓ be the random variable giving the leaf node of T that the deterministic query scheme ends up in on input Π. We have $\varphi(\ell) = \Omega(n \log\log n)$ with probability at least $3/4$.*

5.3 Putting Things together

Finally, we show how Lemma 2 and Lemma 3 combine to give our lower bound. Essentially this boils down to showing that if the query scheme is too efficient, then the query asked at some node of T increases the potential by $\omega(1)$ in expectation, contradicting Lemma 2. To show this explicitly, define $\mathbf{t}$ as the random variable giving the number of queries asked on input Π. We have $\mathbb{E}[\mathbf{t}] = t$, where t was the expected number of queries needed for the deterministic query scheme. Also let $\ell_1, \ldots, \ell_{4t}$ be the random variables giving the first $4t$ nodes of T traversed on input Π, where $\ell_1 = r$ is the root node and ℓ_i denotes the node traversed at the ith level of T. If only $m < 4t$ nodes are traversed, define $\ell_i = \ell_m$ for $i > m$; i.e., $\varphi(\ell_i) = \varphi(\ell_m)$. From Lemma 3, Markov's inequality and a union bound, we may now write

$$\begin{aligned}\mathbb{E}[\varphi(\ell_{4t})] &= \mathbb{E}\left[\varphi(\ell_1) + \sum_{i=1}^{4t-1} \varphi(\ell_{i+1}) - \varphi(\ell_i)\right] = \mathbb{E}[\varphi(r)] + \mathbb{E}\left[\sum_{i=1}^{4t-1} \varphi(\ell_{i+1}) - \varphi(\ell_i)\right] \\ &= \sum_{i=1}^{4t-1} \mathbb{E}[\varphi(\ell_{i+1}) - \varphi(\ell_i)] = \Omega(n \log\log n).\end{aligned}$$

Hence there exists a value i^*, where $1 \leq i^* \leq 4t-1$, such that

$$\mathbb{E}[\varphi(\ell_{i^*+1}) - \varphi(\ell_{i^*})] = \Omega(n \log\log n/t).$$

But

$$\mathbb{E}[\varphi(\ell_{i^*+1}) - \varphi(\ell_{i^*})] = \sum_{v \in T_{i^*} \mid v \text{ non-leaf}} \Pr[\Pi \in S_v]\, \mathbb{E}[\varphi(w_{\mathbf{i}_v}) - \varphi(v) \mid \Pi \in S_v],$$

where T_{i^*} is the set of all nodes at depth i^* in T, $w_0, \ldots, w_n$ are the children of v and $\mathbf{i}_v$ is the random variable giving the score of $F(x_v)$ on an input $\Pi \in S_v$ and 0 otherwise. Since the events $\Pi \in S_v$ and $\Pi \in S_u$ are disjoint for $v \neq u$, we conclude that there must exist a node $v \in T_{i^*}$ for which

$$\mathbb{E}[\varphi(w_{\mathbf{i}_v}) - \varphi(v) \mid \Pi \in S_v] = \Omega(n \log\log n/t).$$

Combined with Lemma 2 this shows that $n \log\log n/t = O(1)$; i.e., $t = \Omega(n \log\log n)$. This concludes the proof of Theorem 5.

References

1. Doerr, B., Winzen, C.: Black-box complexity: Breaking the $o(n \log n)$ barrier of LeadingOnes. In: Hao, J.-K., Legrand, P., Collet, P., Monmarché, N., Lutton, E., Schoenauer, M. (eds.) EA 2011. LNCS, vol. 7401, pp. 205–216. Springer, Heidelberg (2012)
2. Goodrich, M.T.: The Mastermind attack on genomic data. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP 2009), pp. 204–218. IEEE (2009)
3. Focardi, R., Luccio, F.L.: Cracking bank PINs by playing Mastermind. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 202–213. Springer, Heidelberg (2010)
4. Erdős, P., Rényi, A.: On two problems of information theory. Magyar Tudományos Akadémia Matematikai Kutató Intézet Közleményei 8, 229–243 (1963)
5. Knuth, D.E.: The computer as master mind. Journal of Recreational Mathematics 9, 1–5 (1977)
6. Chvátal, V.: Mastermind. Combinatorica 3, 325–329 (1983)
7. Chen, Z., Cunha, C., Homer, S.: Finding a hidden code by asking questions. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 50–55. Springer, Heidelberg (1996)
8. Goodrich, M.T.: On the algorithmic complexity of the Mastermind game with black-peg results. Information Processing Letters 109, 675–678 (2009)
9. Viglietta, G.: Hardness of Mastermind. In: Kranakis, E., Krizanc, D., Luccio, F. (eds.) FUN 2012. LNCS, vol. 7288, pp. 368–378. Springer, Heidelberg (2012)
10. Pelc, A.: Searching games with errors — fifty years of coping with liars. Theoretical Computer Science 270, 71–109 (2002)
11. Spencer, J.: Randomization, derandomization and antirandomization: Three games. Theoretical Computer Science 131, 415–429 (1994)
12. Doerr, B., Spöhel, R., Thomas, H., Winzen, C.: Playing Mastermind with many colors. In: SODA 2013, pp. 695–704. SIAM (2013)
13. Yao, A.C.C.: Probabilistic computations: Toward a unified measure of complexity. In: FOCS 1977, pp. 222–227. IEEE (1977)

Bounds for Scheduling Jobs on Grid Processors*

Joan Boyar[1] and Faith Ellen[2]

[1] University of Southern Denmark
joan@imada.sdu.dk
[2] University of Toronto
faith@cs.toronto.edu

Abstract. In the Grid Scheduling problem, there is a set of jobs each with a positive integral memory requirement. Processors arrive in an online manner and each is assigned a maximal subset of the remaining jobs such that the sum of the memory requirements of those jobs does not exceed the processor's memory capacity. The goal is to assign all the jobs to processors so as to minimize the sum of the memory capacities of the processors that are assigned at least one job. Previously, a lower bound of $\frac{5}{4}$ on the competitive ratio of this problem was achieved using jobs of size S and $2S-1$. For this case, we obtain matching upper and lower bounds, which vary depending on the ratio of the number of small jobs to the number of large jobs.

1 Introduction

The Grid is a computing environment comprised of various processors which arrive at various times and to which jobs can be assigned. We consider the problem of scheduling a set of jobs, each with a specific memory requirement. When a processor arrives, it announces its memory capacity. Jobs are assigned to the processor so that the sum of the requirements of its assigned jobs does not exceed its capacity. In this way, the processor can avoid the expensive costs of paging when it executes the jobs. The charge for a set of jobs is (proportional to) the memory capacities of the processors to which they are assigned. There is no charge for processors whose capacities are too small for any remaining jobs. The goal is to assign all the jobs to processors in a manner that minimizes the total charge.

The Grid Scheduling problem was motivated by a problem in bioinformatics in which genomes are compared to a very large database of DNA sequences to identify regions of interest [2]. In this application, an extremely large problem is divided into a set of independent jobs with varying memory requirements.

* This research was supported in part by the Danish Council for Independent Research, Natural Sciences (FNU), the VELUX Foundation, and the Natural Science and Engineering Research Council of Canada (NSERC). Parts of this work were carried out while Joan Boyar was visiting University of Waterloo and University of Toronto and while Faith Ellen was visiting University of Southern Denmark.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 12–26, 2013.

The Grid Scheduling problem can also be rephrased as a bin packing problem for a given set of items, using variable-sized bins, which arrive one by one, in an online manner. In contrast, usual bin packing problems assume the bins are given and the items arrive online.

The Grid Scheduling problem was introduced by Boyar and Favrholdt in [1]. They gave an algorithm to solve this problem with competitive ratio $\frac{13}{7}$. This solved an open question in [8], which considered a similar problem. They also proved that the competitive ratio of any algorithm for this problem is at least $\frac{5}{4}$. The lower bound proof uses $s = 2\ell$ items of size S and ℓ items of size $L = 2S - 1$, where $S > 1$ is an integer and $M = 2L - 1$ is the maximum bin size. If $S = 1$, then $L = 2S - 1 = 1$, so all items have the same size, making the problem uninteresting. Likewise, if there is no bound on the maximum bin size, the problem is uninteresting, because the competitive ratio is unbounded: Once enough bins for an optimal packing have arrived, an adversary can send bins of arbitrarily large size, which the algorithm would be forced to use.

In many applications of bin packing, there are only a small number of different item sizes. A number of papers have considered the problem of packing a sequence of items of two different sizes in bins of size 1 in an online matter. In particular, there is a lower bound of 4/3 on the competitive ratio [6,4] and a matching upper bound [4]. When both item sizes are bounded above by $1/k$, the competitive ratio can be improved to $\frac{(k+1)^2}{k^2+k+1}$ [3].

In this paper, we consider the Restricted Grid Scheduling problem, a version of the Grid Scheduling problem where the input contains exactly s items of size S and ℓ items of size $L = 2S - 1$ and the maximum bin size is $M = 2L - 1$, which includes the scenario used in the $\frac{5}{4}$ lower bound. Two natural questions arise for this problem:

1. Is there an algorithm matching the $\frac{5}{4}$ lower bound on the competitive ratio or can this lower bound be improved?
2. What is the competitive ratio for the problem when there are initially s items of size S and ℓ items of size L, when the ratio of s to ℓ is arbitrary, rather than fixed to 2?

We obtain matching upper and lower bounds on the competitive ratio of the Restricted Grid Scheduling problem.

Theorem 1. *For unbounded S, the competitive ratio of the Restricted Grid Scheduling problem is*

$$\begin{cases} 1 + \frac{1}{2+s/\ell} & \text{if } \ell \le s/2, \\ 1 + \frac{1}{4} & \text{if } s/2 < \ell \le 5s/6, \text{ and} \\ 1 + \frac{2}{3+6\ell/s} & \text{if } 5s/6 < \ell. \end{cases}$$

We begin with some preliminaries, including a formal definition of the Grid Scheduling problem and some properties of optimal packings. In Section 3, we prove the lower bound on the competitive ratio of the Restricted Grid Scheduling problem. Then, in Section 4, we present properties of optimal packings for the

Restricted Grid Scheduling problem. Some of these provide motivation for our design, while others are important for the analysis. We show why a few simple algorithms are not optimal in Section 5. Our algorithm, 2-Phase-Packer, appears in Section 6, together with part of the analysis. We conclude with some open questions. The omitted proofs and the rest of the analysis are in the full paper.

2 Preliminaries

The competitive ratio [7,5] of an on-line algorithm is the worst-case ratio of the on-line performance to the optimal off-line performance, up to an additive constant. More precisely, for a set I of items (or, equivalently, a multi-set of item sizes), a sequence σ of bins, and an algorithm $\mathbb{A}$ for the Grid Scheduling problem, let $\mathbb{A}(I, \sigma)$ denote the total size of all the bins used by $\mathbb{A}$ when packing I in the sequence σ of bins. Then, the *competitive ratio* $\mathrm{CR}_{\mathbb{A}}$ of $\mathbb{A}$ is

$$\mathrm{CR}_{\mathbb{A}} = \inf \{c \mid \exists b, \forall I, \forall \sigma, \mathbb{A}(I, \sigma) \leq c \cdot \mathrm{OPT}(I, \sigma) + b\},$$

where OPT denotes an optimal off-line algorithm. For specific choices of families of sets I_n and sequences σ_n, the performance ratios, $\frac{\mathbb{A}(I_n, \sigma_n)}{\mathrm{OPT}(I_n, \sigma_n)}$, can be used to prove a lower bound on the competitive ratio of $\mathbb{A}$.

Given a set of items and a sequence of bins, each with a size in $\{1, \ldots, M\}$, the goal of the Grid Scheduling problem is to pack the items in the bins so that the sum of the sizes of the items packed in each bin is at most the size of the bin and the sum of the sizes of bins used is minimized. The bins in the sequence arrive one at a time and each must be packed before the next bin arrives, without knowledge of the sizes of any future bins. If a bin is at least as large as the smallest unpacked item, it must be packed with at least one item. There is no charge for a bin that is smaller than this. It is assumed that enough sufficiently large bins arrive so that any algorithm eventually packs all items. For example, it suffices that every sequence ends with enough bins of size M to pack every item one per bin.

Given a set of items and a sequence of bins, a *partial packing* is an assignment of some of the items to bins such that the sum of the sizes of the items assigned to each bin is at most the size of the bin. A *packing* is a partial packing that assigns every item to a bin. If p is a packing of a set of items I into a sequence of bins σ and p' is a packing of a disjoint set of items I' into a sequence of bins σ', then we use pp' to denote the packing of $I \cup I'$ into the sequence of bins $\sigma\sigma'$, where each item in I is assigned to the same bin as in p and each item in I' is assigned to the same bin as in p'.

If every item has size at least S, there is no loss of generality in assuming that every bin has size at least S. This is because no packing can use a bin of size less than S.

A packing is *valid* if every bin that can be used is used. In other words, in a valid packing, if a bin is empty, then all remaining items are larger than the bin.

A bin in a packing p for σ is *wasteful* if its empty space is at least the size of the smallest remaining item. A packing is *thrifty* if it contains no wasteful bins. Since a packing is valid if and only if it has no wasteful empty bin, every thrifty packing is valid.

A packing is *optimal* if it is valid and the sum of the sizes of the bins it uses is at least as small as any other valid packing.

Lemma 1. *For any sequence of bins and any optimal packing pp' of these bins, there is an optimal packing pq that uses the same set of bins, in which q has no wasteful bins.*

Taking p to be empty in Lemma 1 yields the following result.

Corollary 1. *For any optimal packing of a sequence of bins, there is an optimal thrifty packing of that sequence using the same set of bins.*

3 Lower Bounds

Theorem 2. *For unbounded S, any algorithm for the Restricted Grid Scheduling problem has competitive ratio at least*

$$\begin{cases} 1 + \frac{1}{2+s/\ell} & \text{if } \ell \le s/2, \\ 1 + \frac{1}{4} & \text{if } s/2 < \ell \le 5s/6, \text{ and} \\ 1 + \frac{2}{3+6\ell/s} & \text{if } 5s/6 < \ell. \end{cases}$$

Proof. Consider an instance of the Grid Scheduling problem in which there are s items of size S, ℓ items of size $L = 2S - 1$, and maximum bin size $M = 2L - 1$. We start with the case when $\ell \le s/2$ and then handle the case when $\ell > s/2$. In both cases, we consider two subcases, depending on how the algorithm packs the first batch of bins.

Case I: $\ell \le s/2$.
The adversary begins by giving ℓ bins of size $2S$. In each of these bins, the algorithm must pack either two items of size S or one item of size L. Let $0 \le k \le \ell$ be the number of these bins in which the algorithm packs two items of size S. Then the algorithm has $s - 2k$ items of size S and k items of size L left to pack.

Case I.1: $k \le \ell/2$.
Next, the adversary gives $s - 2\ell$ bins of size S, followed by 2ℓ bins of size L. The algorithm must pack one item of size S in each bin of size S and must use one bin of size L for each of the remaining $s - 2k - (s - 2\ell) = 2(\ell - k)$ items of size S and k items of size L. The total cost incurred by the algorithm is $\ell \cdot 2S + (s - 2\ell) \cdot S + (2\ell - k) \cdot L = s \cdot S + 2\ell \cdot L - k \cdot L \ge s \cdot S + 3\ell \cdot L/2$.

For this sequence, OPT packs two items of size S in each of the ℓ bins of size $2S$, one item of size S in each of the $s - 2\ell$ bins of size S, and one item of

size L in each of the next ℓ bins of size L, for total cost $s \cdot S + \ell \cdot L$. Thus, the performance ratio of the algorithm is at least

$$\frac{s \cdot S + 3\ell \cdot L/2}{s \cdot S + \ell \cdot L} = 1 + \frac{\ell \cdot L}{2\ell \cdot L + 2s \cdot S} = 1 + \frac{\ell}{2\ell + s + s/L}$$
$$= 1 + \frac{1}{2 + \frac{s}{\ell} + \frac{s/\ell}{2S-1}} \to 1 + \frac{1}{2 + s/\ell} \text{ as } S \to \infty.$$

Case I.2: $k > \ell/2$.
Next, the adversary gives s bins of size S, followed by ℓ bins of size M. The algorithm packs one item of size S in the first $s - 2k \geq 2\ell - 2k \geq 0$ of these bins, using up all its items of size S. It discards the remaining $2k$ bins of size S, because it has no remaining elements that are small enough to fit in them. Then the algorithm packs its remaining k items of size L into k bins of size $M = 4S - 3$. The total cost incurred by the algorithm is

$$\ell \cdot 2S + (s - 2k) \cdot S + k \cdot (4S - 3)$$
$$= (2\ell + s) \cdot S + k \cdot (2S - 3)$$
$$> (2\ell + s) \cdot S + \ell \cdot (S - 3/2).$$

For this sequence, OPT packs one item of size L in each of the ℓ bins of size $2S$ and one item of size S in each of the next s bins. The total cost used by OPT is $\ell \cdot 2S + s \cdot S$. Thus, the performance ratio of the algorithm is at least

$$\frac{(3\ell + s) \cdot S - 3\ell/2}{(2\ell + s) \cdot S} = 1 + \frac{1 - 3/(2S)}{2 + s/\ell} \to 1 + \frac{1}{2 + s/\ell} \text{ as } S \to \infty.$$

Case II: $\ell > s/2$. The adversary begins by giving $\lfloor s/2 \rfloor$ bins of size $2S$. In each of these bins, the algorithm must pack either two items of size S or one item of size L. Let $0 \leq k \leq \lfloor s/2 \rfloor$ be the number of these bins in which the algorithm packs two items of size S. Then the algorithm has $s - 2k$ items of size S and $\ell - \lfloor s/2 \rfloor + k$ items of size L left to pack.

Case II.1: $k \leq \lfloor s/2 \rfloor - s/8 - \ell/4 + 1$ or $k \leq \lfloor s/2 \rfloor - s/3 + 1$.
Next, the adversary next gives $\lceil s/2 \rceil - \lfloor s/2 \rfloor$ bins of size S (i.e. one bin of size S if s is odd and no bins of size S if s is even), $\lfloor s/2 \rfloor - k + \ell - 1$ bins of size L, and 1 bin of size M. Since $(s-2k)+(\ell-\lfloor s/2 \rfloor+k) = (\lceil s/2 \rceil - \lfloor s/2 \rfloor)+(\lfloor s/2 \rfloor - k+\ell-1)+1$, the algorithm packs one of its remaining items in each of these bins, so the total cost it incurs is $\lfloor s/2 \rfloor \cdot 2S + (\lceil s/2 \rceil - \lfloor s/2 \rfloor) \cdot S + (\lfloor s/2 \rfloor - k + \ell - 1) \cdot L + M = s \cdot S + \ell \cdot L + (\lfloor s/2 \rfloor - k + 1) \cdot L - 1$.

If $s \geq 4$, then $\lfloor s/2 \rfloor - k \geq \min\{s/8 + \ell/4, s/3\} - 1 > s/4 - 1 \geq 0$, so there are at least ℓ bins of size L. Because of the additive constant in the definition of the competitive ratio, the case $s \leq 3$ can be ignored. For this sequence, OPT packs two items of size S in each of the $\lfloor s/2 \rfloor$ bins of size $2S$, one item of size S in the bin of size S, if s is odd, and one item of size L in each of the next ℓ bins of size L. Its total cost is $s \cdot S + \ell \cdot L$.

If $k \le \lfloor s/2 \rfloor - s/8 - \ell/4 + 1$, then $(\lfloor s/2 \rfloor - k + 1) \cdot L \ge (s/8 + \ell/4) \cdot L = S \cdot s/4 - s/8 + L \cdot \ell/4$, so the performance ratio of the algorithm is at least

$$\frac{(s \cdot S + \ell \cdot L) \cdot 5/4 - s/8 - 1}{s \cdot S + \ell \cdot L} = \frac{5}{4} - \frac{s/8 + 1}{s \cdot S + \ell \cdot (2S - 1)} \to \frac{5}{4} \text{ as } S \to \infty.$$

Similarly, if $k \le \lfloor s/2 \rfloor - s/3 + 1$, then $(\lfloor s/2 \rfloor - k + 1) \cdot L \ge (s/3) \cdot L$, so the performance ratio of the algorithm is at least

$$1 + \frac{(s/3) \cdot L}{s \cdot S + \ell \cdot L} = 1 + \frac{2 - 1/(s \cdot S)}{3 + 6\ell/s - 3\ell/(s \cdot S)} \to 1 + \frac{2}{3 + 6\ell/s} \text{ as } S \to \infty.$$

Case II.2: $k > \lfloor s/2 \rfloor - s/8 - \ell/4 + 1$ and $k > \lfloor s/2 \rfloor - s/3 + 1 > s/6$.
Next, the adversary gives $\max\{s - 2k, s - \ell + \lfloor s/2 \rfloor\}$ bins of size S, followed by $\min\{2k, \ell - \lfloor s/2 \rfloor\}$ bins of size $L + S$, $\max\{\ell - \lfloor s/2 \rfloor - 2k, 0\}$ bins of size L, and finally k bins of size $M = 4S - 3$. The algorithm packs its $s - 2k$ items of size S into bins of size S. Since $\min\{2k, \ell - \lfloor s/2 \rfloor\} + \max\{\ell - \lfloor s/2 \rfloor - 2k, 0\} + k = \ell - \lfloor s/2 \rfloor + k$, the algorithm packs one item of size L in each bin of size $L + S$, L, and M. The total cost incurred by the algorithm is

$$\begin{aligned}
&\lfloor s/2 \rfloor \cdot 2S + (s - 2k) \cdot S + \min\{2k, \ell - \lfloor s/2 \rfloor\} \cdot (L + S) \\
&\quad + \max\{\ell - \lfloor s/2 \rfloor - 2k, 0\} \cdot L + k \cdot M \\
&= (2\lfloor s/2 \rfloor + s) \cdot S + (\ell - \lfloor s/2 \rfloor) \cdot L + k \cdot (2S - 3) + \min\{2k, \ell - \lfloor s/2 \rfloor\} \cdot S.
\end{aligned}$$

For this sequence, OPT fills every bin it uses except for the $\lfloor s/2 \rfloor$ bins of size $2S$, in which it puts items of size $L = 2S - 1$. Therefore, the total cost used by OPT is $s \cdot S + \ell \cdot L + \lfloor s/2 \rfloor$.

If $2k \ge \ell - \lfloor s/2 \rfloor$, the performance ratio of the algorithm is at least

$$\begin{aligned}
&\frac{(2\lfloor s/2 \rfloor + s) \cdot S + (\ell - \lfloor s/2 \rfloor) \cdot L + k \cdot (2S - 3) + (\ell - \lfloor s/2 \rfloor) \cdot S}{s \cdot S + \ell \cdot L + \lfloor s/2 \rfloor} \\
&> \frac{(\ell + s + \lfloor s/2 \rfloor) \cdot S + (\ell - \lfloor s/2 \rfloor) \cdot L + (\lfloor s/2 \rfloor - s/8 - \ell/4 + 1) \cdot (2S - 3)}{s \cdot S + \ell \cdot L + \lfloor s/2 \rfloor} \\
&= \frac{5}{4} + \frac{(\lfloor s/2 \rfloor - s/2 + 2) \cdot S + \ell + 3s/8 - 13\lfloor s/2 \rfloor/4 - 3}{(s + 2\ell) \cdot S - \ell + \lfloor s/2 \rfloor} \\
&> \frac{5}{4} + \frac{\ell + 3s/8 - 13\lfloor s/2 \rfloor/4 - 3}{(s + 2\ell) \cdot S - \ell + \lfloor s/2 \rfloor} \to \frac{5}{4} \text{ as } S \to \infty.
\end{aligned}$$

If $2k \le \ell - \lfloor s/2 \rfloor$, the performance ratio of the algorithm is at least

$$\begin{aligned}
&\frac{(2\lfloor s/2 \rfloor + s) \cdot S + (\ell - \lfloor s/2 \rfloor) \cdot L + k \cdot (4S - 3)}{s \cdot S + \ell \cdot L + \lfloor s/2 \rfloor} \\
&= 1 + \frac{k \cdot (4S - 3)}{(s + 2\ell) \cdot S - \ell + \lfloor s/2 \rfloor} \\
&> 1 + \frac{(s/6) \cdot (4S - 3)}{(s + 2\ell) \cdot S + \lfloor s/2 \rfloor - \ell} \\
&= 1 + \frac{2 - 3/2S}{3 + 6\ell/s + 3(\lfloor s/2 \rfloor - \ell)/sS} \to 1 + \frac{2}{3 + 6\ell/s} \text{ as } S \to \infty.
\end{aligned}$$

Note that $\frac{5}{4} \leq 1 + \frac{2}{3+6\ell/s}$ if and only if $\ell \leq 5s/6$. Thus, $\frac{5}{4}$ is a lower bound on the competitive ratio when $s/2 < \ell \leq 5s/6$ and $1 + \frac{2}{3+6\ell/s}$ is a lower bound on the competitive ratio when $\ell > 5s/6$.

4 Properties of Optimal Packings for Restricted Grid Scheduling

We say that a bin used in a packing is *bad* if it contains an item of size S, it has empty space at least $L - S$, and it occurs while items of size L remain. Note that a bin containing an item of size L and an item of size S has empty space at most $M - L - S = L - S - 1$, so it is not bad.

Lemma 2. *For any set of items and any sequence of bins, there exists an optimal thrifty packing that contains no bad bin.*

Proof. Fix a set of items and a sequence of bins and suppose every optimal thrifty packing contains a bad bin. Consider an optimal thrifty packing p which has the longest prefix without bad bins. Let b' be the last bin to which p assigns an item of size L. Then the first bad bin b occurs before b'. Since p is thrifty, the empty space in b is less than L.

Note that p has no empty bins (of size at least S) between b and b'. Otherwise, let b'' be the first empty bin following b. Since p is valid, $size(b'') < L$ and, when bin b'' arrives, only items of size L remain. Then each nonempty bin after b'', including b, contains exactly one item, which is of size L. Consider the packing p' obtained from p by moving one item of size S in b to bin b'' and moving the item of size L in b' to bin b. Then b' is empty in the packing p'. Since p is valid, p and p' are the same prior to bin b, there are no empty bins between b and b'' in p', and no item which occurred before b'' in p has been moved after b'' in p', it follows that p' is valid. But the cost of p' is equal to the cost of p − the size of b' + size of b'', which is lower than the cost of p, since $size(b'') < L \leq size(b')$. By Corollary 1, there is an optimal thrifty packing with the same cost as p', which contradicts the optimality of p.

Let p' be the packing obtained from p by switching an item of size S in b with the item of size L in b'. Note that p' is optimal, since p is. Since b contains an item of size L, it is not bad in p'. Furthermore, bin b is not wasteful, since its empty space, which was less than L in p, is less than $L - (S - L) = S$ in p'. None of the bins in p are wasteful, since p is thrifty, so none of the bins in p' prior to b are wasteful. By Lemma 1, there is an optimal packing p'' that is identical to p' up to and including bin b and which has no wasteful bins after b. This implies that p'' is thrifty. But p'' has a longer prefix without bad bins than p does. Thus, it contradicts the choice of p.

This motivates the following definition.

Definition 1. *A partial packing is* reasonable *if every bin b contains*

- *one item of size S, if $size(b) \in [S, L-1]$,*

- *one item of size L, if $size(b) = L$,*
- *two items of size S or one item of size L, if $size(b) \in [L+1, L+S-1]$,*
- *one item of size S and one item of size L, if $size(b) = L+S$, and*
- *three items of size S or one item of size S and one item of size L, if $size(b) \in [L+S+1, 2L-1]$.*

Note that a reasonable partial packing contains no wasteful, bad, or empty bins.

For any packing, consider the first bin after which there are no items of size L remaining or at most 2 items of size S remaining. This bin is called the *key bin* of the packing. The partial packing that assigns the same items into each bin up to and including its key bin and assigns no items to any subsequent bin is called the *front* of the packing. We say that a packing is *reasonable* if it is thrifty and its front is reasonable.

Corollary 2. *For any set of items and any sequence of bins, there exists an optimal packing that is reasonable.*

From now on, we will restrict attention to reasonable packings.

Given a set of items and a sequence of bins, two reasonable partial packings can differ as to whether they use one item of size L or two items of size S in certain bins. Therefore, the numbers of items of size S and items of size L they do not assign may differ. However, if both have at least one item of size S and at least one item of size L unassigned, the set of bins they have used is the same and there is a simple invariant relating the numbers of unassigned items of size S and unassigned items of size L they have.

Lemma 3. *Consider two reasonable partial packings of a set of items into a sequence of bins. Suppose that before bin b, each packing has at least one unassigned item of size S and at least one unassigned item of size L or at least three unassigned items of size S. Then immediately after bin b has been packed, the number of items of size S available plus twice the number of items of size L available is the same for both packings.*

Once a packing runs out of items of size S, it may be impossible for it to completely use some bins, so this relationship does not necessarily hold.

For any sequence of bins σ and any nonnegative integers s and ℓ, let $OPT(\sigma, s, \ell)$ denote the cost of an optimal packing of s items of size S and ℓ items of size $L = 2S - 1$ using σ. This must be at least the sum of the sizes of all the items.

Proposition 1. $OPT(\sigma, s, \ell) \geq sS + \ell L$.

Given any optimal packing for a set of items, a packing for a subset of these items can be obtained by removing the additional items from bins, starting from the end.

Proposition 2. $OPT(\sigma, s', \ell') \leq OPT(\sigma, s, \ell)$ *for all* $s' \leq s$ *and* $\ell' \leq \ell$.

For any sequence of bins σ and any nonnegative integers s and ℓ, let $R(\sigma, s, \ell)$ denote the maximum cost of any reasonable packing of s items of size S and ℓ items of size $L = 2S - 1$ using σ.

When all items have the same size, all thrifty algorithms, including OPT, behave exactly the same.

Proposition 3. *For all integers* $s, \ell > 0$, $R(\sigma, s, 0) = OPT(\sigma, s, 0)$ *and* $R(\sigma, 0, \ell) = OPT(\sigma, 0, \ell)$.

Thus, if R and OPT both run out of items of size L at the same time or they both run out of items of size S at the same time, then they will use the same set of bins and, hence, have the same cost. The following four lemmas describe the relationship between the costs incurred by R and OPT when one of them has run out of one size of items.

Lemma 4. *For any sequence of bins* σ *and any integers* $s, \ell \geq 0$, $R(\sigma, s, \ell) \leq OPT(\sigma, s + 2\ell, 0) + \ell(2S - 3)$.

Lemma 5. *For any sequence of bins* σ *and any integers* $s, \ell \geq 0$, *if* $2k = s + 2\ell$, *then* $R(\sigma, s, \ell) \leq OPT(\sigma, 0, k) + (s + \ell - k - 1)L + M$.

Lemma 6. *For any sequence of bins* σ *and any integers* $s, \ell \geq 0$, $R(\sigma, s + 2\ell, 0) \leq OPT(\sigma, s, \ell) + (\ell - 1)L + M$.

Lemma 7. *For any sequence of bins* σ *and any integers* $s, \ell \geq 0$, *if* $2k = s + 2\ell$, *then* $R(\sigma, 0, k) \leq OPT(\sigma, \min\{s, \ell\}, \ell) + (k - \ell)M$.

5 Simple Non-optimal Algorithms

It is helpful to understand why simple reasonable algorithms are not optimal for the Restricted Grid Scheduling problem. The following examples show why a number of natural candidates don't work well enough.

Example 1. Consider the reasonable algorithm that always uses items of size S, when there is a choice. Let $s = 2\ell$ and let σ consist of ℓ bins of size $2S$, followed by s bins of size S, and then ℓ bins of size M. For this instance, the algorithm has a performance ratio of $\frac{\ell \cdot 2S + \ell \cdot M}{\ell \cdot 2S + s \cdot S} = \frac{3}{2} - \frac{3}{4S}$, which is greater than $5/4$ for large S.

Example 2. Consider the reasonable algorithm that always uses items of size L, when there is a choice. Let $s = 2\ell$ and let σ consist of ℓ bins of size $2S$, followed by $s - 1$ bins of size L and then 1 bin of size M. For this instance, the algorithm has a performance ratio of $\frac{\ell \cdot 2S + (s-1) \cdot L + M}{\ell \cdot 2S + \ell \cdot L} = \frac{3}{2} + \frac{1}{\ell} - \frac{1/\ell + 1/2}{2S - 1}$, which is also greater than $5/4$ for large S.

For the two instances considered above, the reasonable algorithm which alternates between using two items of size S and one item of size L, when it has a choice, would do well, achieving a performance ratio of $5/4$. However, it doesn't do as well on other instances.

Example 3. Let $2s = 3\ell$ and let σ consist of ℓ bins of size $2S$, followed by s bins of size S and $\ell/2$ bins of size M. For this instance, the algorithm which alternates between using two items of size S and one item of size L, when it has a choice, has a performance ratio of $\frac{\ell \cdot 2S + (s-\ell)\cdot S + (\ell/2)\cdot M}{\ell \cdot 2S + s \cdot S} = 1 + \frac{\frac{\ell}{2}\cdot(M-2S)}{(2\ell+s)\cdot S} = 1 + \frac{2}{7} - \frac{3}{7S}$, which is larger than $5/4$ for S sufficiently large.

As the above examples partially illustrate, once either the online algorithm or OPT has run out of one type of item (items of size S or items of size L), the adversary can give bins which make the online algorithm waste a lot of space. The algorithm we present in the next section aims to postpone this situation long enough to get a good ratio. The following example indicates the need for a second phase in order to obtain the optimal competitive ratios, which are less than $5/4$ in some cases.

Example 4. Consider the reasonable algorithm which uses one item of size L twice as often as two items of size S, when it has a choice. Let $3s = 2\ell$ and let σ consist of ℓ bins of size $2S$, followed by s bins of size S and $\ell/3$ bins of size M. For this instance, the algorithm packs $\ell/3 = s/2$ bins of size $2S$ with two items of size S, so it has a performance ratio of $\frac{\ell \cdot 2S + (\ell/3)\cdot M}{\ell \cdot 2S + s \cdot S} = 1 + \frac{\frac{s}{2}\cdot M - sS}{(2\ell+s)\cdot S} = 1 + \frac{S-3/2}{4\cdot S} = \frac{5}{4} - \frac{3}{8S}$, which exceeds the lower bound of $1 + \frac{2}{3+6\ell/s} = \frac{7}{6}$ for S sufficiently large.

6 A Matching Upper Bound

In this section, we present a reasonable algorithm, 2-Phase-Packer, for the Restricted Grid Scheduling problem. It is asymptotically optimal: the competitive ratio matches the lower bound in Section 3 for all three ranges of the ratio s/ℓ of the initial numbers of items of size S and items of size L. Not surprisingly, 2-Phase-Packer has two phases.

In the first phase, it attempts to balance the number of items of size S and the number of items of size L it uses, aiming for the ratio indicated by the lower bound. When it receives bins where it has a choice of using one item of size L or two items of size S in part or all of that bin (i.e., bins with sizes in the ranges $[2S, 3S-2]$ and $[3S, 4S-3]$), it uses one item of size L in a certain fraction of them (and increments the variable L-bins) and uses two items of size S in the remaining fraction (and increments S-bins). The fraction varies depending on the original ratio s/ℓ: at least 2, between 2 and 6/5, and less than 6/5. It is enforced by a macro called UseLs, which indicates that an item of size L should be used if and only if L-bins $\leq r$S-bins, where r is the target ratio of L-bins to S-bins. For example, when equal numbers of each should be used, we have UseLs = (L-bins $\leq$ S-bins). Both L-bins and S-bins start at zero, so, in this case, 2-Phase-Packer starts by choosing to use one item of size L and then alternates. Note that S-bins and L-bins do not change after Phase 1 is completed.

In the middle range for the ratio s/ℓ, there are two different fractions used. The first fraction is used until S-bins reaches a specific value. Afterwards, its

choices alternate. To do so, for the rest of Phase 1, it records the number of times it chooses to pack two items of size S in a bin and the number of times it chooses to pack one item of size L in late-S-bins and late-L-bins, respectively. The variables late-S-bins and late-L-bins are also zero initially.

2-Phase-Packer uses countS and countL throughout the algorithm to keep track of the total numbers of items of size S and items of size L it has used, whether or not it had a choice. (Specifically, countS is incremented every time an item of size S is used and countL is incremented every time an item of size L is used.) It continues with Phase 1 until it has used a certain number of items of size S or items of size L (depending on the relationship between s and ℓ). For each of the three ranges of s/ℓ, we define a different condition for ending Phase 1. In Phase 2, only items of size S or only items of size L are used where a reasonable algorithm has a choice, depending on whether one would expect an excess of items of size S or items of size L, given the ratio s/ℓ.

The definitions of the end of Phase 1 for the various ranges of s/ℓ imply these inequalities.

Lemma 8. *If $s/2 < \ell \leq 5s/6$, then S-bins $\leq 3s/8 - \ell/4 + 1$.*
If $s < 6\ell/5$, then S-bins $\leq s/6 + 1$.

The following simple invariants can be proved inductively.

Lemma 9. *L-bins $\leq$ countL and 2S-bins $\leq$ countS.*
If $\ell \leq s/2$, then S-bins $\leq$ L-bins $\leq$ S-bins $+ 1$.
If $s/2 < \ell \leq 5s/6$, then late-S-bins $\leq$ late-L-bins $\leq$ late-S-bins $+ 1$ and $\lfloor c(\text{S-bins} - 1) \rfloor + 1 \leq \text{L-bins} \leq \lfloor c\ \text{S-bins} \rfloor + 1$, where $1 < c = \frac{10\ell - 3s}{s + 2\ell} \leq 2$.
If $5s/6 < \ell$, then 2S-bins $\leq$ L-bins $\leq$ 2S-bins $+ 1$.

We analyse the performance of 2-Phase-Packer as compared to the cost of an optimal reasonable packing on an arbitrary sequence, σ, using the three different ranges for the relationship between s and ℓ in the lower bound.

Both 2-Phase-Packer and OPT use exactly the same bins until one of them runs out of either items of size L or items of size S. Thus, there are four cases to consider.

1. OPT runs out of items of size L at or before the point where 2-Phase-Packer runs out of anything.
2. OPT runs out of items of size S at or before the point where 2-Phase-Packer runs out of anything.
3. 2-Phase-Packer runs out of items of size L before OPT runs out of anything.
4. 2-Phase-Packer runs out of items of size S before OPT runs out of anything.

We only present the analysis for Case 1. The remaining cases are similar.

Consider an arbitrary sequence of bins σ in which 2-Phase-Packer packs s items of size S and ℓ items of size L. Suppose that, in this instance, OPT packs its last item of size L in bin b', but prior to bin b', both OPT and 2-Phase-Packer have unpacked items of both sizes. Let countL$'$ and countS$'$ denote the number of items of size L and items of size S, respectively, that 2-Phase-Packer has used

$$\textbf{macro}\ \text{Phase1done} = \begin{cases} \text{countL} \geq \lfloor \ell/2 \rfloor & \text{if } \ell \leq s/2 \\ \text{countS} \geq \lfloor 3s/4 - \ell/2 \rfloor & \text{if } s/2 < \ell \leq 5s/6 \\ \text{countS} \geq \lfloor s/3 \rfloor & \text{if } 5s/6 < \ell \end{cases}$$

$$\textbf{macro}\ \text{UseLs} = \begin{cases} \text{L-bins} \leq \text{S-bins} & \text{if } \ell \leq s/2 \\ \textbf{if } (\text{S-bins} < \lfloor (s+2\ell)/16 \rfloor) & \text{if } s/2 < \ell \leq 5s/6 \\ \textbf{then } \text{L-bins} \leq (10\ell - 3s)\text{S-bins}/(s+2\ell) & \\ \textbf{else } \text{late-L-bins} \leq \text{late-S-bins} & \\ \text{L-bins} \leq 2\text{S-bins} & \text{if } 5s/6 < \ell \end{cases}$$

countS % counts the number of items of size S used; initially 0
countL % counts the number of items of size L used; initially 0
S-bins $\leftarrow$ L-bins $\leftarrow$ late-S-bins $\leftarrow$ late-L-bins $\leftarrow$ 0

for each arriving bin b
  if only one type of item still remains **then** use as many items as fit in b
  else if $size(b) \in [S, 2S-2]$ **then** use 1 item of size S
  else if $size(b) = 2S-1$ **then** use 1 item of size L
  else if $size(b) = 3S-1$ **then** use 1 item of size L and 1 item of size S
  else if only one item of size S still remains **then**
    use 1 item of size L
    if remaining space in b is at least S **then** use 1 item of size S
  else if only two items of size S still remain and $size(b) \in [3S, 4S-3]$ **then**
    use 1 item of size L and 1 item of size S
  else if (not Phase1done) **then**
    % Use range determined ratio
    if $6\ell/5 \leq s < 2\ell$ and S-bins $\geq \lfloor (s+2\ell)/16 \rfloor$ **then**
      if UseLs **then** late-L-bins ++
        else late-S-bins ++
    if $size(b) \in [2S, 3S-2]$ **then**
      if UseLs **then** use 1 item of size L; L-bins ++;
      else use 2 items of size S; S-bins ++;
    if $size(b) \in [3S, 4S-3]$ **then**
      if UseLs **then** use 1 item of size L and 1 item of size S; L-bins ++;
      else use 3 items of size S; S-bins ++;
  else % In Phase 2
    if $\ell \leq s/2$ **then** use as many items of size S as fit in bin b
    else use 1 item of size L
      if remaining space in b is at least S **then** use 1 item of size S
end for

Fig. 1. The algorithm 2-Phase-Packer

up to and including bin b'. Let S-bins$'$ denote the number of bins at or before b' where 2-Phase-Packer had a choice and used two items of size S instead of one of size L and let L-bins$'$ denote the number where it used one item of size L and could have used two of size S instead. Let σ' denote the portion of σ after b'.

Both algorithms use all bins up to and including bin b', since all bins have size at least S. If X is the total cost of these bins, then $X \geq (\text{countS}')S + (\text{countL}')L$ and 2-Phase-Packer $(\sigma, s, \ell) = X +$ 2-Phase-Packer $(\sigma', s - \text{countS}', \ell - \text{countL}')$. In this case, Lemma 3 implies that, after bin b', OPT has $(s - \text{countS}') + 2(\ell - \text{countL}')$ items of size S remaining, so $OPT(\sigma, s, \ell) = X + OPT(\sigma', (s - \text{countS}') + 2(\ell - \text{countL}'), 0)$.

Since 2-Phase-Packer is reasonable, Lemma 4 implies that 2-Phase-Packer$(\sigma', s - \text{countS}', \ell - \text{countL}') \leq R(\sigma', s - \text{countS}', \ell - \text{countL}') \leq \text{OPT}(\sigma', (s - \text{countS}') + 2(\ell - \text{countL}'), 0) + (\ell - \text{countL}')(2S - 3)$, so 2-Phase-Packer$(\sigma, s, \ell) \leq \text{OPT}(\sigma, s, \ell) + (\ell - \text{countL}')(2S - 3)$. By Proposition 1, $\text{OPT}(\sigma, s, \ell) \geq sS + \ell L \geq sL/2 + \ell L = (s + 2\ell)L/2$. When computing the competitive ratio, we will choose the additive constant to be at least L, so we subtract L from 2-Phase-Packer's cost in the ratio. Thus,

$$\begin{aligned}\frac{\text{2-Phase-Packer}(\sigma, s, \ell) - L}{\text{OPT}(\sigma, s, \ell)} &\leq \frac{\text{OPT}(\sigma, s, \ell) + (\ell - \text{countL}')(2S - 3) - L}{\text{OPT}(\sigma, s, \ell)} \\ &\leq 1 + \frac{(\ell - \text{countL}' - 1)(L - 2)}{(s + 2\ell)L/2} \\ &\leq 1 + \frac{2(\ell - \text{countL}' - 1)}{s + 2\ell}.\end{aligned}$$

It remains to bound $\ell - \text{countL}' - 1$ in each of the three ranges for the ratio s/ℓ. We use the fact that, immediately after bin b', OPT has run out of items of size L, so at least ℓ bins of size at least L are in σ up to and including bin b'.

First, consider the case when $\ell \leq s/2$. If $\text{countL}' < \lfloor \ell/2 \rfloor$, then Phase 1 was not completed when bin b' arrived. Therefore, each time a bin of size at least L arrives up to and including bin b', 2-Phase-Packer either packs an item of size L in it and, hence, increments countL, or it increments S-bins. Hence, $\text{countL}' + \text{S-bins}' \geq \ell$. By Lemma 9, L-bins$' \geq$ S-bins$'$. Since countL$' \geq$ L-bins$'$, by Lemma 9, it follows that $\text{countL}' \geq \ell/2 \geq \lfloor \ell/2 \rfloor$. This is a contradiction. Thus, $\text{countL}' \geq \lfloor \ell/2 \rfloor$ and

$$\begin{aligned}\frac{\text{2-Phase-Packer}(\sigma, s, \ell) - L}{\text{OPT}(\sigma, s, \ell)} &\leq 1 + \frac{2(\ell - \text{countL}' - 1)}{s + 2\ell} \\ &\leq 1 + \frac{\ell}{s + 2\ell} = 1 + \frac{1}{2 + s/\ell}.\end{aligned}$$

So, suppose that $s/2 < \ell$. During Phase 1, each time a bin of size at least L arrives, 2-Phase-Packer either packs an item of size L in it and, hence, increments countL, or it increments S-bins. During Phase 2, 2-Phase-Packer packs an item of size L in each such bin until it runs out of items of size L. Therefore, $\text{countL}' + \text{S-bins}' \geq \ell$.

If $\ell \leq 5s/6$, then Lemma 8 implies that $\ell - \text{countL}' \leq \text{S-bins}' \leq 3s/8 - \ell/4 + 1$ and

$$\frac{\text{2-Phase-Packer}(\sigma, s, \ell) - L}{\text{OPT}(\sigma, s, \ell)} \leq 1 + \frac{2(\ell - \text{countL}' - 1)}{s + 2\ell} \leq 1 + \frac{2(3s/8 - \ell/4)}{s + 2\ell}$$
$$= 1 + \frac{s/4 + \ell/2 + s/2 - \ell}{s + 2\ell} < 1 + \frac{s/4 + \ell/2}{s + 2\ell} = \frac{5}{4}.$$

Similarly, if $s < 6\ell/5$, then $\ell - \text{countL}' \leq \text{S-bins}' \leq s/6 + 1$, by Lemma 8, and

$$\frac{\text{2-Phase-Packer}(\sigma, s, \ell) - L}{\text{OPT}(\sigma, s, \ell)} \leq 1 + \frac{2(\ell - \text{countL}' - 1)}{s + 2\ell}$$
$$\leq 1 + \frac{s/3}{s + 2\ell} < 1 + \frac{2}{3 + 6\ell/s}.$$

7 Conclusions and Open Problems

We have shown that varying the proportion of items of size S to items of size L does not lead to a larger competitive ratio if the maximum bin size is at most $4S - 3$. This may also be the case for an arbitrary maximum bin size, but there are complications. First, in bins of size $2kS$, where $k \geq 2S - 1$, it is possible to pack more than k items of size L. In addition, it may be an advantage to have mixed bins which contain more than one item of size S and more than one item of size L. So, when bins can be large, one needs to consider how to maintain a good ratio of items of size L to items of size S, as they are used. We conjecture that maintaining the ratios specified in 2-Phase-Packer is sufficient.

One could also consider the the Grid Scheduling problem for two bin sizes, S and $L \neq 2S - 1$. For example, when L is a multiple of S, then the algorithm of Example 2, which always uses as many items of size L as possible, is optimal and, hence, has competitive ratio 1. It would be interesting to see if changing the ratio of S to L could improve the lower bound. We conjecture that it does not.

For the general problem, where there could be more than two sizes of items, we would like to close the gap between our lower bound and the upper bound of $\frac{13}{7}$ in [1]. Using many item sizes, it is not hard to prove a lower bound of $\frac{3}{2}$ on the strict competitive ratio (the competitive ratio where the additive constant in the definition is zero). However, the strict competitive ratio is not very interesting for this problem, since the ratio can be made larger simply by increasing the size of M and giving a last bin of size M.

Finally, it would be interesting to consider the competitive ratio of randomized algorithms for the Grid Scheduling problem or Restricted Grid Scheduling problem against an oblivious adversary.

References

1. Boyar, J., Favrholdt, L.M.: Scheduling jobs on Grid processors. Algorithmica 57(4), 819–847 (2010)
2. Boyar, J., Favrholdt, L.M.: A new variable-sized bin packing problem. Journal of Scheduling 15, 273–287 (2012)
3. Epstein, L., Levin, A.: More online bin packing with two item sizes. Discrete Optimization 5(4), 705–713 (2008)
4. Gutin, G., Jensen, T.R., Yeo, A.: On-line bin packing with two item sizes. Algorithmic Operations Research 1(2) (2006)
5. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive snoopy caching. Algorithmica 3(1), 79–119 (1988)
6. Liang, F.M.: A lower bound for on-line bin packing. Inform. Process. Lett. 10, 76–79 (1980)
7. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Comm. of the ACM 28(2), 202–208 (1985)
8. Zhang, G.: A new version of on-line variable-sized bin packing. Discrete Applied Mathematics 72, 193–197 (1997)

Quake Heaps: A Simple Alternative to Fibonacci Heaps

Timothy M. Chan

Cheriton School of Computer Science, University of Waterloo,
Waterloo, Ontario N2L 3G1, Canada
tmchan@uwaterloo.ca

Abstract. This note describes a data structure that has the same theoretical performance as Fibonacci heaps, supporting decrease-key operations in $O(1)$ amortized time and delete-min operations in $O(\log n)$ amortized time. The data structure is simple to explain and analyze, and may be of pedagogical value.

1 Introduction

In their seminal paper [5], Fredman and Tarjan investigated the problem of maintaining a set S of n elements under the operations

- insert(x): insert an element x to S;
- delete-min(): remove the minimum element x from S, returning x;
- decrease-key(x, k): change the value of an element x to a smaller value k.

They presented the first data structure, called *Fibonacci heaps*, that can support insert() and decrease-key() in $O(1)$ amortized time, and delete-min() in $O(\log n)$ amortized time.

Since Fredman and Tarjan's paper, a number of alternatives have been proposed in the literature, including Driscoll et al.'s *relaxed heaps* and *run-relaxed heaps* [1], Peterson's *Vheaps* [9], which is based on AVL trees (and is an instance of Høyer's family of *ranked priority queues* [7]), Takaoka's *2-3 heaps* [11], Kaplan and Tarjan's *thin heaps* and *fat heaps* [8], Elmasry's *violation heaps* [2], and most recently, Haeupler, Sen, and Tarjan's *rank-pairing heaps* [6]. The classical *pairing heaps* [4,10,3] are another popular variant that performs well in practice, although they do not guarantee $O(1)$ decrease-key cost.

Among all the data structures that guarantee constant decrease-key and logarithmic delete-min cost, Fibonacci heaps have remained the most popular to teach. The decrease-key operation uses a simple "cascading cut" strategy, which requires an extra bit per node for marking. For the analysis, the potential function itself is not complicated, but one needs to first establish bounds on the maximum degree of the trees (Fibonacci numbers come into play here), and this requires understanding some subtle structural properties of the trees formed (a node may lose at most one child when it is not a root, but may lose multiple children when it is a root). In contrast, Vheaps are more straightforward to analyze,

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 27–32, 2013.

for those already acquainted with AVL trees, but the decrease-key() algorithm requires division into multiple cases, like the update algorithms of most balanced search trees. The recent rank-pairing heaps interestingly avoid cascading cuts by performing cascading rank changes, which may lead to a simpler implementation, but from the teaching perspective, the analysis appears even more complicated than for Fibonacci heaps (and there are also divisions into multiple cases).

In this note, we describe a data structure that is arguably the easiest to understand among all the existing methods. There is no case analysis involved, and no "cascading" during decrease-key(). We use a very simple, and rather standard, idea to ensure balance: be lazy during updates, and just rebuild when the structure gets "bad". Previous methods differ based on what local structural invariants are imposed. Our method is perhaps the most relaxed, completely forgoing local constraints, only requiring the tracking of some global counters. (Violation heaps [2] are of a similar vein but require multiple local counters; our method is simpler.)

In Section 2, we give a self-contained presentation of our method,[1] which should be helpful for classroom use; it only assumes basic knowledge of amortized analysis. We find a description based on tournament trees the most intuitive, although the data structure can also be expressed more traditionally in terms of heap-ordered trees or half-ordered trees, as noted in Section 3.

2 Quake Heaps

The Approach. We will work with a collection of *tournament trees*, where each element in S is stored in exactly one leaf, and the element of each internal node is defined as the minimum of the elements at the children. We require that at each node x, all paths from x to a leaf have the same length; this length is referred to as the *height* of x. We also require that each internal node has degree 2 or 1. See Figure 1.

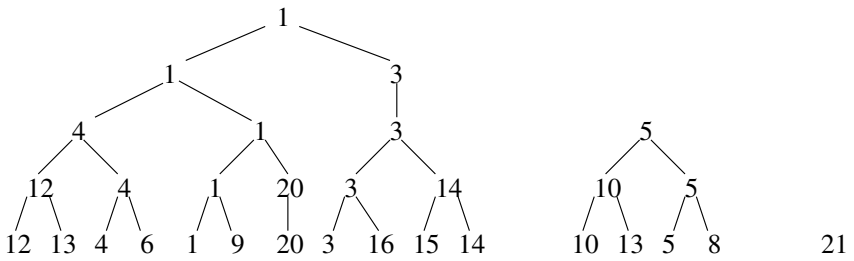


Fig. 1. An example

Two basic operations are easy to do in constant time under these requirements: First, given two trees of the same height, we can *link* them into one, simply

[1] Tradition demands a name to be given. The one in the title will hopefully make some sense after reading Section 2.

by creating a new root pointing to the two roots, storing the smaller element among the two roots. Secondly, given a node x whose element is different from x's parent's, we can *cut* out the subtree rooted at x. Note that x's former parent's degree is reduced to 1, but our setup explicitly allows for degree-1 nodes.

Inserting an element can be trivially done by creating a new tree of size 1. The number of trees in the collection increases by 1, but can be reduced by linking at a convenient time later.

For a delete-min operation, we can just remove the path of nodes that store the minimum element. The number of trees in the collection grows, and this is the time to do repeated linking operations to reduce the number of trees. Namely, whenever there are two trees of the same height, we link them.

For a decrease-key operation on an element, let x be the highest node that stores the element. It would be too costly to update the elements at all the ancestors of x. Instead we can perform a cut at x. Then we can decrease the value of x at will in the separate new tree.

We need to address one key issue: after many decrease-key operations, the trees may become too off-balanced. Let n_i denote the number of nodes at height i. (In particular, $n_0 = n = |S|$.) Our approach is simple—we maintain the following invariant for some fixed constant $\alpha \in (1/2, 1)$:

$$n_{i+1} \leq \alpha n_i.$$

(To be concrete, we can set $\alpha = 3/4$, for example.) The invariant clearly implies that the maximum height is at most $\log_{1/\alpha} n$. When the invariant is violated for some i, a "seismic" event occurs and we remove everything from height $i+1$ and up, to allow rebuilding later. Since $n_{i+1} = n_{i+2} = \cdots = 0$ now, the invariant is restored. Intuitively, events of large "magnitude" (i.e., events at low heights i) should occur infrequently.

Pseudocode. We give pseudocode for all three operations below:

insert(x):
1. create a new tree containing $\{x\}$

decrease-key(x, k):
1. cut the subtree rooted at the highest node storing x [yields 1 new tree]
2. change x's value to k

delete-min():
1. $x \leftarrow$ minimum of all the roots
2. remove the path of nodes storing x [yields multiple new trees]
3. while there are 2 trees of the same height:
4. link the 2 trees [reduces the number of trees by 1]
5. if $n_{i+1} > \alpha n_i$ for some i then:
6. let i be the smallest such index
7. remove all nodes at heights $> i$ [increases the number of trees]
8. return x

We can explicitly maintain a pointer to the highest node storing x for each element x; it is easy to update these pointers as linkings are performed. It is also easy to update the n_i's as nodes are created and removed. Lines 3–4 in delete-min() can be done in time proportional to the current number of trees, by using an auxiliary array of pointers to trees indexed by their heights.

Analysis. In the current data structure, let N be the number of nodes, T be the number of trees, and B be the number of degree-1 nodes (the "bad" nodes). Define the potential to be $N + T + \frac{1}{2\alpha-1}B$. The amortized cost of an operation is the actual cost plus the change in potential.

For insert(), the actual cost is $O(1)$, and N and T increase by 1. So, the amortized cost is $O(1)$.

For decrease-key(), the actual cost is $O(1)$, and T and B increase by 1. So, the amortized cost is $O(1)$.

For delete-min(), we analyze lines 1–4 first. Let $T^{(0)}$ be the value of T just before the operation. Recall that the maximum height, and thus the length of the path in line 2, is $O(\log n)$. We can bound the actual cost by $T^{(0)} + O(\log n)$. Since after lines 3–4 there can remain at most one tree per height, T is decreased to $O(\log n)$. So, the change in T is $O(\log n) - T^{(0)}$. Since linking does not create degree-1 nodes, the change in B is nonpositive. Thus, the amortized cost is $O(\log n)$.

For lines 5–7 of delete-min(), let $n_j^{(0)}$ be the value of n_j just before these lines. We can bound the actual cost of lines 5–7 by $\sum_{j>i} n_j^{(0)}$. The change in N is at most $-\sum_{j>i} n_j^{(0)}$. The change in T is at most $+n_i^{(0)}$. Let $b_i^{(0)}$ be the number of degree-1 nodes at height i just before lines 5–7. Observe that $n_i^{(0)} \geq 2n_{i+1}^{(0)} - b_i^{(0)}$. Thus, $b_i^{(0)} \geq 2n_{i+1}^{(0)} - n_i^{(0)} \geq (2\alpha - 1)n_i^{(0)}$. Hence, the change in B is at most $-(2\alpha - 1)n_i^{(0)}$. Thus, the net change in $T + \frac{1}{2\alpha-1}B$ is nonpositive. We conclude that the amortized cost of lines 5–7 is nonpositive. Therefore, the overall amortized cost for delete-min() is $O(\log n)$.

3 Comments

Like Fibonacci heaps, our method can easily support the meld (i.e., merge) operation in $O(1)$ amortized time, by just concatenating the lists of trees.

Many variations of the method are possible. Linking of equal-height trees can be done at other places, for example, immediately after an insertion or after lines 5–7 of delete-min(), without affecting the amortized cost. Alternatively, we can perform less linking in lines 3–4 of delete-min(), as long as the number of trees is reduced by a fraction if it exceeds $\Theta(\log n)$.

We can further relax the invariant to $n_{i+c} \leq \alpha n_i$ for any integer constant c. In the analysis, the potential can be readjusted to $N + T + \frac{1}{c(2\alpha-1)}B$. It is straightforward to check that the amortized number of comparisons per decrease-key() is at most $1 + \frac{1}{c(2\alpha-1)}$, which can be made arbitrarily close to 1 at the

expense of increasing the constant factor in delete-min(). (A similar tradeoff of constant factors is possible with Fibonacci heaps as well, by relaxing the "lose at most one child per node" property to "at most c children" [5].)

In the tournament trees, it is convenient to assume that the smaller child of each node is always the left child (and if the node has degree 1, its only child is the left child).

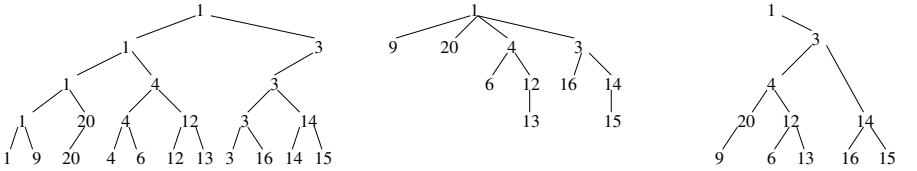


Fig. 2. Transforming a tournament tree into a heap-ordered tree or a half-ordered tree

The tournament trees require a linear number of extra nodes, but more space-efficient representations are possible where each element is stored in only one node. One option is to transform each tournament tree T into a heap-ordered, $O(\log n)$-degree tree T': the children of x in T' are the right children of all the nodes storing x in T. See Figure 2 (middle). Binomial heaps and Fibonacci heaps are usually described for trees of this form.

Another option is to transform T into a binary tree T'' as follows: after short-cutting degree-1 nodes in T, the right child of x in T'' is the right child of the highest node storing x in T; the left child of x in T'' is the right child of the sibling of the highest node storing x in T. See Figure 2 (right). The resulting tree T'' is a *half-ordered* binary tree: the value of every node x is smaller than the value of any node in the right subtree of x. Høyer [7] advocated the use of such trees in implementation. It is straightforward to redescribe our method in terms of half-ordered binary trees. For example, see [7] on the analogous linking and cutting operations.

While our method is simple to understand conceptually, we do not claim that it would lead to the shortest code, nor the fastest implementation in practice, compared to existing methods.

Philosophically, Peterson's and Høyer's work demonstrated that a heap data structure supporting decrease-key() in constant amortized time can be obtained from techniques for balanced search trees supporting deletions in constant amortized time. The moral of this note is that the heap problem is in fact simpler than balanced search trees—a very simple lazy update algorithm suffices to ensure balance for heaps.

References

1. Driscoll, J., Gabow, H., Shrairman, R., Tarjan, R.: Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM 31, 1343–1354 (1988)

2. Elmasry, A.: The violation heap: a relaxed Fibonacci-like heap. Discrete Math., Alg. and Appl. 2, 493–504 (2010)
3. Elmasry, A.: Pairing heaps with $O(\log \log n)$ decrease cost. In: Proc. 20th ACM–SIAM Sympos. Discrete Algorithms, pp. 471–476 (2009)
4. Fredman, M., Sedgewick, R., Sleator, D., Tarjan, R.: The pairing heap: a new form of self-adjusting heap. Algorithmica 1, 111–129 (1986)
5. Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34, 596–615 (1987)
6. Haeupler, B., Sen, S., Tarjan, R.E.: Rank-pairing heaps. SIAM J. Comput. 40, 1463–1485 (2011)
7. Høyer, P.: A general technique for implementation of efficient priority queues. In: Proc. 3rd Israel Sympos. Theory of Comput. Sys., pp. 57–66 (1995)
8. Kaplan, H., Tarjan, R.: Thin heaps, thick heaps. ACM Trans. Algorithms 4(1), 3 (2008)
9. Peterson, G.: A balanced tree scheme for meldable heaps with updates. Tech. Report GIT-ICS-87-23, Georgia Institute of Technology (1987)
10. Pettie, S.: Towards a final analysis of pairing heaps. In: Proc. 46th IEEE Sympos. Found. Comput. Sci., pp. 174–183 (2005)
11. Takaoka, T.: Theory of 2-3 heaps. Discrete Applied Math. 126, 115–128 (2003)

Variations on Instant Insanity

Erik D. Demaine[1], Martin L. Demaine[1], Sarah Eisenstat[1],
Thomas D. Morgan[2], and Ryuhei Uehara[3]

[1] MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar St., Cambridge, MA 02139, USA
{edemaine,mdemaine,seisenst}@mit.edu
[2] Harvard University School of Engineering and Applied Sciences,
33 Oxford St., Cambridge, MA 02138, USA
tdmorgan@seas.harvard.edu
[3] School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST),
Asahidai 1-1, Nomi, Ishikawa 923-1292, Japan
uehara@jaist.ac.jp

Abstract. In one of the first papers about the complexity of puzzles, Robertson and Munro [14] proved that a generalized form of the then-popular Instant Insanity puzzle is NP-complete. Here we study several variations of this puzzle, exploring how the complexity depends on the piece shapes and the allowable orientations of those shapes.

Prepared in honor of Ian Munro's 66th birthday.

1 Introduction

In the late 1960s, the company Parker Brothers popularized a puzzle known as *Instant Insanity*[1]. Instant Insanity is composed of four cubes, where each face of each cube is colored red, green, white, or blue. The goal is to arrange the cubes in a tower with dimensions $1 \times 1 \times 4$ so that on each of the four long sides of the tower, every color appears (exactly once per side). This puzzle has a rich history — the name *Instant Insanity* dates back to 1967 [13], but there were many earlier variants, released under such names as *Katzenjammer*, *Groceries*, and *The Great Tantalizer* [12].

The mathematics behind the Instant Insanity puzzle have been studied extensively, and the puzzle is used as a sample application of graph theory in some textbooks [1,4,16]. In 1978, Robertson and Munro [14] showed that, by generalizing the number of colors and cubes from 4 to n, the puzzle becomes NP-complete to solve; they also proved a two-player variant PSPACE-complete. Their paper was one of the first to study the computational complexity of puzzles and games [5,8,11]. More recently, there have been two studies of variants

[1] The name "Instant Insanity" was originally trademarked by Parker Brothers in 1967. The trademark is currently owned by Winning Moves, Inc.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 33–47, 2013.

Fig. 1. A modern instance of *Instant Insanity* (left, distributed by Winning Moves, Inc., 2010), and a much older *The Great Tantalizer* (right, indicated only as British Made, consisting of wooden blocks).

of Instant Insanity. In 2002, Jebasingh and Simoson [10] studied variants of the problem with Platonic solid pieces. In 2008, Berkove et al. [3] studied a problem in which the goal is to combine cubes to form a larger box with consistent colors on each side.

Our Results. Inspired by Robertson and Munro [14], our goal is to explore how the complexity of the puzzle changes with the shape of the pieces and the set of allowable motions. In particular, we consider puzzles in which all of the pieces are shaped like identical right prisms, and the goal is to stack the pieces to form a taller prism with the same base. In Section 2, we establish a combinatorial definition for the pieces of a puzzle, and give a formal definition of the Instant Insanity problem.

In Section 3.2, we examine the case of pieces where the base of the prism is a regular polygon. When the base is an equilateral triangle, we show that the problem is easy to solve. When the base is a square (but the piece is not a cube), we prove that the problem is NP-complete, even though the number of allowable configurations for each piece is only a third as large as the number of configurations for the original Instant Insanity problem.

In Section 3.3, we consider the case of regular polygon prisms where the motion is restricted, and show that even in the case of equilateral triangle pieces with three possible configurations per piece, the problem remains NP-complete. Finally, in Section 4, we prove results about some irregular prism pieces, using a technique for solving any puzzle in which each piece has two possible configurations.

2 Definitions

2.1 Instant Insanity

Let C be a finite set of colors. Given a polyhedral *piece shape*, let k_1 be the number of potentially visible sides, and let k_2 be the number of sides that are

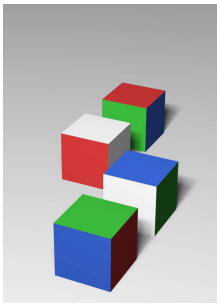


Fig. 2. A rendering of the original Instant Insanity puzzle. Although the back of the tower is not shown, each color occurs exactly once on each of the four long sides of the tower of blocks.

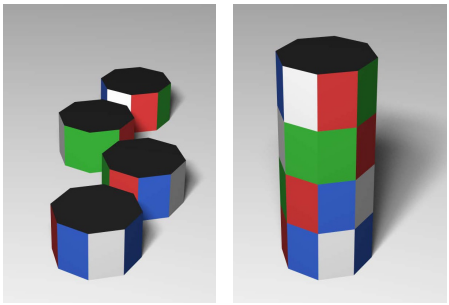


Fig. 3. A generalized version of the puzzle with octagonal prism pieces. The top and bottom of each prism can never lie on any of the eight sides of the tower. The other eight faces are visible at all times.

visible in any given configuration. In this paper, we restrict ourselves to pieces in the shape of a right prism, which often implies that $k_1 = k_2$. For each (potentially visible) side of the piece shape, we assign a unique number from the set $\{1, \ldots, k_1\}$. Hence, a single piece can be represented by a tuple in C^{k_1} assigning a color to each of the k_1 sides. When defining a piece, we sometimes use the special symbol $*$, which represents a unique color used once in the entire puzzle.

The set of possible configurations of a single piece is given by the *piece configurations* $P \subseteq \{1, \ldots, k_1\}^{k_2}$. Each piece configuration indicates how the colors in the original piece should be assigned to the visible sides. Specifically, a piece configuration is a mapping from each of the k_2 visible sides to the index of one of the sides of the piece shape. A single side of a single piece cannot appear on multiple sides of the puzzle, so each piece configuration has the additional restriction that no element of the tuple is repeated.

For each visible side $1 \leq i \leq k_2$, we define the function $F_i : C^{k_1} \times P \to C$ to return the color of side i, given a piece and its configuration. Formally, we say:

$$F_i(\langle a_1, \ldots, a_{k_1} \rangle, \langle q_1, \ldots, q_{k_2} \rangle) = a_{q_i}.$$

As an example, we consider the original Instant Insanity puzzle. Each cube has $k_1 = 6$ sides with colors on them; only $k_2 = 4$ of those sides are visible when the cubes are stacked vertically. Suppose that we number the sides as depicted in Fig. 4. Then there are 24 possible piece configurations: 6 ways to choose the side that faces down, and 4 possible ways to rotate the visible faces. These 24 piece configurations are listed in Fig. 5.

Many piece shapes, including the cube, have a number of symmetries. In particular, many have the following property:

Definition 1. *The set of piece configurations P is* rotationally symmetric *if P is closed under cyclic shifts.*

Using this combinatorial definition of piece configurations, we can formally define two variants of the Instant Insanity problem:

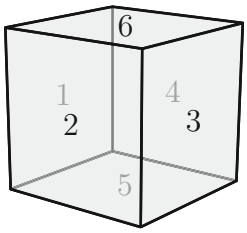


Fig. 4. Labels for each of the faces on a cube

$\{\langle 1,2,3,4\rangle, \langle 2,3,4,1\rangle, \langle 3,4,1,2\rangle, \langle 4,1,2,3\rangle,$
$\langle 1,4,3,2\rangle, \langle 2,1,4,3\rangle, \langle 3,2,1,4\rangle, \langle 4,3,2,1\rangle,$
$\langle 1,5,3,6\rangle, \langle 5,3,6,1\rangle, \langle 3,6,1,5\rangle, \langle 6,1,5,3\rangle,$
$\langle 1,6,3,5\rangle, \langle 5,1,6,3\rangle, \langle 3,5,1,6\rangle, \langle 6,3,5,1\rangle,$
$\langle 2,5,4,6\rangle, \langle 5,4,6,2\rangle, \langle 4,6,2,5\rangle, \langle 6,2,5,4\rangle,$
$\langle 2,6,4,5\rangle, \langle 5,2,6,4\rangle, \langle 4,5,2,6\rangle, \langle 6,4,5,2\rangle\}$

Fig. 5. The 24 different piece configurations for the cube piece depicted in Fig. 4

Definition 2. *The* Complete-Insanity(P) *problem is defined as follows:*

Input: *A set of colors C and a sequence of pieces $A_1, \ldots, A_n$, where $n = |C|$.*
Output: Yes *if and only if there is a sequence of configurations $p_1, \ldots, p_n \in P$ such that for each side $1 \leq i \leq k_2$, the set of visible colors $\{F_i(A_j, p_j) \mid 1 \leq j \leq n\} = C$.*

Definition 3. *The* Partial-Insanity(P) *problem is defined as follows:*

Input: *A set of colors C and a sequence of pieces $A_1, \ldots, A_n$, where $n \leq |C|$.*
Output: Yes *if and only if there is a sequence of configurations $p_1, \ldots, p_n \in P$ such that for each side $1 \leq i \leq k_2$, all of the visible colors $F_i(A_j, p_j)$ on side i are distinct.*

Note that both problems require all visible colors on a single side to be distinct; however, the Partial-Insanity(P) problem only requires a subset of all of the colors to be visible on a single side, while the Complete-Insanity(P) problem requires all colors to be visible. Clearly, the Partial-Insanity(P) problem is at least as hard as the Complete-Insanity(P) problem, and both are contained in the complexity class NP.

2.2 Positive Not-All-Equal Satisfiability

In this paper, we prove that several variants of the Instant Insanity puzzle are NP-complete to solve by reducing from a known NP-complete problem. The problem we use for these reductions is a variant of the 3-Sat problem known as Positive-Not-All-Equal-Sat, or Positive-NAE-Sat.

Definition 4. *The* Positive-NAE-Sat *problem is defined as follows:*

Input: *A set of n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. Each clause C_i is composed of three positive literals $c_{i,1}, c_{i,2}, c_{i,3} \in \{x_1, \ldots, x_n\}$.*
Output: Yes *if there exists some mapping $\phi : \{x_1, \ldots, x_n\} \to \{T, F\}$ such that each clause C_i has at least one literal c_{i,j_1} such that $\phi(c_{i,j_1}) = T$, and at least one literal c_{i,j_2} such that $\phi(c_{i,j_2}) = F$.*

This problem was one of many Boolean satisfiability problems shown to be NP-complete by Schaefer [15].

3 Regular Prism Pieces

3.1 Partial versus Complete Insanity

In this section, we show that for many types of regular pieces, if the Partial-Insanity(P) problem is NP-complete, then the Complete-Insanity(P) problem is NP-complete. We do this by way of a third problem in which the use of each color is restricted. Specifically, we restrict the *number of occurrences of* $c \in C$: the sum over all pieces A_i of the number of sides of A_i with the color c.

Definition 5. *The* One-or-All-Insanity(P) *problem is defined as follows:*

Input: *A set of colors C and a sequence of pieces $A_1, \ldots, A_n$, where $n \leq |C|$. Furthermore, the number of occurrences of each color $c \in C$ is either 1 or k_2.*

Output: Yes *if and only if there is a sequence of configurations $p_1, \ldots, p_n \in P$ such that for each side $1 \leq i \leq k_2$, all of the visible colors $F_i(A_j, p_j)$ on side i are distinct.*

Lemma 1. *Suppose that $k_1 = k_2$ and the set of piece configurations P is rotationally symmetric. Then there exists a polynomial-time reduction from the problem* Partial-Insanity(P) *to the problem* One-or-All-Insanity(P).

Proof. Suppose that we are given an instance of the Partial-Insanity(P) problem consisting of colors C and pieces $A_1, \ldots, A_n$. Using these pieces, we construct an instance of the One-or-All-Insanity(P) problem as follows:

1. The first n pieces are identical to the pieces $A_1, \ldots, A_n$.
2. For each color $c \in C$, let $\#(c)$ be the number of occurrences of c in the set of pieces $A_1, \ldots, A_n$. If $\#(c) = 1$ or $\#(c) = k_2$, then we do not need to do anything. If $\#(c) > k_2$, then the puzzle must be impossible to solve. Otherwise, we generate $k_2 - \#(c)$ new pieces, each defined as follows:

$$\langle c, \underbrace{*, *, \ldots, *}_{k_2 - 1 \text{ times}} \rangle.$$

(Recall that each $*$ symbol represents a unique color used exactly once.)

Let D be the set of colors generated by this process, and let $B_1, \ldots, B_m$ be the pieces. Our construction ensures that the pieces $B_1, \ldots, B_m$ form a valid instance of the One-or-All-Insanity(P) problem. We must additionally show that the puzzle formed by $A_1, \ldots, A_n$ can be solved if and only if $B_1, \ldots, B_m$ can be solved.

Suppose that we have a solution $p_1, \ldots, p_n$ to the problem $A_1, \ldots, A_n$. If we use $p_1, \ldots, p_n$ to configure $B_1, \ldots, B_n$, then we are guaranteed that for the first n pieces, no color is used twice on the same side. So our goal is to find a way to configure the remaining $m - n$ pieces. All of the new colors generated in step 2 of the reduction occur exactly once, so they can be used on any side of the puzzle. Because the set of piece configurations P is rotationally symmetric, the

first color in each piece B_i for $n+1 \leq i \leq m$ can be placed on any side of the puzzle. Each color occurs at most k_2 times, so it is straightforward to arrange the pieces $B_{n+1}, \ldots, B_m$ to ensure that no color occurs twice on any side.

Furthermore, if there exists a solution $q_1, \ldots, q_m$ to the problem $B_1, \ldots, B_m$, then we can find a solution to $A_1, \ldots, A_n$ using the piece configurations $q_1, \ldots, q_n$. This means that our polynomial-time reduction is correct. □

Lemma 2. *Suppose that $k_1 = k_2$ and the set of piece configurations P is rotationally symmetric. Then there exists a polynomial-time reduction from* One-or-All-Insanity(P) *to* Complete-Insanity(P).

Proof. Suppose that we are given an instance of the One-or-All-Insanity(P) problem consisting of colors C and pieces $A_1, \ldots, A_n$. Let C_1 be the set of colors that are used once, and let C_2 be the set of colors that are used k_2 times. For each color $c \in C_2$, let $g(c)$ be a unique index in $\{1, \ldots, |C_2|\}$.

For each $1 \leq i \leq k_2$, we construct n pieces $B_{i,1}, \ldots, B_{i,n}$ and $|C_2|$ distinct colors $D_i = \{d_{i,1}, \ldots, d_{i,|C_2|}\}$. If the piece A_j consists of the sequence of colors $a_{j,1}, \ldots, a_{j,k_2}$, then we define the colors $b_{i,j,1}, \ldots, b_{i,j,k_2}$ for the piece $B_{i,j}$ as follows:

$$b_{i,j,k} = \begin{cases} d_{i,g(a_{j,k})}, & \text{if } a_{j,k} \in C_2; \\ a_{j,k}, & \text{otherwise.} \end{cases}$$

This ensures that each color $c \in (C_1 \cup D_1 \cup \cdots \cup D_{k_2})$ is used exactly k_2 times in the set of pieces $B_{1,1}, \ldots, B_{k_2,n}$. Hence the number of colors is $k_2 n$, which is the same as the number of pieces, making this an instance of Complete-Insanity(P).

Now we wish to show that $A_1, \ldots, A_n$ is solvable if and only if $B_{1,1}, \ldots, B_{k_2,n}$ is solvable. First, suppose that $B_{1,1}, \ldots, B_{k_2,n}$ is solvable. This means that there is a way to configure $B_{1,1}, \ldots, B_{1,n}$ such that no color is used twice on a single side. The pieces $B_{1,1}, \ldots, B_{1,n}$ are almost identical to the pieces $A_1, \ldots, A_n$, with the exception that each color $c \in C_2$ is replaced with the color $d_{1,g(c)}$. Hence, a configuration of $B_{1,1}, \ldots, B_{1,n}$ that does not reuse colors on any side must also be a configuration of $A_1, \ldots, A_n$ that does not reuse colors on any side. So if $B_{1,1}, \ldots, B_{k_2,n}$ is a solvable instance of Complete-Insanity(P), then $A_1, \ldots, A_n$ is a solvable instance of One-or-All-Insanity(P).

Next, suppose that $A_1, \ldots, A_n$ is solvable. This means that there is a sequence of configurations $p_1, \ldots, p_n$ such that if we apply $p_1, \ldots, p_n$ to the pieces $A_1, \ldots, A_n$, then no color is used twice on a single side. Let σ_i be the circular shift of k_2 elements by i so that $\sigma_i(\langle a_1, \ldots, a_{k_2} \rangle) = \langle a_{i+1 \bmod k_2}, \ldots, a_{i+k_2 \bmod k_2} \rangle$. Because P is rotationally symmetric, for any $1 \leq i \leq k_2$ and any $1 \leq j \leq n$, $\sigma_i(p_j) \in P$.

Suppose that we use the configuration $\sigma_i(p_j)$ for the piece $B_{i,j}$. Suppose, for the sake of contradiction, that there is some pair of distinct pieces B_{i_1,j_1} and B_{i_2,j_2} that assigns the same color c to the same side of the puzzle. Note that the colors of B_{i_1,j_1} must be a subset of $C_1 \cup D_{i_1}$, while the colors of B_{i_2,j_2} must be a subset of $C_1 \cup D_{i_2}$. We consider three cases:

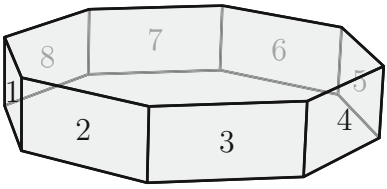


Fig. 6. Labels for each of the faces on an 8-sided regular prism

Case 1: $i_1 \neq i_2$. Then it must be that $c \in C_1$. Because $A_1, \ldots, A_n$ is an instance of One-or-All-Insanity(P), there is exactly one piece A_{j_3} that uses the color c. By construction of the pieces B_{i_1,j_1} and B_{i_2,j_2}, it must be that $j_1 = j_2 = j_3$, and the index of c is the same in both pieces. The configuration of B_{i_1,j_1} is $\sigma_{i_1}(p_{j_1})$, while the configuration of B_{i_2,j_2} is $\sigma_{i_2}(p_{j_2}) = \sigma_{i_2}(p_{j_1})$. Hence the configurations would map the color c to different sides of the puzzle, contradicting our assumption that c occurs twice on the same side.

Case 2: $i_1 = i_2$ and $c \in C_1$. Because $c \in C_1$, it is used exactly once in the original set of pieces $A_1, \ldots, A_n$. Therefore, it is also used exactly once in the set of pieces $B_{i_1,1}, \ldots, B_{i_1,n}$, and so if it occurs both in B_{i_1,j_1} and $B_{i_2,j_2} = B_{i_1,j_2}$, then we must have $j_1 = j_2$. Again, this means that c is not used twice.

Case 3: $i_1 = i_2$ and $c \notin C_1$. Let $i = i_1 = i_2$. By our assumption, we know that there exists some ℓ such that $F_\ell(B_{i,j_1}, \sigma_i(p_{j_1})) = F_\ell(B_{i,j_2}, \sigma_i(p_{j_2})) = c$. Because σ_i is a cyclic shift, this means that there is some ℓ' such that $F_{\ell'}(B_{i,j_1}, p_{j_1}) = F_{\ell'}(B_{i,j_2}, p_{j_2}) = c$. By construction of B_{i,j_1} and B_{i,j_2}, this means that $F_{\ell'}(A_{j_1}, p_{j_1}) = F_{\ell'}(A_{j_2}, p_{j_2})$, which is a contradiction.

□

The combination of Lemmas 1 and 2 yields the following theorem:

Theorem 1. *Suppose that $k_1 = k_2$ and the set of piece configurations P is rotationally symmetric. Then there exists a polynomial-time reduction from the* Partial-Insanity(P) *problem to the* Complete-Insanity(P) *problem.*

3.2 Regular Prism Pieces

Definition 6. *The* k-sided regular prism *is a right prism whose base is a regular k-sided polygon. We leave the bases of the prism unlabeled, because they cannot become visible. The other k faces we label in order, as depicted in Fig. 6. We define $\mathcal{R}_k$ to be the set of piece configurations of a k-sided regular prism: in particular, $\mathcal{R}_k$ is the union of all cyclic shifts of $\langle 1, \ldots, k \rangle$ and its reverse.*

Note that in the case of $k = 4$, we assume that the height of the prism and the side length of the square base are distinct, so that the pieces do not have the full range of motion of the original Instant Insanity puzzle.

Our first result concerns right prism pieces with an equilateral triangle base:

Theorem 2. *The* Partial-Insanity($\mathcal{R}_3$) *problem can be solved in polynomial time.*

Proof. Let $\mathcal{P}_k$ be the set of all permutations of $\{1, \ldots, k\}$. By definition, $\mathcal{R}_3 = \mathcal{P}_3$. Furthermore, for any k, the set of piece configurations $\mathcal{P}_k$ is rotationally symmetric, so by Theorem 1, if the Complete-Insanity($\mathcal{P}_k$) problem is solvable in polynomial time, then the Partial-Insanity($\mathcal{P}_k$) problem is also solvable in polynomial time. Hence we may examine the Complete-Insanity($\mathcal{P}_k$) instead.

In particular, our goal is to show that Complete-Insanity($\mathcal{P}_k$) can be solved if and only if every color $c \in C$ occurs exactly k times in the pieces $A_1, \ldots, A_n$. It is easy to see that if a particular series of pieces $A_1, \ldots, A_n$ can be solved, then the number of occurrences of each color must be k. We may show the converse by induction on k.

First, consider the base case Complete-Insanity($\mathcal{P}_1$). If all colors occur exactly once, then the only possible configuration of the puzzle is in fact a solved state. By induction, we assume that Complete-Insanity($\mathcal{P}_{k-1}$) can be solved if each color is used exactly $k-1$ times. Suppose that we have an instance $A_1, \ldots, A_n$ of the Complete-Insanity($\mathcal{P}_k$) puzzle in which each color is used exactly k times. We construct a multigraph G as follows:

1. For each index $1 \leq i \leq n$, construct a node u_i.
2. For each color $c \in C$, construct a node v_c.
3. For each piece $A_i = (a_1, \ldots, a_k)$ and each side $1 \leq j \leq k$, construct an edge from u_i to v_{a_j}. (Note that this may result in the creation of parallel edges.)

Because each piece has k colors, and each color is used exactly k times, G is a k-regular bipartite multigraph. By applying Hall's theorem [7], we know that G has a perfect matching, which can be computed in $O(kn\sqrt{n})$ time with the Hopcroft-Karp algorithm [9]. For each edge (u_i, v_c) in the matching, we wish to configure piece A_i so that the color on the first visible side is c. By using the matching to determine the colors on side 1, we ensure that no color will occur twice. Furthermore, because the set of piece configurations $\mathcal{P}_k$ consists of all permutations, we know that for each piece A_i, the remaining $k-1$ sides can be arbitrarily assigned to the remaining $k-1$ visible sides of the puzzle. In particular, we are left with an instance of the Complete-Insanity($\mathcal{P}_{k-1}$) problem in which each color occurs $k-1$ times. Hence, by induction, the puzzle is solvable. In particular, the Partial-Insanity($\mathcal{R}_3$) problem can be solved in $O(n\sqrt{n})$ time. □

Note that the structure of this proof reveals some interesting properties about the problem Complete-Insanity($\mathcal{R}_3$). In particular, we don't need to know anything about the assignment of colors to pieces to determine whether a given Complete-Insanity($\mathcal{R}_3$) puzzle can be solved — we need only check whether the number of occurrences of each color is 3. Furthermore, careful analysis reveals that the reduction of Theorem 1 will produce an Complete-Insanity($\mathcal{R}_3$) puzzle with 3 copies of each color if and only if the original Partial-Insanity($\mathcal{R}_3$) puzzle has at most 3 copies of each color. Hence, to determine whether a given

instance of PARTIAL-INSANITY($\mathcal{R}_3$) can be solved, it is sufficient to check that every color occurs ≤ 3 times.

The PARTIAL-INSANITY($\mathcal{R}_4$) problem is not as easy to solve:

Theorem 3. *The* PARTIAL-INSANITY($\mathcal{R}_4$) *problem is NP-complete.*

Proof. We show that PARTIAL-INSANITY($\mathcal{R}_4$) is NP-complete by a reduction from the POSITIVE-NAE-SAT problem. Suppose that we are given a POSITIVE-NAE-SAT instance θ with n clauses, each containing three literals. Let m be the number of variables. Then we can represent which literals are associated with which variables by a function $\gamma : \{0, \ldots, 3n-1\} \to \{0, \ldots, m-1\}$ mapping the index of each literal to the index of the corresponding variable. Let $L(i) = \{x \mid \gamma(x) = i\}$, and let $\ell(i,k)$ be the kth literal index in $L(i)$. Then our reduction proceeds as follows.

For each literal index $0 \leq x \leq 3n-1$, add three colors: a_x, b_x, and c_x. For each variable $0 \leq i \leq m-1$ and each index $0 \leq h \leq |L(i)| - 1$, let $x = \ell(i,h)$ and $y = \ell(i, (h+1) \bmod |L(i)|)$. Use these values to construct two pieces:

$$A_{x,0} = \langle a_x, b_x, c_x, b_x \rangle$$
$$A_{x,1} = \langle a_x, c_x, b_y, c_x \rangle$$

Intuitively, the placement of the color a_x will determine whether or not the corresponding literal is true — in particular, the structure of the pieces $A_{x,0}$ and $A_{x,1}$ ensures that the two copies of the color a_x must occur on opposite sides of the puzzle. Hence the color a_x can either be placed on sides 1 and 3, or on sides 2 and 4. These two possibilities represent the two possible assignments to literal x. The use of the color b_y in $A_{x,1}$ ensures that the assignment to literal x is the same as the assignment to literal y, so that the assignment is consistent.

We must also add pieces to ensure that the assignment is satisfying. For each clause $0 \leq j \leq n-1$, add a new color d_j and construct the following three pieces:

$$B_{3j+0} = \langle d_j, a_{3j+0}, *, * \rangle$$
$$B_{3j+1} = \langle d_j, a_{3j+1}, *, * \rangle$$
$$B_{3j+2} = \langle d_j, a_{3j+2}, *, * \rangle$$

We wish to show that, when taken together, these pieces form a puzzle that can be solved if and only if the original POSITIVE-NAE-SAT problem can be solved.

Suppose that we have a consistent solution to the original POSITIVE-NAE-SAT instance θ. We can represent this solution as a function $\phi : \{1, \ldots, 3n\} \to \{T, F\}$ assigning a value of true or false to each of the literals. This assignment has the property that for any pair of indices j_1, j_2, if $\gamma(j_1) = \gamma(j_2)$, then $\phi(j_1) = \phi(j_2)$. Using this assignment, we wish to construct configurations $p_{0,0}, p_{0,1}, \ldots, p_{3n-1,0}, p_{3n-1,1}$ for the pieces $A_{0,0}, A_{0,1}, \ldots, A_{3n-1,0}, A_{3n-1,1}$ and configurations $q_0, \ldots, q_{3n-1}$ for the pieces $B_0, \ldots, B_{3n-1}$. We define these configurations as follows:

- For each literal $0 \leq x \leq 3n-1$, if $\phi(x) = T$, then we set $p_{x,0} = \langle 1, 2, 3, 4 \rangle$ and $p_{x,1} = \langle 3, 4, 1, 2 \rangle$. Otherwise, we set $p_{x,0} = \langle 2, 3, 4, 1 \rangle$ and $p_{x,1} = \langle 4, 1, 2, 3 \rangle$.

– For each clause $0 \le j \le n-1$, we assign q_{3j+0}, q_{3j+1}, and q_{3j+2} according to the following table:

$\phi(3j+0)$	$\phi(3j+1)$	$\phi(3j+2)$	q_{3j+0}	q_{3j+1}	q_{3j+2}
T	T	F	$\langle 1,2,3,4\rangle$	$\langle 3,4,1,2\rangle$	$\langle 2,3,4,1\rangle$
T	F	T	$\langle 1,2,3,4\rangle$	$\langle 2,3,4,1\rangle$	$\langle 3,4,1,2\rangle$
T	F	F	$\langle 1,2,3,4\rangle$	$\langle 2,3,4,1\rangle$	$\langle 4,1,2,3\rangle$
F	T	T	$\langle 4,1,2,3\rangle$	$\langle 3,4,1,2\rangle$	$\langle 1,2,3,4\rangle$
F	T	F	$\langle 4,1,2,3\rangle$	$\langle 3,4,1,2\rangle$	$\langle 2,3,4,1\rangle$
F	F	T	$\langle 4,1,2,3\rangle$	$\langle 2,3,4,1\rangle$	$\langle 1,2,3,4\rangle$

It may be verified that by configuring the pieces in this way, we ensure that no color appears twice on the same side of the puzzle. Hence we have shown that the existence of a solution to the Positive-NAE-Sat instance implies the existence of a solution to the constructed Partial-Insanity($\mathcal{R}_4$) puzzle.

Suppose now that the Partial-Insanity($\mathcal{R}_4$) problem can be solved. Let $p_{0,0}, \ldots, p_{3n-1,1}$ be the sequence of configurations for the pieces $A_{0,0}, \ldots, A_{3n-1,1}$, and let $q_0, \ldots, q_{3n-1}$ be the sequence of configurations for $B_0, \ldots, B_{3n-1}$. Then we construct an assignment function $\phi : \{1, \ldots, 3n\} \to \{T, F\}$ as follows: for each literal $0 \le x \le 3n-1$, if the configuration $p_{x,0} = (s_1, s_2, s_3, s_4)$, then we define $\phi(x) = T$ if and only if $s_1 \equiv 1 \pmod 2$. To see that this is a valid assignment, we must show that this assignment is both consistent and satisfying.

Suppose, for the sake of contradiction, that the assignment is not consistent. This means that there exists some variable i such that $\{\phi(j) \mid \gamma(j) = i\} = \{T, F\}$. Then there must be some index h such that $\phi(\ell(i,h)) \neq \phi(\ell(i,(h+1) \bmod |L(i)|))$. Let $x = \ell(i,h)$ and let $y = \ell(i,(h+1) \bmod |L(i)|)$. Without loss of generality, we may assume that $\phi(x) = F$ and $\phi(y) = T$. Hence $F_2(A_{y,0}, p_{y,0}) = F_4(A_{y,0}, p_{y,0}) = b_y$. No color is used twice on the same side, so we know that either $F_1(A_{x,1}, p_{x,1}) = b_y$ or $F_3(A_{x,1}, p_{x,1}) = b_y$. In either case, all possible configurations $p_{x,1}$ have the property that $F_2(A_{x,1}, p_{x,1}) = F_4(A_{x,1}, p_{x,1}) = c_x$. By assumption, $\phi(x) \neq \phi(y)$, so either $F_2(A_{x,0}, p_{x,0}) = c_x$ or $F_4(A_{x,0}, p_{x,0}) = c_x$. In either case, we have a contradiction — colors cannot be repeated on the same side of the puzzle. Hence, the assignment of values to literals must be consistent.

Now suppose that there is some clause j such that $\phi(3j+0) = \phi(3j+1) = \phi(3j+2)$. Without loss of generality, we may assume that all three of these values are T. For each literal index $x \in \{3j+0, 3j+1, 3j+2\}$, pieces $A_{x,0}$ and $A_{x,1}$ must be configured so that they do not place the color c_x on the same side of the puzzle. Furthermore, because $\phi(x) = T$, we know that the configuration of $A_{x,0}$ must place the color a_x on side 1 or side 3. Together, these constraints ensure that the configuration of $A_{x,1}$ must also place the color a_x on side 1 or side 3. Hence between the pieces $A_{x,0}$ and $A_{x,1}$, the color a_x must be placed both on side 1 and on side 3. Then B_x must place a_x on either side 2 or side 4, and must therefore place d_j on side 1 or 3. This holds for all $x \in \{3j+0, 3j+1, 3j+2\}$. As a result, d_j must show up at least twice on either side 1 or side 3. This contradicts our assumption that the puzzle was solved. □

3.3 Regular Prism Pieces with Restricted Motion

In this section, we consider what happens when the motion of the pieces is limited. In particular, we consider what happens when the pieces of the puzzle are mounted on a central dowel, so that the set of allowable motions is restricted to rotations around the dowel.

Definition 7. *We define $\mathcal{U}_k$ to be the* unflippable *piece configurations of the k-sided regular prism: in particular, $\mathcal{U}_k$ is the set of all cyclic shifts of $\langle 1, \ldots, k\rangle$.*

Theorem 4. *The* Partial-Insanity$(\mathcal{U}_k)$ *problem is NP-complete for all $k \geq 3$.*

Proof. Suppose that we are given a Positive-NAE-Sat instance θ with n clauses, each containing three literals. Let m be the number of variables. Again, we represent the relationship between variables and the corresponding literal indices with a function $\gamma : \{0, \ldots, 3n-1\} \to \{0, \ldots, m-1\}$ mapping the index of each literal to the index of the corresponding variable.

Let $L(i) = \{x \mid \gamma(x) = i\}$, and let $\ell(i, j)$ be the jth literal index in $L(i)$. Then our reduction proceeds as follows. For each literal index $0 \leq x \leq 3n-1$, construct six colors: a_x, $b_{x,1}$, $b_{x,2}$, $c_{x,1}$, $c_{x,2}$, and d_x. We then use these colors to define three pieces for each literal index x:

$$A_x = \langle \underbrace{a_x, \ldots, a_x}_{k-2 \text{ times}}, b_{x,1}, b_{x,2}\rangle$$

$$B_x = \langle \underbrace{*, \ldots, *}_{k-3 \text{ times}}, a_x, c_{x,1}, c_{x,2}\rangle$$

$$C_x = \langle \underbrace{*, \ldots, *}_{k-3 \text{ times}}, a_x, d_x, *\rangle$$

To ensure that all of the pieces $A_0, \ldots, A_{3n-1}$ have the same configuration, we use a series of gadgets, each of which consists of $k-1$ identical pieces. Specifically, for every $x \in \{0, \ldots, 3n-2\}$, we create $k-1$ copies of the following piece:

$$G_x = \langle \underbrace{*, \ldots, *}_{k-2 \text{ times}}, b_{x,1}, b_{x+1,2}\rangle$$

These $k-1$ copies of G_x, combined with the piece A_x, all have the color $b_{x,1}$ on side $k-1$. Hence, each piece must have a different configuration. There are k possible configurations, and k pieces, so each configuration is used exactly once. The configurations are rotationally symmetric, so we may suppose without loss of generality that the configuration of A_x is $\langle 1, \ldots, k\rangle$. Every other possible cyclic shift is used to configure one of the pieces G_x. Hence, the color $b_{x+1,2}$ occurs on sides $\{1, 2, 3, \ldots, k-1\}$ — everything except side k. So the only possible configuration for piece A_{x+1} is $\langle 1, \ldots, k\rangle$. By induction, for any pair x_1, x_2, the configuration of A_{x_1} must be the same as the configuration of A_{x_2}.

We also wish to ensure that for any x_1, x_2 such that $\gamma(x_1) = \gamma(x_2)$, the configuration of B_{x_1} is the same as the configuration of B_{x_2}. We accomplish

this with a nearly identical construction. For each variable i, and each index $h \in \{0, \ldots, |L(i)| - 2\}$, let $x = \ell(i, h)$ and let $y = \ell(i, h + 1)$. Then we create $k - 1$ copies of the following piece:

$$H_x = \langle \underbrace{*, \ldots, *}_{k-2 \text{ times}}, c_{x,1}, c_{y,2} \rangle$$

By a similar argument, this enforces the desired constraint.

Finally, for each clause $j \in \{0, \ldots, n-1\}$, we add one more piece:

$$D_j = \langle \underbrace{d_{3j+0}, \ldots, d_{3j+0}}_{k-3 \text{ times}}, d_{3j+0}, d_{3j+1}, d_{3j+2} \rangle.$$

This completes the construction. Next, we must show that this construction is in fact a reduction: that there is a solution to this instance of the Partial-Insanity($\mathcal{U}_k$) problem if and only if there is a solution to the original Positive-NAE-Sat problem.

Suppose that we are given an assignment $\phi : \{0, \ldots, 3n-1\} \to \{T, F\}$ mapping each literal to true or false. Then we can construct a solution to the Partial-Insanity($\mathcal{U}_k$) puzzle as follows. For each A_x, use the configuration $\langle 1, \ldots, k \rangle$. If $\phi(x) = T$, then the configuration of B_x is $\langle k, 1, \ldots, k-1 \rangle$ and the configuration of C_x is $\langle k-1, k, 1, \ldots, k-2 \rangle$. Otherwise, the configuration of B_x is $\langle k-1, k, 1, \ldots, k-2 \rangle$ and the configuration of C_x is $\langle k, 1, \ldots, k-1 \rangle$. The configuration of D_j is determined according to the following table:

$\phi(3j+0)$	$\phi(3j+1)$	$\phi(3j+2)$	configuration of D_j
T	T	F	$\langle k, 1, \ldots, k-2, k-1 \rangle$
T	F	T	$\langle k-1, k, 1, \ldots, k-2 \rangle$
T	F	F	$\langle k-1, k, 1, \ldots, k-2 \rangle$
F	T	T	$\langle 1, \ldots, k-2, k-1, k \rangle$
F	T	F	$\langle k, 1, \ldots, k-2, k-1 \rangle$
F	F	T	$\langle 1, \ldots, k-2, k-1, k \rangle$

It can be verified that by configuring the pieces in this way, we ensure that each color occurs at most once per side.

Because the configurations of A_{x_1} and A_{x_2} are the same for any $x_1, x_2 \in \{0, \ldots, 3n-1\}$, there is a way to configure the gadgets G_x so that the colors $b_{x,1}$ and $b_{x,2}$ occur exactly once on each visible side of the puzzle. Similarly, because the configurations of B_{x_1} and B_{x_2} are the same for any $x_1, x_2 \in \{0, \ldots, 3n-1\}$ with $\gamma(x_1) = \gamma(x_2)$, there is a way to configure the gadgets H_x.

We have shown that if there exists a satisfying assignment for the original Positive-NAE-Sat problem, then there is a solution to the corresponding Partial-Insanity($\mathcal{U}_k$) puzzle. Now we wish to show the converse. To that end, suppose that we have a solution to the puzzle. For each literal $x \in \{0, \ldots, 3n-1\}$, let p_x be the configuration of A_x, let q_x be the configuration of B_x, and let r_x be the configuration of C_x. Furthermore, for each clause $j \in \{0, \ldots, n-1\}$, let s_j be the configuration of D_j. Without loss of generality, suppose that $p_0 = \langle 1, \ldots, k \rangle$.

Then, as we have argued previously, the $k-1$ copies of the pieces $G_0, \ldots, G_{3n-2}$ ensure that $p_x = \langle 1, \ldots, k \rangle$ for all literal indices x.

We define our assignment function $\phi : \{0, \ldots, 3n-1\} \to \{T, F\}$ as follows: $\phi(x) = T$ if and only if $q_x = \langle k, 1, \ldots, k-1 \rangle$. To see that this assignment function is consistent, we recall that the $k-1$ copies of the pieces $H_0, \ldots, H_{3n-2}$ ensure that $q_{x_1} = q_{x_2}$ for all $x_1, x_2 \in \{0, \ldots, 3n-1\}$ with $\gamma(x_1) = \gamma(x_2)$. Hence, if $\gamma(x_1) = \gamma(x_2)$, then $\phi(x_1) = \phi(x_2)$.

It remains to show that our assignment function is satisfying: for all clauses $j \in \{0, \ldots, n-1\}$, $\{\phi(3j+0), \phi(3j+1), \phi(3j+2)\} = \{T, F\}$. Suppose, for the sake of contradiction, that there is some clause with $\phi(3j+0) = \phi(3j+1) = \phi(3j+2)$. We know that $p_x = \langle 1, \ldots, k \rangle$ for all $x \in \{3j+0, 3j+1, 3j+2\}$, so the color a_x already shows up on sides $\{1, \ldots, k-2\}$. Hence the configurations q_x and r_x must be in the set $\{\langle k, 1, \ldots, k-1 \rangle, \langle k-1, k, 1, \ldots, k-2 \rangle\}$. We also know that $q_x \neq r_x$ for all $x \in \{3j+0, 3j+1, 3j+2\}$.

If $\phi(3j+0) = \phi(3j+1) = \phi(3j+2) = T$, then $q_{3j+0} = q_{3j+1} = q_{3j+2} = \langle k, 1, \ldots, k-1 \rangle$, and $r_{3j+0} = r_{3j+1} = r_{3j+2} = \langle k-1, k, 1, \ldots, k-2 \rangle$. This means that the three pieces C_{3j+0}, C_{3j+1}, and C_{3j+2} map the three colors d_{3j+0}, d_{3j+1}, and d_{3j+2} to side 1. Hence, there is no way to configure D_j to avoid duplicating one of the colors on one of the sides. A similar argument shows that setting $\phi(3j+0) = \phi(3j+1) = \phi(3j+2) = F$ also leads to contradiction. Hence, the assignment is satisfying as well as consistent. □

4 Irregular Prism Pieces

In this section, we consider less regular pieces. Because the symmetries of the shape determine the number of possible configurations, puzzles with highly asymmetric shapes are easy to solve. For instance, if all edge lengths are distinct (as in the case of a generic k-gon), then there is exactly one possible configuration for each piece, making the puzzle trivial to solve. Nonetheless, there are a few shapes with interesting symmetries, some of which we discuss here.

The results in this section derive from the following theorem:

Theorem 5. *For any set of piece configurations P, if $|P| = 2$, then* Partial-Insanity(P) *can be solved in polynomial time.*

Proof. We may show this by a reduction to 2-Sat. Let p_1, p_2 be the two piece configurations in P. Suppose that we are given a set of pieces $A_1, \ldots, A_n$. Then we construct two variables for each piece A_i: a variable $x_{i,1}$ that is true if and only if A_i is in configuration p_1, and a variable $x_{i,2}$ that is true if and only if A_i is in configuration p_2. We additionally construct two clauses for each A_i: $(x_{i,1} \vee x_{i,2})$ and $(\neg x_{i,1} \vee \neg x_{i,2})$. This ensures that A_i must be placed in exactly one of the two configurations.

Next, we construct constraints to ensure that no color is used twice. For each pair of piece indices $1 \leq i_1 \neq i_2 \leq n$ and for each pair of (not necessarily distinct) configurations $j_1, j_2 \in \{1, 2\}$, we examine the k_2 visible sides. If there exists some $\ell \in \{1, \ldots, k_2\}$ such that $F_\ell(A_{i_1}, p_{j_1}) = F_\ell(A_{i_2}, p_{j_2})$, then we know

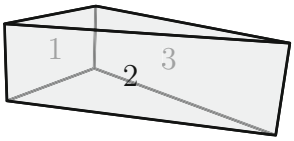


Fig. 7. Labels for the faces on a right prism with an isoceles triangle base

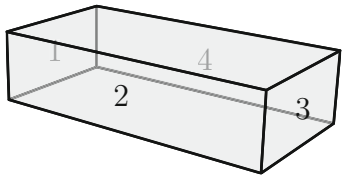


Fig. 8. Labels for each of the potentially visible faces on a box

that the pair of assigned configurations is invalid, and so we add a constraint $(\neg x_{i_1,j_1} \vee \neg x_{i_2,j_2})$ to prevent the reuse of colors on a single side.

Because these constraints completely encapsulate the requirements of the Partial-Insanity(P) problem, we may use a standard 2-Sat algorithm to find a satisfying assignment [2,6], then use the satisfying assignment to determine how we configure the pieces. □

For example, we can make use of Theorem 5 to help analyze the complexity of solving a puzzle with the following shape:

Definition 8. *Suppose that the piece shape is a right prism with an isoceles triangle base. If we number the sides of the prism in order around the base, as depicted in Fig. 7, then the set of configurations is* $\mathcal{T} = \{\langle 1,2,3\rangle, \langle 1,3,2\rangle\}$.

Because $|\mathcal{T}| = 2$, we can show the following:

Theorem 6. *The problem* Partial-Insanity($\mathcal{T}$) *can be solved in polynomial time.*

We can also use Theorem 5 to help us analyze more complex shapes:

Definition 9. *Suppose that the piece shape is a* box *with dimensions* $w \times h \times d$*, where the goal is to stack pieces into a tower with base* $w \times h$*. If we number the sides of the prism in order around the base, as depicted in Fig. 8, then the set of configurations is* $\mathcal{B} = \{\langle 1,2,3,4\rangle, \langle 3,4,1,2\rangle, \langle 1,4,3,2\rangle, \langle 3,2,1,4\rangle\}$.

Theorem 7. *The problem* Partial-Insanity($\mathcal{B}$) *can be solved in polynomial time.*

Proof. The set of configurations $\mathcal{B}$ is generated by allowing side 1 to swap freely with side 3, and allowing side 2 to swap freely with side 4. Hence, we can decompose the Partial-Insanity($\mathcal{B}$) problem into two instances of the Partial-Insanity($\mathcal{P}_2$) problem: one for sides 1 and 3, and one for sides 2 and 4. If we solve both halves of the problem with the technique of Theorem 5, it is straightforward to combine the solutions to compute the configuration of each piece. □

5 Conclusion

In this paper, we have examined several variants of Instant Insanity, exploring how the complexity changes as the geometry of the pieces changes. We have also

explored how restricting the motion of the pieces can change the complexity. In particular, we have analyzed several types of triangular prism puzzles and rectangular prism puzzles, discovering which variants are NP-complete and which can be solved in polynomial time.

Our results leave open a few problems. In particular, the complexity of PARTIAL-INSANITY($\mathcal{R}_k$) for $k \geq 5$ is still unknown. Because the PARTIAL-INSANITY($\mathcal{R}_4$) problem is NP-complete, it is likely (but not yet proven) that the PARTIAL-INSANITY($\mathcal{R}_k$) problem is NP-complete for $k \geq 5$ as well.

References

1. Arlinghaus, W.C.: The tantalizing four cubes. In: Michaels, J.G., Rosen, K.H. (eds.) Applications of Discrete Mathematics, ch. 16. McGraw-Hill Higher Education (1991)
2. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters 8(3), 121–123 (1979)
3. Berkove, E., Van Sickle, J., Hummon, B., Kogut, J.: An analysis of the (colored cubes)^3 puzzle. Discrete Mathematics 308(7), 1033–1045 (2008)
4. Chartrand, G.: Introductory Graph Theory, pp. 125–132. Courier Dover Publications (1985); Originally published in 1977 as Graphs as Mathematical Models
5. Eppstein, D.: Computational complexity of games and puzzles (April 2013), `http://www.ics.uci.edu/~eppstein/cgt/hard.html`
6. Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. SIAM Journal on Computing 5(4), 691–703 (1976)
7. Hall, P.: On representatives of subsets. Journal of the London Mathematical Society s1-10(1), 26–30 (1935)
8. Hearn, R.A., Demaine, E.D.: Games, Puzzles, and Computation. A. K. Peters (July 2009)
9. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing 2(4), 225–231 (1973)
10. Jebasingh, A., Simoson, A.: Platonic solid insanity. In: Congressus Numerantium, vol. 154, pp. 101–112 (2002)
11. Kendall, G., Parkes, A., Spoerer, K.: A survey of NP-complete puzzles. ICGA Journal 31(1), 13–34 (2008)
12. O'Beirne, T.H.: Puzzles and paradoxes. New Scientist 247, 358–359 (1961)
13. United States Patent and Trademark Office. Trademark Electronic Search System (TESS) (April 2013), `http://tmsearch.uspto.gov/`
14. Robertson, E., Munro, I.: NP-completeness, puzzles and games. Utilitas Mathematica 13, 99–116 (1978)
15. Schaefer, T.J.: The complexity of satisfiability problems. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, vol. 14, pp. 216–226 (1978)
16. van Lint, J.H., Wilson, R.M.: A Course in Combinatorics. Cambridge University Press (2001)

A Simple Linear-Space Data Structure for Constant-Time Range Minimum Query*

Stephane Durocher

University of Manitoba, Winnipeg, Canada
durocher@cs.umanitoba.ca

Abstract. We revisit the range minimum query problem and present a new $O(n)$-space data structure that supports queries in $O(1)$ time. Although previous data structures exist whose asymptotic bounds match ours, our goal is to introduce a new solution that is simple, intuitive, and practical without increasing asymptotic costs for query time or space.

1 Introduction

1.1 Motivation

Along with the mean, median, and mode of a multiset, the minimum (equivalently, the maximum) is a fundamental statistic of data analysis for which efficient computation is necessary. Given a list $A[0 : n-1]$ of n items drawn from a totally ordered set, a *range minimum query (RMQ)* consists of an input pair of indices (i, j) for which the minimum element of $A[i : j]$ must be returned. The objective is to preprocess A to construct a data structure that supports efficient response to one or more subsequent range minimum queries, where the corresponding input parameters (i, j) are provided at query time.

Although the complete set of possible queries can be precomputed and stored using $\Theta(n^2)$ space, practical data structures require less storage while still enabling efficient response time. For all i, if $i = j$, then a range query must report $A[i]$. Consequently, any range query data structure for a list of n items requires $\Omega(n)$ storage space in the worst case [7]. This leads to a natural question: how quickly can an $O(n)$-space data structure answer a range minimum query?

Previous $O(n)$-space data structures exist that provide $O(1)$-time RMQ (e.g., [4–6, 14, 18], see Section 2). These solutions typically require a transformation or invoke a property that enables the volume of stored precomputed data to be reduced while allowing constant-time access and RMQ computation. Each such solution is a conceptual organization of the data into a compact table for efficient reference; essentially, the algorithm reduces to a clever table lookup. In this paper our objective is not to minimize the total number of bits occupied by the data structure (our solution is not succinct) but rather to present a simple and more intuitive method for organizing the precomputed data to support

* Work supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 48–60, 2013.

RMQ efficiently. Our solution combines new ideas with techniques from various previous data structures: van Emde Boas trees [16], resizable arrays [10], range mode query [11, 12, 23], one-sided RMQ [4], and a linear-space data structure that supports RMQ in $O(\sqrt{n})$ time. The resulting RMQ data structure stores efficient representations of the data to permit direct lookup without requiring the indirect techniques employed by previous solutions (e.g., [1, 4–6, 18, 22, 27]) such as transformation to a lowest common ancestor query, Cartesian trees, Eulerian tours, or the Four Russians speedup. The data structure's RMQ algorithm is astonishingly simple: it can be implemented as a single if statement with four branches, each of which returns the minimum of at most three values retrieved from precomputed tables (see the pseudocode for Algorithm 2 in Section 3.3).

The RMQ problem is sometimes defined such that a query returns only the index of the minimum element instead of the minimum element itself. In particular, this is the case for succinct data structures that support $O(1)$-time RMQ using only $O(n)$ bits of space [14, 19, 20, 26] (see Section 2). In order to return the actual minimum element, say $A[i]$, in addition to its index i, any such data structure must also store the values from the input array A, corresponding to a lower bound of $\Omega(n \log u)$ bits of space in the worst case when element are drawn from a universe of size u or, equivalently, $\Omega(n)$ *words* of space (this lower bound also applies to other array range query problems [7]). Therefore, a range query data structure that uses $o(n)$ words of space requires storing the input array A separately, resulting in total space usage of $\Theta(n)$ words of space in the worst case. In this paper we require that a RMQ return the minimum element. Our RMQ data structure stores all values of A internally and matches the optimal asymptotic bounds of $O(n)$ words of space and $O(1)$ query time.

1.2 Definitions and Model of Computation

We assume the RAM word model of computation with word size $\Theta(\log u)$, where elements are drawn from a universe $U = \{-u, \ldots, u-1\}$ for some fixed positive integer $u > n$. Unless specified otherwise, memory requirements are expressed in word-sized units. We assume the usual set of $O(1)$-time primitive operations: basic integer arithmetic (addition, subtraction, multiplication, division, and modulo), bitwise logic, and bit shifts. We do not assume $O(1)$-time exponentiation nor, consequently, radicals. When the base operand is a power of two and the result is an integer, however, these operations can be computed using a bitwise left or right shifts. All arithmetic computations are on integers in U, and integer division is assumed to return the floor of the quotient. Finally, our data structure only requires finding the binary logarithm of integers in the range $\{0, \ldots, n\}$. Consequently, the complete set of values can be precomputed and stored in a table of size $O(n)$ to provide $O(1)$-time reference for the log (and log log) operations at query time, regardless of whether logarithm computation is included in the RAM model's set of primitive operations.

A common technique used in array range searching data structures (e.g., [4, 11,23]) is to partition the input array $A[0 : n-1]$ into a sequence of $\lceil n/b \rceil$ blocks, each of size b (except possibly for the last block whose size is $[(n-1) \bmod b]+1$).

A query range $A[i:j]$ spans between 0 and $\lceil n/b \rceil$ complete blocks. We refer to the sequence of complete blocks contained within $A[i:j]$ as the *span*, to the elements of $A[i:j]$ that precede the span as the *prefix*, and to the elements of $A[i:j]$ that succeed the span as the *suffix*. See Figure 1. One or more of the prefix, span, and suffix may be empty. When the span is empty, the prefix and suffix can lie either in adjacent blocks, or in the same block; in the latter case the prefix and suffix are equal.

We summarize the asymptotic resource requirements of a given RMQ data structure by the ordered pair $\langle f(n), g(n) \rangle$, where $f(n)$ denotes the storage space it requires measured in words and $g(n)$ denotes its worst-case RMQ time for an array of size n. Our discussion focuses primarily on these two measures of efficiency; other measures of interest include the preprocessing time and the update time. Note that similar notation is sometimes used to pair precomputation time and query time (e.g., [4, 18]).

2 Related Work

Multiple $\langle \omega(n), O(1) \rangle$ solutions are known, including precomputing RMQs for all query ranges in $\langle O(n^2), O(1) \rangle$, and precomputing RMQs for all ranges of length 2^k for some $k \in \mathbb{Z}^+$ in $\langle O(n \log n), O(1) \rangle$ (Sparse Table Algorithm) [4,18]. In the latter case, a query is decomposed into two (possibly overlapping) precomputed queries. Similarly, $\langle O(n), \omega(1) \rangle$ solutions exist, including the $\langle O(n), O(\sqrt{n}) \rangle$ data structure described in Section 3.1, and a tournament tree which provides an $\langle O(n), O(\log n) \rangle$ solution. This latter data structure (known in RMQ folklore, e.g., [25]) consists of a binary tree that recursively partitions the array A such that successive array elements are stored in order as leaf nodes, and each internal node stores the minimum element in the subarray of A stored in leaves below it. Given an arbitrary pair of array indices (i, j), a RMQ is processed by traversing the path from i to j in the tree and returning the minimum value stored at children of nodes on the path corresponding to subarrays contained in $A[i:j]$.

Several $\langle O(n), O(1) \rangle$ RMQ data structures exist, many of which depend on the equivalence between the RMQ and lowest common ancestor (LCA) problems. Harel and Tarjan [22] gave the first $\langle O(n), O(1) \rangle$ solution to LCA. Their solution was simplified by Schieber and Vishkin [27]. Berkman and Vishkin [6] showed how to solve the LCA problem in $\langle O(n), O(1) \rangle$ by transformation to RMQ using an Euler tour. This method was simplified by Bender and Farach-Colton [4] to give an ingenious solution which we briefly describe below. Comprehensive overviews of previous solutions are given by Davoodi [13] and Fischer [17].

The array $A[0 : n-1]$ can be transformed into a Cartesian tree $\mathcal{C}(A)$ on n nodes such that a RMQ on $A[i:j]$ corresponds to the LCA of the respective nodes associated with i and j in $\mathcal{C}(A)$. When each node in $\mathcal{C}(A)$ is labelled by its depth, an Eulerian tour on $\mathcal{C}(A)$ (i.e., the depth-first traversal sequence on $\mathcal{C}(A)$) gives an array $B[0 : 2n-2]$ for which any two adjacent values differ by ± 1. Thus, a LCA query on $\mathcal{C}(A)$ corresponds to a ± 1-RMQ on B. Array B is partitioned into blocks of size $(\log n)/2$. Separate data structures are constructed

to answer queries that are contained within a single block of B and those that span multiple blocks, respectively. In the former case, the ± 1 property implies that the number of unique blocks in B is $O(\sqrt{n})$; all $O(\sqrt{n}\log^2 n)$ possible RMQs on blocks of B are precomputed (the Four Russians technique [3]). In the latter case, a query can be decomposed into a prefix, span, and suffix (see Section 1.2). RMQs on the prefix and suffix are contained within respective single blocks, each of which can be answered in $O(1)$ time as in the former case. The span covers zero or more blocks. The minimum of each block of B is precomputed and stored in $A'[0 : 2n/\log n - 1]$. A RMQ on A' (the minimum value in the span) can be found in $\langle O(n), O(1)\rangle$ using the $\langle O(n'\log n'), O(1)\rangle$ data structure mentioned above due to the shorter length of A' (i.e., $n' = 2n/\log n$).

Fischer and Heun [18] use similar ideas to give a $\langle O(n), O(1)\rangle$ solution to RMQ that applies the Four Russians technique to any array (i.e., it does not require the ± 1 property) on blocks of length $\Theta(\log n)$. Yuan and Atallah [29] examine RMQ on multidimensional arrays and give a new one-dimensional $\langle O(n), O(1)\rangle$ solution that uses a hierarchical binary decomposition of $A[0 : n-1]$ into $\Theta(n)$ canonical intervals, each of length 2^k for some $k \in \mathbb{Z}^+$, and precomputed queries within blocks of length $\Theta(\log n)$ (similar to the Four Russians technique).

When only the minimum's index is required, Sadakane [26] gives a succinct data structure requiring $4n + o(n)$ bits that supports $O(1)$-time RMQ. Fischer and Heun [19, 20] and Davoodi et al. [14] reduce the space requirements to $2n + o(n)$ bits. Finally, the RMQ problem has been examined in the dynamic setting [8,13], in two and higher dimensions [2,9,15,21,26,29], and on trees and directed acyclic graphs [5,8,15].

Various array range query problems have been examined in addition to range minimum query. See the survey by Skala [28].

3 A New $\langle O(n), O(1)\rangle$ RMQ Data Structure

The new data structure is described in steps, starting with a previous $\langle O(n), O(\sqrt{n})\rangle$ data structure, extending it to $\langle O(n\log\log n), O(\log\log n)\rangle$ by applying the technique recursively, eliminating recursion to obtain $\langle O(n\log\log n), O(1)\rangle$, and finally reducing the space to $\langle O(n), O(1)\rangle$. To simplify the presentation, suppose initially that the input array A has size $n = 2^{2^k}$, for some $k \in \mathbb{Z}^+$; as described in Section 3.5, removing this constraint and generalizing to an arbitrary n is easily achieved without any asymptotic increase in time or space.

3.1 $\langle O(n), O(\sqrt{n})\rangle$ Data Structure

The following $\langle O(n), O(\sqrt{n})\rangle$ data structure is known in RMQ folklore (e.g., [25]) and has similar high-level structure to the ± 1-RMQ algorithm of Bender and Farach-Colton [4, Section 4]. While subobtimal and often overlooked in favour of more efficient solutions, this data structure forms the basis for our new $\langle O(n), O(1)\rangle$ data structure.

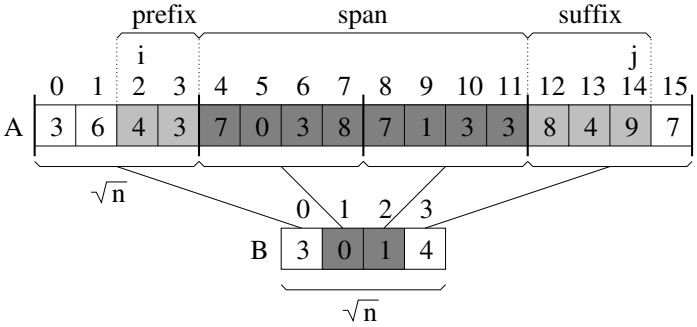


Fig. 1. A $\langle O(n), O(\sqrt{n})\rangle$ data structure: the array A is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in array B. A range minimum query $A[2:14]$ is processed by finding the minimum of the respective minima of the prefix $A[2:3]$, the span $A[4:11]$ (determined by examining $B[1:2]$), and the suffix $A[12:14]$. In this example this corresponds to $\min\{3,0,4\} = 0$.

The input array $A[0:n-1]$ is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in a table $B[0:\sqrt{n}-1]$. See Figure 1. A query range spans between zero and $\sqrt{n}$ complete blocks. The minimum of the span is computed by iteratively scanning the corresponding values in B. Similarly, the respective minima of the prefix and suffix are computed by iteratively examining their elements. The range minimum corresponds to the minimum of these three values. Since the prefix, suffix, and array B each contain at most $\sqrt{n}$ elements, the worst-case query time is $\Theta(\sqrt{n})$. The total space required by the data structure is $\Theta(n)$ (or, more precisely, $n + \Theta(\sqrt{n})$). Precomputation requires only a single pass over the input array in $\Theta(n)$ time. Updates (e.g., set $A[i] \leftarrow x$) require $\Theta(\sqrt{n})$ time in the worst case; whenever an array element equal to its block's minimum is increased, the block must be scanned to identify the new minimum.

3.2 $\langle O(n \log\log n), O(\log\log n)\rangle$ Data Structure

One-sided range minimum queries (where one endpoint of the query range coincides with one end of the array A) are trivially precomputed [4] and stored in arrays C and C', each of size n, where for each i,

$$C[i] = \begin{cases} \min\{A[i], C[i-1]\} & \text{if } i > 0, \\ A[0] & \text{if } i = 0, \end{cases}$$

$$\text{and} \quad C'[i] = \begin{cases} \min\{A[i], C'[i+1]\} & \text{if } i < n-1, \\ A[n-1] & \text{if } i = n-1. \end{cases} \tag{1}$$

Any subsequent one-sided RMQ on $A[0:j]$ or $A[j:n-1]$ can be answered in $O(1)$ time by referring to $C[j]$ or $C'[j]$.

The $\langle O(n), O(\sqrt{n})\rangle$ solution discussed in Section 3.1 includes three range minimum queries on subproblems of size $\sqrt{n}$, of which at most one is two-sided.

In particular, if the span is non-empty, then the query on array B is two-sided, and the queries on the prefix and suffix are one-sided. Similarly, if the query range is contained in a single block, then there is a single two-sided query and no one-sided queries. Finally, if the query range intersects exactly two blocks, then there are two one-sided queries (one each for the prefix and suffix) and no two-sided queries.

Thus, upon adding arrays C and C' to the data structure, at most one of the three (or fewer) subproblems requires $\omega(1)$ time to identify its range minimum. This search technique can be applied recursively on two-sided queries. By limiting the number of recursive calls to at most one and by reducing the problem size by an exponential factor of $1/2$ at each step of the recursion, the resulting query time is bounded by the following recurrence (similar to that achieved by van Emde Boas trees [16]):

$$T(n) \leq \begin{cases} T(\sqrt{n}) + O(1) & \text{if } n > 2, \\ O(1) & \text{if } n \leq 2 \end{cases}$$
$$\in O(\log \log n). \tag{2}$$

Each step invokes at most one recursive RMQ on a subarray of size $\sqrt{n}$. Each recursive call is one of two types: i) a recursive call on array B (a two-sided query to compute the range minimum of the span) or ii) a recursive call on the entire query range (contained within a single block).

Recursion can be avoided entirely for determining the minimum of the span (a recursive call of the first type). Since there are $\sqrt{n}$ blocks, $\binom{\sqrt{n}+1}{2} < n$ distinct spans are possible. As is done in the range mode query data structure of Krizanc et al. [23], the minimum of each span can be precomputed and stored in a table D of size n. Any subsequent RMQ on a span can be answered in $O(1)$ time by reference to table D. Consequently, tables C and D suffice, and table B can be eliminated.

The result is a hierarchical data structure containing $\log \log n + 1$ levels[1] which we number $0, \ldots, \log \log n$, where the xth level[2] is a sequence of $b_x(n) = n \cdot 2^{-2^x}$ blocks of size $s_x(n) = n/b_x(n) = 2^{2^x}$. See Table 1.

Table 1. The xth level is a sequence of $b_x(n)$ blocks of size $s_x(n)$

x	0	1	2	...	i	...	$\log \log n - 2$	$\log \log n - 1$	$\log \log n$
$b_x(n)$	$n/2$	$n/4$	$n/16$	...	$n2^{-2^i}$	...	$n^{3/4}$	$\sqrt{n}$	1
$s_x(n)$	2	4	16	...	2^{2^i}	...	$n^{1/4}$	$\sqrt{n}$	n

[1] Throughout this manuscript, $\log a$ denotes the binary logarithm $\log_2 a$.

[2] Level $\log \log n$ is included for completeness since we refer to the size of the parent of blocks on level x, for each $x \in \{0, \ldots, \log \log n - 1\}$. The only query that refers to level $\log \log n$ directly is the complete array: $i = 0$ and $j = n-1$. The global minimum for this singular case can be stored using $O(1)$ space and updated in $O(\sqrt{n})$ time as described in Section 3.1.

Generalizing (1), for each $x \in \{0, \ldots, \log\log n\}$ the new arrays C_x and C'_x are defined by

$$C_x[i] = \begin{cases} \min\{A[i], C_x[i-1]\} & \text{if } i \neq 0 \bmod s_x(n), \\ A[i] & \text{if } i = 0 \bmod s_x(n), \end{cases}$$

$$\text{and} \quad C'_x[i] = \begin{cases} \min\{A[i], C'_x[i+1]\} & \text{if } (i+1) \neq 0 \bmod s_x(n), \\ A[i] & \text{if } (i+1) = 0 \bmod s_x(n). \end{cases}$$

We refer to a sequence of blocks on level x that are contained in a common block on level $x+1$ as *siblings* and to the common block as their *parent*. Each block on level $x+1$ is a parent to $s_{x+1}(n)/s_x(n) = s_x(n)$ siblings on level x. Thus, any query range contained in some block at level $x+1$ covers at most $s_x(n)$ siblings at level x, resulting in $\Theta(s_x(n)^2) = \Theta(s_{x+1}(n))$ distinct possible spans within a block at level $x+1$ and $\Theta(s_{x+1}(n) \cdot b_{x+1}(n)) = \Theta(n)$ total distinct possible spans at level $x+1$, for any $x \in \{0, \ldots, \log\log n - 1\}$. These precomputed range minima are stored in table D, such that for every $x \in \{0, \ldots, \log\log n - 1\}$, every $b \in \{0, \ldots, b_{x+1}(n) - 1\}$, and every $\{i, j\} \subseteq \{0, \ldots, s_x(n) - 1\}$, $D_x[b][i][j]$ stores the minimum of the span $A[b \cdot s_{x+1}(n) + i \cdot s_x(n) : b \cdot s_{x+1}(n) + (j+1)s_x(n) - 1]$.

This gives the following recursive algorithm whose worst-case time is bounded by (2):

Algorithm 1

```
RMQ(i, j)
 1 if i = 0 and j = n − 1                 // query is entire array
 2    return minA                         // precomputed array minimum
 3 else
 4    return RMQ(log log n − 1, i, j)     // start recursion at top level

RMQ(x, i, j)
 1 if x > 0
 2    b_i ← ⌊i/s_x(n)⌋                    // blocks containing i and j
 3    b_j ← ⌊j/s_x(n)⌋
 4    if b_i = b_j                        // i and j in same block at level x
 5       return RMQ(x − 1, i, j)          // two-sided recursive RMQ: T(√n) time
 6    else if b_j − b_i ≥ 2               // span is non-empty
 7       b ← i mod s_{x+1}(n)
 8       return min{C'_x[i], C_x[j], D_x[b][b_i + 1][b_j − 1]}
          // 2 one-sided RMQs + precomputed span: O(1) time
 9    else
10       return min{C'_x[i], C_x[j]}      // 2 one-sided RMQs: O(1) time
11 else
12    return min{A[i], A[j]}              // base case (block size ≤ 2): O(1) time
```

The space required by array D_x for each level $x < \log\log n$ is

$$O\left(s_x(n)^2 \cdot b_{x+1}(n)\right) = O\left(s_{x+1}(n) \cdot b_{x+1}(n)\right) = O(n).$$

Since arrays C_x and C'_x also require $O(n)$ space at each level, the total space required is $O(n)$ per level, resulting in $O(n \log\log n)$ total space for the complete data structure.

For each level $x < \log\log n$, precomputing arrays C_x, C'_x, and D_x is easily achieved in $O(n \cdot s_x(n)) = O(n \cdot 2^{2^x})$ time per level, or $O(n^{3/2})$ total time. Each update requires $O(s_x(n))$ time per level, or $O(\sqrt{n})$ total time per update.

3.3 $\langle O(n \log\log n), O(1)\rangle$ Data Structure

Each step of Algorithm 1 described in Section 3.2 invokes at most one recursive call on a subarray whose size decreases exponentially at each step. Specifically, the only case requiring $\omega(1)$ time occurs when the query range is contained within a single block of the current level. In this case, no actual computation or table lookup occurs locally; instead, the result of the recursive call is returned directly (see Line 5 of Algorithm 1). As such, the recursion can be eliminated by jumping directly to the level of the data structure at which the recursion terminates, that is, the highest level for which the query range is not contained in a single block. Any such query can be answered in $O(1)$ time using a combination of at most three references to arrays C and D (see Lines 8 and 10 of Algorithm 1). We refer to the corresponding level of the data structure as the *query level*, whose index we denote by ℓ.

More precisely, Algorithm 1 makes a recursive call whenever $b_i = b_j$, where b_i and b_j denote the respective indices of the blocks containing i and j in the current level (see Line 5 of Algorithm 1). Thus, we seek to identify the highest level for which $b_i \neq b_j$. In fact, it suffices to identify the highest level $\ell \in \{0, \ldots, \log\log n - 1\}$ for which no query of size $j - i + 1$ can be contained within a single block. While the query could span the boundary of (at most) two adjacent blocks at higher levels, it must span at least two blocks at all levels less than or equal to ℓ. In other words, the size of the query range is bounded by

$$\begin{aligned}
& & s_\ell(n) < j - i + 1 &\leq s_{\ell+1}(n) \\
\Leftrightarrow & & 2^{2^\ell} < j - i + 1 &\leq 2^{2^{\ell+1}} \\
\Leftrightarrow & \log\log(j - i + 1) - 1 \leq & \ell &< \log\log(j - i + 1) \\
\Rightarrow & & \ell &= \lfloor \log\log(j - i) \rfloor.
\end{aligned}$$

As discussed in Section 1.2, since we only require finding binary logarithms of positive integers up to n, these values can be precomputed and stored in a table of size $O(n)$. Consequently, the value ℓ can be computed in $O(1)$ time at query time, where each logarithm is found by a table lookup.

This gives the following simple algorithm whose worst-case running time is constant (note the absence of loops and recursive calls):

Algorithm 2

RMQ(i, j)
1 **if** $i = 0$ **and** $j = n - 1$ // query is entire array
2 **return** $minA$ // precomputed array minimum
3 **else if** $j - i \geq 2$
4 $\ell \leftarrow \lfloor \log \log(j - i) \rfloor$
5 $b_i \leftarrow \lfloor i / s_\ell(n) \rfloor$
6 $b_j \leftarrow \lfloor j / s_\ell(n) \rfloor$ // blocks containing i and j
7 **if** $b_j - b_i \geq 2$ // span is non-empty
8 $b \leftarrow i \bmod s_{\ell+1}(n)$
9 **return** $\min\{C'_\ell[i], C_\ell[j], D_\ell[b][b_i + 1][b_j - 1]\}$
// 2 one-sided RMQs + precomputed span: $O(1)$ time
10 **else**
11 **return** $\min\{C'_\ell[i], C_\ell[j]\}$ // 2 one-sided RMQs: $O(1)$ time
12 **else**
13 **return** $\min\{A[i], A[j]\}$ // query contains ≤ 2 elements

Although the query algorithm differs from Algorithm 1, the data structure remains unchanged except for the addition of precomputed values for logarithms which require $O(n)$ additional space total space. As such, the space remains $O(n \log \log n)$ while the query time is reduced to $O(1)$ in the worst case. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively.

3.4 $\langle O(n), O(1) \rangle$ Data Structure

The data structures described in Sections 3.2 and 3.3 store exact precomputed values in arrays C_x, C'_x, and D_x. That is, for each a and each x, $C_x[a]$ stores $A[b]$ for some b (similarly for C'_x and D_x). If the array A is accessible during a query, then it suffices to store the relative index $b - a$ instead of storing $A[b]$. Thus, $C_x[a]$ stores $b - a$ and the returned value is $A[C_x[a] + a] = A[(b - a) + a] = A[b]$. Since the range minimum is contained in the query range $A[i : j]$ we get that $\{a, b\} \subseteq \{i, \ldots, j\}$ and, therefore,

$$|b - a| \leq j - i + 1 \leq s_{\ell+1}(n).$$

Consequently, for each level x, $\log(s_{x+1}(n)) = 2^{x+1}$ bits suffice to encode any value stored in C_x, C'_x, or D_x. Therefore, for each level x, each table C_x, C'_x, and D_x can be stored using $O(n \cdot 2^{x+1})$ bits. Observe that

$$\sum_{x=0}^{\log \log n - 1} n \cdot 2^{x+1} < 2n \log n < 2n \log u, \tag{3}$$

where $\log u$ denotes the word size under the RAM model. Therefore, the total space occupied by the tables C_x, C'_x, and D_x can be compacted into $O(n \log u)$ bits or, equivalently, $O(n)$ words of space. We now describe how to store this

compact representation to enable efficient access. For each $i \in \{0, \ldots, n-1\}$, the values $C_0[i], \ldots, C_{\log\log n-1}[i]$ can be stored in two words by (3). Specifically, the first word stores $C_{\log\log n-1}[i]$ and for each $x \in \{0, \ldots, \log\log n-2\}$, bits $2^{x+1}-1$ through $2^{x+2}-2$ store the value $C_x[i]$. Thus, all values $C_0[i], \ldots, C_{\log\log n-2}[i]$ are stored using

$$\sum_{i=0}^{\log\log n-2} 2^{x+1} = \log n - 2 < \log u$$

bits, i.e., a single word. The value $C_x[i]$ can be retrieved using a bitwise left shift followed by a right shift or, alternatively, a bitwise logical AND with the corresponding mask sequence of consecutive 1 bits (all $O(\log\log n)$ such bit sequences can be precomputed). An analogous argument applies to the arrays C'_x and D, resulting in $O(n)$ space for the complete data structure.

To summarize, the query algorithm is unchanged from Algorithm 2 and the corresponding query time remains constant, but the data structure's required space is reduced to $O(n)$. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively. This gives the following lemma:

Lemma 1. *Given any $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$ and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

3.5 Generalizing to an Arbitrary Array Size n

To simplify the presentation in Sections 3.1 to 3.4 we assumed that the input array had size $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$. As we show in this section, generalizing the data structure to an arbitrary positive integer n while maintaining the same asymptotic bounds on space and time is straightforward.

Let m denote the largest value no larger than n for which Lemma 1 applies. That is,

$$\begin{aligned} & m = 2^{2^{\lfloor \log\log n \rfloor}} \\ \Rightarrow \quad & m \le n < m^2 \\ \Rightarrow \quad & n/m < \sqrt{n}. \end{aligned} \tag{4}$$

Define a new array $A'[0 : n'-1]$, where $n' = m\lceil n/m \rceil$, that corresponds to the array A padded with dummy data[3] to round up to the next multiple of m. Thus,

$$\forall i \in \{0, \ldots, n'-1\},\ A'[i] = \begin{cases} A[i] & \text{if } i < n \\ +\infty & \text{if } i \ge n. \end{cases}$$

Since $n' = 0 \bmod m$, partition array A' into a sequence of blocks of size m. The number of blocks in A' is $\lceil n/m \rceil < \lceil \sqrt{n} \rceil$.

[3] For implementation, it suffices to store $u-1$ (the largest value in the universe U) instead of $+\infty$ as the additional values.

By (4) and Lemma 1, for each block we can construct a data structure to support RMQ on that block in $O(1)$ time using $O(m)$ space per block. Therefore, the total space required by all blocks in A' is $O(\lceil n/m \rceil \cdot m) = O(n)$. Construct arrays C, C', and D as before on the top level of array A' using the blocks of size m. The arrays C and C' each require $O(n') = O(n)$ space. The array D requires $O(\lceil n/m \rceil^2) \subseteq O(n)$ space by (4). Therefore, the total space required by the complete data structure remains $O(n)$.

Each query is performed as in Algorithm 2, except that references to C, C', and D at the top level access the corresponding arrays (which are stored separately from C_x, C'_x, and D_x for the lower levels). Therefore, the query time is increased by a constant factor for the first step at the top level, and the total query time remains $O(1)$.

This gives the following theorem:

Theorem 1 (Main Result). *Given any $n \in \mathbb{Z}^+$, and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

4 Directions for Future Work

4.1 Succinctness

The data structure presented in this paper uses $O(n)$ words of space. It is not currently known whether its space can be reduced to $O(n)$ bits if a RMQ returns only the index of the minimum element. As suggested by Patrick Nicholson (personal communication, 2011), each array C_x and C'_x can be stored using binary rank and select data structures in $O(n)$ bits of space (e.g., [24]). That is, we can support references to C_x and C'_x in constant time using $O(n)$ bits of space per level or $O(n \log \log n)$ total bits. It is not known whether the remaining components of the data structure can be compressed similarly, or whether the space can be reduced further to $O(n)$ bits.

4.2 Higher Dimensions

As shown by Demaine et al. [15], RMQ data structures based on Cartesian trees cannot be generalized to two or higher dimensions. The data structure presented in this paper does not involve Cartesian trees. Although it is possible that some other constraint may preclude generalization to higher dimensions, this remains to be examined.

4.3 Dynamic Data

As described, our data structure structure requires $O(\sqrt{n})$ time per update (e.g., set $A[i] \leftarrow x$) in the worst case. It is not known whether the data structure can be modified to support efficient queries and updates without increasing space.

Acknowledgements. The author thanks Timothy Chan and Patrick Nicholson with whom this paper's results were discussed. The author also thanks the students of his senior undergraduate class in advanced data structures at the University of Manitoba; preparing lecture material on range searching in arrays inspired him to revisit solutions to the range minimum query problem. Finally, the author thanks an anonymous reviewer for helpful suggestions.

References

1. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest commmon ancestors: a survey and a new algorithms for a distributed environment. Theory of Computing Systems 37(3), 441–456 (2004)
2. Amir, A., Fischer, J., Lewenstein, M.: Two-dimensional range minimum queries. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 286–294. Springer, Heidelberg (2007)
3. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradžev, I.A.: On economical construction of the transitive closure of a directed graph. Soviet Mathematics—Doklady 11(5), 1209–1210 (1970)
4. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms 57(2), 75–94 (2005)
6. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structures. SIAM Journal on Computing 22(2), 221–242 (1993)
7. Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 377–388. Springer, Heidelberg (2005)
8. Brodal, G.S., Davoodi, P., Srinivasa Rao, S.: Path minima queries in dynamic weighted trees. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 290–301. Springer, Heidelberg (2011)
9. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica 63(4), 815–830 (2012)
10. Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.D.: Resizable arrays in optimal time and space. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 37–48. Springer, Heidelberg (1999)
11. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. In: Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS). Leibniz International Proceedings in Informatics, vol. 14, pp. 291–301 (2012)
12. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. Theory of Computing Systems (to appear, 2013)
13. Davoodi, P.: Data Structures: Range Queries and Space Efficiency. PhD thesis, Aarhus University (2011)
14. Davoodi, P., Raman, R., Satti, S.R.: Succinct representations of binary trees for range minimum queries. In: Gudmundsson, J., Mestre, J., Viglas, T. (eds.) COCOON 2012. LNCS, vol. 7434, pp. 396–407. Springer, Heidelberg (2012)

15. Demaine, E.D., Landau, G.M., Weimann, O.: On Cartesian trees and range minimum queries. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 341–353. Springer, Heidelberg (2009)
16. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Letters 6(3), 80–82 (1977)
17. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
18. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
19. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
20. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
21. Golin, M., Iacono, J., Krizanc, D., Raman, R., Rao, S.S.: Encoding 2D range maximum queries. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 180–189. Springer, Heidelberg (2011)
22. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
23. Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. Nordic Journal of Computing 12, 1–17 (2005)
24. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
25. Pasailă, D.: Range minimum query and lowest common ancestor, `http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor`
26. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5, 12–22 (2007)
27. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
28. Skala, M.: Array range queries. In: Brodnik, A., Lopez-Ortiz, A., Raman, V., Viola, A. (eds.) Munro Festschrift. LNCS, vol. 8066, pp. 337–354. Springer, Heidelberg (2013)
29. Yuan, H., Atallah, M.J.: Data structures for range minimum queries. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 150–160 (2010)

Closing a Long-Standing Complexity Gap for Selection: $V_3(42) = 50$

David Kirkpatrick

Department of Computer Science, University of British Columbia,
201-2366 Main Mall, Vancouver, B.C., V6T 1Z4, Canada
kirk@cs.ubc.ca

Abstract. The selection problem has a long history, with deep roots in concrete complexity theory, and important practical applications. Nevertheless, some very basic questions remain unresolved. Included among these is the exact specification of the worst-case complexity of selecting the third largest of a set of n elements, a problem (originally formulated in terms of tennis tournaments) that dates back to 1883.

Inspired, in part, by the contributions of J. Ian Munro, to the selection problem as well as many other problems in concrete complexity, we revisit a question concerning the complexity of selecting the third largest element. The question was raised, and only partially solved, in the author's Ph.D. thesis, at a time that marks the beginning of a long friendship with Ian, and research journeys with numerous parallel interests. In this paper, we settle one very specific instance of this question that is interesting, in part, because it constitutes (i) a new counterexample to a natural conjecture about the exact complexity of this problem, and (ii) what the author now believes is the only remaining counterexample.

The author hopes, in this modest way, to reflect his deep admiration for Ian's many contributions to the theory, practice and appreciation of algorithm design and analysis.

1 Introduction

The *selection problem*, determining the i-th largest, for specified i, of a collection of n elements drawn from some totally ordered universe, has a rich history that predates its seminal role in the popularization and development of concrete complexity theory. It captures the essence of tournaments whose objective is to correctly identify or rank the top competitors, and appeals to our everyday familiarity with the design (or mis-design) of such competitions. Within complexity theory, it has served as a benchmark that not only sheds light on the more specific problem of determining the maximum and the more general problem of sorting, but also plays a fundamental role, quite distinct from sorting, in the design of divide-and-conquer algorithms.

For more than three decades, the research of J. Ian Munro has advanced our understanding of a variety of aspects of the selection problem. Early work with M. Paterson [15] and D. Dobkin [5], concerning selection and sorting on

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 61–76, 2013.

space-constrained computation models, set the stage for the development of rich and well-motivated sub-disciplines of space efficient data structures and algorithms, including streaming algorithms. The complexity of the *median* problem, the special case where where $i = \lceil n/2 \rceil$, was the focus of joint work with P. V. Poblete [16]. Subsequent work, with W. Cunto [3] and M. Ray [4], altered the conventional focus to consider other practical considerations, like average case behaviour and the minimization of data movement. More recently, Munro and collaborators have achieved breakthroughs in our understanding of some generalizations of selection, including multiple selection [8] and the production of arbitrary partial orders [2].

Munro's research has served as a source of both insight and inspiration for the author over these same decades. This is a least part of the motivation for picking up the selection problem thread, set aside by the author a long time ago.

1.1 The Selection Problem

We denote by $V_i(n)$ the minimum, over all comparison-based algorithms (tournaments) $\mathcal{A}$, of number of comparisons required by $\mathcal{A}$ to identify the i-th largest element in a set of n distinct inputs, in *worst case* over all presentations (permutations) of that input set. Similarly, $W_i(n)$ denotes the minimum number of comparisons to solve the multiple selection variant: determine the j-th largest element, for all j, $1 \leq j \leq i$, collectively. As we shall see, the exact value of $V_i(n)$, when $i = 1, 2$, has been known for some time. Furthermore, much is known about the asymptotic behavior of $V_i(n)$, particularly for the special case of the median; Knuth [13] remains an excellent overview of work in the area. Our focus here will be on small values of i, in particular $i = 3$.

1.2 Background on Selecting the Third Largest

It would defy tradition not to mention the pioneering role of Charles Dodgson, better known as Lewis Carroll, whose 1883 essay in *St. James's Gazette* posed the question of the correct allocation of second and third prizes in tennis tournaments. Hugo Steinhaus is credited with formulating, as early as 1929, the complexity question that amounts to determining an exact description of $W_2(n)$ (which, it is easy to see, coincides with $V_2(n)$).

The bound $W_2(n) \leq n - 2 + \lceil \lg n \rceil$ was established by J. Schreier [17], in 1932, but more than three decades passed before S. S. Kislitsyn showed that this bound is tight [11]. Knuth [12] provided a simpler proof of Kislitsyn's lower bound, with the use of an "oracle" (or adversary) argument, a technique that has had a significant impact on a good deal of subsequent work in this and related areas.

Kislitsyn also generalized Schreier's upper bound by demonstrating that

$$W_i(n) \leq n - i + \sum_{t=0}^{i-2} \lceil \lg(n-t) \rceil.$$

This was subsequently refined, by A. Hadian and M. Sobel [7], to give

$$V_i(n) \leq n - i + (i-1)\lceil \lg(n-i+2) \rceil,$$

which is tight (like Kislitsyn's bound), when $i = 2$.

The remaining gap, in the case where $i = 3$, was narrowed by F. Yao [18], who presented an adversary strategy showing

$$V_3(n) \geq \begin{cases} n - 3 + 2\lceil \lg(n-1) \rceil , & n - 1 = 4 \cdot 2^k \\ n - 5 + 2\lceil \lg(n-1) \rceil , & 4 \cdot 2^k < n - 1 < 6 \cdot 2^k \\ n - 4 + 2\lceil \lg(n-1) \rceil , & 6 \cdot 2^k \leq n - 1 < 8 \cdot 2^k , \end{cases}$$

for $k \geq 0$. This left $V_3(n)$ specified to within two, for all values of n, and exactly, whenever $n = 4 \cdot 2^k + 1$.

In the author's 1974 Ph.D. thesis [9] (see also [10]), Yao's bound was strengthened and generalized to

$$V_i(n) \geq n + i - 3 + \sum_{j=0}^{i-2} \lceil \lg \frac{n-i+2}{i+j} \rceil.$$

In the special case where $i = 3$ this gives

$$V_3(n) \geq \begin{cases} n - 4 + 2\lceil \lg(n-1) \rceil , & 4 \cdot 2^k < n - 1 \leq 6 \cdot 2^k \\ n - 3 + 2\lceil \lg(n-1) \rceil , & 6 \cdot 2^k < n - 1 \leq 8 \cdot 2^k . \end{cases} \tag{1}$$

In addition, Hadian and Sobel's upper bound on $V_3(n)$ was lowered to

$$V_3(n) \leq \begin{cases} n - 4 + 2\lceil \lg(n-1) \rceil , & 4 \cdot 2^k < n - 1 \leq 5 \cdot 2^k \\ n - 3 + 2\lceil \lg(n-1) \rceil , & 5 \cdot 2^k < n - 1 \leq 8 \cdot 2^k \end{cases} \tag{2}$$

or, equivalently,

$$V_3(n) \leq n + 1 + \lceil \lg \frac{n-1}{4} \rceil + \lceil \lg \frac{n-1}{5} \rceil.$$

Together (1) and (2) determine $V_3(n)$ to within one, when $5 \cdot 2^k < n - 1 \leq 6 \cdot 2^k$, and exactly, otherwise.

The author went on to conjecture that the upper bound (2) is tight (and, in an Appendix of [9], outlined a, regrettably incomplete, proof that the conjecture holds for all $n \geq 50$).

M. Aigner [1], apparently unaware of [9], re-derived (2). He went on to provide a lengthy argument that (2) is, in fact, tight for all $n \geq 6$. Unfortunately, Aigner's argument too seems to fall short of a complete proof [14]; in particular, J. Eusterbrock [6], trying to re-establish Aigner's claim with a computer-assisted proof search, discovered a counterexample: $V_3(22) = 28$.

1.3 The New Result for $V_3(42)$

For obvious reasons, the author feels motivated to try and provide a complete and convincing proof, specifying $V_3(n)$ exactly, for all $n \geq 3$. This paper is a modest, but important step in that direction. It proves that $V_3(42) = 50$, providing a second, and the author believes last, counterexample to Aigner's claim (which would require $V_3(42) = 51$).

2 The Algorithm and Its Analysis

2.1 Representation and Realization of Posets

We follow the usual convention and represent the partial orders realized at intermediate steps in an algorithm as *Hasse diagram*s. These are directed acyclic graphs on the underlying set of elements, where an arrow is drawn from element u to element v if it has been determined that $u < v$ *by a direct comparison*; for ease of reading elements will be positioned in such a way that all arrows point upwards. (Note that our Hasse diagrams may include some edges that do not belong to the transitive reduction of the associated partial ordering, but for the sake of clarity, and convention, we include these in our Hasse diagrams.)

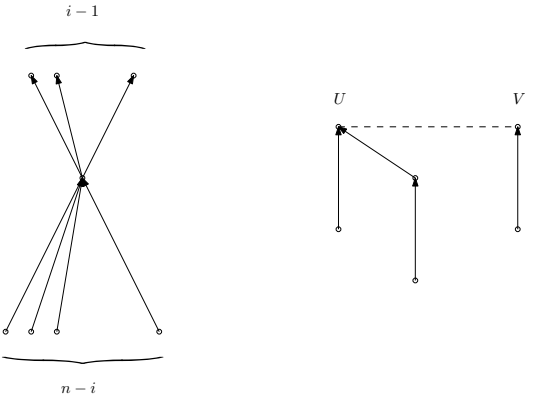


Fig. 1. (left) the partial order S_{n-i}^{i-1}, and (right) illustrating poset refinement

The problem of selecting the i-th largest element is equivalent to that of producing the partial order S_{n-i}^{i-1}, which has a central element with $i-1$ above (larger) and $n-i$ below (smaller) (see Fig. 1, left). We say that one (unlabeled) partial order P_1 *refines* another (unlabeled) partial order P_2 (denoted $P_1 \Rightarrow P_2$), if there is a labeling of P_1 and P_2 such that every linear extension of P_1 is a linear extension of P_2. We take advantage of the fact that an algorithm

that behaves correctly for inputs consistent with partial order P must (after suitable re-labeling) also behave correctly on refinements of P. So, for example, in considering the outcomes of a comparison between elements U and V in Fig. 1 (right), it will suffice to consider only the case where $U > V$, since the partial order that results in the case $V > U$ is a refinement of the partial order that results when $U > V$.

With only a slight abuse of our earlier notation, we can describe a generalization of the Schreier-Kislitsyn formula for $V_2(n)$. Specifically, let $V_2(S^0_{r_1}, S^0_{r_2}, \ldots, S^0_{r_t})$ denote the minimum number of comparisons to select the second largest element from a collection of elements for which a partial ordering $S^0_{r_1} \cup S^0_{r_2} \cup \ldots \cup S^0_{r_t}$ is given. Then

$$V_2(S^0_{r_1}, S^0_{r_2}, \ldots, S^0_{r_t}) = t - 2 + \lceil \lg(2^{r_1} + 2^{r_2} + \cdots + 2^{r_t}) \rceil. \tag{3}$$

The straightforward inductive proof of this result (attributed to R. W. Floyd), in fact a modest generalization that considers an arbitrary initial partial ordering, is an exercise (p. 219) in Knuth [12].

2.2 A Decision Tree Formulation of the Algorithm

Our algorithm has two phases. In the first phase, elements are partitioned into four sets, consisting of 32, 4, 4 and 2 elements respectively. Each set provides input to a balanced (single knockout) tournament on the elements of that set, for a total of $31 + 3 + 3 + 1 = 38$ comparisons. The (partially) labeled Hasse diagram describing the result of phase 1 is shown in Fig. 2.

The second phase involves a tournament (see Fig. 3) that refines the partial order produced in phase 1, in a way that suffices to identify one of the two largest elements in the set. To avoid unnecessary clutter the upper (resp., lower) branch following a comparison $E_1 : E_2$ corresponds consistently to the $>$ (resp., $<$)

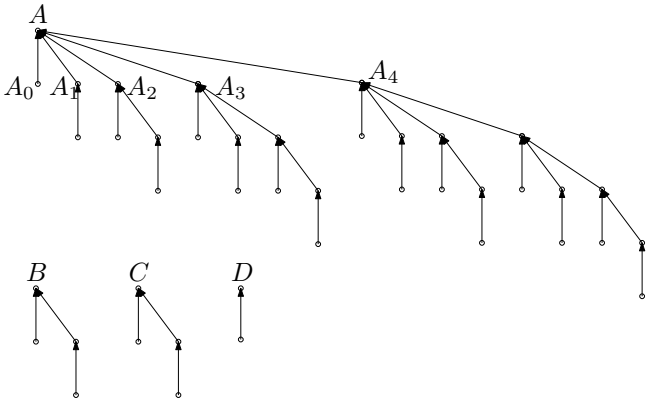


Fig. 2. Posets after phase 1

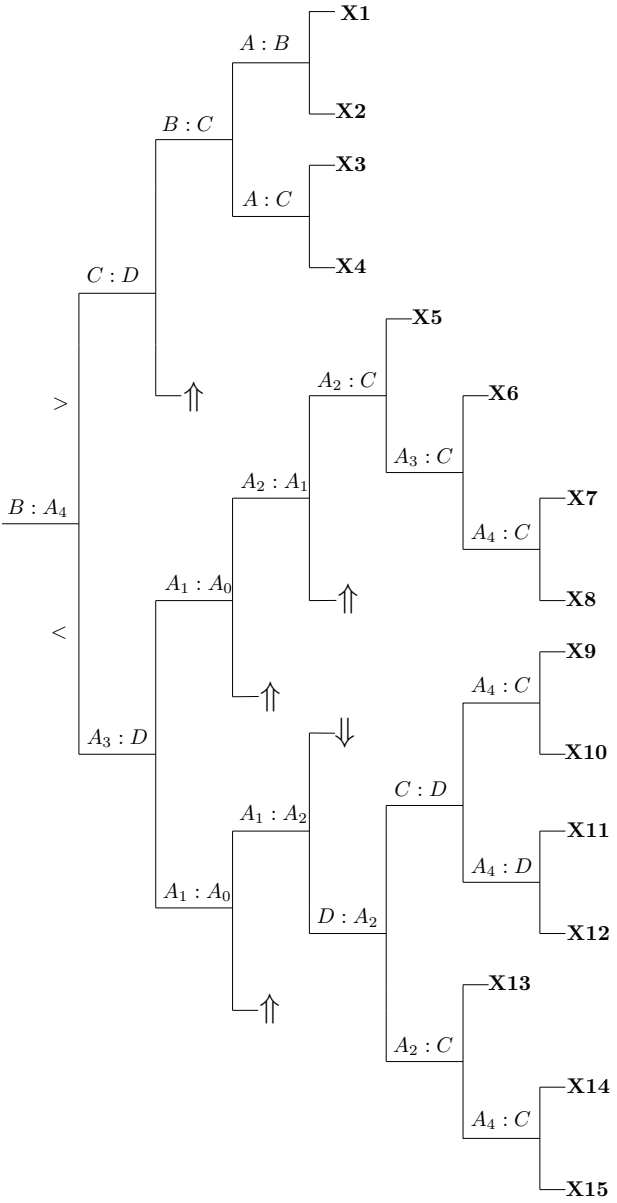


Fig. 3. Comparison tree in phase 2

outcome, i.e. the case where $E_1 > E_2$ (resp., $E_1 < E_2$). Note that some branches (marked by $\Uparrow$ or $\Downarrow$) are not described, because their associated poset is a refinement of that associated with their sibling branch (and hence can be treated in a completely symmetric fashion).

It remains to determine, at each of the fifteen leaves of this tournament, the second largest of the remaining elements. Figures 4 through 18 (see Appendix A) illustrate the Hasse diagrams associated with each of the fifteen leaves, with the results of phase 2 comparisons highlighted (in bold red). In each case, a parallel figure is drawn below that (i) highlights (by a circle) the element that has been identified as one of the two largest elements, and (ii) removes those elements that are known to be smaller than three or more other elements. What remains, in each case, is a collection of partial orders of the form S_j^0, so we can appeal directly to (3).

Table 1. Analysis of phase 2 leaf configurations

Phase 2 Cost	Subproblem	V_2 Equivalent	Completion Cost	Total
4	**X1**	$V_2(S_0, S_1, S_2, S_3, S_4)$	8	12
4	**X2**	$V_2(S_0, S_1, S_3, S_5)$	8	12
4	**X3**	$V_2(S_0, S_1, S_2, S_3, S_4)$	8	12
4	**X4**	$V_2(S_0, S_1, S_1, S_2, S_4)$	8	12
5	**X5**	$V_2(S_4, S_4, S_5)$	7	12
6	**X6**	$V_2(S_5, S_5)$	6	12
7	**X7**	$V_2(S_6)$	5	12
7	**X8**	$V_2(S_0, S_1, S_3)$	5	12
7	**X9**	$V_2(S_6)$	5	12
7	**X10**	$V_2(S_0, S_1, S_1, S_1)$	5	12
7	**X11**	$V_2(S_6)$	5	12
7	**X12**	$V_2(S_0, S_2, S_3)$	5	12
6	**X13**	$V_2(S_5, S_5)$	6	12
7	**X14**	$V_2(S_6)$	5	12
7	**X15**	$V_2(S_0, S_1, S_2)$	4	11

2.3 Analysis

We have reduced our problem to fifteen cases, the leaf configurations in Fig. 3. In each case, we have recorded in the Table 1 above (i) the phase 2 cost, the number of phase 2 comparisons leading to that leaf, (ii) the generalized V_2 problem that remains to be solved at that leaf, (iii) the completion cost, the number of comparisons sufficient to solve this generalized V_2 problem, and (iv) the total number of comparisons used (after the 38 in phase 1).

In summary, we have shown the following

Theorem 1. $V_3(42) = 50$.

3 Conclusion

This paper establishes the exact number of comparisons necessary and sufficient (in the worst case) to determine the third largest of a set of 42 elements. This is only the second known counterexample to a natural conjecture, made by the author almost 40 years ago, concerning the exact value of $V_3(n)$. The result in this paper was discovered in a renewed effort to finally establish this conjecture in a complete and convincing fashion.

Acknowledgements. The author acknowledges the generous support of the Natural Sciences and Engineering Research Council of Canada.

This also seems like an opportune time to express the author's deep appreciation for the wide-ranging contributions of Donald Knuth. His efforts not only introduced research on discrete algorithms to a broad audience, but have also served, for many decades, as a model of clarity and precision.

References

1. Aigner, M.: Selecting the top three elements. Discrete Applied Mathematics 4, 247–267 (1982)
2. Cardinal, J., Florini, S., Joret, G., Jungers, R.M., Munro, J.I.: An efficient algorithm for partial order production. SIAM J. on Computing 39, 2927–2940 (2010)
3. Cunto, W., Munro, J.I.: Average case selection. J. ACM 36, 270–279 (1989)
4. Cunto, W., Munro, J.I., Rey, M.: Selecting the median and two quartiles in a set of numbers. Software: Practice and Experience 22, 439–454 (1992)
5. Dobkin, D., Munro, J.I.: Optimal time minimal space selection algorithms. J. ACM 28, 454–461 (1981)
6. Eusterbrock, J.: Errata to "Selecting the top three elements" by M. Aigner: A result of a computer-assisted proof search. Discrete Applied Mathematics 41, 131–137 (1993)
7. Hadian, A., Sobel, M.: Selecting the i-th largest using binary errorless comparisons. Colloquia Mathematica Societatis Janos Bolyai 4, 585–599 (1969)
8. Kaligosi, K., Mehlhorn, K., Munro, J.I., Sanders, P.: Towards optimal multiple selection. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 103–114. Springer, Heidelberg (2005)
9. Kirkpatrick, D.G.: Topics in the complexity of combinatorial algorithms. Technical Report 74, Department of Computer Science, University of Toronto (1974)
10. Kirkpatrick, D.G.: A unified lower bound for selection and set partitioning problems. J. ACM 28, 150–165 (1981)
11. Kislitsyn, S.S.: On the selection of the k-th element of an ordered set by pairwise comparisons. Sibirsk. Mat. Zh. 5, 557–564 (1964) (in Russian)
12. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)
13. Knuth, D.E.: Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1998)
14. Knuth, D.E.: Private correspondence (July 1996)
15. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)

16. Munro, J.I., Poblete, P.V.: A lower bound for determining the median. Technical Research Report CS-82-21, University of Waterloo (1982)
17. Schreier, J.: On tournament elimination systems. Mathesis Polska 7, 154–160 (1932) (in Polish)
18. Yao, F.F.: On lower bounds for selection problems. Technical Report MAC TR-121, M. I. T. (1974)

Appendix A: The partial orders at X1...X15

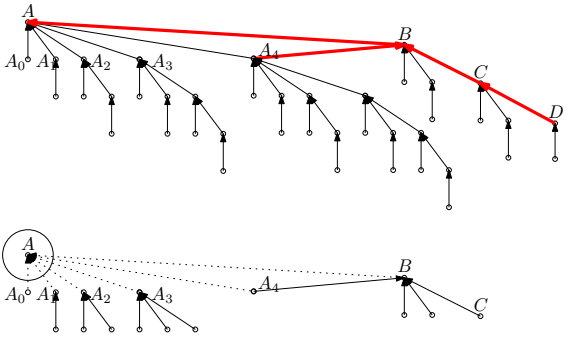


Fig. 4. The poset at **X1** (above) and its reduction (below)

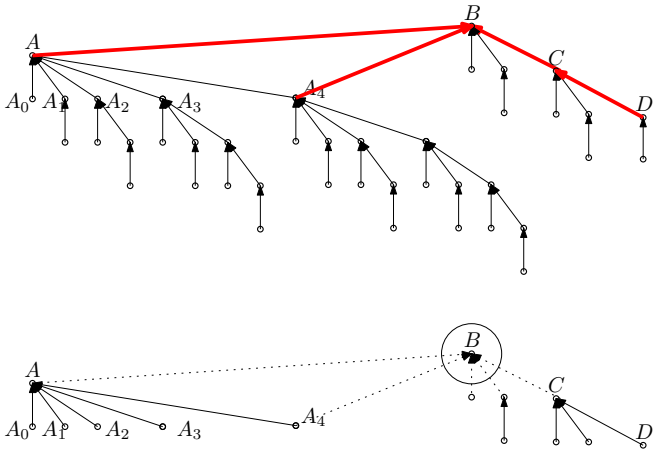


Fig. 5. The poset at **X2** (above) and its reduction (below)

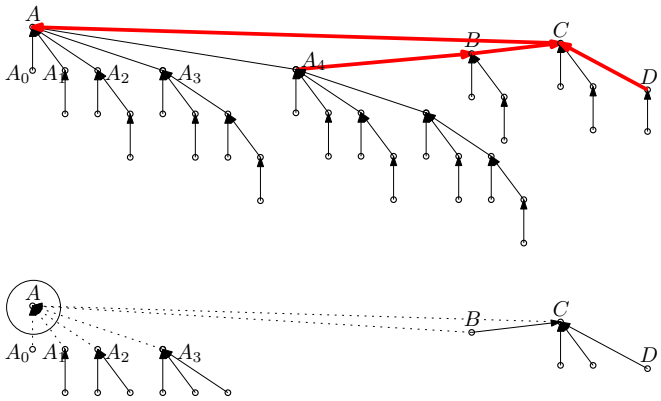


Fig. 6. The poset at **X3** (above) and its reduction (below)

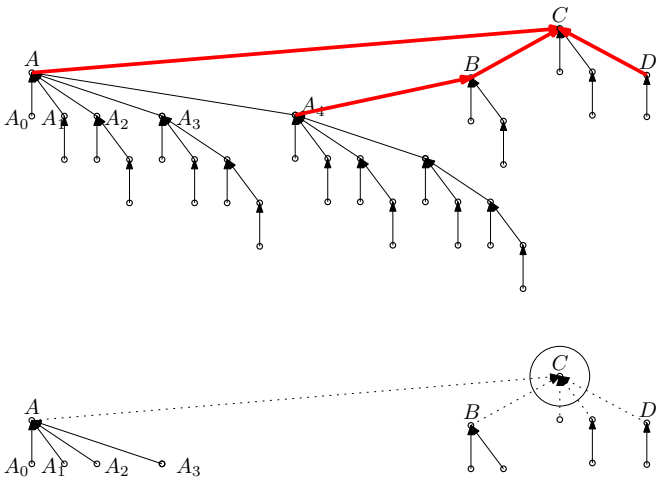


Fig. 7. The poset at **X4** (above) and its reduction (below)

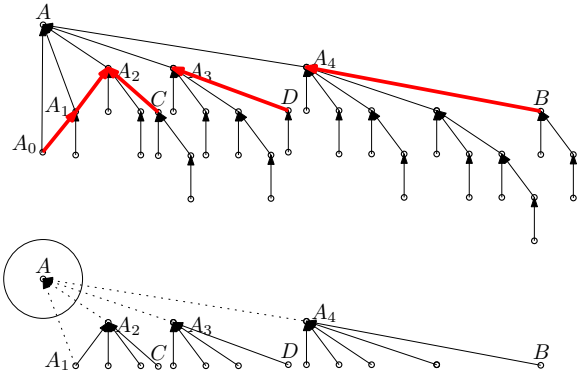


Fig. 8. The poset at **X5** (above) and its reduction (below)

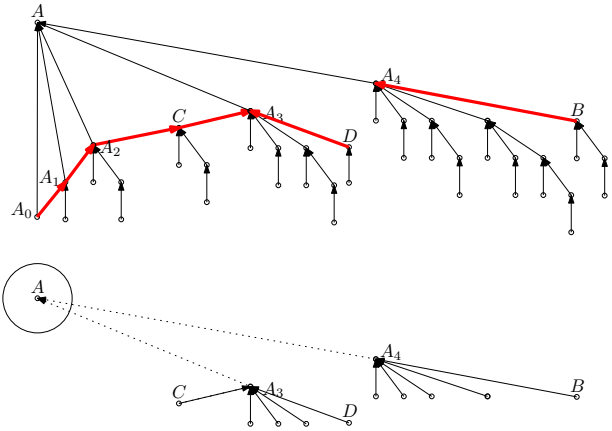


Fig. 9. The poset at **X6** (above) and its reduction (below)

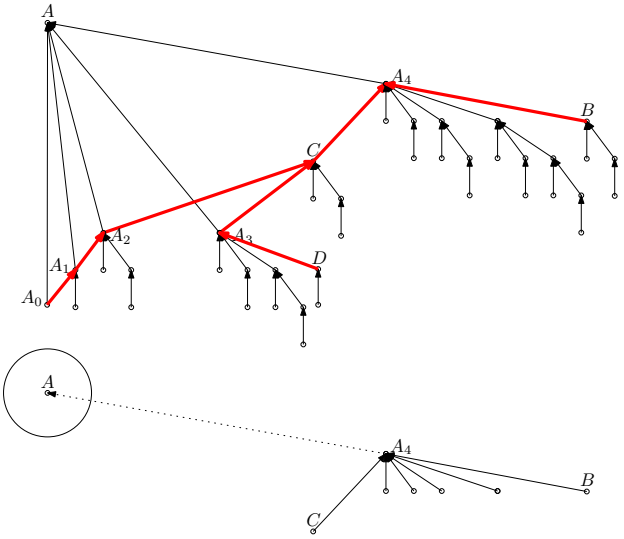


Fig. 10. The poset at **X7** (above) and its reduction (below)

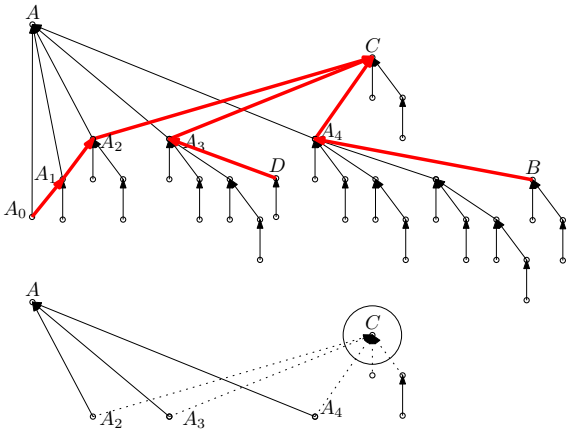


Fig. 11. The poset at **X8** (above) and its reduction (below)

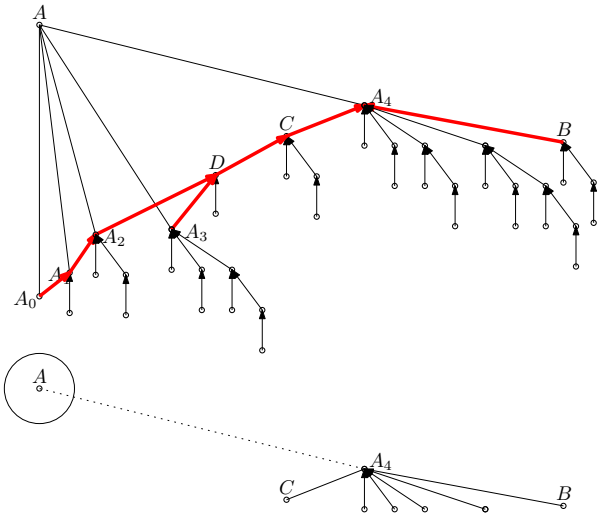


Fig. 12. The poset at **X9** (above) and its reduction (below)

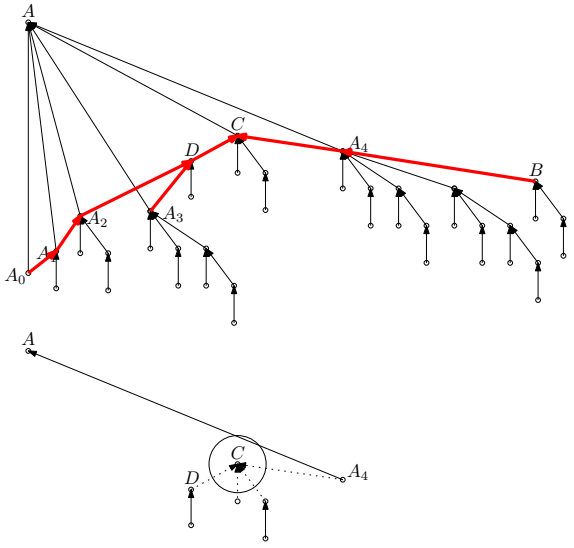


Fig. 13. The poset at **X10** (above) and its reduction (below)

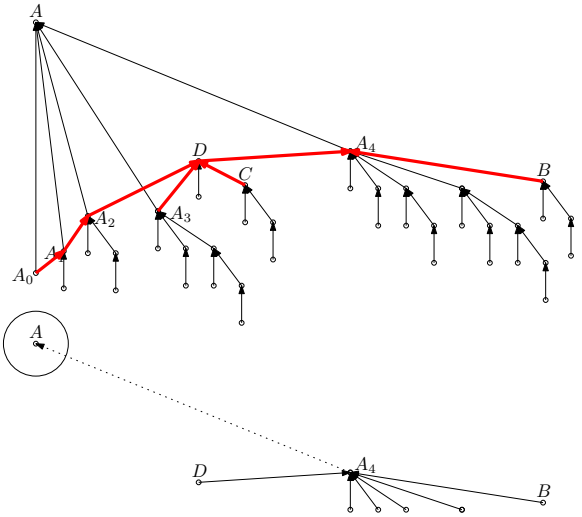


Fig. 14. The poset at **X11** (above) and its reduction (below)

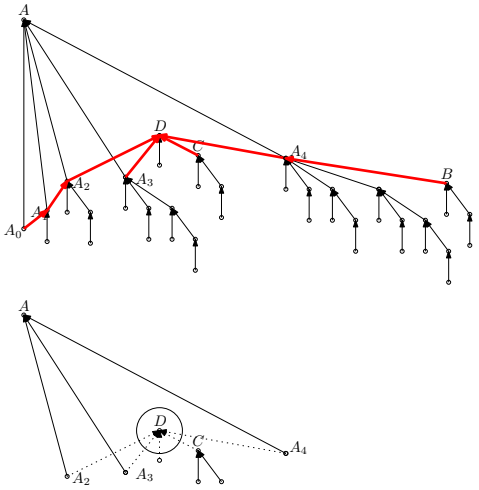


Fig. 15. The poset at **X12** (above) and its reduction (below)

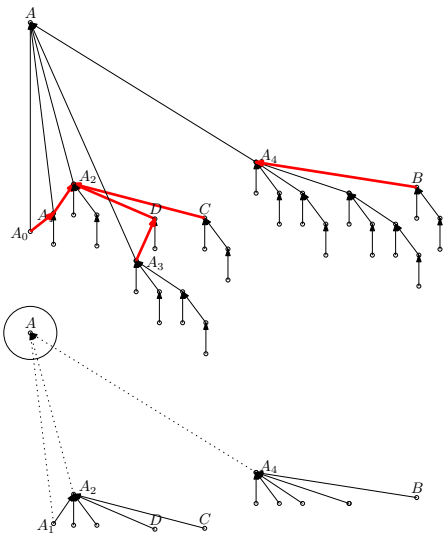


Fig. 16. The poset at **X13** (above) and its reduction (below)

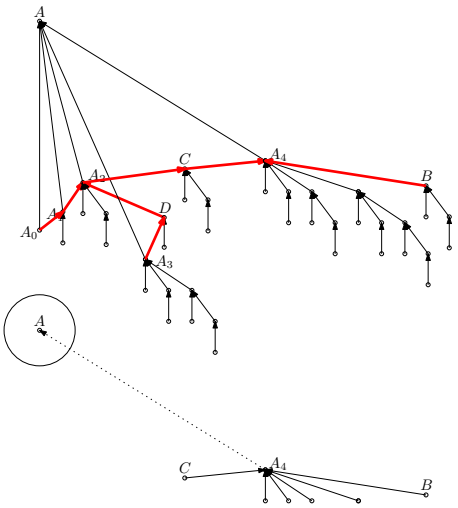


Fig. 17. The poset at **X14** (above) and its reduction (below)

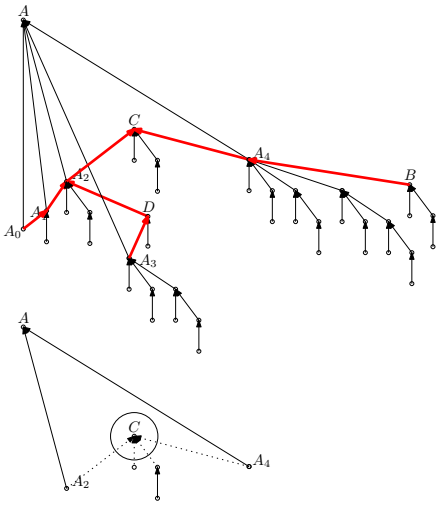


Fig. 18. The poset at **X15** (above) and its reduction (below)

Frugal Streaming for Estimating Quantiles

Qiang Ma[1], S. Muthukrishnan[1], and Mark Sandler[2]

[1] Rutgers University, Piscataway, NJ 08854, USA
{qma,muthu}@cs.rutgers.edu
[2] Google Inc. New York, NY 10011, USA
sandler@google.com

Abstract. Modern applications require processing streams of data for estimating statistical quantities such as quantiles with small amount of memory. In many such applications, in fact, one needs to compute such statistical quantities for each of a large number of groups (*e.g.,*network traffic grouped by source IP address), which additionally restricts the amount of memory available for the stream for any particular group. We address this challenge and introduce *frugal streaming*, that is algorithms that work with tiny – typically, sub-streaming – amount of memory per group.

We design a frugal algorithm that uses *only one* unit of memory per group to compute a quantile for each group. For stochastic streams where data items are drawn from a distribution independently, we analyze and show that the algorithm finds an approximation to the quantile rapidly and remains stably close to it. We also propose an extension of this algorithm that uses *two* units of memory per group. We show experiments with real world data from HTTP trace and Twitter that our frugal algorithms are comparable to existing streaming algorithms for estimating any quantile, but these existing algorithms use far more space per group and are unrealistic in frugal applications; further, the two memory frugal algorithm converges significantly faster than the one memory algorithm.

1 Introduction

Modern applications require processing streams of data for estimating statistical quantities such as quantiles with small amount of memory. A typical application is in IP packet analysis systems such as Gigascope [8] where an example of a query is to find the median packet (or flow) size for IP streams from some given IP addresses. Since IP addresses send millions of packets in reasonable time windows, it is prohibitive to store all packet or flow sizes and estimate the median size. Another application is in social networking sites such as Facebook or Twitter where there are rapid updates from users, and one is interested in median time between successive updates from a user. In yet another example, search engines can model their search traffic and for each search term, want to estimate the median time between successive instances of that search.

Motivated by applications such as these, there has been extensive work in the database community on theory and practice of approximately estimating quantiles of streams with limited memory [1–4, 6, 7, 9–11, 13, 14, 17]. Taken together, this body of research has generated methods for approximating quantiles to $1 + \epsilon$ approximation with space roughly $O(1/\epsilon)$ in various models of data streams.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 77–96, 2013.

Our work here begins with our experience that while the algorithms above are useful, in reality, they get used within GROUPBYs, that is, there are a large number of groups and each group defines a stream within which we need to compute quantiles. In example applications above, this is evident. In IP traffic analysis, one wishes to find median packet size from *each* of the source IP addresses, and therefore the number of "groups" is upto 2^{32} (or 2^{128}). Similarly, in social network application, we wish to compute the median time between updates for *each* user, and the number of users is in 100's of millions for Facebook or Twitter. Likewise, the number of "groups" of interest to search engines is in 100's of millions of search terms. Now, the bottleneck of high speed memory manifests in a different way. We can no longer allocate a lot of memory to any of the groups! In real systems such as Gigascope, low level aggregation engines keep in memory as many groups as they can and rely on higher level aggregation to aggregate partial answers from various groups, which ends up essentially forcing the higher level aggregator to work as a high speed streamer, and proves ineffective.

Motivated by this, we introduce the new direction of *frugal streaming*, that is streaming algorithms that work with tiny amount of memory per group, memory that is far less than is used by typical streaming algorithms. In fact, we will work with 1 or 2 memory locations per group. Our contributions are as follows.

- We present two frugal streaming algorithms for estimating a quantile of a stream. One uses 1 unit of memory for the data stream item, and the other uses 2 units of memory.
- For stochastic streams, that is streams where each item is drawn independently from a distribution, we can mathematically analyze and show how our algorithms converge rapidly to the desired quantile and how they stably oscillate around the quantile as stream progresses.
- We evaluate our algorithms on real datasets from Twitter. We show that our frugal streaming algorithms perform accurately and quickly. Regular streaming algorithms known previously either are highly inadequate given our memory constraints or need significantly more memory to be comparable in accuracy. Further, our frugal algorithms have an intriguing "memoryless" property. Say the stream abruptly changes and now represents a new distribution; *irrespective of the past*, at any given moment, our frugal algorithms move towards the median of the new distribution without waiting for the new streaming items to drown out the old median. We also experimentally evaluate the performance of our frugal streaming algorithms with changing streams.

Ian Munro and Mike Paterson [16], very early on, introduced and solved the problem of estimating quantiles in one or more passes with small memory. This influential paper was a prelude to the area of streaming that was to emerge 15+ years later. Our paper here is an homage to this classical paper.

In Section 2 we introduce definitions and notations. We present our 1 unit of memory frugal streaming algorithm in Section 3. It is analyzed for stochastic streams in Section 4 to give insights about its speed in approaching true quantile and its stability in the long run. Section 5 gives an extension to 2 units of memory frugal streaming algorithm. We discuss related algorithms and present our experimental study in Section 6 and 7. Section 8 has concluding remarks.

2 Background and Notations

Suppose values in domain D are integers[1] distributed over $\{1, 2, 3, \ldots, N\}$. Given a random variable X in domain D, denote its cumulative distribution function (CDF) as $F(x)$, and its quantile function as $Q(x)$. In other words, $F(Q(x)) = x$ if the CDF is strictly monotonic.

h-th p-quantile is x such that $Pr(X < x) = F(x) = \frac{h}{p}$. For convenience we use $\frac{h}{p}$-quantile for the hth p-quantile. S is a sampled set from D. Define a rank function that gives the number of items in S which are smaller than x, $R(x) = |S'|$ where $S' = \{s_i \in S, s_i < x\}$. So when size of S grows to infinity, $F(x) = \frac{R(x)}{|S|}$.

In this paper we consider rank p-quantiles, so the $\frac{h}{p}$-quantile approximation returned by algorithm is considered correct even if the approximation value has zero probability in domain D. For example, if D is distributed over two values 1 and 1000 with equal probabilities. Under value quantile evaluation, a median estimation at 1000 would be considered accurate (throughout our paper, upper median is used for even sample sizes). But any value between 1 and 1000 can also give us good estimation in terms of ranking, hence they are considered correct estimation under rank quantile evaluation.

Throughout when we refer to memory use of algorithms, each memory unit has sufficient bits to store the input domain, that is, each memory unit is $\log N$ bits. This is standard in data stream literature where a method uses f words, it is really f words each of which has sufficient bits to store the input, or $f \log N$ bits.

3 Frugal Streaming Algorithm

We start from median estimation problem and then generalize our algorithm to estimate any quantile of stream S.

3.1 1 Unit Memory Algorithm to Estimate Median

Our algorithm maintains only one unit of memory $\tilde{m}$ which contains its estimate for the stream median, m_S. When a new stream item s_i arrives, consider what our algorithm can do? Since it has no memory of the past beyond $\tilde{m}$, it can do very little. The algorithm adjusts its estimate so that the absolute difference with the new stream item is decreased. C-style pseudo code of this algorithm is described in Algorithm 1, $Frugal$-$1U$-$Median$.

Example 1. To illustrate how $Frugal$-$1U$-$Median$ works, let us consider the example in Figure 1. The estimated median from $Frugal$-$1U$-$Median$ algorithm starts from $\tilde{m}_0 = 0$, and gets updated on each arriving stream item. For example, when $s_4 = 5$ comes, it is larger than $\tilde{m}_3$ whose value is 1, therefore $\tilde{m}_4 = \tilde{m}_3 + 1 = 2$. In this example, $\tilde{m}$ starts from 0, and after reading 5 items from the stream it reaches the stream median for the first time.

[1] For domains with non-integer values, their values can be rewritten to keep desired precision and converted to integers.

Item index i =		1	2	3	4	5	6	7	8	... ...
Stream Items s_i		4	2	1	5	3	2	5	4	... ...
True Medians m_i		4	4	2	4	3	3	3	4	... ...
Median Estimates $\tilde{m}_i$	0	1	2	1	2	3	2	3	4	... ...

Fig. 1. Estimate stream median

Algorithm 1 $Frugal\text{-}1U\text{-}Median$

Input: Data stream S, 1 unit of memory $\tilde{m}$
Output: $\tilde{m}$

```
Initialization m̃ = 0
for each s_i in S do
    if s_i > m̃ then
        m̃ = m̃ + 1;
    else if s_i < m̃ then
        m̃ = m̃ − 1;
    end if
end for
```

3.2 1 Unit of Memory to Estimate Any Quantile

Following the same intuition as above, we can use 1 unit of memory to estimate any $\frac{h}{k}$-quantile, where $1 \leq h \leq k-1$. If the current stream item is larger than estimation, we need to increase estimation by 1; otherwise, we need to decrease estimation by 1. The trick to generalize median estimation to any $\frac{h}{k}$-quantile estimation is that not every stream item seen will cause an update. If the current stream item is larger than estimation, an increment update will be triggered only with probability $\frac{h}{k}$. The rationale behind it is that if we are estimating $\frac{h}{k}$-quantile, and if the current estimate is at stream's true $\frac{h}{k}$-quantile, we will expect to see stream items larger than the current estimate with probability $1-\frac{h}{k}$. If the probability of seeing larger stream items is greater than $1-\frac{h}{k}$, it is caused by the fact that the current estimate is smaller than stream's true $\frac{h}{k}$-quantile. Similarly, a smaller stream item will cause a decrement update only with probability $1-\frac{h}{k}$. Our general 1 unit of memory quantile estimation algorithm is described in Algorithm 2, $Frugal\text{-}1U$.

We need to make a few observations. Besides $\tilde{m}$, this algorithm uses $rand$ and $\frac{h}{k}$. Notice that we can implement the algorithm without explicitly storing $rand$ value, $\frac{h}{k}$ is a constant across all the groups, no matter how many, and can be kept in registers.

Update taken by $\tilde{m}$ in $Frugal\text{-}1U$ is 1, it is a small change at each step when the stream quantile to estimate is large. When it is allowed one extra unit of memory, we can use it to store the size of update to take, denoted as `step`. The extension to two units of memory algorithm is presented in Section 5.

Algorithm 2 $Frugal\text{-}1U$

Input: Data stream S, h, k, 1 unit of memory $\tilde{m}$
Output: $\tilde{m}$

```
Initialization m̃ = 0
for each s_i in S do
    rand = random(0,1); // get a random value in [0,1]
    if s_i > m̃ and rand > 1 − h/k then
        m̃ = m̃ + 1;
    else if s_i < m̃ and rand > h/k then
        m̃ = m̃ − 1;
    end if
end for
```

4 Analysis of $Frugal\text{-}1U\text{-}Median$

Our frugal algorithm for estimating a quantile can behave badly on certain streams. For example, if the true stream quantile value has high probability, even if current estimation is at the true stream quantile, an update of 1 to our estimation will cause large change in rank quantile error. Also any adversary that can remember the entire past and reorder the stream items, they can constantly mislead our algorithm by spreading out the median. This is expected because our algorithm has no memory of the past. The real intuition and strength of our algorithm comes from elsewhere. We say a stream is Stochastic if each stream item is drawn from some distribution D, randomly and independently from other stream items. We will analyze and show that for Stochastic streams, our algorithm quickly converges to an estimate of the target quantile, and further, stably remains in the neighborhood of the quantile as stream progresses.

4.1 Approaching Speed

For our 1 memory algorithm, each update size is 1. At any time t_i, our algorithm estimation has non-zero probabilities to move towards or away from true quantile. Therefore for sufficiently large t, the probability that algorithm estimation moves continuously in one direction is very low. When current algorithm estimation is far away from true quantile, the speed of approaching the true quantile is high, since every update is highly biased towards true quantile. But as the estimation gets closer to true quantile, the bias to move towards true quantile gets weaker so the speed of approaching the true quantile is low. In other words, we are likely to see algorithm estimation showing an oscillating trajectory towards true quantile. The analysis of our algorithm is non-trivial and challenging because the rate of the convergence to an estimate is not constant and depends on a number of varying factors. We rely on the concept of stochastic dominance and we show that in fact the algorithm will approach the true quantile with linear speed.

Recall our notations from Section 2, $F(t)$ is the CDF of distribution, $Q(x)$ is quantile. Let x_i be an indicator variable for the direction of i-th step of the algorithm, where $x_i = 1$ for increment and $x_i = -1$ for decrement. Let $\tilde{m}_t = \sum_{i=1}^{t} x_i$, in other words $\tilde{m}_t$ is the estimation of the quantile at time t. Let $|F(i) - F(i+1)| \leq \delta$, so δ is the

maximum single location probability in distribution and $0 \leq \delta < 1$. Suppose the algorithm is to estimate $\frac{h}{k}$ quantile, whose value is M. Assume the algorithm estimate starts from position $\tilde{m}_0$, where $\tilde{m}_0 < M$. The distance from start position to true quantile is $M - \tilde{m}_0$, but the analysis trivially generalizes to the case where the distance from start position to the true quantile is M.

Lemma 1. *For median estimation, assume the algorithm estimate starts from position $\tilde{m}_0$, where $F(\tilde{m}_0) < \frac{1}{2} - \delta$. After $T = \frac{M|\log 1/\varepsilon|}{\delta}$ steps of algorithm, the probability that $F(\tilde{m}_t) < \frac{1}{2} - \delta$ for all $t < T$ is at most ε. In other words, after $O(M)$ steps it is likely the algorithm has crossed vicinity of the true quantile, $\frac{1}{2} - \delta$, at least once.*

Proof. Let $M' = Q(\frac{1}{2} - \delta)$, we can compute the expectation of a move whenever the algorithm is below M'.

$$\mathbf{Pr}\left[x_i = 1\right] \geq \frac{1}{2}(1 - (\frac{1}{2} - \delta)) = \frac{1}{2} - \frac{1}{2^2} + \delta * \frac{1}{2}$$

we denote it by θ, then

$$\mathbf{Pr}\left[x_i = -1\right] \leq (1 - \frac{1}{2})(\frac{1}{2} - \delta) = \theta - \delta$$

Therefore we have

$$\mathbf{E}\left[x_i\right] \geq \delta \tag{1}$$

In other words the expected shift of each x_i before it hits M' is then at least δ. To prove our lemma, we therefore can use tail inequalities to bound the deviation of $\tilde{m}_t = \sum x_i$ from the expectation. The main difficulty, however arises from the fact x_i are not independent from each other and the constraint (1) holds only when $\tilde{m}_t \leq M'$. Consider an arbitrary sequence of moves x_i. Define $y_i = x_i$ for all $i < i_0$, where i_0 is the time where $\tilde{m}_{i_0}$ crossed M' for the first time, and $y_i = 1$ with probability θ, $y_i = -1$ with probability $\theta - \delta$, and 0 otherwise. Similarly we define $Y_t = \sum y_i$ for all $i < i_0$. Then we have $\mathbf{Pr}\left[\tilde{m}_i < M',\ \forall i \in [T]\right] = \mathbf{Pr}\left[Y_i < M',\ \forall i \in [T]\right]$. Therefore it is enough for us to prove our statement for Y_i. However, Y_i are still not necessarily independent from each other, before they cross M', however all of them satisfy $\mathbf{E}\left[y_i\right] \geq \delta$ and $\mathbf{Pr}\left[y_i = 1\right] \geq \theta$, and $\mathbf{Pr}\left[y_i = -1\right] \leq \theta - \delta$. Define z_i (and Z_i respectively), such that z_i is stochastically dominated by y_i and each z_i is 1 with probability θ and -1 with probability $\theta - \delta$. Using the Hoeffding inequality we have:

$$\mathbf{Pr}\left[|Z_t - \mathbf{E}\left[Z_t\right]| > C\right] \leq \exp(-\frac{tC}{2})$$

using the fact

$$\mathbf{E}\left[Z_t\right] \geq \delta t \geq M|\log 1/\varepsilon| = M - (M \log 1/\varepsilon + M)$$

and using $C = (M + M \log 1/\varepsilon)$ and using union bound over all t we have desired result immediately for Z_t. Using the fact that $Y_t \geq Z_t$ we have the probability that Y_t never crosses the vicinity is less than ε and hence lemma holds.

Note, that our constraints are spelled in terms of probability mass inequality rather than absolute error. This is required, since for any function $f(M)$, it is possible to devise a distribution, such that the algorithm will be $f(M)^2$ far away from true quantile in absolute steps, and yet it will be very close to it in terms of probability mass.

Lemma 2. *For median estimation, algorithm estimation starts from a position $\tilde{m}_0$, where $F(\tilde{m}_0) > \frac{1}{2} + \delta$. After $T = \frac{M|\log 1/\varepsilon|}{\delta}$ steps of algorithm, the probability that $F(\tilde{m}_t) > \frac{1}{2} + \delta$ for all $t < T$ is at most ε.*

Proof. Proof is similar to Lemma 1.

Theorem 1. *For median estimation, algorithm estimation starts from a position $\tilde{m}_0$, where $F(\tilde{m}_0)$ is outside of region $[\frac{1}{2} - \delta, \frac{1}{2} + \delta]$. After $T = \frac{M|\log \varepsilon|}{\delta}$ steps the algorithm, the probability that $F(\tilde{m}_t)$ is outside of this close region $[\frac{1}{2} - \delta, \frac{1}{2} + \delta]$ for all $t < T$ is at most ε.*

Proof. Proof is directly obtained from Lemma 1 and Lemma 2.

In approaching speed analysis, we do not need assumptions on algorithm's starting estimation. Therefore this actually implies for $Frugal$-1U algorithm, quantile estimations adjust to new distribution quantile when the underlying distribution changes, regardless of current estimation position. The speed of approaching new distribution quantile can be determined by Theorem 1. We verified this feature of $Frugal$-1U in experiments on streams with changing distribution, but omit the results in the interest of space.

4.2 Stability

Next we show that after algorithm estimate once reaches true median, the probability of estimate drifting far away from true median is low. Note that Theorem 1 is affecting this estimation drifting process the whole time.

Lemma 3. *To estimate the median, suppose algorithm estimate starts from true median, after t steps the algorithm estimate is at position $F(\tilde{m}_t)$, where*

$$\mathbf{Pr}\left[F(\tilde{m}_t) > \frac{1}{2} + 2\sqrt{\delta \ln \frac{t}{\varepsilon}}\right] \leq \varepsilon.$$

Proof. Define $\omega = 2\sqrt{\delta \ln \frac{t}{\varepsilon}}$. Let us split the interval $[\frac{1}{2}, \frac{1}{2} + \omega]$ into two, $[\frac{1}{2}, \frac{1}{2} + \omega/2]$ and $[\frac{1}{2} + \omega/2, \frac{1}{2} + \omega]$. Our approach is to show that once the algorithm reaches the boundary of the first interval, it is very unlikely to continue through the second interval, without ever dipping back into the first. First of all we note that we need at least $T = \frac{\omega}{\delta}$ more steps of increment than decrement to reach outside of the second interval, and by the way we select the probabilistic weight of the interval, we will need at least $T/2$ to pass through each.

Consider arbitrary outcome of the algorithm where $\tilde{m}_t > T$. Since x changes by at most 1 at every step, there exists j, such that $\tilde{m}_j = \frac{T}{2}$. Therefore the entire space of

events can be decomposed based on the value of j where $\tilde{m}_j = \lfloor T/2 \rfloor$ and for all $i > j$, $\tilde{m}_i > \tilde{m}_j$. Thus:

$$\begin{aligned}\mathbf{Pr}\left[\tilde{m}_t > T\right] &= \sum_{j=0}^{t} \mathbf{Pr}\left[\tilde{m}_t > T, \tilde{m}_i > \tilde{m}_j, \forall i > j\right] \times \mathbf{Pr}\left[\tilde{m}_j = \left\lfloor \tfrac{T}{2} \right\rfloor\right] \\ &\leq \sum_{j=0}^{t} \mathbf{Pr}\left[\tilde{m}_t > T, \tilde{m}_i > \tilde{m}_j, \forall i > j\right]\end{aligned}$$

Let us consider individual term for a fixed j in the sum above. We want to show that each term is at most ε/t. Define $Y_i^{(j)}$ for $i \geq j$, where $Y_i^{(j)} = \tilde{m}_j + \sum_{k=j+1}^{i} y_j$, and $y_i^{(j)} = x_i$ if $X_i' > \tilde{m}_j$, for all $i' < i$, and for the remainder of the segment $y_i^{(j)}$ is random variable that is -1 with probability $p = \frac{1}{2} + \frac{\omega}{2}$ and 1 otherwise. In other words Y_i agrees with $\tilde{m}_i$ until $\tilde{m}_i = \tilde{m}_j$ for the first time after j, after that $Y_i^{(j)}$ becomes independent of $\tilde{m}_i$. We have:

$$\begin{aligned}&\mathbf{Pr}\left[\tilde{m}_t > T, \tilde{m}_i > \tilde{m}_j, \forall i > j\right] \\ &= \mathbf{Pr}\left[Y_t^{(j)} > T, Y_i^{(j)} > Y_j^{(j)}, \forall i > j\right] \\ &\leq \mathbf{Pr}\left[Y_t^{(j)} > T\right]\end{aligned}$$

therefore it is sufficient to compute an upper bound for $\mathbf{Pr}\left[Y_t^{(j)} > T\right]$ for all j. Let Z_i^j be a variable which both stochastically dominates $Y_i^{(j)}$, and is -1 with probability p and 1 otherwise. Since $Y_i^{(j)}$ is -1 with probability of *at least* p, so such variable Z_i^j always exists. Note that Z_i^j are independent from each other for all i, thus we can use standard tail inequality to upper bound $Z_t^{(j)}$, and because of the dominance the result will immediately apply to $Y_i^{(j)}$. Since $Z_i^{(j)}$ only depends on j at the starting point, we can shift it to zero and rewrite out constraint as:

$$\sum_{j=0}^{t} \mathbf{Pr}\left[Z_j > T/2\right] \leq \varepsilon$$

where Z_j is defined as sum $\sum_{i=0}^{j} z_i$, and z_i is -1 with probability p and 1 otherwise. The expected value of Z_j is $(1-p)j - pj = (1-2p)j = -\omega j$. Furthermore by our assumption, $\omega \geq \frac{\delta T}{2}$. Therefore using Hoeffding inequality we have $\mathbf{Pr}\left[Z_j > T/2\right] \leq \exp - \frac{(\omega j + T)^2}{4j}$. Thus it is sufficient for us to show that

$$\exp\left(-\frac{(\omega j + T)^2}{4j}\right) \leq \frac{\varepsilon}{t}, \text{ for all } j < t$$

This constraint is automatically satisfied for all j such that

$$j \geq \frac{4}{\omega^2} \ln \frac{t}{\varepsilon} = j_0.$$

Indeed, if $j > j_0$ we have $(\omega j + T)/4j \geq \frac{\omega^2}{4j} \geq \ln t/\varepsilon$.

On the other hand if $j \leq j_0$, then we have

$$\frac{(\omega j + T)^2}{4j} \geq \frac{T^2\omega^2}{16 \ln t/\varepsilon}$$

but $T \geq \omega/\delta$ and substituting the expression for ω we have:

$$\frac{T^2\omega^2}{4 \ln t/\varepsilon} \geq \frac{\omega^4}{16\delta^2 \ln t/\varepsilon} = \ln t/\varepsilon$$

Thus $\mathbf{Pr}\left[Z_j > T/2\right] \leq \varepsilon/t$, for $j < j_0$, completing the proof.

Lemma 4. *To estimate the median, suppose algorithm estimate starts from true median, after t steps, the algorithm estimate is at position $F(\tilde{m}_t)$, where*

$$\mathbf{Pr}\left[F(\tilde{m}_t) < \frac{1}{2} - 2\sqrt{\delta \ln \frac{t}{\varepsilon}}\right] \leq \varepsilon.$$

Proof. Following the same reasoning in the proof of LEMMA 3, we can prove that the probability of estimation moving far to the left is small. Where we can split the interval $[\frac{1}{2} - \omega, \frac{1}{2}]$ into two $[\frac{1}{2} - \omega, \frac{1}{2} - \omega/2]$ and $[\frac{1}{2} - \omega/2, \frac{1}{2}]$. We can show that once the algorithm reaches the boundary of the first interval, it is very unlikely to continue through the second interval without ever dipping back into the first.

Theorem 2. *To estimate median, after t steps, the probability of the algorithm current position*

$$\mathbf{Pr}\left[\left|F(\tilde{m}_t) - \frac{1}{2}\right| > 2\sqrt{\delta \ln \frac{t}{\varepsilon}}\right] \leq \varepsilon.$$

Proof. This theorem is obtained from Lemma 3 and 4.

These properties of median estimation can be generalized to any quantile $\frac{h}{k}$.

5 Algorithm Extensions

The $Frugal$-$1U$ algorithm described in Section 3 uses 1 unit of memory and is intuitive, and we managed to analyze it; however it has linear convergence to the true quantile. This is effectively by design, because the algorithm does not have the capability to remember anything except the current location. A simple extension to our algorithm is to keep a current step size in memory, and modify it if the new samples are consistently on one side of the current estimate.[2] In this section we describe a 2 units of memory algorithm that we use in experiments for comparison.

Generally the algorithm uses two variables to keep quantile estimate and update size, and one extra bit to keep sign, which indicates the increment or decrement direction of

[2] Another approach that we do not explore here, is to use multiplicative update on step size instead of additive.

Algorithm 3 $Frugal\text{-}2U$

Input: Data stream S, h, k, $\tilde{m}$, step, $sign$
Output: $\tilde{m}$

Initialization $\tilde{m} = 0$, step $= 1$, $sign = 1$
for each s_i in S **do**
  $rand$ = random(0,1);
  if $s_i > \tilde{m}$ **and** $rand > 1 - h/k$ **then**
    step += $(sign > 0)$? f(step) : $-f$(step);
    $\tilde{m}$ += (step> 0) ? $\lceil$step$\rceil$: 1;
    $sign = 1$;
    if $\tilde{m} > s_i$ **then**
      step += $s_i - \tilde{m}$;
      $\tilde{m} = s_i$;
    end if
  else if $s_i < \tilde{m}$ **and** $rand > h/k$ **then**
    step += $(sign < 0)$? f(step) : $-f$(step);
    $\tilde{m}$ - = (step> 0) ? $\lceil$step$\rceil$: 1;
    $sign = -1$;
    if $\tilde{m} < s_i$ **then**
      step += $\tilde{m} - s_i$;
      $\tilde{m} = s_i$;
    end if
  end if
  if $(\tilde{m} - s_i) * sign < 0$ **and** step> 1 **then**
    step$= 1$;
  end if
end for

estimate. Empirically this algorithm has much better convergence and stability property than 1 unit of memory algorithm, however the precise convergence/stability analysis of it is one of our future work. On the intuitive level the algorithm for finding the median works as follows. As before it maintains the current estimate of median but in addition it also maintains an update step that increases or decreases based on the observed values, determined by a function f. More precisely, the step increases if the next element from the stream is on the same side of the current estimate, and decreases otherwise. When estimation is close to true quantiles, step can be decreased to extremely small value.

The increment and decrement factors to be applied to step remains an open problem. step can potentially grow to very large values, so the randomness of the order which stream items appear affects estimation accuracy. For example, if let $\texttt{step}_\texttt{i}$ be the step value at ith update, a multiplicative update of $\texttt{step}_{\texttt{i}+1} = 2 \times \texttt{step}_\texttt{i}$ might be a good choice for a random order stream, which intuitively needs $O(\log M)$ updates to reach true quantile at distance M from current estimate. However in empirical data periodic pattern might be apparent in the stream, for example social network users might have shorter activity intervals at evening, but longer intervals at early morning. Then step can easily get increased to a huge value. It will make the algorithm estimate drift far away from true quantile, hence estimates will have large oscillations.

Therefore to trade off convergence speed for estimation stability we present a version of 2 units of memory algorithm that applies constant factor additive update to step size, where $f(\texttt{step}) = 1$. Full details of the algorithm are described in Algorithm 3. Lines 4-11 handle stream items larger than algorithm estimation, and lines 12-19 handle smaller stream items. For brevity we only look at lines 4-11 in detail. Similar to Algorithm $Frugal$-1U , the key to make $Frugal$-2U able to estimate any quantile is that not every stream item will cause an estimation update, so line 4 enables updates only on "unexpected" larger stream items. `step` is cumulatively updated in line 5. Line 6 ensures minimum update to estimation is 1, and `step` size is only applied in update when it is positive. The reason is that when algorithm estimation is close to true quantile, $Frugal$-2U updates are likely to be triggered by larger and smaller (than estimation) stream items with largely equal chances. Therefore `step` is decreased to a small negative value and it serves as a buffer for value bursts (*e.g.,* a short series of very large values) to stabilize estimations. Lines 8-11 are to ensure estimation do not go beyond empirical value domain when `step` gets increased to very large value. At the end of the algorithm, we reset `step` if its value is larger than 1 and two consecutive updates are not in the same direction. This is to prevent large estimate oscillations if `step` gets accumulated to a large value. This checking is implemented by lines 21-23.

Note that $Frugal$-1U and $Frugal$-2U algorithms are initialized by 0, but in practice they can be initialized by the first stream item to reduce the time needed to converge to true quantiles.

6 Related Work and Algorithms to Compare

There has been extensive work in the database community on theory and practice of approximately estimating quantiles of streams with limited memory (e.g.., [1–4,6,7,9–11, 13, 14, 17]). This body of research has generated methods for approximating quantiles to $1 + \epsilon$ approximation with space roughly $O(1/\epsilon)$ in various models of data streams.

We compare our algorithms with existing algorithms that use constant memory for stochastic streams [11], and also non-constant memory algorithms described in [10,17]. However all the non-constant memory algorithms above use considerably more than 2 persistent variables. While some of the algorithms such as the one described in [1] have a tuning parameter allowing to decrease memory utilization, the algorithm then performs poorly when used with less than 20 variables. Here we briefly overview the algorithms we compare.

6.1 GK Algorithm

Greenwald and Khanna [10] proposed an online algorithm to compute ϵ-approximate quantile summaries with worst-case space requirement of $O(\frac{1}{\epsilon}log(\epsilon N))$. Greenwald-Khanna algorithm (GK) maintains a list of tuples (v_i, g_i, $\triangle_i$), where v_i is a value seen from the stream and tuples are order by v in ascending order. $\sum_{j=1}^{i} g_j$ gives the minimum rank of v_i, and its maximum rank is $\sum_{j=1}^{i} g_j + \triangle_i$. GK is composed of two main operations which are to insert a new tuple in to tuple list when sees a new value, and do compression on the tuple list to achieve the minimum space as possible. Throughout

the updates it is kept invariant that for any tuple we have $\sum_{j=1}^{i} g_j + \triangle_i \leq 2\epsilon N$ to ensure the ϵ-approximate query answers. To make it comparable with our $Frugal$-1U and $Frugal$-2U , we limit the number of tuples maintained by GK . When this memory budget is exceeded we gradually increase ϵ (increment by 0.001) to force compression operation get conducted repeatedly until number of tuples used is within specified budget. In our comparison, we limit the number of tuples to be $t = 20$.

6.2 *q-digest* Algorithm

Tree based stream summary algorithms were studied by Manku et al. [14], Munro and Paterson [16], Alsabti et at. [2], Shrivastava et al. [17] and Huang et al. [12]. In this paper we compare with q-$digest$ algorithm proposed in [17], which is most relevant to our comparison aspects. Their proposed algorithm builds a binary tree on a value domain σ, with depth $\log \sigma$. Each node v in this tree is considered as a bucket representing a value range in the domain, associated with a counter indicating the number of items falling in this bucket. A leaf node represents a single value in domain, and associated with the number of items having this value. Each parent node represents the union of the ranges of children nodes, root node represents the full domain range. This algorithm then keeps merging and removing nodes in the tree to meet memory budget requirement.

For every new stream sample we make a trivial q-$digest$ and merge it with q-$digest$ built so far. Therefore, at any time we can query for a quantile based on the most recently updated q-$digest$. For our evaluation we used number of buckets of $b = 20$ to build tree digests.

6.3 *Selection* Algorithm

Guha and McGregor [11] proposed an algorithm that uses constant memory and operates on random order streams, where the order of elements of the stream have not been chosen by adversary. Their approach is a single pass algorithm that uses constant space and their guarantee is that for a given r (the rank of element of interest) their algorithm returns an element that is within $O(n^{1/2})$ rank of r with probability at least $1 - \delta$. The algorithm does not require prior knowledge of the length of the stream, nor the distribution, which is in common with our $Frugal$-1U and $Frugal$-2U .

This single-pass algorithm ($Selection$) processes the stream in phases, and each phase is composed of three sub-phases namely, *sample*, *estimate* and *update*. Throughout the process, algorithm maintains an interval (a, b) which encloses the true quantile and each phase tries to narrow this interval. At any time algorithm has to keep four variables which are the boundaries a and b, estimation u, and a counter to estimate rank of u. For this algorithm, data size n should be given in order to decide how to divide stream into pieces. By adding one more variable, one can remove this requirement of knowing n beforehand. The proved accuracy guarantee can be achieved when the overall stream is very large. In experiments, to relax the requirement of very large streams we set $\delta = 0.99$, and the version without knowing n in advance is evaluated. [3]

[3] McGregor and Valiant [15] gave a new algorithm using the same space, proving improved approximation with accuracy $n^{1/3+o(1)}$ can be achieved. This algorithm behaves qualitatively similar to the algorithm $Selection$ we have implemented here.

7 Empirical Evaluations

In this section we evaluate our algorithms on both synthetic and two real world data sets. For synthetic data we consider two scenarios, one when data arrive from a static distribution, and one when the distribution changes mid-stream. These tests demonstrate that our algorithms perform well for both scenarios. For real world data we evaluate on HTTP streams [5] and Twitter user tweet streams, where our goals are to evaluate median and 90-% quantile estimates of TCP-flow durations and tweet intervals. As mentioned earlier the structure of our algorithms allow us to estimate quantiles for every stream with 1 or 2 in-memory variables, and the quantile to estimate can be shared by all streams.

Instead of evaluating the absolute error of quantile estimation, we evaluate how far the estimate is from the true quantile by the relative mass error. For example if the estimate of 90-% quantile turned out to be 89-% quantile then the error is 0.01. Throughout our evaluations, we initialize $Frugal\text{-}1U$ and $Frugal\text{-}2U$ algorithm estimates with 0 [4]. For non-constant memory algorithms GK and $q\text{-}digest$, we limit the memory budget to 20 units of their in-memory data structure.

7.1 Synthetic Data

In this section we evaluate algorithms on data streams from a Cauchy distribution (density function $f(x) = \frac{\gamma}{\pi(\gamma^2+(x-x_0)^2}$). The reason we picked Cauchy is because it has a high probability of outliers, the expected value of a Cauchy random variable is infinity, thus we can demonstrate that our algorithms work well in the presence of outliers.

Static Distribution. For our experiments we fix $x_0 = 10000$ and $\gamma = 1250$, and draw 3×10^4 samples. Figure 2 shows the evaluation results. Not only for $Frugal\text{-}1U$ and $Frugal\text{-}2U$, but also most of the algorithms in comparison need some time (some amount of stream items) before getting to a stable quantile estimation. When memory is insufficient for the non-constant memory algorithms, estimation performance degrades much. Due to smaller fixed update size of $Frugal\text{-}1U$, it takes much longer travel than $Frugal\text{-}2U$ to reach stream quantiles.

Dynamic Distribution. Since the algorithms in comparison are not built for estimating changing distributions, we only evaluate $Frugal\text{-}1U$ and $Frugal\text{-}2U$ in the scenario where the underlying distribution of stream changes. We generate three sub-streams drawn from three different Cauchy distributions and feed them one by one to our algorithms. For each of the three sub-streams we sample 2×10^4 items in value domains [10000, 15000], [15000, 20000] and [20000, 25000] respectively.

Figure 3 shows the median and 90-% quantile estimations for $Frugal\text{-}1U$ and $Frugal\text{-}2U$ algorithms. Those sub-streams are ordered by their medians in the sequence of highest, lowest and middle, then they are feed to algorithms one by one. For other algorithms they either need to know the value domain as input or they try to learn upper and lower bounds for the quantile in query, therefore if the stream underlying distribution changes their knowledge about stream are out-dated hence quantile

[4] In practice we can also initialize them with the first stream item.

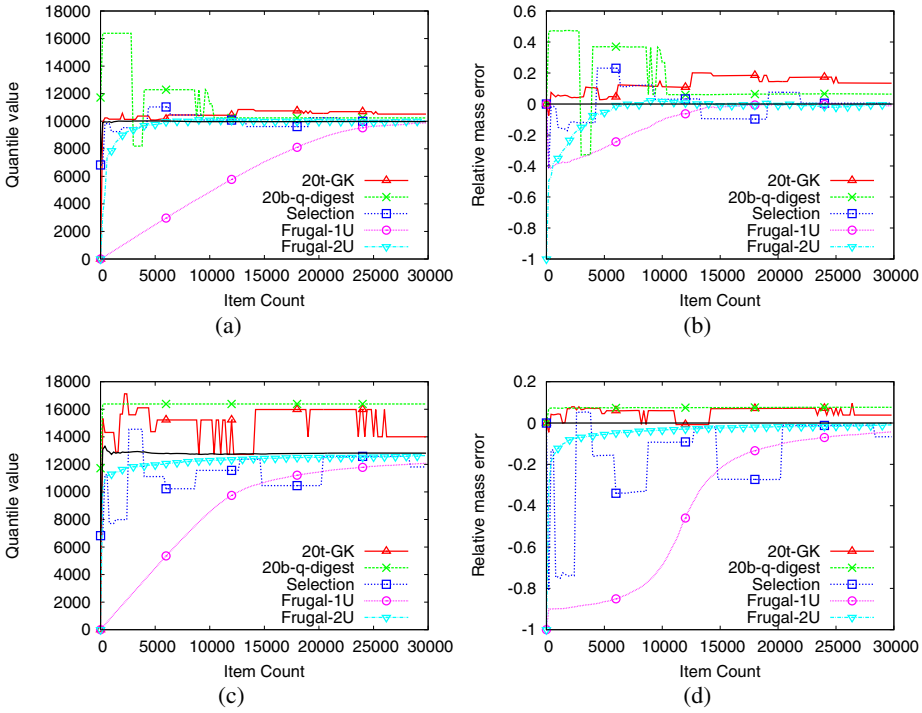


Fig. 2. Evaluation on a stream from one Static Cauchy Distribution. (a) median estimation. (b) relative mass error for (a). (c) 90-% quantile estimation. (d) relative mass error for (c).

approximations are probably not accurate. $Stream\text{-}quantile$ curve shows the cumulative stream quantile, and this is the curve which the other algorithms try to approximate if the combined stream is of interest at the beginning. But in this figure we want to show that our $Frugal\text{-}1U$ and $Frugal\text{-}2U$ are doing a different job. $Use\text{-}Distrib$ curve shows the quantile values for each sub-distribution. The change of $Use\text{-}Distrib$ curve indicates the change of underlying distribution. We can see that our algorithms are trying to reach new distribution's quantile when the stream underlying distribution changes. It is only that $Frugal\text{-}1U$ takes longer time to approach new distribution's quantiles, while $Frugal\text{-}2U$ can make "sharper" turns in its quantile estimations when distribution changes. $Frugal\text{-}1U$ in Figure 3.(b) leaves a steeper approaching trace to 90-% quantile than estimating median in Figure 3.(a), because it is more biased to move estimate towards one direction (getting larger).

One counter argument is that the property of adapting to changing distribution's new quantile also might be a disadvantage, because it makes the algorithms vulnerable to short bursts of "noise". However since the adjustment taken by $Frugal\text{-}1U$ is 1, when the true stream quantile is large the shifting from the true stream quantile caused by short bursts will not affect much in terms of relative mass error. For $Frugal\text{-}2U$ it

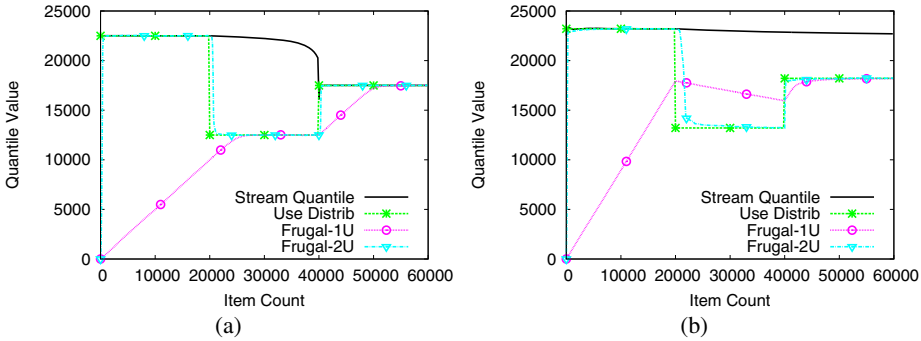


Fig. 3. Evaluation on one stream generated from three Cauchy distributions. (a) Median estimation. (b) 90-% quantile estimation. The change of $Use\text{-}Distrib$ curve indicates the change of underlying distribution. $Frugal\text{-}2U$ algorithm converges to new distribution quantiles significantly faster than $Frugal\text{-}1U$.

is true that `step`'s increment and decrement function f should be picked to trade-off between convergence speed and stability when bursts or periodic patterns are apparent in streams. But once after reaching a close estimate of true quantile, the decreasing `step` value is able to buffer the impact of some value bursts.

7.2 HTTP Streams Data

From an HTTP request and response trace [5] collected for over a period of 6 months, spanning 2003-10 to 2004-03, we extract out TCP-flow durations (in millisecond) between local clients and 100 remote sites, and order them by connections set up time to form streams. In this experiment we first evaluate on streams generated with each of those 100 sites in each of the 6 months. Therefore in total we have 600 streams. But in final performance evaluations we filter out streams with length less than 2000 items and end up with 419 usable streams. Finally we collect the last estimations for median and 90-% quantile by all algorithms.

Figure 4 shows the relative mass error and cumulative percent of 419 TCP-flow duration streams. Figure 4.(a) and (b) show that $Frugal\text{-}2U$ performances are better than or comparable with other algorithms. Whereas $Frugal\text{-}1U$ largely makes underestimations for most of the streams, because in evaluations we initiated $Frugal\text{-}1U$ and $Frugal\text{-}2U$ quantile estimations from 0, however duration stream median (and 90-% quantile) values can easily be tens of thousands. In comparison, $t = 20$ for GK and $b = 20$ for $q\text{-}digest$ are not enough to get close estimations. Note that in relative mass error, the overestimate errors are bounded by 0.5 and 0.1 respectively for median and 90-% quantile estimations.

In the situations where there are millions of streams to be processed simultaneously, statistical quantities about more general groups can help understand the characteristics of different groups. In HTTP request and response trace, streams generated by remote site can also be considered as GROUPBY application to understand the communication patterns from local clients to different remote sites.

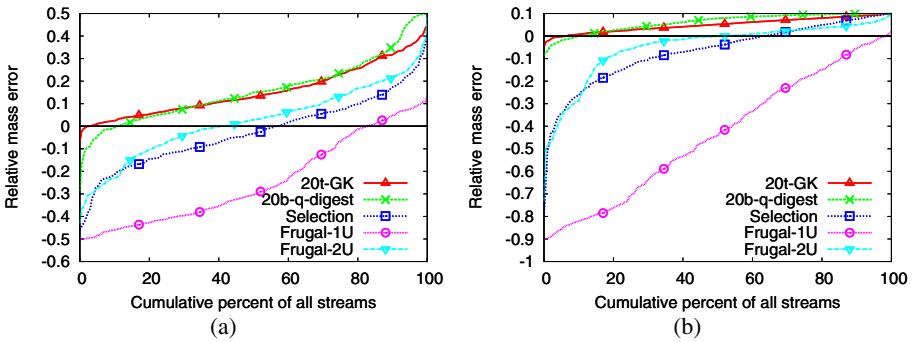


Fig. 4. Evaluation on 419 TCP-flow duration streams by cumulative percent of all streams at different relative mass errors. (a) median estimation. (b) 90-% quantile estimation.

We evaluate all algorithms for one GROUPBY application on this HTTP trace data, where connections with all 100 sites in each month are combined by their creation time. This simulates the viewpoint from trace collecting host. For brevity here we present the results from evaluation on combined stream of month 2004-03, and the results are similar for other months. This combined stream has about 1.6×10^6 items. Figure 5 presents the results on estimating median and 90-% quantile of this stream. In this stream we have median and 90-% quantile values at about 544,267 and 1,464,793 (in microsecond) respectively. Due to the large quantile value $Frugal\text{-}1U$ shows a slower convergence to true stream quantile, while $Frugal\text{-}2U$ handles this problem much better. $Selection$ converges to [-0.1, 0.1] relative mass error region after about 2×10^5 items, but it is oscillatory thereafter and needs much more items to stabilize. In contrast, although $Frugal\text{-}1U$ and $Frugal\text{-}2U$ need relatively more stream items to reach a large true quantile their estimations are relatively stabler. In Figure 5.(a), $b = 20$ $q\text{-}digest$ gives very oscillatory median estimation around 8×10^5, and from the curve it seems converging to stream median but apparently it needs much more stream items. Overall, 20 units of in-memory variables are not sufficient for GK and $q\text{-}digest$ to make accurate quantile estimations.

7.3 Twitter Data Set

From an on-line twitter user directory, we collected 4554 users over 80 directories (e.g. Food and Business). Those tweets from individual users form 4554 sub-streams in the ocean of all tweets. We extracted the intervals (in seconds) between two consecutive tweets for every user and then run our algorithms on those interval streams. This allows us to answer the question of "what is the median inactive time for a given user across all?".

Among the total 4554 twitter users, we filtered out the users with less than 2000 tweets since we need a decent number of data items to reflect the true distribution and allow our algorithms to reach true quantiles. Since twitter does not store more than 3200 tweets of a single user, therefore at the time of data collection the maximum length of a

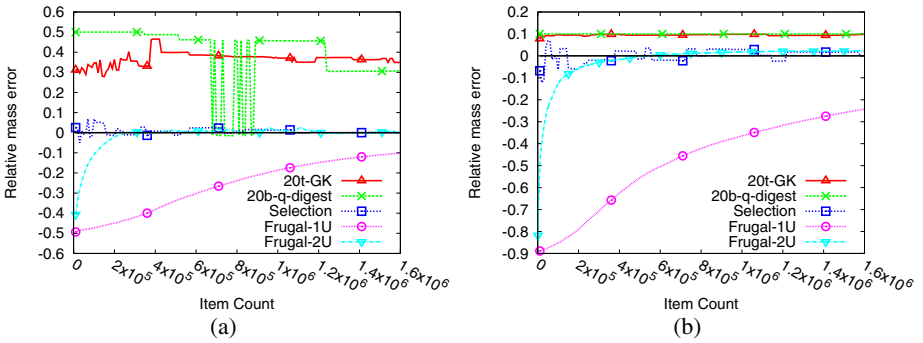


Fig. 5. Relative mass error evaluation on TCP-flow duration stream of month 2004-03. (a) median estimation. (b) 90-% quantile estimation.

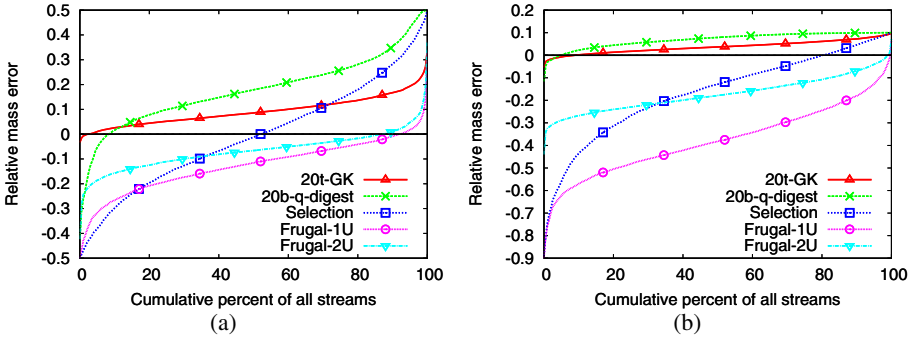


Fig. 6. Evaluation on 4414 twitter users' tweet interval streams, by cumulative percent of all streams at different relative mass errors. (a) median estimation. (b) 90-% quantile estimation.

single user's interval stream is 3200. Finally we evaluated our algorithms on 4414 tweet interval streams, and collected the last estimations for median and 90-% quantile.

Figure 6 shows the relative mass error and cumulative percent of all 4414 interval streams. In Figure 6.(a) we see that about 70 percent of the last median estimation by $Frugal$-1U are under-estimating (less than -0.1). In evaluations we initiated $Frugal$-1U and $Frugal$-2U quantile estimations from 0, however interval stream median (and 90-% quantile) values can easily be tens of thousands. Therefore within 2000 steps they can not fully reach true median. $Frugal$-2U applies dynamic $step$ size hence it performs much better than $Frugal$-1U algorithm, with more than 70 percent of the last median estimations in error range [-0.1, 0.1]. In comparison, $b = 20$ for q-$digest$ are not enough to get close estimations, and $Selection$ does not work well on these short streams. Figure 6.(b) shows that when estimating 90-% quantile, which are much larger values, as expected $Frugal$-1U cannot reach true quantile when the stream items are few (94% of twitter user interval streams have 90-% quantiles larger than 3,200). Again $Frugal$-2U shows its advantages over $Frugal$-1U but it also needs longer streams to reach true quantiles. In comparison, $t = 20$ for GK and $b = 20$ for q-$digest$ are

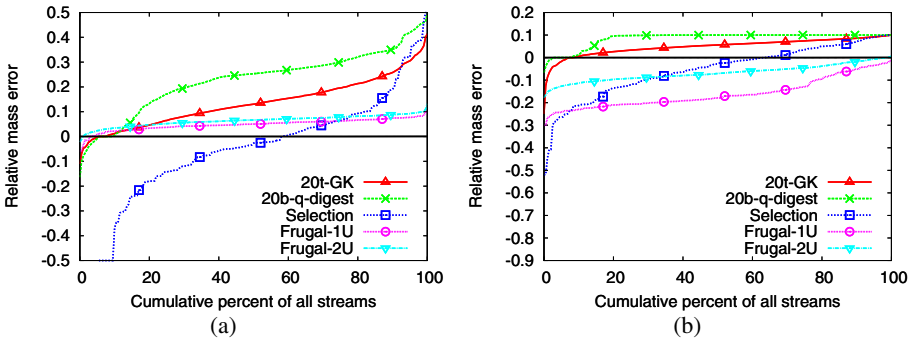


Fig. 7. Evaluation on 905 daily tweet interval streams, by cumulative percent of all streams at different relative mass errors. (a) Median estimation. (b) 90-% quantile estimation.

not affected by stream sizes, however $Selection$ algorithm needs much longer streams. Again note that from this figure, the overestimate errors are bounded by 0.5 and 0.1 respectively for median and 90-% quantile estimations, because relative mass error is measured.

For a database there are various meaningful GROUPBY applications, such as group by geo-location and age for an on-line social network database. To simulate such applications, we evaluate our algorithms on the combined tweet interval streams on each day. We merge tweet interval streams from all 4554 twitter users in our dataset, and sort all the intervals based on the their creation time. We divide the combined interval stream into segments by day, and in total our tweet interval data spans 1328 days from 2008 to 2011. We ran our algorithms on each day's stream and take the last estimations from algorithms to evaluate their accuracy. We filter out the days that have less than 2000 intervals in the daily stream, with similar reason to filter individual tweeter user's tweet interval streams. After filtering process, we have 905 days left. Figure 7 shows the cumulative percent of all days against relative mass error, we can see that median and 90-% quantile under-estimation problems in individual user interval streams are alleviated (in daily interval streams about 67% of the streams have size larger than 3,200). Figure 7.(a) demonstrates that $Frugal$-$1U$ can reach close estimation before the daily interval streams end. $Frugal$-$2U$ does not have much advantage over $Frugal$-$1U$ in these streams, so it just shows similar performance with $Frugal$-$1U$. In Figure 7.(b), for 90-% quantile on most of the days $Frugal$-$1U$ algorithm underestimates the true quantiles by using update size of 1. For median and 90-% quantile estimations by $Frugal$-$2U$ almost all last estimates are in relative mass error range [-0.1, 0.1]. In comparison, $t = 20$ for GK and $b = 20$ for q-$digest$ are not enough to get close estimations, and $Selection$ algorithm needs much more stream items.

Throughout our extensive experiments on synthetic and real-world HTTP trace and twitter data, for streams given enough number of data items in the stream, our 1 and 2 variables stochastic algorithms can achieve quite comparative accuracy against other non-constant and constant memory algorithms, while using much less memory and being very efficient for per item update.

8 Conclusions and Future Directions

We have introduced the concept of frugal streaming and presented algorithms that can estimate arbitrary quantiles using 1 or 2 unit memories. This is very useful when we need to estimate quantiles for each of many groups, as applications demand in reality. These algorithms do not perform well with adversarial streams, but we have mathematically analyzed the 1 unit of memory algorithm and shown fast approach and stability properties for stochastic streams. Our analysis is non-trivial, and we believe it provides a framework for analysis of other statistical estimates with stochastic streams. Further we have reported extensive experiments with our algorithms and several prior quantile algorithms on synthetic data as well as real dataset from HTTP trace and Twitter.

To the best of our knowledge our algorithms are the first that perform well with 2 or less persistent variables per group. In contrast, other regular streaming algorithms, while having other desirable properties, perform poorly when pushed to the extreme on memory consumption like we do with our frugal streaming algorithms.

Our work has initiated frugal streaming, but much remains to be done. First, we need mathematical analysis of 2 or more memory algorithms and at this moment, it looks quite non-trivial. We also need frugal streaming algorithms for other problems such as distinct count estimation and others, that are critical for streaming applications. Finally, as our experiments and insights indicate, frugal streaming algorithms work with so little memory of the past that they are adaptable to changes in the stream characteristics. It will be of great interest to understand this phenomenon better.

References

1. Agrawal, R., Swami, A.: A one-pass space-efficient algorithm for finding quantiles. In: Proc. 7th Intl. Conf. Management of Data, COMAD 1995 (1995)
2. Alsabti, K., Ranka, S., Singh, V.: A one-pass algorithm for accurately estimating quantiles for disk-resident data. In: Proc. 23rd VLDB Conference, pp. 346–355 (1997)
3. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2004, pp. 286–296. ACM, New York (2004)
4. Babcock, B., Datar, M., Motwani, R., O'Callaghan, L.: Maintaining variance and k-medians over data stream windows. In: Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2003, pp. 234–243. ACM, New York (2003)
5. Bissias, G.D., Liberatore, M., Jensen, D., Levine, B.N.: Privacy vulnerabilities in encrypted HTTP streams. In: Danezis, G., Martin, D. (eds.) PET 2005. LNCS, vol. 3856, pp. 1–11. Springer, Heidelberg (2006)
6. Cormode, G., Korn, F., Muthukrishnan, S., Srivastava, D.: Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2006, pp. 263–272. ACM, New York (2006)
7. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55(1), 58–75 (2005)
8. Cranor, C., Johnson, T., Spataschek, O.: Gigascope: a stream database for network applications. In: SIGMOD, pp. 647–651 (2003)

9. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.J.: How to summarize the universe: dynamic maintenance of quantiles. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002, pp. 454–465. VLDB Endowment (2002)
10. Greenwald, M., Khanna, S.: Space-efficient online computation of quantile summaries. SIGMOD Rec. 30, 58–66 (2001)
11. Guha, S., Mcgregor, A.: Stream order and order statistics: Quantile estimation in random-order streams. SIAM Journal on Computing 38, 2044–2059 (2009)
12. Huang, Z., Wang, L., Yi, K., Liu, Y.: Sampling based algorithms for quantile computation in sensor networks. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 745–756. ACM, New York (2011)
13. Lin, X., Lu, H., Xu, J., Yu, J.X.: Continuously maintaining quantile summaries of the most recent n elements over a data stream. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, pp. 362–374. IEEE Computer Society, Washington, DC (2004)
14. Manku, G.S., Rajagopalan, S., Lindsay, B.G.: Approximate medians and other quantiles in one pass and with limited memory. SIGMOD Rec. 27, 426–435 (1998)
15. Mcgregor, A., Valiant, P.: The shifting sands algorithm. In: SODA (2012)
16. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science 12(3), 315–323 (1980)
17. Shrivastava, N., Buragohain, C., Agrawal, D., Suri, S.: Medians and beyond: new aggregation techniques for sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, pp. 239–249. ACM, New York (2004)

From Time to Space: Fast Algorithms That Yield Small and Fast Data Structures

Jérémy Barbay

Departamento de Ciencias de la Computación (DCC),
Universidad de Chile, Santiago, Chile
jbarbay@dcc.uchile.cl

Abstract. In many cases, the relation between encoding space and execution time translates into combinatorial lower bounds on the computational complexity of algorithms in the comparison or external memory models. We describe a few cases which illustrate this relation in a distinct direction, where fast algorithms inspire compressed encodings or data structures. In particular, we describe the relation between searching in an ordered array and encoding integers; merging sets and encoding a sequence of symbols; and sorting and compressing permutations.

Keywords: Adaptive (Analysis of) Algorithms, Compressed Data Structures, Permutation, Set Union, Sorting, Unbounded Search.

1 Introduction

The worst-case analysis of algorithmic constructs is the theoretical equivalent of a grumpy and bitter fellow who always predicts the worst outcome possible for any actions you might take, however far-fetched the prediction.

Consider for instance the case of data compression. In a modern digital camera, one does not assume that each picture of width w and height h measured in pixels of b bytes will require $h \times w \times b$ bytes of storage space. Rather, the design assumes that the pictures taken have some regularities (e.g. many pixels of similar colors grouped together), which permit to encode it in less space. Only a truly random set of pixels will require $h \times w \times b$ bytes of space, and those are usually not meaningful in the usage of a camera.

Similar to compression techniques, some opportunistic algorithms perform faster than the worst case on some instances. As a simple example, consider the sequential search algorithm on a sorted array of n elements, which performs n comparisons in the worst case but much less in other cases. More sophisticated search algorithms [9] uses $\mathcal{O}(\log n)$ comparisons in the worst case, yet much less in many particular cases. The study and comparison of the performance of such algorithms requires finer analysis techniques than the worst case among instances of fixed sizes. An analysis technique reinvented several times (under names such as *parameterized complexity*, *output-sensitive complexity*, *adaptive* (analysis of)

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 97–111, 2013.

algorithms, *distribution-sensitive algorithms* and others) is to study the worst-case performance among finer classes of instances, defined not only by a bound on their size but also by bounds on some defined measure of their *difficulty*.

This concept of opportunism, in common between encodings and algorithms, has yielded some dual analysis, where any fine-analysis of an encoding scheme implies a similar analysis of an algorithm, and vice-versa. For instance, already in 1976 Bentley and Yao [9] described *adaptive algorithms* inspired by Elias codes [13] for searching in a (potentially unbounded) sorted array. More recently in 2009, Laber and Avila [1] showed that any comparison-based merging algorithm can be adapted into a compression scheme for bit vectors. In particular, they discovered a relation between previous independent results on the distinct problems of comparison-based merging and compression schemes for bit vectors: the `Binary Merging` algorithm proposed by Hwang and Lin [24] is closely related to a runlength-based coder using `Rice coding` [35], while `Recursive Merging` is closely related to a runlength-based coder using the `Binary Interpolative Coder`. The very same year, Barbay and Navarro [7], observing that any comparison-based sorting algorithm yields an encoding for permutations, exploited the similarity between the *Wavelet Tree* [22], a data structure used for compression, and the execution of an adaptive variant of `Merge Sort`, to develop and analyze various *compressed data structures* for permutations and *adaptive sorting algorithms*. In this case, the construction algorithm of each data structure is in fact an adaptive algorithm sorting the permutation, in a number of comparisons roughly the same as the number of bits used by the data structure. Given those results, the following questions naturally arise:

1. All comparison-based adaptive sorting algorithms yield compressed encodings. One can wonder if all comparison-based adaptive sorting algorithm, and not only those based on `MergeSort`, can inspire compressed data structures, and if a similar relationship exists for all comparison-based algorithms on other problems than sorting. **Which adaptive algorithms yield compressed data structures?**
2. In the cases of permutations [7], techniques from data compression inspired improvements on the design of algorithms and the analysis of their performance. **Are there cases where such relations between compressed data structures and adaptive algorithms are bijective?**
3. Ignoring the relation between comparison-based merging algorithms and compressed encodings of bit vectors led to the independent discovery of similar techniques [1]. **Could a better understanding of such relations simplify future research?**

This survey aims to be a first step toward answering those questions, by reviewing some adaptive techniques (Section 2), some related data structures (Section 3), and various relations between them in the comparison model, such as between sorted search algorithms and encodings for sequences of integers (Section 4.1), merging algorithms and string data structures (Section 4.2), and sorting algorithms and permutation data structures (Section 4.3). We conclude with a selection of open problems in Section 5.

2 Adaptive Analysis

2.1 Sorted Search

A regular implementation of binary search returns the insertion rank r (defined as $r \in [1..n]$ such that $r = 1$ and $x \leq A[1]$ or $r > 1$ and $A[r-1] < x \leq A[r]$) of an element x in a sorted array A of n elements in $\lceil \lg n \rceil + 1 \in \mathcal{O}(\log n)$ comparisons[1] in the worst case and in $\lfloor \lg n \rfloor + 1 \in \mathcal{O}(\log n)$ comparisons in the best case. The performance of sequential search is much more variable: the algorithm will perform $\min(r, n) + 1$ comparisons (between 2 and $n + 1$), which corresponds to a worst-case complexity of $\mathcal{O}(n)$ comparisons. An interesting (if minor) fact is that, whenever the insertion rank of x is less than $\lceil \lg n \rceil + 1$, *linear search outperforms binary search*.

In 1976, inspired by Elias' codes [13] for sequences of integers, Bentley and Yao [9] described a family of algorithms for *unbounded search* in a (potentially infinite) sorted array. Among those, the doubling search algorithm [9] returns the insertion rank after $1 + 2\lfloor \lg r \rfloor$ comparisons. Hence, doubling search outperforms binary search whenever the insertion rank of x is less than $\sqrt{n}$, and never performs worse than twice the complexity of the binary search. It is widely used in practice, whereas its asymptotic worst-case complexity is the same as binary search, both optimal in the comparison model: the traditional worst-case analysis for a fixed value of n fails to distinguish the performance of those algorithms.

2.2 Union of Sorted Sets

A problem where the output size can vary but is not a good measure of difficulty, is the *description* of the *sorted union of sorted sets*: given k sorted sets, describe their union. On the one hand, the sorted union of $A = \{0, 1, 2, 3, 4\}$ and $B = \{5, 6, 7, 8, 9\}$ is easier to describe (all values from A in their original order, followed by all values from B, in their original order) than the union of $C = \{0, 2, 4, 6, 8\}$ and $D = \{1, 3, 5, 7, 9\}$. On the other hand, a deterministic algorithm must *find* this description, which can take much more time than to output it when computing the union of many sets at once.

Carlsson *et al.* [11] defined the adaptive algorithm `Adaptmerge` to compute the union of two sorted sets in adaptive time. This can be used to compute the union of k sets by merging them two by two, but Demaine *et al.* [12] proposed an algorithm whose complexity depends on the minimal encoding size $\mathcal{C}$ of a *certificate*, a set of comparisons required to check the correctness of the output of the algorithm (yielding worst-case complexity $\Theta(\mathcal{C})$) over instances of fixed size n and certificate-encoding-size $\mathcal{C}$). An alternative approach is to consider the non-deterministic complexity [6], the number of steps δ performed by a non deterministic algorithm to solve the instance, or equivalently the minimal number of comparisons of a certificate (yielding worst-case complexity $\Theta(\delta k \log(n/\delta k))$ over instances of fixed size n and certificate-comparison-size δ).

[1] We note $\lg n = \log_2 n$, $\lg^{(k)}(n) =$ the logarithm iterated k times, and $\log n$ when the base do not matter, such as in asymptotic notations.

2.3 Sorting

Sorting an array A of numbers is a basic problem, where the size of the output of an instance is always equal to its input size. Still, some instances are easier than others to sort (e.g. a sorted array, which can be checked/sorted in linear time). Instead of the output size, one can consider the disorder in an array as a measure of the difficulty of sorting this array [10, 28].

There are many ways to measure this disorder: one can consider the number of exchanges required to sort an array; the number of adjacent exchanges required; the number of pairs (i, j) such that $A[i] > A[j]$, but there are many others [33]. For each disorder measure, the logarithm of the number of instances with a fixed size and disorder forms a natural lower bound to the worst-case complexity of any sorting algorithm in the comparison model, as a correct algorithm must at least be able to distinguish all instances. As a consequence, there could be as many optimal algorithms as there are difficulty measures. Instead, one can reduce difficulty measures between themselves, which yields a hierarchy of disorder measures [29].

An algorithm is *adaptive with respect to* a given measure of presortedness M if its running time is a function of both the measure M and the size n of the input list $X = \langle x_1, \ldots, x_n \rangle$, being linear for small values of M and at most polylogarithmic in n for larger ones. Many measures have been defined [29], we list here only a few relevant ones:

- `nSRuns`, the number of *Strict Runs*, subsequences of consecutive positions in the input with a gap between successive values exactly 1, from beginning to end (e.g. $(1, 2, 6, 7, 8, 9, 3, 4, 5)$ is composed of $\texttt{nSRuns} = 3$ strict runs);
- `nRuns`, the number of *Runs*, subsequences of consecutive positions in the input with a positive gap between successive values, from beginning to end (e.g. $(1, 2, 6, 7, 8, 9, 3, 4, 5)$ is composed of $\texttt{nRuns} = 2$ runs);
- `nSSUS`, the number of *Strict Shuffled Up Sequences*, subsequences in the input with a gap between successive values exactly 1, from beginning to end (e.g. $(1, 5, 2, 6, 3, 8, 4, 9, 7)$ is composed of $\texttt{nSSUS} = 4$ strict shuffled up sequences);
- `nSUS`, the number of *Shuffled Up Sequences*, subsequences in the input with a positive gap between successive values, from beginning to end (e.g. $(1, 5, 2, 6, 3, 8, 4, 9, 7)$ is composed of $\texttt{nSUS} = 2$ shuffled up sequences);
- `nSMS`, the number of *Shuffled Monotone Sequences*, subsequences in the input with a positive gap between successive values, from beginning to end or from end to beginning (e.g. $(1, 9, 2, 8, 3, 7, 4, 6, 5)$ is composed of $\texttt{nSMS} = 2$ shuffled monotone sequences);
- `nInv`, the number $\texttt{nInv}(X) = |\{(i, j) : i < j \wedge x_i > x_j\}|$ of inversions (i.e., pairs in wrong order) in the input (e.g. $(2, 3, 4, 5, 6, 7, 8, 9, 1)$ has $\texttt{nInv} = 8$ inversions); and
- `nRem`, the minimum number of elements that must be removed from the input in order to obtain a sorted subsequence (e.g. $(2, 3, 4, 5, 6, 7, 8, 9, 1)$ needs only $\texttt{nRem} = 1$ removal to be sorted).

Moffat and Petersson [33] proposed a framework to compare those measures of presortedness, based on the cost function $C_M(|X|, k)$ representing the minimum

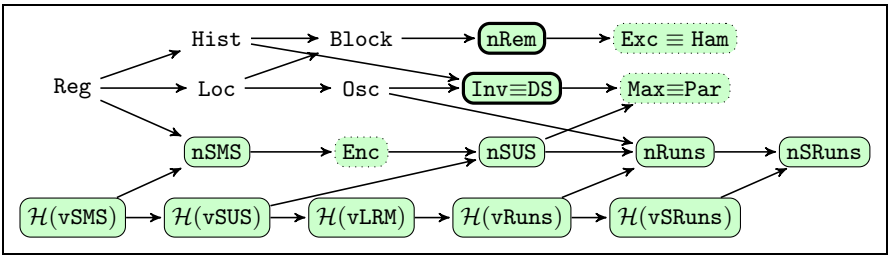


Fig. 1. Partial order on some measures of disorder for adaptive sorting, completed from Moffat and Petersson's survey [29] in 1992. A measure A dominates a measure B ($A \to B$ or $A \supseteq B$) if superior *asymptotically*. Round lined boxes signal the measures for which a compressed data structure supporting $\pi()$ and $\pi^{-1}()$ in sublinear time is known [3, 4, 7], round dotted boxes signal the results that can be inferred, and bold round boxes signal the additional compressed data structure presented in this article (see Section 4.3).

number of comparisons to sort a list X such that $M(X) = k$, where M is a measure of presortedness. Given two measures of disorder M_1 and M_2:

- $M_1 \supseteq M_2$, M_1 is *superior* to M_2 if $C_{M_1}(|X|, M_1(X)) = O(C_{M_2}(|X|, M_2(X)))$;
- $M_1 \supset M_2$, M_1 is *strictly superior* to M_2 if $M_1 \supseteq M_2$ and $M_2 \not\subseteq M_1$;
- $M_1 \equiv M_2$, M_1 and M_2 are *equivalent* if $M_1 \supseteq M_2$ and $M_2 \supseteq M_1$;
- M_1 and M_2 are *independent* if $M_1 \not\supseteq M_2$ and $M_2 \not\supseteq M_1$.

This organization yields a partial on those measures and the corresponding algorithms, such as the one given in Figure 1.

3 Encodings and Data Structures

3.1 Integers

In 1975, Elias introduced the concept of *universal* prefix-free codeword set [13], such that given any countable set M of messages and any probability distribution P on M, the codewords in order of increasing length yield a code set of average cost within a constant factor of the source entropy, for any alphabet B of size $|B| \geq 2$. In the binary case ($|B| = 2$), he discussed the properties of existing representation of integers such as unary (code α) and binary (code $\hat{\beta}$), and presented several new techniques of binary representations of integers (γ, γ', δ, and ω), showing that the δ and ω codes are asymptotically optimal universal.

The γ code of an integer i is constructed by inserting after each of the bit of the binary representation $\hat{\beta}(i)$ of i a single bit of the unary representation $\alpha(|\hat{\beta}(i)|)$ of the length $|\hat{\beta}(j)|$ of $\hat{\beta}(j)$. Dropping the first bit (always equal to 1) of $\hat{\beta}(i)$ yields a code of length $1+2\lfloor \lg i \rfloor$ for any positive integer $i > 0$. Reordering the bits of $\gamma(i)$ yields a variant of the code, $\gamma'(i) = \alpha(|\hat{\beta}(i)|).\hat{\beta}(i)$ with the same

length which "is easier for people to read" [13] and is usually referred to as the "Gamma Code". Both γ and γ' codes are universal but neither is asymptotically optimal, as the length of the codes stays at a factor of 2 of the optimal value of $\lfloor \lg i \rfloor$ suggested by information theory. The δ and ω codes below reduce this.

The δ code of an integer i is constructed in a similar way to the γ code, only encoding the length $|\hat{\beta}(i)|$ of the binary representation $\hat{\beta}(i)$ of i through the γ code $\gamma(|\hat{\beta}(i)|)$ instead of the unary code $\alpha(|\hat{\beta}(i)|)$, i.e. replacing $\alpha(|\hat{\beta}(i)|)$ by $\gamma(|\hat{\beta}(i)|)$. The δ code $\delta(i)$ of an integer i has length $|\delta(i)| = 1 + \lfloor \lg i \rfloor + 2\lfloor \lg(1 + \lfloor \lg i \rfloor) \rfloor$. This code is, this time, asymptotically optimal as the ratio $1 + \frac{1+2\lfloor \lg(1+\lfloor \lg i \rfloor) \rfloor}{\lg i}$ of its length to the information theory lower bound tends to 1 for large values of i. The same improvement techniques can be applied again in order to improve the convergence rate of this ratio.

The ω code is constructed by concatenating several groups of bits, the rightmost group being the binary representation $\hat{\beta}(i)$ of i, and each other group being the binary encoding $\hat{\beta}(l-1)$ of the length l less one of the following group, the process halting on the left with a group of length 2. Elias points out that there is a code performing even better than the ω code, but only for very large integer values ("much larger than Eddington's estimate of the number of protons and electrons in the universe").

3.2 Sets and Bit Vectors

Jacobson introduced the concept of succinct data structures encoding *Sets* and *bit vectors* [25] within space close to the information lower bounds while supporting efficiently some basic operators on it, as a constructing block for other data-structures, such as tree structures and planar graphs. Given a bit vector $B[1, \ldots, n]$ (potentially representing a set $S \subset [1..n]$ such that $\alpha \in S$ if and only if $B[\alpha] = 1$), a bit $\alpha \in \{0, 1\}$, a position $x \in [1..n] = \{1, \ldots, n\}$ and an integer $r \in \{1, \ldots, n\}$, the operator `bin_rank`(B, α, x) returns the number of occurrences of α in $B[1..x]$, and the operator `bin_select`(B, α, r) returns the position of the r-th label α in B, or n if none.

An index of $\frac{n \lg \lg n}{\lg n} + \mathcal{O}(\frac{n}{\log n})$ additional bits [19] support those operators in *constant time* in the $\Theta(\log n)$-word RAM model on a bit vector of length n. This space is asymptotically negligible (i.e. it is within $o(n)$) compared to the n bits required to encode the bit vector itself, in the worst case over all bit vectors of n bits, and is optimal up to asymptotically negligible terms among the encodings keeping the index separated from the encoding of the binary string [19]. As the index is separated from the encoding of the binary string, the result holds even if the binary string has exactly v bits set to one and is compressed to $\lceil \lg \binom{n}{v} \rceil$ bits, as long as the encoding supports in constant time the access to a machine word of the string [34].

3.3 Permutations and Functions

Another basic building block for various data structures is the representation of a *permutation* of the integers $\{1, \ldots, n\}$, denoted by $[1..n]$. The basic operators on

a permutation are the image of a number i through the permutation, through its inverse or through $\pi^k()$, the k-th power of it (i.e. $\pi()$ iteratively applied k times starting at i, where k can be any integer so that $\pi^{-1}()$ is the inverse of $\pi()$).

A straightforward array of n words encodes a permutation π and supports the application of the operator $\pi()$ in constant time. An additional index composed of n/t shortcuts [18] cutting the largest cycles of π in loops of less than t elements supports the inverse permutation $\pi^{-1}()$ in at most t word-accesses. Using such an encoding for the permutation mapping a permutation π to one of its cyclic representations, one can also support the application of the operator $\pi^k()$, the k times iterated application of operator $\pi()$, in at most t word-accesses (i.e. in time independent of k), with the same space constraints [31]. Those results extend to functions on finite sets [31] by a simple combination with the tree encodings from Jacobson [25].

3.4 Strings

Another basic abstract data type is the *string*, composed of n characters taken from an alphabet of arbitrary size σ (as opposed to binary for the bit vector). The basic operations on a string are to access it, and to search and count the occurrences of a pattern, such as a simple character from $[1..\sigma]$ in the simplest case [22]. Formally, for any character $\alpha \in [1..\sigma]$, position $x \in [1..n]$ in the string and positive integer r, those operations are performed via the operators `string_access`(x), the x-th character of the string; `string_rank`(α, x), the number of occurrences of α before position x; and `string_select`(α, r), the position of the r-th α-occurrence.

Golynski *et al.* [21] showed how to encode a string of length n over the alphabet $[1..\sigma]$ via n/σ permutations over $[1..\sigma]$ and a few bit vectors of length n; and how to support the string operators using the operators on those permutations. Choosing a value of $t = \lg \sigma$ in the encoding of the permutation from Munro *et al.* [31] yields an encoding using $n\big(\lg \sigma + o(\log \sigma)\big)$ bits in order to support the operators in at most $\mathcal{O}(\lg \lg \sigma)$ word accesses. Observing that the encoding of permutations already separates the data from the index, Barbay *et al.* [5] properly separated the data and the index of strings, yielding a succinct index using the same space and supporting the operators in $\mathcal{O}(\lg \lg \sigma \lg \lg \lg \sigma(f(n, \sigma) + \lg \lg \sigma))$ word accesses, if each word of the data can be accessed in $f(n, \sigma)$ word accesses. The space used by the resulting data-structures is optimal up to asymptotically negligible terms, among all possible succinct indexes [20] of fixed alphabet size.

A large body of work has further compressed strings to within entropy limits, culminating (or close) with full independence on the alphabet size from both the redundancy space of the compressed indexes and the time of support of the operators [8].

4 Fast Algorithms That Yield Compression Schemes

4.1 From Unbounded Search to Integer Compression

Bentley and Yao [9] mentioned that each comparison-based unbounded search algorithm A implies a corresponding encoding for integers, by memorizing the bit result of each comparison performed by A: simulating A's execution with those bits will yield the same position in the array, hence those bits code the position of the searched element. Their search algorithms are clearly inspired from Elias' codes [13] yet Bentley and Yao do not give explicitly the correspondence between the codes generated by their unbounded search algorithms and the codes from Elias. We remedy this in the following table:

Search Algorithm	cmps and bits	Encoding Scheme
binary	$\lfloor \lg \rfloor n + 1$	binary
sequential	$\lfloor p \rfloor + 1$	unary
B_1 [9]	$2\lfloor \lg p \rfloor + 1$	γ' [13]
B_2 [9]	$2\lfloor \lg \lg p \rfloor + \lfloor \lg p \rfloor + 1$	δ [13]
U [9]	$\sum_i \lfloor \lg^{(i)} p \rfloor + \lfloor \lg^{\lfloor \lg^* p \rfloor} p \rfloor + \lfloor \lg^* p \rfloor + 1$	ω [13]

Each of those codes is readily extendable to a compressed data structure for integers supporting algebraic operations such as the sum, difference, and product in time linear in the sum of the sizes of the compressed encodings of the operands and results. The most advanced codes γ, γ', δ and ω even support, by construction, the extraction of the integer part of the logarithm in base two, in time linear in the sum of the sizes of the compressed encodings of the operands.

4.2 From Merging Algorithms to Set and String Compression

Consider k sorted arrays of integers $A_1, \ldots, A_k$. A *k-Merging Algorithm* computes the sorted union $A = A_1 \cup \ldots \cup A_k$ of those k arrays, with no repetitions.

Ávila and Laber observed that any comparison-based merging algorithm yields an encoding for strings [1]. The transformation is simple: given a string S of n symbols from alphabet $[1..k]$ and a comparison-based merging algorithm M, define the k sorted arrays $A_1, \ldots, A_k$ such that for each value $i \in [1..k]$, A_i is the set of positions in S of symbol i. Running algorithm M on input $(A_1, \ldots, A_k)$ yields the elements from $[1..n]$ in sorted order. Let C be the bit string formed by the sequence of results of each comparison performed by M. Since the execution of M on $(A_1, \ldots, A_k)$ can be simulated via C *without any access to* $(A_1, \ldots, A_k)$, the bit vector C encodes the string S, potentially in less than $n\lceil \lg k \rceil$ bits.

Many merging algorithms perform sublinearly in the size of the input on particular instances: this means that some strings of n symbols from an alphabet $[1..\sigma]$ of size σ are encoded in less than $n\lceil \lg \sigma \rceil$ bits, i.e. that the encoding implied by the merging algorithm is *compressing* the string. Ávila and Laber focused on binary merging algorithms and the compression schemes they implied for bit vectors and sets (i.e. binary sources). They observed that Hwang and

Lin's binary merging algorithm [24] yields an encoding equivalent to using Rice coding [35] in a runlength-based encoder of the string; and that the `Recursive Merging` algorithm [2] yields an encoding equivalent to Moffat and Stuiver's `Binary Interpolative coder` [30]. Furthermore, they note that at least one merging algorithm [15] yields a new encoding scheme (probabilistic in this case) for bit vectors, which was not considered before!

Merging Algorithm	cmps and bits	Source Encoding Scheme
Hwang and Lin [24]	$n \lg(1 + \frac{m}{n})$	Rice coding+runlength [35]
`Recursive Merging` [2]	$n \lg(1 + \frac{m}{n})$	`Binary Interpolative` [30]
`Probabilistic Binary` [15]	$(m+n) \lg(\frac{m}{m+n})+0.2355$	Randomized Rice Code [1]

4.3 From Sorting Algorithms to Permutations Data Structures

Barbay and Navarro [7] observed that each adaptive sorting algorithm in the comparison model also describes an encoding of the permutation π that it sorts, so that it can be used to compress permutations from specific classes to less than the information-theoretic lower bound of $\lg(n!) \in n \log n - \frac{n}{\ln 2} + \frac{\log(n)}{2} + \Theta(1)$ bits. Furthermore they used the similarity of the execution of the `Merge Sort` algorithm with a wavelet tree [22], to support the application of the operator $\pi()$ and its inverse $\pi^{-1}()$ in time logarithmic in the disorder of the permutation π (as measured by `nRuns`, `nSRuns`, `nSUS`, `nSSUS` or `nSMS`) in the worst case. We describe below their results and some additional on additional preorder measures.

$\mathcal{H}$(`vRuns`)-Adaptive Sorting and Compression: The simplest way to partition a permutation into sorted chunks is to divide it into *runs* of consecutive positions forming already sorted blocks, in $n-1$ comparisons. For example, the permutation $(8, 9, 1, 4, 5, 6, 7, \mathbf{2}, \mathbf{3})$ contains $\texttt{nRuns} = 3$ ascending runs, of lengths forming the vector $\texttt{vRuns} = \langle 2, 5, 2 \rangle$.

Using a simple partition of the permutation into *runs*, merging those via a wavelet tree sorts the permutation and yields a data structure compressing a permutation to $n\mathcal{H}(\texttt{vRuns}) + \mathcal{O}(\texttt{nRuns} \log n) + o(n)$ bits in time $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})))$, which is worst-case optimal in the comparison model. Furthermore, this data structure supports the operators $\pi()$ and $\pi^{-1}()$ in sublinear time $\mathcal{O}(1 + \log \texttt{nRuns})$, with the average supporting time $\mathcal{O}(1 + \mathcal{H}(\texttt{vRuns}))$ decreasing with the entropy of the partition of the permutation into runs [7].

Strict-Runs-Adaptive Sorting and Compression: A two-level partition of the permutation yields further compression [7]. The first level partitions the permutation into *strict ascending runs* (maximal ranges of positions satisfying $\pi(i + k) = \pi(i) + k$). The second level partitions the *heads* (first position) of those strict runs into conventional ascending runs.

For example, the permutation $\pi = (8, 9, 1, 4, 5, 6, 7, 2, 3)$ has $\texttt{nSRuns} = 4$ strict runs of lengths forming the vector $\texttt{vSRuns} = \langle 2, 1, 4, 2 \rangle$. The run heads are

$\langle 8, 1, 4, 2\rangle$, which form 3 monotone runs, of lengths forming the vector $\texttt{vHRuns} = \langle 1, 2, 1\rangle$. The number of strict runs can be anywhere between $\texttt{nRuns}$ and n: for instance the permutation $(\mathit{6}, \mathit{7}, \mathit{8}, \mathit{9}, \mathit{10}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$ contains $\texttt{nSRuns} = \texttt{nRuns} = 2$ strict runs while the permutation $(\mathit{1}, \mathit{3}, \mathit{5}, \mathit{7}, \mathit{9}, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\texttt{nSRuns} = 10$ strict runs, each of length 1, and 2 runs, each of length 5.

$\mathcal{H}$(vSUS)-Adaptive Sorting and Compression: The preorder measures seen so far have considered runs which group contiguous positions in π: this does not need to be always the case. A permutation π over $[1..n]$ can be decomposed in n comparisons into a minimal number $\texttt{nSUS}$ of *Shuffled Up Sequences*, defined as a set of, not necessarily consecutive, subsequences of increasing numbers that have to be removed from π in order to reduce it to the empty sequence [26]. Then those subsequences can be merged using the same techniques as above, which yields a new adaptive sorting algorithm and a new compressed data structure [7]. For example, the permutation $(\mathit{1}, \mathbf{6}, \mathit{2}, \mathbf{7}, \mathit{3}, \mathbf{8}, \mathit{4}, \mathbf{9}, \mathit{5}, \mathbf{10})$ contains $\texttt{nSUS} = 2$ shuffled up sequences of lengths forming the vector $\texttt{vSUS} = \langle 5, 5\rangle$, but $\texttt{nRuns} = 5$ runs, all of length 2.

Note that it is a bit more complicated to partition a permutation π over $[1..n]$ into a minimal number $\texttt{nSMS}$ of *Shuffled Monotone Sequences*, sequences of not necessarily consecutive subsequences of increasing or decreasing numbers: an optimal partition is NP-hard to compute [27].

$\mathcal{H}$(vLRM)-Adaptive Sorting and Compression: LRM-Trees partition a sequence of values into consecutive sorted blocks, and express the relative position of the first element of each block within a previous block. They were introduced under this name as an internal tool for basic navigational operations in ordinal trees [36] and, under the name "2d-Min Heaps", to index integer arrays in order to support range minimum queries on them [16]. Such a tree can be computed in $2(n-1)$ comparisons within the array and overall linear time, through an algorithm similar to that of Cartesian Trees [17].

The interest of LRM trees in the context of adaptive sorting and permutation compression is that the values are increasing in each root-to-leaf branch: they form a partition of the array into sub-sequences of increasing values. Barbay *et al.* [4] described how to compute the partition of the LRM-tree of minimal size-vector entropy, which yields a sorting algorithm asymptotically better than $\mathcal{H}(\texttt{vRuns})$-adaptive sorting, and better in practice than $\mathcal{H}(\texttt{vSUS})$-adaptive sorting; as well as a smaller compressed data structure.

nRem-Adaptive Sorting and Compression: The preorder measures described above are all variants of `MergeSort`, exploiting the similarity of its execution with a wavelet tree: they are all situated on the same "branch" of the graph from Figure 1 representing the measures of preorder and their relation.

The preorder measure $\texttt{nRem}$ counts how many elements must be removed from a permutation so that what remains is already sorted. Its exact value is n minus

the length of the *Longest Increasing Subsequence*, which can be computed in time $n \lg n$, but in order to adaptively sort in time faster than this, $\texttt{nRem}$ can be approximated within a factor of 2 in n comparisons by an algorithm very similar to the one building a LRM-tree, which returns a partition of π into one part of $2\texttt{nRem}$ unsorted elements, and $n - 2\texttt{nRem}$ elements in increasing order. Sorting those $2\texttt{nRem}$ unsorted elements using any n-worst-case optimal comparison-based algorithm (ideally, one of the adaptive algorithms described above), and merging its result with the $n - 2\texttt{nRem}$ elements found to be already in increasing order, yields an adaptive sorting algorithm that performs $2n + 2\texttt{nRem} \lg(n/\texttt{nRem} + 1)$ comparisons [14, 29]. Similarly, partitioning π into those two parts by a bit vector of n bits; representing the order of the $2\texttt{nRem}$ elements in a wavelet tree (using any of the data structures described above) and representing the merging of both into n bits yields a compressed data structure using $2n + 2\texttt{nRem} \lg(n/\texttt{nRem}) + o(n)$ bits and supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time, within $\mathcal{O}(\log(\texttt{nRem} + 1) + 1)$.

$\texttt{nInv}$-Adaptive Sorting and Compression: The preorder measure $\texttt{nInv}$ counts the number of pairs (i, j) of positions $1 \leq i < j \leq n$ in a permutation π over $[1..n]$ such that $\pi(i) > \pi(j)$. Its value is exactly the number of comparisons performed by the algorithm `Insertion Sort`, between n and n^2 for a permutation over $[1..n]$. A variant of `Insertion Sort`, named `Local Insertion Sort`, sorts π in $n(1 + \lg(\texttt{nInv}/n))$ comparisons [14, 29].

As before, the bit vector B listing the binary results of the comparisons performed by `Local Insertion Sort` on a permutation π identifies exactly π, because B is sufficient to simulate the execution of `Local Insertion Sort` on π without access to it. This yields an encoding of π into $n(1 + \lceil \lg(\texttt{nInv}/n) \rceil)$ bits, which is smaller than $n\lceil \lg n \rceil$ bits for permutations such that $\texttt{nInv} \in o(n^2)$. Yet it is not clear how to support the operator $\pi()$ (yet even its inverse $\pi^{-1}()$) on such an encoding without reading all the $n(1 + \lg(\texttt{nInv}/n))$ bits of B: the bits deciding of a single value can be spread in the whole encoding.

But reordering those bits does yield a compressed data structure supporting the operator $\pi()$ in constant time, by simply encoding the n values $(\pi(i)-i)_{i\in[1..n]}$ using the γ' code from Elias [13], and indexing the positions of the beginning of each code by a compressed bit vector. Following the execution of `Linear Insertion Sort` algorithm over a permutation π over $[1..n]$ presenting $\texttt{nInv}$ inversions, the number of swaps of the i-th element $\pi(i)$ required to reach its final position in the sorted list is $\pi(i) = i + g_i^+(\pi) - g_i^-(\pi)$, where $g_i^+(\pi) = |\{j \in [1..n] : \ j > i \text{ and } \pi(j) < \pi(i)\}|$ is the number of swaps to the right; and $g_i^-(\pi) = |\{j \in [1..n] : \ j < i \text{ and } \pi(j) > \pi(i)\}|$ is the number of swaps to the left. By definition of $\texttt{nInv}$, g^+ and g^-, $\sum_{i=1}^n g_i^+(\pi) = \sum_{i=1}^n g_i^-(\pi) = \texttt{nInv}$, and by property of the γ' code, the number of bits used to store the values of $g_i^+(\pi)$ (or $g_i^-(\pi)$) is $\texttt{Gap}(g_i^+(\pi))_{i\in[1..n]} = \sum_{i=1}^n \lg g_i^+(\pi) \leq n \lg\left(\frac{\sum_{i=1}^n g_i^+(\pi)}{n}\right) = n \lg \frac{\texttt{nInv}}{n}$, by concavity of the logarithm. Since $\pi(i)-i = g_i^+(\pi)-g_i^-(\pi)$, the data structure uses $n+\texttt{Gap}((\pi(i)-i)_{i\in[1..n]})+o(n) \subset n+2n \lg \frac{\texttt{nInv}}{n}+o(n) = n(1+2 \lg \frac{\texttt{nInv}}{n})+o(n)$ bits.

Note that the compressed data structure described so far supports the operator $\pi()$ in constant time, which is faster than the compressed data structure described above, but not the operator $\pi^{-1}()$ (other than in at least linear time, by reconstructing π). By definition of $\texttt{nInv}$, the inverse permutation π^{-1} has the same number $\texttt{nInv}$ of inversions than π: the data structure for π^{-1} uses the same space as for π. Hence encoding both the permutations π and its inverse π^{-1} as described above yields a data structure using space within $2n(1+2\lg\frac{\texttt{nInv}}{n})+o(n)$ which supports both operators $\pi()$ and $\pi^{-1}()$ in constant time. This space is less than $n\log n + o(n)$ for instance where $\texttt{nInv}[0..n^{\frac{5}{4}}]$, but of course in the worst case where $\texttt{nInv}$ is close to n^2, this space is getting close to $8n\log n + 2n + o(n)$, a solution four times as costly as merely encoding in a raw form both π and its inverse π^{-1}.

Other Adaptive Sorting and Compression: As we observed in Section 2.3, Moffat and Petersson [29] list many other measures of preorder and adaptive sorting techniques. Each measure explored above yields a compressed data structure for permutation supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time. Figure 1 shows the relation between those measures, and the table below shows the relation between a selection of adaptive sorting algorithms and some permutation data structure (omitting both the $o()$ order terms in the space and the support time for the operators for sake of space).

Sorting Algorithm	cmps=bits	Permutation Data Structure
Natural MergeSort [23]	$n(1+\lg\texttt{nRuns})$	Runs [7]
BN-MergeSort [7]	$n(1+\mathcal{H}(\texttt{vRuns}))$	Huffman Runs [7]
$\mathcal{H}(\texttt{vSUS})$-Sort [7]	$2n\mathcal{H}(\texttt{vSUS})$	Huffman SUS [7]
$\mathcal{H}(\texttt{SMS})$-Sort [3]	$2n\mathcal{H}(\texttt{vSMS})$	Huffman SMS [7]
$\texttt{Rem}$-Sort (here)	$2n + 2\texttt{nRem}\lg(n/\texttt{nRem})$	$\texttt{Rem}$-encoding (here)
Local Ins Sort (here)	$2n(1 + 2\lg(\texttt{nInv}/n))$	$\texttt{Inv}$-encoding (here)

Note that all comparison-based adaptive sorting algorithms yield a compression scheme for permutations, but not all yield one for which we know how to support useful operators (such as $\pi()$ and $\pi^{-1}()$) in sublinear time.

5 Selected Open Problems

From Compression Schemes to Compressed Data Structures: In most current applications, compressed data is useless if it needs to be totally decompressed in order to access a small part of it. We saw how to support some operators on compressed data inspired from adaptive algorithms in the cases of permutations, integers and strings, but there are many other operators to study, from the iterated operator $\pi^k()$ on a compressed permutation π, non-algebraic operators on compressed integers, patern matching operators on compressed strings, etc...

From Compression Schemes to Adaptive Algorithms: Bentley and Yao [9] were already asking in 1976 if there are cases where such relations between compressed data structures and adaptive algorithms are bijective. Such a question comes naturally and applies to many other Abstract Data Types than integers or permutations, as both compression schemes and adaptive algorithms take advantage of forms of regularity in the instances considered. If a systematic transformation generating a distinct adaptive algorithm from each distinct compression scheme might not exist, at least one should be able to define a subclass of compression schemes which are in bijection with adaptive algorithms.

Other Compressed Data Structures for Permutations: Each adaptive sorting algorithm in the comparison model yields a compression scheme for permutations, but the encoding thus defined does not necessarily support the simple application of the permutation to a single element without decompressing the whole permutation, nor the application of the inverse permutation. Figure 1 represents the preorder measures for which opportunistic sorting algorithms are known, and in round boxes the ones for which compressed data structures for permutations (supporting in sublinear time the operators $\pi()$ and $\pi^{-1}()$) are known. **Are there compressed data structures for permutation**s, supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time and **using space proportional to the other preorder measures?** What about other useful operators on permutations, such as $\pi^k()$?

Sorting and Encoding Multisets: Munro and Spira [32] showed how to sort multisets through `MergeSort`, `Insertion Sort` and `Heap Sort`, adapting them with counters to sort in time $\mathcal{O}(n(1+\mathcal{H}(\langle m_1, \ldots, m_r\rangle)))$ where m_i is the number of occurrences of i in the multiset (note this is totally different from our results, that depend on the distribution of the lengths of monotone runs). It seems easy to combine both approaches (e.g. on `Merge Sort` in a single algorithm using both runs and counters), yet quite hard to *analyze* the complexity of the resulting algorithm. The difficulty measure must depend not only on both the entropy of the partition into runs and the entropy of the repartition of the values of the elements, but also on their interaction.

References

[1] Ávila, B.T., Laber, E.S.: Merge source coding. In: Proceedings of IEEE International Symposium on Information Theory (ISIT), pp. 214–218. IEEE (2009)
[2] Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
[3] Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. Algorithmica (to appear, 2013)
[4] Barbay, J., Fischer, J., Navarro, G.: LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. Elsevier Theoretical Computer Science (TCS) 459, 26–41 (2012)

[5] Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
[6] Barbay, J., Kenyon, C.: Deterministic algorithm for the t-threshold set problem. In: Proceedings of the 14th International Symposium Algorithms and Computation (ISAAC), pp. 575–584 (2003)
[7] Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: 26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009), vol. 3, pp. 111–122 (2009)
[8] Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. ACM Transactions on Algorithms (TALG) (to appear, 2013)
[9] Bentley, J.L., Yao, A.C.-C.: An almost optimal algorithm for unbounded searching. Information Processing Letters 5(3), 82–87 (1976)
[10] Burge, W.H.: Sorting, trees, and measures of order. Information and Control 1(3), 181–197 (1958)
[11] Carlsson, S., Levcopoulos, C., Petersson, O.: Sublinear merging and natural mergesort. Algorithmica 9(6), 629–648 (1993)
[12] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 743–752 (2000)
[13] Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)
[14] Estivill-Castro, V., Wood, D.: A survey of adaptive sorting algorithms. ACM Computing Surveys 24(4), 441–476 (1992)
[15] Fernandez de la Vega, W., Kannan, S., Santha, M.: Two probabilistic results on merging. In: Asano, T., Imai, H., Ibaraki, T., Nishizeki, T. (eds.) SIGAL 1990. LNCS, vol. 450, pp. 118–127. Springer, Heidelberg (1990)
[16] Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
[17] Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. STOC, pp. 135–143. ACM Press (1984)
[18] Gennaro, R., Trevisan, L.: Lower bounds on the efficiency of generic cryptographic constructions. In: IEEE Symposium on Foundations of Computer Science, pp. 305–313 (2000)
[19] Golynski, A.: Optimal lower bounds for rank and select indexes. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 370–381. Springer, Heidelberg (2006)
[20] Golynski, A.: Upper and Lower Bounds for Text Indexing Data Structures. PhD thesis. University of Waterloo (2007)
[21] Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 368–373. ACM (2006)
[22] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850. ACM (2003)
[23] Harris, J.D.: Sorting unsorted and partially sorted lists using the natural merge sort. Software: Practice and Experience 11(12), 1339–1340 (1981)
[24] Hwang, F.K., Lin, S.: A simple algorithm for merging two disjoint linearly-ordered sets. SIAM J. Comput. 1(1), 31–39 (1972)
[25] Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 549–554 (1989)

[26] Levcopoulos, C., Petersson, O.: Sorting shuffled monotone sequences. In: Gilbert, J.R., Karlsson, R. (eds.) SWAT 1990. LNCS, vol. 447, pp. 181–191. Springer, Heidelberg (1990)
[27] Levcopoulos, C., Petersson, O.: Sorting shuffled monotone sequences. Inf. Comput. 112(1), 37–50 (1994)
[28] Mannila, H.: Measures of presortedness and optimal sorting algorithms. IEEE Trans. Computers 34(4), 318–325 (1985)
[29] Moffat, A., Petersson, O.: An overview of adaptive sorting. Australian Computer Journal 24(2), 70–77 (1992)
[30] Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Inf. Retr. 3(1), 25–47 (2000)
[31] Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations and functions. Theor. Comput. Sci. 438, 74–88 (2012)
[32] Munro, J.I., Spira, P.M.: Sorting and searching in multisets. SIAM J. Comput. 5(1), 1–8 (1976)
[33] Petersson, O., Moffat, A.: A framework for adaptive sorting. Discrete Applied Mathematics 59, 153–179 (1995)
[34] Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
[35] Rice, R.F., Plaunt, J.R.: Adaptive variable-length coding for efficient compression of spacecraft television data. IEEE Trans. Commun. COM-19, 889–897 (1971)
[36] Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149. ACM/SIAM (2010)

Computing (and Life) Is All about Tradeoffs

A Small Sample of Some Computational Tradeoffs

Allan Borodin

University of Toronto
bor@cs.toronto.edu

Abstract. A pervasive theme in Compute Science (as in any science or for that matter in life) is tradeoffs. This is, of course, a recurring theme in many of Ian Munro's papers (e.g. in data structures, streaming algorithms). In one form or another, time space tradeoffs can be found in many settings. Another traditional research area concerns tradeoffs between performance (e.g. approximation bounds) vs complexity bounds. Newer areas of research consider various issues of tradeoffs involving concepts relating to fairness, privacy, conceptual simplicity, robustness, and self-interest (e.g. truthfulness). We will review some of the well established results as well as some of the many open questions involving tradeoffs.

1 Introduction

The title of this talk is not meant to presumptiously suggest that I will have anything useful to say about all the tradeoffs we face in our lives. Rather, I just want to put into perspective that tradeoffs in computation are to be expected and the fact that there are so many papers that implicitly or explicitly concern tradeoffs, is certainly not a surprise. In fact, this talk is even more narrowly focused than "tradeoffs in computation" in that I will only discuss tradeoffs that can be considered within the context of "theoretical computer science". By this, I mean that the discussion is limited to tradeoffs where there has been an attempt to quantify the concepts involved and establish provable results for such concepts. Even within this perhaps more modest goal, we will necessarily be limited to only a small subset of what is truly a pervasive theme in theoretical computer science. I apologize in advance for all glaring omissions.

I wanted to choose a topic that would have a substantial history as well as being current. But mainly, I wanted to choose a topic that would relate well to some of Ian Munro's research interests and contributions. As I will discuss, even though there is only one paper title in Ian's DBLP that contains the word "tradeoff", I would argue that his work is directly related to or has inspired a number of seminal tradeoff results, especially for tradeoffs in the area of data structures.

Just like someone in queuing theory who sees lines everywhere, when I started to think about this survey paper I began to see quantifiable tradeoff issues everywhere. Indeed, for any algorithm, one can usually formalize (in a meaningful way)

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 112–132, 2013.

a model capturing that algorithm and then quantify and measure the resources being used and the quality of the results obtained. This leads naturally to ask whether one can reduce one or more of the resources and the impact that would have on the quality. Similarly, for impossibility results, where a model needs to be explicitly stated a priori, one might have results that consider extreme cases (e.g. "no space" vs unrestriced space, deterministic vs randomized algorithms) and interpolate (e.g. for small space, for a limited number of random bits) to try to gain a better understanding of the inherent tradeoffs. The ideal goal is to have matching positive (i.e. algorithmic) and impossibility results, but this is relatively rare. Given the perceived difficulty of establishing "robust" impossibilty results in most areas of research, we tend to believe that our positive results are closer to "truth" than the weaker impossibility results. However, our religious belief (in the religion of theoretical computer science) is that our impossibility results often lead to new and unexpected algorithms improving upon the state of the art (and what may have also been conjectured). I think a good example of this phenomenon can be found in many of Ian's data structure results.

Unless otherwise stated, I will only consider worst case results. (However, some of the lower bounds mentioned also apply to random inputs.) My organization (and choice of examples) is arbitrary but still I wanted to give some sense of the diversity of tradeoff results in the world of theoretical computer science. I will try to provide the latest journal version reference for papers, although this may distort the time line of some results.

2 Complexity Theory: Time Space Tradeoffs

Since computer science is such a relatively new academic subject, calling any results classical is an abuse of language. But let's start with what might be called "classical issues" in complexity theory. Perhaps the most natural tradeoff issue is that of time vs space. Of course, time space issues are also prominent in many areas and in particular within the context of data structures. But here I am referring to time vs space within some general model of discrete computation. Unless otherwise noted, I am considering complexity theory when all inputs and outputs are encoded as strings over some finite (often binary) alphabet. With a slight abuse of notation, we will refer to this as Boolean complexity theory.

2.1 Time vs Space in Turing Machine Models

We first consider this issue within the context of a multi-tape Turing machine (TM) model. Unfortunately, the most basic questions are essentially completely open. It will be a refreshing contrast to see how results can be established within more structured models as in the theory of data structures.

The TM model is such a precise model that it immediately provides a definition of time, by which we mean sequential time. In order to study space, we usually assume a read only input tape and write only output tape and define space as the number of tape squares on the work tapes or the maximum length of

any work tape. We measure time $T(n)$ and space $S(n)$ as functions of the length n of the input and output and often focus on decision problems. Some fundamental issues immediately arise. What functions are computable in space $S(n)$ and/or time $T(n)$? Unfortunately, given the current state of complexity theory, we do not know how to *prove* that a given "explicit" function (say a decision problem within the class NP) *cannot* be computed in space $S(n) = O(\log n)$ or in time $T(n) = O(n \log n)$.

It is immediate that a problem computable in space $S(n) \geq \log n$ is also simultaneously computable in time $c^{T(n)}$ for some constant c depending on the cardinality of the work tape symbols. But when is such an exponential blow up in time required, especially if we wish to maintain small space? In this regard, let me mention one fundamental open question. Namely, the STCON problem (does there exist a directed path from s to t) is computable in linear time and linear space by say breadth or depth first search and, by Savitch's theorem [87] in space $O(\log^2 n)$. But can we compute STCON simulataneously in space $O(\log^2 n)$ and time that is substantially less than $c^{\log^2(n)} = n^{O(\log n)}$ and, in particular, can we achieve $\log^2(n)$ space and polynomial time simutaneously or must there be a tradeoff? The general problem is whether all problems computable in polynomial time and also in polylog space are computable simultaneously in polynomial time and polylog space or must there be tradeoffs?

In the terminology of complexity theory, the class of decision problems simultaneously computable in time $T(n)$ and space $S(n)$ is called TISP(T, S). The question above asks whether or not the inclusion TISP$(poly, polylog) \subseteq P \cap polylogspace$ is proper and the conjecture is that the inclusion is proper. Since such a grand challenge question does not seem within current reach, for what bounds T and S can we prove negative results about TISP(T, S)? In this regard, Cobham [30] showed that even for a simple function such as deciding if an input string w is a palindrome requires $TS = \Omega(n^2)$. But one can easily argue that such a result is more about the restrictive sequential nature of accessing the input tape than about the complexity of the problem.

2.2 The Branching Program Model

There is a substantial history of time space tradeoffs that go beyond the limitations of the basic TM model including results on k head Turiing machines (e.g. [37,60]). One thread of results concerns the time space complexity of sorting and the related decision problem of determining if a set of elements are all distinct. This thread begins with a more structured model, namely that of comparison branching programs as studied in [18]. This model extends the comparison tree model where nodes indicate pairwise comparisons. (This is, of course, the basic model in which we show the $n \log n$ lower bound for the number of comparisons to sort n elements.) In braching programs, the underlying model is a DAG, rather than a tree with nodes again indicating pairwise comparisons. Now different computation paths can lead to the same node. Each node in such a branching program represents a state of the computation and space is defined as the log of the number of states. This is a purely information theoretic way to define space

and in some sense this model can be considered as the ultimate comparison based model for studying space or time space lower bounds. For sorting, edge labels in the branching program are used to indicate ranks for elements of the input. For element distinctness, edges are unlabelled and leaves are are labelled by accept or reject. Borodin et al [18] show that $TS = \Omega(n^2)$ for sorting and Yao [94] shows that $TS = \Omega(n^{2-O(1/\sqrt{\log n})})$ for element distinctness following a weaker bound of $TS = \Omega(n^{3/2})$ in [16]. More recently, Chan [28] proves that time $T = \Omega(n \log \log_S(n))$ for the median (and other selection problems). The Chan lower bound corresponds to positive time space results for selection as will be discussed in section 5.1.

Returning to "Boolean complexity", the comparison branching program is extended by Borodin and Cook [15] to a model called R-way branching programs where now inputs are integers in the range $[1, R]$ and each node queries a specific input and branches R-ways according to the value of that input. They show a lower bound of $TS = \Omega(n^2/\log n)$ in this model for sorting integers in the range $[1, n^2]$. (We ignore the additional $O(\log n)$ cost to read each integer.) Beame [10] shows $TS = \Omega(n^2)$ for a slightly different variant of sorting and that this bound can be met exactly for all space bounds $\log n \leq S(n) \leq n$. This is an example of a tight time space tradeoff, where provably it is shown precisely how time must increase as space decreases. Such tight results do not exist for Boolean *decision problems* with respect to a general model. However, there has been a significant sequence of results initiated by Ajtai [3] establishing some limited but non trivial time space lower bounds for Boolean decision problems showing that non linear time is required for sublinear space. I believe the best lower bound in this regard with respect to a general branching program is the Beame and Vee [12] result for a specific decision problem that requires time $T = \Omega(n \log^2 n)$ when the space $S = n^{1-\epsilon}$ for any $\epsilon > 0$.

Before leaving the topic of time space tradeoffs, we note that the branching program models (like circuit models) are non uniform models. That is, for every n, one can have a different algorithm. Starting with the work of Nepomnjaščiĭ [74] and Fortnow [45] there are some very interesting TISP lower bound results for uniform models that allow random access of the input. Specifically, there is a sequence of TISP results for the NP hard problem SAT. This culminates thus far in the paper of Williams [91] where SAT is proven not to be in $\mathrm{TISP}(n^c, n^\epsilon)$ for various values of $c < 2$ and $\epsilon < 1$. The best lower bound in [91] is $c = cos(\pi/7) \approx 1.8019$. It is conjectured in [91] that this is optimal for the proof technique of *alternation trading proofs*, an indirect diagonalization method based on simulating nondeterminism for time and space bounded deterministic algorithms. Surprisingly, Buss and Williams [23] show that this "strange constant" conjecture is indeed optimal for the proofs based on such alternation trading.

3 Complexity Theory: Parallel vs Sequential Complexity

Booelan circuits are a basic (non-uniform) model in which there are precise analogs of parallel (circuit depth) and sequential complexity (circuit size).

Pippinger [82] defined the compleixty classes NC^k and their union $NC = \cup_{k\geq 0} NC^k$. NC is the class of problems that can be computed by circuits within simultaneous polylog depth and polynomial size. This is analogous to but different from the class $TISP(poly, polylog)$ discussed in section 2.1. Uniform versions of these classes arise when one considers various PRAM models. The basic Booelan circuit and complexity theory question is whether or not every problem computable in polynomial (TM) time or polynomial (circuit) size can be computed in NC or ignoring the size constraint whether or not all polynomial time problems can be computed in polylog depth. It is widely conjectured that this is not the case but currently it is not even known that polynomial time is not contained in NC^1 which is the same as the class of problems having polynomial size formulas. The immediate tradeoff question is how much must size suffer in order to achieve some amount of parallelization.

There is also an arithmetic version of circuits where the basic binary operations are $+, -, \times$. Such circuits compute polynomials over some underlying ring. (The arithmetic circuit model can also include a division operation for computing polynomials or rational functions.) This leads to arithmetic analogs of parallel vs sequential complexity. One early result in this regard is the result of Munro and Paterson [68] that establishes a very tight (within one operation) size bound for the evaluation of a univariate polynomial in terms of the depth of the arithmetic circuit. For multivariate polynomials, we consider the arithmetic versions of the NC^k and NC classes. It is easy to see that circuits of depth d computing a multivariate polynomial have size and degree at most 2^d. A reasonable analogue of the above poly size vs NC question is then whether every polynomial $P(x_1, \ldots, x_n)$ of polynomial degree (in n) that is computable by polynomial size arithmetic circuits is computable in the arithmetic analogues of NC or even in some fixed NC^k. Surprisingly, such polynomials are all computed within the class NC^2 as shown by Valiant et al [89] following a previous result by Hyafil [55] showing that all such polynomials are computable in $O(\log^2 n)$ depth but with size $n^{O(\log n)}$. More specifically, Valiant et al show that every polynomial of degree d that is computable in size c is also computable simultaneously in depth $O((\log c) \cdot (\log d))$ and size $O(c^3)$. It is open if depth $O(\log c + \log d)$ is possible.

4 Randomization vs Time and Accuracy

The previous section only considered deterministic algorithms. Of course, a central question in complexity theory (as well as any of the topics mentioned in this paper) is the power of randomization. In some problem settings, randomization is necessary to achieve any meaningful results (e.g. for interactive proofs, probabilistically checkable proofs, cryptography, sublinear time and in many problems within the context of streaming algorithms and online algorithms.). In the complexity theory setting of the previous section, fundamental questions are formulated in terms of an unrestricted version of randomization comparing the complexity of deterministic vs randomized computation. The natural quantification is to consider how many random bits are needed to achieve a stated goal, say a reduction in space or time, while insuring some probability of correctness.

The fundamental complexity theory question is whether more can be done in randomized polynomial time than in deterministic polynomial time. In terms of decision problems, the power of randomization question in complexity theory asks whether P is a proper subset of complexity classes ZPP, RP, BPP. Perhaps the most notable relevant example is the symbolic determinant problem (i.e. is $det(A) \not\equiv 0$ where the entries of matrix A are multivariate polynomials). This problem is easily seen to be in the class RP but not known to be in P. While this question is beyond our current reach, it is interesting to note that some research (see Impagliazzo and Wigderson [56]) surpisingly brings into question what would have been the prevailing view that P is a proper subset of RP. Even if it turns out that say $P = BPP$, there will inevitably be a question as to the tradeoffs between error probability and time.

Consder the problem of primality testing. The question of primality testing is in some sense solved as a complexity theory question by the deterministic polynomial time algorithm of Agarwal, Kayal and Saxena [2]. However, from the view of actual practice, randomized primality testing as initiated independently by Solovay and Strassen [88] and Rabin [83] are much more efficient. I am not aware of any results quantifying a tradeoff between randomness or the error probability and the time complexity of the algorithm.

But there are many areas of research where the power of randomness is better understood and quantified. One such example (see section 6) is the tight tradeoff between randomness and bits communicated as shown in Canetti and Goldreich [24]. Other examples include tradeoffs in oblivious routing in Krizanc, Peleg and Upfal [62]. In the next section, we briefly consider the streaming model where a number of results rely on randomization.

5 The Streaming Model and Other Space Restricted Models

Given the large volumes of data (for example, in analyzing internet routing data), it is often infeasible to store the data and one is forced to process the data in a stream using relatively limited space. Well before the model was being used for such relatively modern applcations, Munro and Paterson [69] introduced the streaming model where data items stream by in one or more passes. Their goal was to study the space complexity of selecting the k^{th} largest number (from an input steram of n distinct numbers) when the input is being processed by some fixed number of passes. The most interesting case is for computing the median, that is when $k = \lceil n/2 \rceil$. They show the following nearly matching results. There is a p pass algorithm that selects the k^{th} largest element using memory $O(n^{\frac{1}{p}}(\log n)^{2-\frac{2}{p}})$. On the other hand, for any k such that $\Omega(n) = k \leq n/2$, any deterministic p pass streaming algorithm for selecting the k^{th} largest element requires space $\Omega(n^{\frac{1}{p}})$.

The streaming model is one of a number of space restricted models that are studied for particular settings in contrast to the general models considered in section 2. See also the specific results concerning data structures in section 8.1.

5.1 Multi-pass and RAM Comparison Based Models

The Munro and Paterson results led to a number of positive results for selection and other problems. One of the early surprising results of complexity analysis was the Blum et al [14] linear time algorithm for selection. I say surprising in that there were conjectures stating that $n \log n$ comparison were necessary in a deterministic comparison computation tree and indeed the linear time algorithm was derived following attempts to establish a lower bound. (See my comment in the Introduction as to our "religious beliefs" about the value of impossibility results.) The Munro and Paterson selection algorithm can be interpreted in terms of space vs. time (say comparisons) and a natural question is to study time bounds for selection when space is limited. Extending the algorithm in [69], Frederickson [49] gave an algorithm that achieved time $O(n \log^* n + n \log_S n)$ in a comparion based, read-only input RAM model for space satisfying $\Omega(\log^2 n) = S(n) = O(n/\log n)$. Here space means the number of additional registers. Chan [28] noted the bound in [49] could be reduced to $O(n \log^*(n/S) + n \log_S n)$ which then yields the Blum et al time bound for unrestricted space. Frederickson's result can be interpreted as a p pass, S space streaming algorithm achieving almost linear time $O(n \log^{(p)}(n/S))$ where $\log^{(p)}$ is the p^{th} iterated logarithm. Munro and Raman [71] and, respectively, Raman and Ramnath [85] derive time bounds for smaller space bounds; specifically time $O(n^{1+\epsilon})$ for constant space (with $\epsilon > 0$ arbitarility small depending on the space) and (respectively) time $O(n \log^2 n)$ for space $O(\log n)$. The Munro and Raman paper also considered the time for a randomized algorithm (or for a random ordering of the inputs) and derived the improved time bound $O(n \log \log_S n)$ for all space $S = \omega(1)$. Hence linear time can be achieved with space n^δ for any $\delta > 0$. Chan's [28] lower bound (as mentioned in section 2) shows that this time bound is optimal for $S = \omega(\log n)$.

5.2 Streaming Space vs Approximation

The most prevalent use of the streaming model is to gather statistics on the stream of data. The desired goal is to use a small amount of time (say even O(1) time) per data item and relatively small space (perhaps $\log(n)$ or polylog space). Often the data must be sampled and the natural goal is to study the tradeoff between the space required and the probabilistic confidence in the statistical approximation being estimated. The seminal paper in this regard is by Alon, Matias and Szegedy [5] who gave several results concerning the computation of frequency moments by a one pass streaming algorithm. The one pass streaming model has generated a mini industry of papers for computing different statistical properties of an input stream, and in particular for what are called frequent item queries (e.g. top k queries, threshold queries) An excellent treatment of the now classic one pass streaming model can be found in the survey by Muthukrishnan [73]. Moreover, variants of the model have also been introduced to capture other requirements in large data streams (see Datar et al [32] and Golab et al [52] for results concerning the sliding window model) and Demaine et al [33] for a number of deterministic and randomized tradeoffs in a more structured streaming

model in which counters are explicit in the model. Given important applications such as estimating statistical properties of packet streams, the precise relation between all the relevant paprameters (space, randomness, time per input item, approximation and error bounds) continues to attract attention.

5.3 Priority Branching Trees and Programs: Space vs Approximation

Motivated by the study of the power of simple dynamic programming and simple back-tracking algorithms for optimization and search problems, Alekhnovich et al [4] study the priority branching tree pBT model. Similar to computation trees, nodes represent input items that are being queried and based on the value of the item, the computation branches. But in contrast, this is a uniform model of computation where the computation unfolds as more and more input items are queried. The essense of the model is to prescibe how the algorithm is allowed to choose the next item to look at. A considerably stronger dynamic programming model is studied in Davis, Impagliazzo and Buresh-Oppenheim [22] where they define priority branching programs (pBPs), a DAG extension of pBTS. The issue in [4,22] as it has been studied thus far mainly studies the width of pBTs and pBPs (which can be viewed as space) required for achieving optimality in these models. But in doing so, there are some very limited results that address the interplay between program width (or the total number of nodes in the pBT or pBP) and the quality of the approximation.

6 Communication Complexity

One of the most important theoretical models is the two party communication model introduced by Yao [93] and its extension to mutli-party communciation and the number on forehead (NOF) model in Chandra, Furst and Lipton [29]. These models play a pivotal role in many if not most aspects of theoretical computer science. The model also lends itself directly to many natural tradeoff questions. In the basic two party model, A and B each hold half of an n bit input w and they alternate sending messages so as to compute a function $f(w)$.

Perhaps the most basic tradeoff in communciation complexity is the tradeoff between rounds and bits communicated (i.e. the communication complexity). Papadimitriou and Sipser [79] considered a pointer chasing problem (to find the k^{th} pointer) that can be easily solved in k rounds and communication complexity $k \log n$ but showed that any 2 round protocol would require $\Omega(n)$ bits of communication. Duris, Galil and Schnitger [38] extend the lower bound to show that $\omega(n/k^2)$ communication is necessary for any $k-1$ round protocol thus establishing a relatively tight tradeoff. Similar exponential gaps between $k-1$ and k round communcaition complexity were established for randomized protocols by Halstenberg and Reischuk [53], and Nisan and Wigderson [76]. Impagliazzo and Williams [57] consider a variant of the basic model allowing synchronized clocks (which allow processors to only pass messages during certain rounds thereby

allowing the equality function to be solved with a single bit of communication but using 2^n rounds). They establish strong tradeoff results between rounds and communication in this model.

Another notable example of tight tradeoffs in communciation complexity is the tradeoff between randomness and bits communicated in Canetti and Goldreich [24]. They consider and establish tight tradeoff results in four models, depending on whether the random bit are private or public (i.e. shared by both parties) and whether the communication is one-way or two-way. With the exception of the one way, public coin model, two lower bounds are derived, for the other three models and which of the two is more meaningful depends on the determinitsic communication complexity. In particular, the more meaningful bound for small values of m (for the worst case communication cost over all input instances and coin tosses) in the two-way public coin model is as follows: for any non-degenerate n by n communication problem (expressed as a $2^n \times 2^n$ matrix with distinct rows and distinct columns), any protocol must satisfy $r \geq \log(\frac{n}{(1-\epsilon)2^m})$ where r is the number of random bits used and ϵ is the advantage in accuracy over probability $\frac{1}{2}$ correctness. That is, for small values of m, $\Omega(\log n)$ random bits are needed and it can be shown that $O(\log n)$ is always sufficient while maintaining the same communciation cost but possibly cutting the advantage to $\epsilon/2$.

The communication complexity setting also provides for tight results concerning the tradeoff between privacy and bits communcated. The study of information-theoretic privacy (i.e. independent of computational complexity assumptions) was begun in Kushilevitz [63] and led to the study of privacy vs communciation results in Feigenbaum, Jaggard and Schapira [40]. The following development is another success story for tradeoff results. Feigenbaum et al. define the privacy approximation ratio (PAR) to measure the loss of privacy in a communication complexity protocol. They study the PAR in both a worst case sense (over all input pairs) and average case sense for (integral valued) Vickery auctions in which two parties $\{A, B\}$ must arrive at a winner revealing the bid of the losing party. If A and B both have n bit integer bids, then the PAR ranges between 1 (perfect privacy) and 2^n (revealing the value of the winner). Kushilevitz showed that PAR = 1 (perfect privacy) requires exponential communciation (and is achievable by the ascending English auction). Feigenbaum et al. describe a binary search protocol that achieves a spectrum of privacy vs communciation tradeoffs and in particular show that linear communication (which is necessary) can be acheived with an exponential PAR. Ada et al. [1] then show the tightness of the positive tradeoffs established in [40] for Vickery auctions. More specifically they show that for every n and p with $2 \leq p \leq n/4$, every deterministic protocol obtaining worst case PAR $< 2^{p-2}$ must have communication complexity at least $2^{\frac{n}{4p}}$. Like the time space tradeoffs in proof complexity (see section 9), and in contrast to what can be proven in computational complexity, these results yield super linear lower bounds. In particular, any deterministic protocol must either have communciation complexity or PAR at least $2^{\Omega(\sqrt{n})}$. Similar tradeoff limitations are shown for average case PAR.

7 Distributed Computing

Distributed computation is another area where tradeoffs are very natural and multi-faceted. Obviously, distributed systems is a field in itself and one that is of growing importance with the platforms for cloud computing. I refer the reader to the Attiya and Welch [8] text and the Fich and Rupert [43] survey of impossibility results. I will retrict attention to synchronous message passing and shared memory models.

For synchronous message passing models, some of the measures of interest are rounds of communication, number and size of messages, and the number of possible faulty processes. As just one example of tradeoffs in this area, let me mention what is one of the most fundamental problems and results in distributed computing. Namely, consider the problem of processes trying to achieve consensus in the presense of possibly faulty processors. Consensus means that all non faulty processors return the same value and that value is held by at least one of the processes (possibly faulty). There is a spectrum of different types of failures. The easiest type of failure to accomodate are crash faults where the processes cease to operate at some point during the computation. The most difficult type are Byzantine faults where a process may be maliciously providing false information during a computation including misreporting their identity. For Byzantine failures, consensus also means that if all processes hold the same value, then the nonfaulty processes must return that value. The consenus problem was introduced and initially studied in the Byzantine fault model by Pease, Shostak and Lamport [80]. Letting n be the total number of processes, and f, the maximum number of faulty processes, there are two basic tradeoffs (depending on the type of failures) as follows: for $n \geq f+2$ (resp. $n \geq 3f+1$) in order to achieve consensus, $f+1$ rounds of communication are necessary and sufficient in a system with at most f crash failures (resp. Byzantine failures). The positive result for Byzantine faults is proven in the defining paper by Peasse, Shostak and Lamport. The negative result for Byzantine faults is due to Fischer and Lynch [44]. The results for crash failures are due to Dolev and Strong [34] who consider the stronger model of Byzantine faults but with authenticaion (so that process identity can be verified). Similar tradeoffs between the number of rounds vs number of faults have been proven for other types of failures (in between crash and Byzantine) and for other distributed problems (e.g. terminating reliable boradcast).

For shared memory models, one is often interested in the difficulty of implementing a given object in terms of the atomic or primitive operations that are allowed in the model. For example, a *snapshot object* stores some m component values and supports two operations, a *scan* (which is a consistent view of all m values) and an *update* of any of the m components. For the synchoronous shared memory model, Brodsky and Fich [20] consider the implementation of a snapshot in terms of the number of atomic read/write operations required. They show an asymtotically tight tradeoff for implementing a snapshot object amongst n processes. Namely, $T_u = \Theta(\log(\min\{m, n\}/T_s))$ where T_u is the time (i.e. number of atomic operations) for an update and T_s is the time for a scan.

8 Tradeoffs in Data Structures

Data structures (both static and dynamic) provide one of the most active areas for tradeoff results. Indeed, this important area naturally raises multi-faceted tradeoff results relating to the relation of space, preprocessing time, update time, and query time (with respect to tradeoffs bewteen a variety of query requests). It is also an area in which Munro and his students have a number of seminal results especially in the area of data structures allowing only limited space.

8.1 Implicit and Semi-implicit Data Structures

In their seminal paper, Munro and Suwanda [72] defined and studied algorithms for *implicit data structures* and semi-implicit data structures. The underlying computational model for Munro and Suwanda (and many subsequent papers) allows comparisons and data and pointer movements. Time is measured in terms of the number of comparisons and moves. The results here do not apply to models with more general operations (as in the cell probe model discussed in the next subsection) that can exploit properties of the data. Implicit data structures (resp. semi-implicit data structures) store elements in an array and use only a constant number of pointers (or more religiously no pointers as in a heap data structure for a priority queue) and constant (or no) additional space beyond the number of elements of the array. Simply stated, an implicit data structure can be viewed as a set of allowable permutations along with methods for search and update. Semi-implicit data structures allow some sublinear number of pointers or additional amount of space and hence provides a model for studying space vs performance. Within the implicit data structure model (or any data structure model), the goal is to study tradeoffs between the operations required by the data structure.

Perhaps the most studied and basic data structure problem is the dictionary where queries are simply to search for the existence of an element (or a key which in turn has an associated value) in a given set. In the case of dynamic dictionaries, we also have insertions and deletions. One way to realize an implicit dictionary is to insure that the contents of the dictionary array are constrained to satisfy some partial order, examples being unsorted lists, sorted lists and heaps. To keep the size n of a dynamic dictionary fixed (for the purpose of the lower bound), updates are replaced by value changes (which can be realized by a deletion followed by an insertion). Munro and Suwanda show that under the partial order contstraint, the product of update and search time is $\Omega(n)$. They show that this product can be realized by a triangular grid organization resulting in $O(\sqrt{n})$ time for both search and updates. They then go beyond the partial order constraint and combine rotated lists with the triangular grid scheme so as to achieve $O(n^{1/3} \log n)$ time for search and update using $O(\log n)$ additional memory.

The Munro and Suwanda paper initated an active research area. Frederickson [48] created an improved implicit data structure for dictionairies, and then Munro [67] provided what was conjectured to be the best possible upper bounds

for implicit dictionaries achieving $O(\log^2 n)$ for both search and update. This conjecture was disproved in the work of Franceschini, et al [46] who were able to achieve time $O(\log^2 n/\log\log n)$ for both search and update. This seems to be the current best implicit dictionary upper bound. Recalling that implicit data structures are structures characterized by the set of allowable permutations, Borodin et al [17] gave a lower bound on the number of comparisons for search in terms of the number of moves and comparisons for an update in any implicit dynamic dictionary. One corollary of this lower bound is that any implict dictionary with constant update time (i.e. moves plus comparisons) requires $\Omega(n^\epsilon)$ search time for some $\epsilon > 0$. This lower bound on update time was improved to $\Omega(n)$ by Radhakrishnan and Raman [84]. Surprisingly (and contraducting a conjecture in [17]), Franceschini and Munro [47] show that it is possible to achieve $O(\log^3 n)$ search time using only $O(1)$ exchanges (and $O(\log^3 n)$) comparisons in an implicit data structure using only a constant amount of additional memory. There is still a significant gap in the upper and lower bounds for implicit dictionaries; in particular, whether they can achieve the same $O(\log n)$ time that is acheived by balanced search trees. And, more generally, how much additional space is required to achieve the best possible tradeoffs between search and update.

One of the most impressive implicit data structures is for the multi-key search problem where n items can be searched according to any one of k keys. Fiat et al provide an implicit data structure for this problem that uses on $O(k \log k \log n)$ comparisons. For fixed k, this $O(\log n)$ comparison method stands in contrast to the Alt, Mehlhorn and Munro [6] lower bound of $\Omega(n^{1-\frac{1}{k}})$ for searching a multi-key table when all comparisons are restricted to be against the search key. Beyond dictionaries, we note that implicit data structures and semi-implicit data structures have been studied for many different data structure problems (e.g. [6,70,25,21]).

8.2 The Cell Probe Model

The cell probe model of Yao [93] is arguably the most general data structure model. It can simulate RAM algorithms over any instruction set and plays the same information theoretic abstraction for data structure problems as does the R-way branching program model discussed in section 2.2. Lower bounds in this model therefore apply generally and upper bounds can be viewed as a starting point for a possible realistic imnplementation. We will simplify the discussion here and just mention some cell probe model results for membership search and the nearest neighbor problem. But in its generality, the cell probe model has probably been used to prove results for every natural data structure problem imagineable. (See Gál and Miltersen [50] for a spectrum of results indicating the generality of the model.) As the name suggests, data itema are encoded in, say, s cells of some word size w and queries are realized by a 2^w way tree probing a cell at each node and branching according to the value of the cell. The query time t is the depth of the computation tree. In the basic membership search problem we have a set of $S = \{x_1, \ldots, x_n\}$ with each x_i in some universe U, say $\{0,1\}^m$, and a query is to simply determine if a given y is in S. For such

problems, it is often assumed that $w = \Theta(m)$. To further refine the results, it is often desireable to state "transdichotomous" bounds that are a function only of n, holding for all values of m. The model immediately raises tradeoffs between the space s and the query time t for different queries. In dynamic problems, updates are allowed and the computation tree allows for a probed cell to be changed. This induces futher tradoffs. For static problems, one can also measure the preprocessing cost to set up the data structure and the tradeoffs between preprocessing, space and query costs. Needless to say, the cell probe model is a test bed for a variety of interesting tradeoff questions. Yao's persuasive question was "should tables be sorted" to answer a membership query. Yao answered that question for the implicit cell probe model (no extra cells) in the affirmative by showing that $\lceil \log(n+1) \rceil$ probes are necessary when the set of possible elements $S = \{1, \ldots, m\}$ is sufficiently large. Curiously, with one extra cell, 2 probes are sufficient when m is sufficently large. Miltersen and Fich [42] give a more precise answer as to when $\Omega(\log n)$ time is necessary for a particular RAM model.

Another widely studied problem studied within the context of the cell probe model is the nearest neighbor search (NNS) problem. In the NNS problem, there is a set S of n points in a d-dimensional Euclidean space. The search problem is to find the nearest point in S to a given query point x. The problem is interesting for static and dynamic data bases, for small and large dimensions d, for discrete and continuous spaces, and for deterministic and randomized algorithms. The goal is to find a nearest neighbor exactly or just to obtain a good approximation to the nearest neighbor (i.e. for some given ϵ to find a point $y \in S$ so that $||y - x|| \leq (1+\epsilon) \cdot ||z - x||$ for all $z \in S$). There is also a decision problem variant (to determine if there is a $y \in S$ within some given distance λ to the search query x) as well as the special case of partial match queries. In the static high dimensional problem, there are time space tradeoff questions relating the time for a query vs the number and size of cells used to store the data base. For example, if the data base $S = \{0,1\}^d$, then two extremes are either to use just nd bits and search exhaustively (with search time nd) or to store an answer to each possible query using 2^d bits but using only $O(d)$ bit operations for the search time. The question becomes whether one can, say, simlutaneously achieve space which is polynomial in nd and search time $O(d)$ (or even polynomial in d). The question is mainly of interest when d is asymptotically larger than $\log n$ and smaller than n^δ for all $\delta > 0$. The conjecture is that there is a curse of dimensionality for the exact NNS problem in that either the space or search time must be exponential in d. For the approximate NNS, the curse of dimensionality has been traded for a curse of approximation bound in the randomized algorithms of Indyk and Motwani [58], and Kushilevitz, Ostrovsky and Rabani [64]. Some cell probe lower bounds have appeared for deterministic and randomized algorithms for both the exact and approximate NNS problem ([19,27,9]) but the question of precise tradeoff results for NNS problems remains largely open.

9 Proof Complexity: Time vs Space Revisited

Initiated by Cook and Reckhow [31], proof complexity was motivated by the "NP = co-NP? " question , and the natural questions relating to the relative power of various proof systems (in terms of the lengths of proofs). In addition, there are interesting time space tradeoffs that have been studied for various proof systems. Such tradeoffs directly impact DPLL-based SAT solvers, since memory and time are often both bottlenecks. Perhaps the most studied proof system is resolution refutation, which is also at least implicitly involved in all DPLL-based solvers. For resolution proofs, the usual notions of time is length (i.e. number of resolution steps) of the proof or the *size* defined as the total number of clauses in the proof. The notion of resolution space is usually given by the maximum (over all resolution steps) number of clauses that are *active* (i.e. derived by some step t but still needed at some step $t' \geq t$).

As in the branching program model, linear space is always achieveable in resolution proofs. On the other hand, the conjecture is that natural CNF formulas (encoding simple mathematical statements) will require exponential size proofs in general proof systems thereby proving NP $\neq$ co-NP. While no lower bounds have been established for say Frege or extended Frege systems, results for proof size have been proven in a number of proof systems including resolution. For resolution, it is interesting to understand what formulas can have small proofs and when such short proofs can be achieved with relatively small space. Time space resolution tradeoffs have been studied for both sublinear space and superlinear space (up to subexponential space). Whereas the "barrier" for computational complexity space and time space studies is log space, for resolution proof systems (and some other restricted proof systems) there are now time space tradeoffs that go beyond linear space! We mention two results in regard to this success story.

Beame, Beck and Impagliazzo [11] break the "linear space barrier" showing that there are formulas of size n, with refutation proofs with simultaneous size and space quasi-polynomial in n (so not polynomial space) but any resolution proof that is restricted to polynomial space requires size that is exponential in n. Following this result, Beck, Nordstrom and Tang [13] provide a similar tradeoff result for the stronger polynomial calculus proof system. Huynh and Nordstrom [54] obtain size space tradeoffs for stronger proof systems (cutting planes) but for a weaker regime of parameters (space at most linear). Nordstrom [77] provides an excellent summary of time space tradeoffs in proof complexity.

10 A Variety of Current and Potential Tradeoff Studies

This survey has necessarily been far from comprehensive given the pervasive nature of tradeoffs. Indeed, I have omitted discussion of entire well studied fields such as approximation algorithms (see, for example, the texts by Vazirani [90], and Shmoys and Williamson [92]), parameterized complexity (see the text by Downey and Fellows [35]), and packet routing networks (see, for example, Peleg

and Upfal [81]). These fields are inherently about tradeoffs and remain active areas of research.

Computer science continues to evolve as the technology advances and the pervasive use of computing and communication continues at what often seems an accelerated pace. Theoretical computer science tries to keep pace and conferences will, for example, contain sections on algorthmic game theory, mechansim design, differential privacy, online social networks, social choice, internet routing, high dimensional data, and search engines. While recent results in these topics may not explicitly emphasize tradeoffs, it seems reasonable to expect more quantifiable tradeoff results in new contexts.

One current area of interest is algorthmic game theory and mechanism design as pioneered by Nisan and Ronen [75]. The classical fields of game theory and mechanism design did not account for computational constraints. For example, the inherent allocation problems that underlie various auctions are NP hard and often NP hard to obtain any non trivial approximation. Hence the truthful VCG mechanism (which requires an optimal allocation along with the VCG payment rule) is not in general computationally feasible. Similarly, the existence of a "good" equilibrium does not discuss efficient methods for reaching such an equilibrium. The tradeoff between computational feasibility and the goals of self-interested agents is the raison d'etre for the area of computational game theory/mechanism design.

Suppose we consider the conbinatorial auction problem where agents have valuations for various subsets of items and no item can be allocated to more than one person. The underlying allocation problem is a multi-minded generalization of the set packing problem, which is NP hard to approximate even for very restricted forms of the problem. The emphasis so far in this area has been to see what approximations to the social welfare (the sum of true valuations for the sets allocated to agents) are possible by mechanisms that run in polynomial time. Here the mechanism has to contend with agents that may not bid truthfully. Incentive compatible mechanisms guarantee that the utility for every agent (i.e. the valuation minus the payment for the allocated set) is dominated by a truthful bid. There have been examples [78] where it is proven that incentive compatible mechanisms cannot provide approximations that are nearly as good as the approximations that can be achieved by computationally efficient algorithms for the underlying allocation problem. While there are some definitions of "approximate truthfullness"(as in McSherry and Talwar [66]), there has not yet been a quantifiable study of tradeoffss between quantifiable measures of truthfulness and computational efficiency.

A major aspect of algorithmic game theory is the computation and properties of equilbria and in particular of pure, mixed and Bayesian Nash equilibria. Although previously studied in game theory by economists (for example as in Dubey [36]), the tradeoff between performance (i.e. approximation guarantees) and solutions achieving equilibrium was launched in algorithmic game theory by Koutsoupias and Papadimitriou [61]. They defined the price of anarchy (POA) as the ratio between the performance (for example, the social welfare of the

game) at the worst possible equilibria to that of the optimal performance for the game. The POA definition was soon exploited in a seminal paper on routing by Roughgarden and Tardos [86]. One can define various notions of ϵ approximate equilibria and then study the tradeoffs between performance and equilibria. The POA concept can be extended to any concept of stability (for example, stability in cooperative and non-cooperative games, stable matchings) and then one can study properties of achieving some degree of stability (as in the stability scores of Feldman, Meir, Tennenholtz [41]). In cooperative game theory, cores capture the concept of stability against coalitions and approximate cores are a subject of interest within economics.

Another important area of recent research concerns the many issues involving privacy especially as it relates to say online social network data bases. In particular, what is the value of privacy? We have already seen the cost of privacy in the setting of communication complexity. The active and important area of differential privacy as initiated by Dwork [39] studies the question of privacy vs accuracy of information in statistical databases. As online social networks and recommendation systems continue to request personal information, there is a need to understand the cost of privacy loss verses the perceived benefit in participating (see, for example, Carrascal et al [26]).

In studying online algorithms, there are two commonly used measures of the quality of the algorithm, namely the competitive ratio (measuring an algorithms performance relative to an optimal solution) and regret (measuring the algorithms limiting cost per online step relative to that of a fixed optimal solution) While these two measures seem quite related, there are problems where it is proved that there is a tradeoff between these two criteria. Lin et al [65] consider a general problem they call "smoothed online convex optimization" (which adds a smoothed switching cost to an online convex optimzation problem). Their tradeoff for an instance of this problem shows that any algorithm that achives constant competitive ratio cannot achieve sublinear regret. Many natural online learning problems require some tradeoff between realizing some immediate gain vs the potential cost/gain for the future. Such considerations constitute the fundamental issue of "exploitation vs exploration", as considered in reinforcement learning. Fundamental tradeoffs in this regard are discussed in the excellent survey by Kaelbling, Littman and Moore [59].

An area of increasing importance concerns tradeoffs between energy and performance. In this regard, Yao, Demers and Shenker [95] introduced "speed scaling", whereby the speed of a system can be set so as to lower energy cost while still trying to preserve say response time. See Andrew, Lin and Wierman [7] for results concerning such tradeoffs.

Finally, there is an important aspect of computation that is not usually quantified but does often play an important criteria in many computational choices. Namely, algorithmic simplicity, style or structure usually induces some degree of understandability, extendability, or maintainability and therefore becomes a critical aspect in computation. But here we do not yet have any quantification or well accepted hierarchy of algorithmic design nor do we have quantifiable

measures for understandability, extendability and maintainability. In this regard, the work concerning categorical data in databases and information systems (see Gibson, Kleinberg and Raghavan [51]) may provide a good starting point.

Acknowledgements. I would like to thank Faith Ellen, Toni Pitassi, Venkatesh Raman and Vinod Vaikunthanathan for many helpful comments and references. I especially appreciate Faith Ellen's reading of a preliminary draft. I would also like to thank the organizers for their patience and for inviting me to speak at this workshop celebrating Ian's 66th birthday.

References

1. Ada, A., Chattopadhyay, A., Cook, S.A., Fontes, L., Kouckỳ, M., Pitassi, T.: The hardness of being private. In: IEEE Conference on Computational Complexity, pp. 192–202 (2012)
2. Agarwal, M., Kayal, N., Saxena, N.: Primes is in p. Annals of Mathematics 160(r21), 781–793 (2004)
3. Ajtai, M.: A non-linear time lower bound for boolean branching programs. Theory of Computing 1(1), 149–176 (2005)
4. Alekhnovich, M., Borodin, A., Buresh-Oppenheim, J., Impagliazzo, R., Magen, A.: Toward a model for backtracking and dynamic programming. Electronic Colloquium on Computational Complexity (ECCC) 16, 38 (2009)
5. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. 58(1), 137–147 (1999)
6. Alt, H., Mehlhorn, K., Munro, J.I.: Partial match retrieval in implicit data structures. Inf. Process. Lett. 19(2), 61–65 (1984)
7. Andrew, L.L.H., Lin, M., Wierman, A.: Optimality, fairness, and robustness in speed scaling designs. In: SIGMETRICS, pp. 37–48 (2010)
8. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley Series on Parallel and Distributed Computing. Wiley (2004)
9. Barkol, O., Rabani, Y.: Tighter lower bounds for nearest neighbor search and related problems in the cell probe model. J. Comput. Syst. Sci. 64(4), 873–896 (2002)
10. Beame, P.: A general sequential time-space tradeoff for finding unique elements. In: STOC, pp. 197–203 (1989)
11. Beame, P., Beck, C., Impagliazzo, R.: Time-space tradeoffs in resolution: superpolynomial lower bounds for superlinear space. In: STOC, pp. 213–232 (2012)
12. Beame, P., Vee, E.: Time-space tradeoffs, multiparty communication complexity, and nearest-neighbor problems. In: IEEE Conference on Computational Complexity, p. 18 (2002)
13. Beck, C., Nordstrom, J., Tang, B.: Some trade-off results for the polynomial calculus. In: STOC (2013)
14. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. 7(4), 448–461 (1973)
15. Borodin, A., Cook, S.A.: A time-space tradeoff for sorting on a general sequential model of computation. SIAM J. Comput. 11(2), 287–297 (1982)
16. Borodin, A., Fich, F.E., Meyer auf der Heide, F., Upfal, E., Wigderson, A.: A time-space tradeoff for element distinctness. SIAM J. Comput. 16(1), 97–99 (1987)

17. Borodin, A., Fich, F.E., Meyer auf der Heide, F., Upfal, E., Wigderson, A.: A tradeoff between search and update time for the implicit dictionary problem. Theor. Comput. Sci. 58, 57–68 (1988)
18. Borodin, A., Fischer, M.J., Kirkpatrick, D.G., Lynch, N.A., Tompa, M.: A time-space tradeoff for sorting on non-oblivious machines. J. Comput. Syst. Sci. 22(3), 351–364 (1981)
19. Borodin, A., Ostrovsky, R., Rabani, Y.: Lower bounds for high dimensional nearest neighbor search and related problems. In: Discrete and Computational Geometry – The Goodman-Polack Festschrift, vol. 25, pp. 255–276 (2003)
20. Brodsky, A., Fich, F.E.: Efficient synchronous snapshots. In: PODC, pp. 70–79 (2004)
21. Brönnimann, H., Chan, T.M., Chen, E.Y.: Towards in-place geometric algorithms and data structures. In: Symposium on Computational Geometry, pp. 239–246 (2004)
22. Buresh-Oppenheim, J., Davis, S., Impagliazzo, R.: A stronger model of dynamic programming algorithms. Algorithmica 60(4), 938–968 (2011)
23. Buss, S.R., Williams, R.: Limits on alternation-trading proofs for time-space lower bounds. In: IEEE Conference on Computational Complexity, pp. 181–191 (2012)
24. Canetti, R., Goldreich, O.: Bounds on tradeoffs between randomness and communication complexity. Computational Complexity 3, 141–167 (1993)
25. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
26. Carrascal, J., Riederer, C., Erramilli, V., Cherubini, M., de Oliveira, R.: Your browsinf behavior for a big mac: Economics of personal information online. In: WWW (2013)
27. Chakrabarti, A., Chazelle, B., Gum, B., Lvov, A.: A lower bound on the complexity of approximate nearest-neighbor searching on the hamming cube. In: STOC, pp. 305–311 (1999)
28. Chan, T.: Comparison-based time-space lower bounds for selection. ACM Transaction on Algorithms 6(n) (2010)
29. Chandra, A.K., Furst, M.L., Lipton, R.J.: Multi-party protocols. In: STOC, pp. 94–99 (1983)
30. Cobham, A.: The recognition problem for the set of perfect squares. In: SWAT (FOCS), pp. 78–87 (1966)
31. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. J. Symb. Log. 44(1), 36–50 (1979)
32. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM J. Comput. 31(6), 1794–1813 (2002)
33. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 348–360. Springer, Heidelberg (2002)
34. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM J. Comput. 12(4), 656–666 (1983)
35. Downey, R.G., Fellows, M.R.: Parameterized Complexity, 530 p. Springer (1999)
36. Dubey, P.: Inefficiency of nash equilibrium. Mathematics of Operations Research 11(1), 1–8 (1986)
37. Duris, P., Galil, Z.: A time-space tradeoff for language recognition. Mathematical Systems Theory 17(1), 3–12 (1984)
38. Duris, P., Galil, Z., Schnitger, G.: Lower bounds on communication complexity. Inf. Comput. 73(1), 1–22 (1987)

39. Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006, Part II. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)
40. Feigenbaum, J., Jaggard, A.D., Schapira, M.: Approximate privacy: foundations and quantification (extended abstract). In: Proceedings of the 11th ACM Conference on Electronic Commerce, EC 2010, pp. 167–178. ACM (2010)
41. Feldman, M., Meir, R., Tennenholtz, M.: Stability scores: measuring coalitional stability. In: AAMAS, pp. 771–778 (2012)
42. Fich, F.E., Miltersen, P.B.: Tables should be sorted (on random access machines). In: Sack, J.-R., Akl, S.G., Dehne, F., Santoro, N. (eds.) WADS 1995. LNCS, vol. 955, pp. 482–493. Springer, Heidelberg (1995)
43. Fich, F.E., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing 16(2-3), 121–163 (2003)
44. Fischer, M.J., Lynch, N.A.: A lower bound for the time to assure interactive consistency. Inf. Process. Lett. 14(4), 183–186 (1982)
45. Fortnow, L.: Time-space tradeoffs for satisfiability. J. Comput. Syst. Sci. 60(2), 337–353 (2000)
46. Franceschini, G., Grossi, R., Munro, J.I., Pagli, L.: Implicit b-trees: a new data structure for the dictionary problem. J. Comput. Syst. Sci. 68(4), 788–807 (2004)
47. Franceschini, G., Munro, J.I.: Implicit dictionaries with $o(1)$ modifications per update and fast search. In: SODA, pp. 404–413 (2006)
48. Frederickson, G.N.: Implicit data structures for the dictionary problem. J. ACM 30(1), 80–94 (1983)
49. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J. Comput. Syst. Sci. 34(1), 19–26 (1987)
50. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379(3), 405–417 (2007)
51. Gibson, D., Kleinberg, J.M., Raghavan, P.: Clustering categorical data: An approach based on dynamical systems. VLDB J. 8(3-4), 222–236 (2000)
52. Golab, L., DeHaan, D., Demaine, E.D., López-Ortiz, A., Munro, J.I.: Identifying frequent items in sliding windows over on-line packet streams. In: Internet Measurement Comference, pp. 173–178 (2003)
53. Halstenberg, B., Reischuk, R.: Different modes of communication. SIAM J. Comput. 22(5), 913–934 (1993)
54. Huynh, T., Nordström, J.: On the virtue of succinct proofs: amplifying communication complexity hardness to time-space trade-offs in proof complexity. In: STOC, pp. 233–248 (2012)
55. Hyafil, L.: On the parallel evaluation of multivariate polynomials. SIAM J. Comput. 8(2), 120–123 (1979)
56. Impagliazzo, R., Wigderson, A.: $P = BPP$ if e requires exponential circuits: Derandomizing the xor lemma. In: STOC, pp. 220–229 (1997)
57. Impagliazzo, R., Williams, R.: Communication complexity with synchronized clocks. In: IEEE Conference on Computational Complexity, pp. 259–269 (2010)
58. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
59. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. J. Artif. Intell. Res. (JAIR) 4, 237–285 (1996)
60. Karchmer, M.: Two time-space tradeoffs for element distinctness. Theor. Comput. Sci. 47(3), 237–246 (1986)
61. Koutsoupias, E., Papadimitriou, C.H.: Worst-case equilibria. Computer Science Review 3(2), 65–69 (2009)

62. Krizanc, D., Peleg, D., Upfal, E.: A time-randomness tradeoff for oblivious routing (extended abstract). In: STOC, pp. 93–102 (1988)
63. Kushilevitz, E.: Privacy and communication complexity. SIAM J. Discrete Math. 5(2), 273–284 (1992)
64. Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. SIAM J. Comput. 30(2), 457–474 (2000)
65. Lin, M., Wierman, A., Roytman, A., Meyerson, A., Andrew, L.L.H.: Online optimization with switching cost. SIGMETRICS Performance Evaluation Review 40(3), 98–100 (2012)
66. McSherry, F., Talwar, K.: Mechanism design via differential privacy. In: FOCS, pp. 94–103 (2007)
67. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in o($\log^2$ n) time. J. Comput. Syst. Sci. 33(1), 66–74 (1986)
68. Munro, J.I., Paterson, M.: Optimal algorithms for parallel polynomial evaluation. J. Comput. Syst. Sci. 7(2), 189–198 (1973)
69. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theor. Comput. Sci. 12, 315–323 (1980)
70. Munro, J.I., Poblete, P.V.: Searchability in merging and implicit data structures. BIT 27(3), 324–329 (1987)
71. Munro, J.I., Raman, V.: Selection from read-only memory and sorting with minimum data movement. Theor. Comput. Sci. 165(2), 311–323 (1996)
72. Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. J. Comput. Syst. Sci. 21(2), 236–250 (1980)
73. Muthukrishnan, S.: Theory of data stream computing: where to go. In: PODS, pp. 317–319 (2011)
74. Nepomnnjaščiĭ, V.: Rudimentary predicates and turing computations. Dokl. Akad. Nauk SSSR 195, 282–284 (1070); English translation in Soviet Math. Dokl. 11, 1462–1465 (1970)
75. Nisan, N., Ronen, A.: Algorithmic mechanism design. Games and Economic Behavior 35(1-2), 166–196 (2001)
76. Nisan, N., Wigderson, A.: Rounds in communication complexity revisited. SIAM J. Comput. 22(1), 211–219 (1993)
77. Nordstrom, J.: Pebble games, proof complexity, and time-space trade-offs. In: Logical Methods in Computer Science (to appear, 2013)
78. Papadimitriou, C.H., Schapira, M., Singer, Y.: On the hardness of being truthful. In: FOCS, pp. 250–259 (2008)
79. Papadimitriou, C.H., Sipser, M.: Communication complexity. J. Comput. Syst. Sci. 28(2), 260–269 (1984)
80. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)
81. Peleg, D., Upfal, E.: A trade-off between space and efficiency for routing tables. J. ACM 36(3), 510–530 (1989)
82. Pippenger, N.: On simultaneous resource bounds (preliminary version). In: FOCS, pp. 307–311 (1979)
83. Rabin, M.O.: Probabilistic algorithm for testing primality. Journal of Number Theory 12(1), 128–138 (1980)
84. Radhakrishnan, J., Raman, V.: A tradeoff between search and update in dictionaries. Inf. Process. Lett. 80(5), 243–247 (2001)
85. Raman, V., Ramnath, S.: Improved upper bounds for time-space trade-offs for selection. Nord. J. Comput. 6(2), 162–180 (1999)

86. Roughgarden, T., Tardos, É.: How bad is selfish routing? J. ACM 49(2), 236–259 (2002)
87. Savitch, W.J.: Deterministic simulation of non-deterministic turing machines (detailed abstract). In: STOC, pp. 247–248 (1969)
88. Solovay, R., Strassen, V.: Erratum: A fast monte-carlo test for primality. SIAM J. Comput. 7(1), 118 (1978)
89. Valiant, L.G., Skyum, S., Berkowitz, S., Rackoff, C.: Fast parallel computation of polynomials using few processors. SIAM J. Comput. 12(4), 641–644 (1983)
90. Vazirani, V.V.: Approximation algorithms. Springer-Verlag New York, Inc., New York (2001)
91. Williams, R.: Alternation-trading proofs, linear programming, and lower bounds. In: STACS, pp. 669–680 (2010)
92. Williamson, D.P., Shmoys, D.B.: The Design of Approximation Algorithms. Cambridge University Press (2011)
93. Yao, A.C.-C.: Some complexity questions related to distributive computing (preliminary report). In: STOC, pp. 209–213 (1979)
94. Yao, A.C.-C.: Near-optimal time-space tradeoff for element distinctness. SIAM J. Comput. 23(5), 966–975 (1994)
95. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS, pp. 374–382 (1995)

A History of Distribution-Sensitive Data Structures*

Prosenjit Bose[1], John Howat[2], and Pat Morin[1]

[1] School of Computer Science
Carleton University
{jit,morin}@scs.carleton.ca
[2] School of Computing
Queen's University
howat@cs.queensu.ca

Abstract. Distribution-sensitive data structures attempt to exploit patterns in query distributions in order to allow many sequences of queries execute faster than in traditional data structures. In this paper, we survey the history of such data structures, outline open problems in the area, and offer some new results.

1 Introduction

Data structures that exploit non-uniform query distributions to solve data structuring problems are typically termed *distribution-sensitive*. The idea of a query distribution being *non-uniform* is a deliberately vague notion; such data structures can take advantage of many different types of underlying patterns in a query distribution, and we will elaborate on the specific types of patterns that are generally studied in Section 3.

Most distribution-sensitive data structures evolve from the following high-level principle. For a given problem, a data structure that does not take advantage of the query distribution typically has a query time that is a function of the size of the data structure, and a (possibly matching) lower bound may also exist. A quantity related to a property of the query distribution is defined. A data structure is designed such that its query time is a function of this quantity as opposed to the size of the data structure. This quantity, in the worst case, is no larger than the number of elements stored in the data structure. As a result, any existing lower bounds are not violated. However, in many cases, it is possible to beat known lower bounds, at least for some queries (or sequences of queries).

Distribution-sensitive data structures are often motivated by practical concerns, since the types of queries handled in real-world applications are generally not uniformly random. For example, imagine a file system on a server that handles a large number of users. One possible source of non-uniformity is that some files get accessed much more frequently than others (*e.g.*, programs essential to the operating system or commonly-accessed documents), and an efficient file

* Research supported in part by NSERC. Dedicated to Ian Munro.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 133–149, 2013.

system may wish to speed up access to these files. The same could be said of users, as well. Users who log-in to the system infrequently (such as guest users) might have their files retrieved more slowly in order to speed up the access of more frequent users.

Another source of non-uniformity could be the relative location of files. During a directory traversal, for example, a user might be likely to access files that are, in some sense, close together (either physically on disk or abstractly in the directory structure). If directory traversal is a common operation, speeding up such accesses would be beneficial.

Perhaps the clearest source of non-uniformity would be an explicit description of the distribution of queries to files. Each file would have some specified probability of being requested by a user. Naturally, we would expect frequently requested files to be accessed quickly, while files that are seldom requested could be accessed relatively slowly. Having complete information about the distribution of queries is, in general, a somewhat unrealistic assumption. Nevertheless, this information can be approximated empirically.

Organization. The remainder of this paper is organized in the following way. We begin with a review of optimum binary search trees in Section 2. We then provide an overview of different distribution patterns that have been considered in the literature in Section 3. We then tour data structures that take advantage of these patterns in Section 4. Finally, in Section 5, we present a new distribution-sensitive data structure.

2 Optimum Binary Search Trees

Perhaps the earliest data structuring result that can be considered distribution-sensitive is that of *optimum binary search trees*, introduced in 1975 by Knuth [1]. An optimum binary search tree is a binary search tree on n keys that has *average* search cost that is no larger than that of any other binary search tree built on those n keys. Note that finding such a tree is not trivial, and an exhaustive approach is hopeless since it is well-known that there are an exponential number of possible binary search trees on a given number of keys.

To construct such a tree, one defines a query probability for each key.[1] The construction algorithm uses dynamic programming to construct the tree in $O(n^2)$ time. The average search cost for a key drawn from the specified probability distribution is the sum (over all keys) of the probability associated with a key multiplied by the cost to search for that key in the tree (*i.e.*, the depth of the node containing that key).

It is important to note that the optimum binary search tree has the minimum average search cost over all binary search trees built on the given keys: there is no asymptotic notation required. This is, in general, a fairly lofty goal (as

[1] One may also wish to consider the case when not all queries are stored in the tree (*i.e.*, some queries are unsuccessful). In this case, one also defines the probability that a query lies between two adjacent keys for each such pair.

evidenced by the large amount of time required to construct the tree). Linear time algorithms were designed to build binary search trees that are optimal to within a constant factor [2–5].

One important limitation of optimum binary search trees is that they require *a priori* knowledge of the query distribution, which (in many applications) may not be available.

3 Distribution Patterns

Optimum binary search trees exploit the most obvious form of non-uniformity: explicit non-uniformity. In this section, we discuss other patterns and properties of queries that are commonly observed in data structuring problems. We defer examples of data structures with these properties until Section 4.

For simplicity and concreteness, we describe these properties as they are used for *static dictionary* data structures. Such data structures maintain a static set $S = \{s_1, s_2, \ldots, s_n\}$ of n keys under the *search* operation, where we are given a key and must return the corresponding element in the data structure. The sequence of queries is given online and is denoted $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$ (for $1 \leq t \leq m$) and m is the length of the query sequence. We also assume a sufficiently long query sequence, so that m is $\Omega(n \log n)$. Throughout this paper, we use $\log x$ to denote $\max\{1, \log_2 x\}$.

3.1 Static and Dynamic Optimality

Consider the *frequency* $f(x)$ of a query $x \in S$, *i.e.*, the number of times that query is made in A: $f(x) = |\{t \mid a_t = x\}|$. The *static optimality property* states that the time to execute A is

$$O\left(m + \sum_{i=1}^{n} f(x_i) \log(m/f(x_i))\right)$$

Note that the quantity $f(x_i)/m$ is essentially the probability $p(x_i)$ that x_i is queried. The sum can therefore be re-written as

$$\sum_{i=1}^{n} f(x_i) \log(m/f(x_i)) = \sum_{i=1}^{n} mp(x_i) \log(1/p(x_i)) = mH$$

where H is the empirical entropy of the query distribution.[2] The average query time is therefore $mH/m = H$. Since the entropy of the query distribution is a lower bound on the average query time (to within lower-order terms) [6], it follows that a data structure with the static optimality property is optimal in this sense. The optimum binary search trees of Knuth match this average query time [1]. Note that, although the query time is stated in terms of query

[2] For this reason, the static optimality property is also known as the *entropy bound*.

frequencies, it is not necessarily required that the data structure be given the frequencies. While optimum binary search trees require the frequencies to be specified in advance, not all data structures require this.

The *dynamic optimality property* enforces a much stronger sense of "optimality." Given some specific class of data structure (*e.g.*, binary search trees), let $OPT(A)$ denote the total query time of the fastest possible data structure in that class to execute the sequence A, given that it may perform arbitrary restructuring after each query (*e.g.*, rotations in a binary search tree), which are included in the access time. Another data structure of that class is *dynamically optimal* if it can execute A in time $O(OPT(A))$, assuming A is sufficiently long. The problem of designing a dynamically optimal binary search tree is a longstanding open problem. We discuss this problem in more detail in Section 4.

3.2 Key-Independent Optimality

Another kind of optimality is that of *key-independent optimality*, which was introduced by Iacono [7]. Let $b : S \to S$ be a random bijection and let $b(A)$ denote the query sequence $\langle b(a_1), b(a_2), \ldots, b(a_m)\rangle$. Recall that $OPT(A)$ represents the fastest possible time that a data structure in a given class can execute A. Now, consider the quantity $E[OPT(b(A))]$: the expected value of the fastest time a data structure in a given class can execute $b(A)$. We say that a data structure is key-independently optimal if it can execute A in time $O(E[OPT(b(A))])$.

Intuitively, a key-independently optimal data structure executes sequences with random key values as fast as any other data structure in that class.

3.3 The Working-Set Property

In the query sequence $A = \langle a_1, a_2, \ldots, a_m\rangle$, we say that a_t is queried at time t for $1 \leq t \leq m$. We define the *working-set number* of x at time t, denoted $w_t(x)$, to be the number of *distinct* elements queried since the last time x was queried (*i.e.*, the last time x appeared in A). If x has not yet appeared in A (prior to time t), then we set $w_t(x) = n$. To be more precise, we slightly modify the notation of Iacono [7]: let $l_t(x) = \min(\{\infty\} \cup \{t' > 0 \mid a_{t-t'} = x\})$ and define

$$w_t(x) = \begin{cases} n & \text{if } l_t(x) = \infty \\ |\{a_{t-l_t(x)+1}, \ldots, a_t\}| & \text{otherwise} \end{cases}$$

The *working set property* states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log w_t(a_t)\right)$$

Intuitively, query sequences that repeat particular queries frequently are executed faster in data structures with the working-set property. Sleator and Tarjan [8] indicate one possible motivation for why this is a desirable property. Suppose that, while the set S is quite large, there exists a subset $S' \subset S$ such

that all queries to the data structure belong to S'. In this case, the working-set property allows queries to run in $O(\log |S'|)$ time instead of $O(\log |S|)$ time, since the working-set number of any query at any time is at most $|S'|$. This essentially means that the elements of $S \setminus S'$ can be ignored, since they are never queried. Put another way, it is as if those elements are not present in the data structure.

This property was first discussed in this context by Sleator and Tarjan [8], although the general idea is similar to that of the move-to-front heuristic [9]. IIacono proved that any data with the working-set property is also statically optimal [10]. Therefore, ensuring the working-set property is one way of circumventing the need for *a priori* knowledge of the query distribution. Furthermore, the bound provided by the working-set property is asymptotically equivalent to that provided by key-independent optimality [7]. Finally, for certain classes of skip-lists and B-trees, the working-set property is asymptotically equivalent to dynamic optimality [11].

3.4 The Queueish Property

In some sense, the queueish property is the opposite of the working-set property. The *queue number* of the element $x \in A$ at time t is denoted $q_t(x)$ and is defined to be $n - w_t(x) - 1$, *i.e.*, the number of distinct elements of S that have *not* been queried since the last time x was queried. The *queueish property* states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log q_t(a_t)\right)$$

Iacono and Langerman proved that no binary search tree has the queueish property [12]. As a result, this particular property has yet to be explored in significantly more detail, at least for the dictionary problem. A weaker version of this property called the *weakly queueish property* exists in at least one dictionary [12]. The weak version of the property states that a query a_t should execute in time $O(\log q_t(a_t)) + o(\log n)$.

3.5 The Static and Dynamic Finger Properties

While both the working-set and queueish properties consider the *time* between queries, the static and dynamic finger properties consider the *distance* between queries. Let $d(x, y)$ denote the *rank distance* between x and y, *i.e.*, the number of elements in S between x and y.

We first consider the *static finger property*. Fix an element $f \in S$, which we shall call a *finger*. The static finger property states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log d(f, a_t)\right)$$

The "static" in "static finger" arises from the fact that the finger f is fixed. Iacono showed that any data structure that has the static optimality property also has the static finger property, for any choice of finger f [13].

The *dynamic finger property* allows the finger to dynamically change over time: the finger is always the previous query. Therefore, the dynamic finger property states that the time required to execute the sequence A is

$$O\left(m + \sum_{t=2}^{m} \log d(a_{t-1}, a_t)\right)$$

3.6 The Unified Property

The *unified property*[3] is a combination of the working-set and dynamic finger properties and was introduced by Bădoiu et al. [14]. The unified bound $u_t(x)$ for a query x at time t is defined as follows

$$u_t(x) = \min_{y \in S}(w_t(y) + d(y, x))$$

Intuitively, $u_t(x)$ is small whenever a recently queried element is close to x. More formally, the unified property states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log u_t(a_t)\right)$$

It is important to note that having the unified property is not the same as simultaneously having both the working-set and dynamic finger properties. To illustrate this, we consider an example of a query sequence due to Bădoiu et al. [14]. Suppose $S = \{1, 2, \ldots, n\}$ for some even n and define A as follows

$$A = \langle 1, n/2+1, 2, n/2+2, 3, n/2+3, \ldots, n/2, n, 1, n/2+1, \ldots \rangle$$

In A, every query (after the first n queries) will have working-set number n, and so the working-set property gives a bound of $O(m \log n)$ for the sequence. Similarly, every query is at distance $n/2$ from the previous query, and so the dynamic finger property gives a bound of $O(m \log n)$ for each sequence as well. However, observe that after the first n queries, query a_{t-2} is at distance 1 from a_t. The unified property therefore gives a bound of only $O(m)$ for the sequence, since both the working-set number and the distance are constant, which is a factor of $O(\log n)$ better than either of the other two bounds.

4 Examples of Distribution-Sensitive Data Structures

In this section, we survey a selection of known distribution-sensitive data structures.

[3] As noted by Bădoiu et al. [14], a "unified bound" appears in the original paper on splay trees by Sleator and Tarjan [8], which is simply the minimum of the bounds provided by static optimality, the working-set property and the static finger property. The property we discuss here is distinct.

4.1 Splay Trees

The splay tree of Sleator and Tarjan [8] is among the first distribution-sensitive data structures and is certainly one of the most interesting. The reasons for this are two-fold: first, splay trees have a considerable number of distribution-sensitive properties, including most of those mentioned in Section 3. Perhaps more interesting is the fact there is still much to be shown about the behaviour of splay trees.

The splay tree fits into the usual binary search tree model, and the *splay* operation, which consists of a series of rotations, is used after every query. We omit the details of the splaying operation. Splay trees support queries in $O(\log n)$ *amortized* time; this means that individual queries may take as much as $O(n)$ time, but the sequence as a whole executes in $O(m \log n)$ time.

Splay trees have many of the properties mentioned in Section 3. Sleator and Tarjan showed that splay trees are statically optimal, have the static finger property and have the working-set property [8]. Iacono later showed that splay trees are key-independently optimal [7].

Sleator and Tarjan conjectured that splay trees have the dynamic finger property [8]; this was verified 15 years later with a fairly difficult proof due to Cole et al. [15, 16], although Tarjan showed that the less general sequence where the keys are queried in sequential order can be executed in $O(n)$ time [17]. This bound is known as the *scanning bound* or *sequential access theorem*.

Given that splay trees have both the working-set and dynamic finger properties, Bădoiu et al. conjectured that splay trees also satisfy the unified property [14]; this question is still open.

Perhaps the most important unresolved conjecture was also posed by Sleator and Tarjan in their original paper [8]: are splay trees dynamically optimal? This problem remains open and is generally viewed as the most important question in this line of research. Note that this problem must, by its nature, be studied with respect to a particular class of data structure. For example, the problem is resolved for some classes of skip lists and B-trees [11].

4.2 Other Data Structures

Many other data structures also exhibit distribution-sensitive properties.

The Working-Set Structure. The working-set structure is due to Bădoiu et al. [14] and provides an even stronger version of the working-set property. Rather than executing the sequence in time $O(m + \sum_{t=1}^{m} \log w_t(a_t))$, the working-set structure guarantees that each individual query a_t is answered in worst-case time $O(\log w_t(a_t))$ per query in the sequence. This can be viewed as de-amortizing the working-set property provided by, for example, splay trees. This means that any query can be answered in time $O(\log n)$, since $w_t(a_t) \leq n$ for all t; this is a significant improvement over the $O(n)$ guarantee offered by splay trees. The working-set structure is not a binary search tree, however. Instead, it is a collection of binary search trees and queues, each of which increases doubly

exponentially in size to ensure that recently queried elements are found quickly. Therefore, the working-set structure is not directly comparable with splay trees. However, Bose et al. showed how to construct and maintain a binary search tree with the same query time as the working-set structure [18].

The Unified Structure. The unified structure was also introduced by Bădoiu et al. [14] and was the first data structure to have the unified property. Like the working-set structure, the unified structure is not a binary search tree and is therefore not directly comparable with splay trees either. However, Derryberry [19] studied binary search trees that support the unified property.

Skip-Splay Trees. The skip-splay tree is due to Derryberry and Sleator [20] and is an attempt to achieve the unified property within the binary search tree model. Skip-splay trees fall slightly short of this goal: they execute the query sequence in time $O(m \log\log n + \sum_{t=1}^{m} \log u_t(a_t))$, and so spend an extra $O(\log\log n)$ amortized time per query.

Nearing Dynamic Optimality. Considering the apparent difficulty of the problem of proving splay trees to be dynamically optimal, attention has turned somewhat to finding other binary search trees that are *competitive* to dynamically optimal data structures: their query times are a (sub-logarithmic) factor away from dynamic optimality. Several data structures come within a factor of $O(\log\log n)$ of dynamic optimality [21–23].

Finger Search. In the finger search problem, we are given a pointer to some element of the data structure before we perform a query for another element. The goal is then to execute the query in time that is a function (usually logarithmic) of d, where d is the rank distance from the query to the finger. This is a generalization of the dynamic finger property: to achieve the dynamic finger property, one can simply use a pointer to the previous answer. Several finger search data structures are known for various models (*e.g.*, [24–26]). Skip lists also allow for finger searches [27]. Kaporis et al. [28] show how to achieve $O(\log\log d)$ queries when the contents of S are chosen according to a certain kind of distribution.

Biased Search Trees. The biased search trees of Bent et al. [29] are similar in spirit to optimum binary search trees. Each element $s_i \in S$ is assigned a positive real weight w_i, and a query for s_i is executed in time $O(\log W/w_i)$, where $W = \sum_{i=1}^{n} w_i$ is the sum of all weights in the data structure. When w_i is viewed as the probability that s_i is queried, then we have $W = 1$ and achieve essentially the same as an optimum binary search tree (to within a constant factor). The key difference is that biased search trees support insertions and deletions of elements into and from S. Splay trees can also be made to support this query time, although only in the amortized sense. Treaps support this query time in the expected sense [30]. Bagchi et al. presented a biased version of skip lists that can also match this query time [31].

Priority Queues. Johnson described a priority queue that allowed for insertion and deletion in time $O(\log \log D)$ [32], where D is the difference in priorities of the elements before and after the element to be inserted or deleted. In this setting, the allowable values of priorities come from a restricted range of integers. This type of distribution sensitivity is similar in spirit to that of the dynamic finger property, except that the measure of distance is in terms of priority and that the distance is measured to both the previous and next elements. Iacono showed that pairing heaps offer a bound very similar to that of the working-set property, except in the priority queue setting [10] . The minimum element can be extracted from a pairing heap in time $O(\log \min\{n, k\})$, where n is the number of elements in the pairing heap and k is the number of heap operations performed since x was inserted. Similar results were obtained by Elmasry [33].

Queaps and Queueish Dictionaries. Iacono and Langerman [12] presented a heap with the queueish property: the minimum element x can be extracted in time $O(\log k)$, where k is the number of elements that have been in the heap longer than x. A dictionary data structure is also presented that supports a query for x at time t in time $O(\log q_t(x) + \log \log n)$. Iacono and Langerman also showed that no binary search tree can have the queueish or even weakly queueish property [12] by showing an access sequence such that having the queueish property would violate the lower bound of Wilber [34].

The Temporal Precedence Problem. For the temporal precedence problem, a list must be maintained over many insertion operations. Queries consist of two pointers into the list and must quickly determine which of the two items pointed to was inserted first. Brodal et al. gave a data structure in the pure pointer machine model that is capable of answering queries in time $O(\log \log \delta)$ [35], where δ is the number of insertions that occurred between the insertions of the query elements.

Random Input and Predecessor Search. The predecessor search problem asks for the largest element of S less than or equal to some query element. Belazzougui et al. [36] showed that this can be accomplished quickly when the set S is chosen according to a certain kind of distribution (as in the work of Kaporis et al. [28]). This differs from the usual notion of distribution sensitivity: rather than considering patterns in the query distribution, patterns in the input itself are considered.

Point Searching. For the (planar) point searching problem, the set S consists of points in the plane and a query consists of a point and the data structure must determine if that point is in the set S. If the point is not in the set S, then the data structure must indicate this. Demaine et al. described how to answer point searching queries in a distribution-sensitive manner [37]. The query time is logarithmic in a function related to the number of points in a certain region defined by the current and previous queries. Therefore, this data structure can be considered to have a two-dimensional analogue of the dynamic finger property.

Point Location. For the (planar) point location problem, we are given a series of regions in the plane. Queries consist of a query point and the data structure must determine which region contains the query point. This particular problem has been thoroughly studied and has had several distribution-sensitive data structures proposed for it. One approach is to obtain an optimal data structure based on the probability distribution of the queries (*e.g.*, [38, 39]). In fact, it is possible to achieve close to this even without knowledge of the distribution [40]. Another approach to the problem is to support an analogue of the dynamic finger property, as proposed by Iacono and Langerman [41]. In this setting, the query time achieved is very similar to that achieved for point searching [37].

Nearest Neighbour Search. For the nearest neighbour problem, we are given a set of points in the plane, and a query asks for the closest point in the set to the query point. In general, it is difficult to find the exact solution quickly using a reasonable amount of space, and so most data structures settle for an approximate solution (*i.e.*, a point that is not much further away than the nearest neighbour). Derryberry et al. proposed a data structure for the (approximate) nearest neighbour problem that executes queries in time that is a (logarithmic) function of the number of points in a certain box containing the query and the answer to the previous query [42]. In fact, this technique works in any constant dimension; the query time has quadratic dependence on the dimension (and, of course, the approximation becomes worse as the dimension increases).

Biased Range Trees. Range searching is a very well-studied problem, but very little has been done in terms of distribution-sensitive data structures for range searching. Recall that for the range searching problem, we are given a set of points in the plane. A query consists of some region (typically of a specific shape) and we must report (or count) the number of points in the region. Dujmović et al. described a data structure that, given a query distribution, can answer quarter-space (*i.e.*, quadrant) queries in time that is optimal (to within a constant factor) [43]. Afshani et al. subsequently extended this result to four-sided queries [44].

5 Searching with Temporal Fingers

In this section, we present a new distribution-sensitive data structure for searching in a static dictionary. In this data structure, the time required to search for a query element is logarithmic in the distance from the query element to a temporal finger, plus a small additive term.

Consider a set S and a sequence $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$. For the purposes of this chapter, we assume $S = \{1, 2, \ldots, n\}$ (which is simply a reduction to rank-space) and that the set S is static.

Recall that the working-set property roughly states that accesses are fast if they have been made recently. Conversely, the queueish property states that accesses are fast if they have *not* been made recently. We propose a generalization of these two properties, where a *temporal finger* is defined and the access time is

a function of the distance in A between the temporal finger and the element to be accessed. Such a property can be viewed as a temporal version of the static finger property.

5.1 Defining Temporal Distance

We briefly review some definitions from Section 3. Recall that our access sequence is $A = \langle a_1, a_2, \ldots, a_m \rangle$. Define

$$l_t(x) = \min\left(\{\infty\} \cup \{t' > 0 \mid a_{t-t'} = x\}\right)$$

One can think of $l_t(x)$ as the most recent time x has been queried in A before time t. We then define

$$w_t(x) = \begin{cases} n & \text{if } l_t(x) = \infty \\ |\{a_{t-l_t(x)+1}, \ldots, a_t\}| & \text{otherwise} \end{cases}$$

Here, $w_t(x)$ is the usual working-set number of element x at time t. We are now ready to define temporal distance. Consider a temporal finger f where $1 \leq f \leq n$. The temporal distance from x to f at time t is defined as

$$\tau_{t,f}(x) = |w_t(x) - f + 1|$$

Observe that if $f = 1$, then $\tau_{t,f}(x) = w_t(x)$. In this case, having query time logarithmic in the temporal distance is equivalent to the working-set property. Conversely, if $f = n$, then $\tau_{t,f}(x) = n - w_t(x) - 1$, which is precisely the queue number $q_t(x)$ defined by Iacono and Langerman [12] and described in Section 4.2. Performing a query in time logarithmic in the temporal distance in this case is equivalent to the queueish property. In general, our goal is a query time of $O(\log \tau_{t,f}(x)) + o(\log n)$.

Another way to view temporal distance is the following. At any time t, the query sequence A defines a permutation of the elements that orders them from most recent query to least recent query. The temporal finger f points to the f-th item in this permutation, and the temporal distance $\tau_{i,f}(x)$ is the distance from the f-th item to x in the permutation.

5.2 Background

The notion of a dictionary with query times that are sensitive to temporal distance is not new: the working-set structure [14] and the queueish dictionary [12] are two well-known examples. Layered working-set trees [18] also fall into this category.

However, the notion of allowing the finger by which temporal distance is measured to be selected in advance is relatively new; prior to this thesis, this problem has only been studied in the context of priority queues. Elmasry et al. developed a priority queue that supports a constant number of temporal fingers [45]. In particular, this shows the existence of a priority queue that supports both the working-set and queueish properties.

6 The Data Structure

Our data structure consists of two parts: OLD and YOUNG. A schematic of the data structure is illustrated in Figure 1.

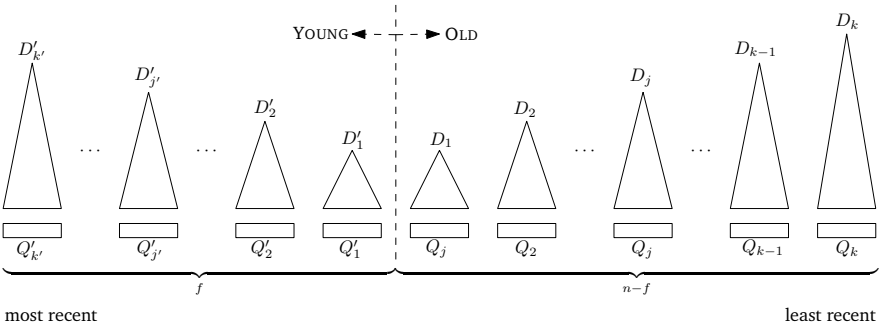


Fig. 1. A schematic of the data structure for temporal finger searching. Pointers between elements in substructures and the corresponding queue elements are not shown. The substructures in the OLD data structure are drawn as trees, but are actually implemented as sorted arrays.

6.1 The Old Data Structure

The OLD data structure contains the $n - f$ elements that were last accessed more than f queries ago. They are stored in a working-set structure [14].

As discussed in Section 4.2, the working-set structure consists of balanced binary search trees (*e.g.*, AVL trees [46]) $D_1, D_2, \ldots, D_k$ of size 2^{2^j} for $j = 1, 2, \ldots, k$. It follows that k is $O(\log\log(n - f))$. Each tree D_j has an accompanying queue Q_j containing the same elements as in the order that they were inserted into D_j. Pointers are maintained between an element in a tree and the corresponding element in the queue. The concatenation of all queues is precisely a list of the $n - f$ elements in the order they were queried, from most recent to least recent.

6.2 The Young Data Structure

The YOUNG data structure contains the f elements that were accessed at most f accesses ago. They are stored in a modified queueish dictionary [12].

The queueish dictionary consists of a series of substructures $D'_1, D'_2, \ldots, D'_{k'}$ and queues $Q'_1, Q'_2, \ldots, Q'_{k'}$. As in the OLD data structure, the concatenation of the queues in this data structure orders the elements in increasing order of last access time. However, the queues no longer correspond exactly to the substructures: all of the elements of $Q'_1 \cup Q'_2 \cup \cdots \cup Q'_j$ are stored in D'_j, but D'_j may contain additional elements. Pointers are maintained between each element

of D'_j and its corresponding entry in a queue (which may not be Q'_j). The size of Q'_j is between 2^{2^j-1} and 2^{2^j}. $D'_{k'}$ will contain all elements in the structure and thus have size f, and $Q'_{k'}$ has size at least $2^{2^{k'}-1}$. Therefore, $k' = O(\log\log f)$. As suggested by Iacono and Langerman, D'_j can be implemented as a sorted array [12]. Note that $D'_{k'}$ will be implemented differently; we address this issue during the analysis.

6.3 Performing a Query

To perform the query x at time t, we search in $D_1, D'_1, D_2, D'_2, \ldots$ and so on, until x is found. At this point x is now the most recently accessed (*i.e.*, *youngest*) element in the data structure, so we delete it from the structure we found it in and insert it into YOUNG. There are two possible cases: either x is found in OLD or YOUNG.

Suppose first that x is found in OLD, say $x \in D_j$. In this case, we delete x from D_j, insert x into YOUNG. We then delete the oldest element in YOUNG and insert it into OLD. In doing so, we will need to shift elements in OLD down to restore the size of D_j. This can be done by taking the oldest element out of each subtree and placing it in the next larger subtree until we reach D_j.

Suppose now that x is found in YOUNG, say $x \in D'_j$. This case is handled exactly as it would be in a regular queueish dictionary: x is removed from Q'_j and inserted at the front of $Q'_{k'}$. In doing so, $|Q'_j|$ may become too small (*i.e.*, less than 2^{2^j-1}). If this occurs, we remove $2^{2^j} - |Q'_j|$ elements from the end of Q'_{j+1}, insert them at the front of Q'_j, and reconstruct D'_j from the elements in $Q'_1, Q'_2, \ldots, Q'_j$. This may result in Q'_{j+1} becoming too small (and so on); these cases are handled identically.

6.4 Access Cost

The cost of an access can be separated into two parts: the cost of finding the query element and the cost of adjusting the data structure.

Finding x. To find the element x at time t, we search in $D_1, D'_1, D_2, D'_2, \ldots$ until x is found in D_j or D'_j. The cost to find the element is therefore $\sum_{l=1}^{j} O\left(\log 2^{2^l}\right) = \sum_{l=1}^{j} O\left(2^l\right) = O\left(2^j\right)$. Again, there are two possible cases: either $x \in D_j$ or $x \in D'_j$.

If $x \in D_j$, then $w_t(x) = f + \Omega\left(2^{2^{j-1}}\right)$. Therefore, $\tau_{t,f}(x) = |w_t(x) - f + 1| = |f + \Omega\left(2^{2^j}\right) - f + 1| = \Omega\left(2^{2^{j-1}}\right)$. Equivalently, $j \leq \log\log\tau_{t,f}(x) + O(1)$. The cost to find the element is therefore $O\left(2^j\right) = O(\log\tau_{t,f}(x))$.

If $x \in D'_j$, then $q_t(x) = (n - f) + \Omega\left(2^{2^{j-1}}\right)$, and since $w_t(x) = n - q_t(x) - 1$, we have $w_t(x) = f - \Omega\left(2^{2^{j-1}}\right) - 1$. Therefore, $\tau_{t,f}(x) = |w_t(x) - f +$

$1| = |f - \Omega\left(2^{2^{j-1}}\right) - 1 - f + 1| = | - \Omega\left(2^{2^{j-1}}\right)| = \Omega\left(2^{2^{j-1}}\right)$. Equivalently, $j \leq \log\log \tau_{t,f}(x) + O(1)$. The cost to find the element is therefore $O\left(2^j\right) = O(\log \tau_{t,f}(x))$.

In either case, we have that the portion of the access cost dedicated to finding x is $O(\log \tau_{t,f}(x))$.

Adjusting the Data Structure. The data structure must now be adjusted. If x is found in YOUNG, then YOUNG can be restructured in the usual manner at amortized cost $O(\log\log f)$ by radix sorting the indices, as suggested by Iacono and Langerman [12]. If x is found in OLD, however, an insertion must be performed into $D'_{k'}$. If $D'_{k'}$ is implemented as a binary search tree or sorted array, this will take time $\Theta(\log f)$, which is too slow.

We therefore describe alternative implementations of $D'_{k'}$ to improve our access time.

The first alternative is to use a y-fast trie [47]. In this case, the word size can be considered $\Theta(\log n)$, since we are effectively operating in rank space. Doing so allows us to insert, delete and search in $D'_{k'}$ in amortized time $O(\log\log n)$. This follows from the fact that we know the rank of x because we have already found it in the previous stage.

Recall that the queueish dictionary still requires a way of rebuilding smaller structures from larger structures. If the other substructures are implemented as arrays, all that is required is following pointers from $Q'_{k'}$ to the oldest elements in $D'_{k'}$, placing them in an array, deleting them from $Q'_{k'}$, and proceeding as usual. This results in an amortized query time of

$$O(\log \tau_{t,f}(x)) + O(\log\log n)$$

The second alternative is to instead use the predecessor search structure of Beame and Fich [48]. Doing so allows us to insert, delete and search in $D'_{k'}$ in time

$$O\left(\min\left\{\frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

This results in an amortized query time of

$$O(\log \tau_{t,f}(x)) + O\left(\min\left\{\frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

At this point we note that, using the first technique, we match the performance of the queueish dictionary described by Iacono and Langerman [12] when $f = n$. Using the second technique, we match the performance of the working-set structure of Bădoiu et al. [14] when $f = 1$. It is also straightforward to determine in advance which technique should be used: if f is $\Omega(\log n)$, then the first technique should be used; otherwise the second technique should be used.

To summarize, we have

Theorem 1. *Let $1 \leq f \leq n$. There exists a static dictionary over the set $\{1, 2, \ldots, n\}$ that supports querying element x in amortized time*

$$O(\log \tau_{t,f}(x)) + O\left(\min\left\{\log\log n, \frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

where $\tau_{t,f}(x)$ denotes the temporal distance between the query x and the element f at time t.

There are several possible directions for future research.

1. Can the additive term in Theorem 1 be reduced? This would be interesting even for specific (ranges of) values of f. When $f = n$, for example, the best known result is $O(\log \tau_{t,n} n + \log\log n)$ [12]. The case when $f = 1$ is fully solved [14].
2. Is it possible to support multiple temporal fingers (*e.g.*, $O(1)$ many)? Simply searching the structures in parallel allows us to find the query element in time proportional to the logarithm of the minimum temporal distance, but it is not obvious how to quickly restructure the data structures and promote the query element to the cheapest substructure in parallel for each structure.
3. Is it possible to maintain the temporal finger property while supporting dynamic update operations?

References

1. Knuth, D.: Optimum binary search trees. Acta Informatica 1, 14–25 (1971)
2. Mehlhorn, K.: Nearly optimal binary search trees. Acta Inf. 5, 287–295 (1975)
3. Mehlhorn, K.: A best possible bound for the weighted path length of binary search trees. SIAM J. Comput. 6(2), 235–239 (1977)
4. Bayer, P.: Improved bounds on the cost of optimal and balanced binary search trees. Master's thesis, MIT (1975)
5. Fredman, M.L.: Two applications of a probabilistic search technique: Sorting x + y and building balanced search trees. In: STOC, pp. 240–244 (1975)
6. Shannon, C.: A mathematical theory of communication. Bell Systems Technical Journal 27, 379–423, 623–565 (1948)
7. Iacono, J.: Key-independent optimality. Algorithmica 42(1), 3–10 (2005)
8. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM 32(3), 652–686 (1985)
9. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2), 202–208 (1985)
10. Iacono, J.: Improved upper bounds for pairing heaps. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 32–45. Springer, Heidelberg (2000)
11. Bose, P., Douïeb, K., Langerman, S.: Dynamic optimality for skip lists and B-trees. In: SODA 2008: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1106–1114 (2008)
12. Iacono, J., Langerman, S.: Queaps. Algorithmica 42(1), 49–56 (2005)
13. Iacono, J.: Distribution Sensitive Data Structures. PhD thesis, Rutgers, The State University of New Jersey (2001)

14. Bădoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theoretical Computer Science 382(2), 86–96 (2007)
15. Cole, R.: On the dynamic finger conjecture for splay trees. Part II: The proof. SIAM Journal on Computing 30(1), 44–85 (2000)
16. Cole, R., Mishra, B., Schmidt, J., Siegel, A.: On the dynamic finger conjecture for splay trees. Part I: Splay Sorting $\log n$-Block Sequences. SIAM Journal on Computing 30(1), 1–43 (2000)
17. Tarjan, R.: Sequential access in splay trees takes linear time. Combinatorica 5, 367–378 (1985)
18. Bose, P., Douïeb, K., Dujmović, V., Howat, J.: Layered working-set trees. Algorithmica 63(1), 476–489 (2012)
19. Derryberry, J.C.: Adapative Binary Search Trees. PhD thesis, Carnegie Mellon University (2009)
20. Derryberry, J.C., Sleator, D.D.: Skip-splay: Toward achieving the unified bound in the BST model. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 194–205. Springer, Heidelberg (2009)
21. Bose, P., Douïeb, K., Dujmović, V., Fagerberg, R.: An $o(\log \log n)$-competitive binary search tree with optimal worst-case access times. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 38–49. Springer, Heidelberg (2010)
22. Demaine, E.D., Harmon, D., Iacono, J., Pătraşcu, M.: Dynamic optimality—almost. SIAM Journal on Computing 37(1), 240–251 (2007)
23. Wang, C.C., Derryberry, J., Sleator, D.D.: $O(\log \log n)$-competitive dynamic binary search trees. In: SODA 2006: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 374–383 (2006)
24. Andersson, A.A., Thorup, M.: Dynamic ordered sets with exponential search trees. Journal of the ACM 54(3) (2007)
25. Brodal, G.S., Lagogiannis, G., Makris, C., Tsakalidis, A., Tsichlas, K.: Optimal finger search trees in the pointer machine. Journal of Computer and System Sciences 67(2), 381–418 (2003)
26. Dietz, P.F., Raman, R.: A constant update time finger search tree. Information Processing Letters 52(3), 147–154 (1994)
27. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
28. Kaporis, A.C., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: Improved bounds for finger search on a RAM. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 325–336. Springer, Heidelberg (2003)
29. Bent, S.W., Sleator, D.D., Tarjan, R.E.: Biased search trees. SIAM Journal on Computing 14, 545–568 (1985)
30. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica 16(4/5), 464–497 (1996)
31. Bagchi, A., Buchsbaum, A.L., Goodrich, M.T.: Biased skip lists. Algorithmica 42, 31–48 (2005)
32. Johnson, D.B.: A priority queue in which initialization and queue operations take $O(\log \log D)$ time. Theory of Computing Systems 15(1), 295–309 (1981)
33. Elmasry, A.: A priority queue with the working-set property. International Journal of Foundations of Computer Science 17(6), 1455–1465 (2006)
34. Wilber, R.: Lower bounds for accessing binary search trees with rotations. SIAM Journal on Computing 18(1), 56–67 (1989)
35. Brodal, G.S., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K.: Optimal solutions for the temporal precedence problem. Algorithmica 33(4), 494–510 (2002)

36. Belazzougui, D., Kaporis, A., Spirakis, P.: Random input helps searching predecessors. arXiv:1104.4353 (2011)
37. Demaine, E.D., Iacono, J., Langerman, S.: Proximate point searching. Computational Geometry: Theory and Applications 28(1), 29–40 (2004)
38. Arya, S., Malamatos, T., Mount, D.M., Wong, K.C.: Optimal expected-case planar point location. SIAM Journal on Computing 37(2), 584–610 (2007)
39. Colette, S., Dujmović, V., Iacono, J., Langerman, S., Morin, P.: Distribution-sensitive point location in convex subdivisions. In: SODA 2008: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 912–921 (2008)
40. Iacono, J.: A static optimality transformation with applications to planar point location. arXiv:1104.5597 (2011)
41. Iacono, J., Langerman, S.: Proximate planar point location. In: SoCG 2003: Proceedings of the 19th Annual ACM Symposium on Computational Geometry, pp. 220–226 (2003)
42. Derryberry, J., Sheehy, D., Woo, M., Sleator, D.D.: Achieving spatial adaptivity while finding approximate nearest neighbors. In: CCCG 2008: Proceedings of the 20th Annual Canadian Conference on Computational Geometry, pp. 163–166 (2008)
43. Dujmović, V., Howat, J., Morin, P.: Biased range trees. In: SODA 2009: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 486–495 (2009)
44. Afshani, P., Barbay, J., Chan, T.M.: Instance-optimal geometric algorithms. In: FOCS 2009: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, pp. 129–138 (2009)
45. Elmasry, A., Farzan, A., Iacono, J.: A unifying property for distribution-sensitive priority queues. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 209–222. Springer, Heidelberg (2011)
46. Adelson-Velskii, G., Landis, E.: An algorithm for the organization of information. Soviet Math. Doklady 3, 1259–1263 (1962)
47. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)
48. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. Journal of Computer and System Sciences 65(1), 38–72 (2002)

A Survey on Priority Queues

Gerth Stølting Brodal

MADALGO*, Department of Computer Science, Aarhus University
gerth@cs.au.dk

Abstract. Back in 1964 Williams introduced the binary heap as a basic priority queue data structure supporting the operations INSERT and EXTRACTMIN in logarithmic time. Since then numerous papers have been published on priority queues. This paper tries to list some of the directions research on priority queues has taken the last 50 years.

1 Introduction

In 1964 Williams introduced "Algorithm 232" [125]—a data structure later known as *binary heaps*. This data structure essentially appears in all introductory textbooks on Algorithms and Data Structures because of its simplicity and simultaneously being a powerful data structure.

A tremendous amount of research has been done within the design and analysis of priority queues over the last 50 years, building on ideas originating back to the initial work of Williams and Floyd in the early 1960s. In this paper I try to list some of this work, but it is evident by the amount of research done that the list is in no respect complete. Many papers address several aspects of priority queues. In the following only a few of these aspects are highlighted.

2 The Beginning: Binary Heaps

Williams' binary heap is a data structure to store a dynamic set of elements from an ordered set supporting the insertion of an element (INSERT) and the deletion of the minimum element (EXTRACTMIN) in $O(\lg n)$ time, where n is the number of elements in the priority queue[1]. Williams' data structure was inspired by Floyd's 1962 paper on sorting using a tournament tree [65], but compared to Floyd's earlier work a binary heap is *implicit*, i.e. the data structure only uses one array of size n for storing the n elements without using any additional space[2]. For a set of size n it is simply an array $A[1..n]$ storing the n elements in a permutation implicitly representing a binary tree satisfying *heap order*, i.e. $A[i] \leq A[2i]$ and

* Center for Massive Data Algorithms, a Center of the Danish National Research Foundation.

[1] We let $\lg x$ denote the binary logarithm of x.

[2] In the following we denote a data structure storing $O(1)$ words of $\lg n$ bits between the operations also as being implicit. The additional space will be stated explicitly in these cases.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 150–163, 2013.

$A[i] \leq A[2i+1]$ for all $1 \leq i \leq n$ (provided $2i \leq n$ and $2i+1 \leq n$, respectively). Williams gave an algorithm for constructing a heap from an array of n elements in $O(n \lg n)$ time [125]. This was subsequently improved by Floyd [66] to $O(n)$.

The average case performance of the operations on a binary heap was studied in a sequence of papers. Porter and Simon [111] introduced the random heap model, and proved that random insertions into a random heap require about 1.61 element exchanges. Bollobás and Simon [10] considered inserting a random sequence of elements into an initially empty binary heap and proved that the average number of exchanges per element inserted is about 1.7645. Doberkat studied the average number of exchanges for ExtractMin in a random heap [38] and for Floyd's heap construction algorithm [39].

Gonnet and Munro considered the constants in the number of comparisons to maintain a binary heap [80], and gave upper and lower bounds proving that Insert requires $\lg \lg n \pm O(1)$ comparisons (the upper bound still requiring $O(\lg n)$ elements to be moved) and ExtractMin requires $\lg n + \lg^* n \pm O(1)$ comparisons. An $1.5n - O(\lg n)$ lower bound for the number of comparisons for constructing a heap was proved by Carlsson and Chen in [22].

Sack and Strothotte [113] showed how to support the merging of two binary heaps (Meld) of size n and k in time $O(k + \lg n \cdot \lg k)$, where $k \leq n$. If the heaps are represented as binary trees using pointers the additive "k" term disappears from their bound.

It is obvious that the k smallest elements in a heap can be extracted by k applications of ExtractMin in $O(k \lg n)$ time. Frederickson [73] proved that the partial order given by a binary heap (or more generally, from any heap ordered binary tree) allows us to select the k smallest elements in $O(k)$ time (reported in arbitrary order).

3 Reducing the Number of Comparisons

All the results in Section 2 assume the partial order of the original binary heaps of Williams. In this section we summarize work on lowering the constants in the number of comparisons by considering priority queues with alternative requirements with respect to the order maintained.

Johnson [88] generalized binary heaps to implicit d-ary heaps with $O(\lg_d n)$ and $O(d \lg_d n)$ time for Insert and ExtractMin, respectively. By setting $d = \omega(1)$, d-ary heaps achieve sublogarithmic insertion time. Subsequently many priority queues achieved constant time Insert and logarithmic time ExtractMin, surpassing the bounds of Johnson.

The weak heaps of Peterson and Dutton [108,41] are not completely implicit like binary heaps. They store the input elements in one array and require one additional bit per element. Edelkamp and Wegener [47] proved that sorting n elements using a weak heap uses $n \lg n + 0.09n$ comparisons, getting very close to the information theoretic lower bound of $n \lg n - 1.44n$ comparisons [67]. A refinement of weak heap sorting performs at most $n \lg n - 0.9n$ comparisons [46]. Edelkamp *et al.* [43] studied variations of weak heaps, in particular they reduced the cost of Insert to be constant.

Elmasry *et al.* [56] studied how to reduce the number of comparisons for priority queues to get close to the optimal number of comparisons, and presented a pointer based spriority queue implementation supporting INSERT with $O(1)$ comparisons and EXTRACTMIN with $\lg n + O(1)$ comparisons. Edelkamp *et al.* [45] recently achieved the same bounds by an implicit priority queue using $O(1)$ extra words of $\lg n$ bits.

4 Double-Ended Priority Queues

Atkinson *et al.* [8] generalized the partial order of binary heaps and introduced the implicit Min-Max heaps supporting both EXTRACTMIN and EXTRACTMAX in logarithmic time and having linear construction time. Essentially Min-Max heaps and all the following double-ended priority queues maintain a Min-heap and a Max-heap for some partition of the elements stored.

Carlsson introduced the implicit Deap [21] as an alternative to Min-Max heaps, improving the number of comparisons for EXTRACTMIN/EXTRACTMAX from $\frac{3}{2}\lg n + \lg\lg n$ to $\lg n + \lg\lg n$. Carlsson *et al.* [23] gave a proof that a Deap can be constructed in linear time.

General techniques to convert single ended priority queues into double-ended priority queues were presented by Chong and Sahni [32] and Elmasry *et al.* [57]. Alternative implementations of implicit double-ended queues include [106,26,37,99,6]. Ding and Weiss [35] presented an implicit double-ended priority queue for multi-dimensional data.

Double-ended priority queues supporting MELD were presented by Ding and Weiss [36], Khoong and Leong [96], and Cho and Sahni [31], which are based on min-max heaps, binomial queues, and leftist trees, respectively.

5 Implicit Data Structures

The original implicit heaps of Williams require $O(\lg n)$ worst-case time for INSERT and EXTRACTMIN. Similar bounds are achieved by several of the above mentioned implicit double-ended priority queues. Carlsson *et al.* [24] described an implicit heap with $O(1)$ time INSERT and $O(\lg n)$ time EXTRACTMIN, storing $O(1)$ extra words of $\lg n$ bits between operations. Edelkamp *et al.* [45] presented an implicit priority queue with the same time bounds and also using $O(1)$ extra words, but only requiring $\lg n + O(1)$ comparisons per EXTRACTMIN. A priority queue with amortized $O(1)$ time INSERT and $O(\lg n)$ time EXTRACTMIN was presented by Harvey and Zatloukal [84], that does not store any additional information between operations.

The existence of efficient implicit priority queue data structures implied the canonical question if efficient implicit dynamic dictionaries also existed. The study of implicit dynamic dictionaries was initiated by Munro and Suwanda [104] who proved tight $\Theta(\sqrt{n})$ bounds on implicit dictionaries satisfying a fixed partial order. The bounds for implicit dictionaries were subsequently improved by Frederickson [71] who achieved logarithmic time searches and $O(n^{\sqrt{2/\lg n}} \cdot \lg^{3/2} n)$

time updates, and the first polylogarithmic bounds were given by Munro in [101] achieving $O(\lg^2 n)$ time for both updates and searches by encoding the bits of pointers for an AVL-tree by the relative order of pairs of elements. Munro and Poblete [103] presented a semi-dynamic implicit dictionary with $O(\lg^2 n)$ time insertions and $O(\lg n)$ time searches. Subsequently update time was improved to $O(\lg^2 n/\lg n)$ by implicit B-trees [69], and eventually logarithmic time bounds where obtained by Franceschini and Grossi [68]. Franceschini and Munro [70] furthermore reduced the number of exchanges to $O(1)$ while keeping the number of comparisons per operation logarithmic (update bounds being amortized).

6 DecreaseKey and Meld

Dijkstra's single source shortest paths algorithm makes essential use of priority queues, and in particular the primitive of lowering the priority of an existing element in the priority queue. Fredman and Tarjan [77] introduced the DecreaseKey operation for this and presented Fibonacci heaps, supporting DecreaseKey in amortized constant time implying a running time of $O(m+n\lg n)$ for Dijkstra's algorithm, improving the previous bound of $O(m\lg_{m/n} n)$ achieved using an m/n-ary heap [89]. Fibonacci heaps also resulted in improved algorithms for computing minimum trees in weighted graphs with running time $O(m\lg^* n)$. Fibonacci heaps are a generalization of the binomial queues of Vuillemin [123], which achieve the same performance as Fibonacci heaps except for the DecreaseKey operation. In particular both data structures support Meld in amortized constant time. The worst-case time for Meld in a binomial queue is $\Theta(\lg n)$, but the amortized time was proven to be constant by Khoong and Leong [96].

A sequence of priority queues achieves the same amortized performance as Fibonacci heaps. Peterson [108] gave a solution based on AVL-trees, Driscoll *et al.* [40] presented the rank-relaxed heaps, Kaplan and Tarjan [94] presented the thin heaps, Chan [25] presented the quake heaps, Haeupler *et al.* [81] presented the rank-pairing heaps, and Elmasry [53] presented the violation heaps. Høyer [85] presented a general technique to construct different data structures achieving time bounds matching those of Fibonacci heaps, using red-black, AVL-trees and (a,b)-trees. Elmasry improved the number of comparisons of Fibonacci heaps by a constant factor [48].

A direction of research has been to develop priority queues with worst-case time guarantees for the operations supported by Fibonacci heaps. The run-relaxed heaps by Driscoll *et al.* [40] achieve worst-case constant time DecreaseKey operations, but Meld takes logarithmic time. The same result was achieved by Kaplan and Tarjan [93] with fat heaps. Elmasry *et al.* presented two-tier relaxed heaps [58] in which the number of comparisons for ExtractMin is reduced to $\lg n + 3\lg\lg n + O(1)$. Elmasry *et al.* [55] achieve similar bounds where DecreaseKey operations are supported with $\lg n + O(\lg\lg n)$ comparisons by introducing structural violations instead of heap order violations. The first priority queue with worst-case $o(\lg n)$ time Meld was a generalization of binomial queues by Fagerberg [63], supporting Meld in $o(\lg n)$ time and ExtractMin in time $\omega(\lg n)$. A priority queue

with constant time INSERT and MELD, and logarithmic time EXTRACTMIN and DECREASEKEY was presented by Brodal [11]. A purely functional implementation of [11] (without DECREASEKEY) was given by Brodal and Okasaki in [17].

Comparison based priority queues with worst-case constant time INSERT, DECREASEKEY and MELD and logarithmic time EXTRACTMIN were presented by Brodal [12], assuming the RAM model. Similar bounds in the RAM model were achieved by Elmasry and Katajainen [59]. Brodal *et al.* [16] recently achieved matching bounds in the pointer machine model.

Common to many of the priority queues achieving good worst-case bounds for MELD and/or DECREASEKEY are that they use some redundant counting scheme [33] to control the number of trees in a forest of heap ordered trees, the number of structural violations and/or heap order violations.

Kaplan *et al.* [92] emphasized the requirement that the DECREASEKEY operation as arguments must take both the element to be deleted and a reference to the priority queue containing this element, since otherwise FINDMIN, DECREASEKEY, or MELD must take non-constant time.

Chazelle [28] introduced the soft heap, a priority queue specialized toward minimum spanning tree computations that is allowed to perform a limited number of internal errors. A simplified version of soft heaps was given by Kaplan and Zwick [95]. Minimum spanning tree algorithms using soft heaps were presented by Chazelle [27] and Pettie and Ramachandran [110], where [110] is an optimal minimum spanning tree algorithm but with unknown complexity.

Mortensen and Pettie [43] presented an implicit priority queue supporting INSERT and DECREASEKEY in amortized constant time and EXTRACTMIN in logarithmic time, using $O(1)$ words of extra storage.

7 Self-adjusting Priority Queues

Crane [34] introduced the leftist heaps. The leftist heaps of Crane are height balanced heap ordered binary trees, where for each node the height of the left subtree is at least the height of the right subtree. Cho and Sahni [30] introduced a weight-balanced version of leftist trees. Okasaki [105] introduced maxiphobic heaps as a very pedagogical and easy to understand priority queue where operations are based on the recursive melding of binary heap ordered trees.

Sleator and Tarjan introduced the skew heaps [117] as a self-adjusting version of leftist heaps [34], i.e. a data structure where no balancing information is stored at the nodes of the structure and where the structure is adjusted on each update according to some local updating role. A tight analysis was given in [91,115] for the amortized number of comparisons performed by EXTRACTMIN and MELD in a skew heap, showing that the amortized number of comparisons is approximately $\lg_\phi n$, where $\phi = (\sqrt{5}+1)/2$ is the golden ratio. The upper bound was given by Kaldewaij and Schoenmakers [91] and the matching lower bound was given by Schoenmakers [115].

Pairing heaps [76] were introduced as a self-adjusting version of Fibonacci heaps, but the exact asymptotic amortized complexity of pairing heaps remains

unsettled. Stasko and Vitter [118] did an early experimental evaluation showing that DecreaseKey was virtually constant. Fredman later disproved this by showing a lower bound of amortized time $\Omega(\lg \lg n)$ for the DecreaseKey operation on pairing heaps [74]. Iacono [86] gave an analysis of pairing heaps achieving amortized constant Insert and Meld, and logarithmic ExtractMin and DecreaseKey operations. Pettie [109] proved an upper bound of amortized $O(2^{2\sqrt{\lg \lg n}})$ time for Insert, Meld and DecreaseKey, and amortized $O(\lg n)$ time for ExtractMin.

Variations of pairing heaps were considered in [118,51,52], all achieving amortized constant time Insert. Stasko and Vitter [118] achieved that Meld, DecreaseKey, and ExtractMin all take amortized $O(\lg n)$ time. Elmasry in [49] examined parameterized versions of skew heaps, pairing heaps, and skew-pairing heaps, both theoretically and experimentally, and in [51] and [52] showed how to improve the time bound for DecreaseKey to amortized $O(\lg \lg n)$ and the time bounds for Meld to amortized $O(\lg \lg n)$ and amortized $O(1)$, respectively.

8 Distribution Sensitive Priority Queues

Priority queues with distribution-sensitive performance have been designed and analyzed (similarly to the working-set properties of splay trees for dictionaries [116]). Fischer and Paterson's fishspear priority queue [64] supports a sequence of Insert and ExtractMin operations, where the amortized cost for handling an element is logarithmic in the "max-depth" of the element, i.e. over time the largest number elements less than the element simultaneously in the priority queue. Iacono [86] proved that for pairing heaps ExtractMin on an element takes amortized logarithmic time in the number of operations performed since the insertion of the element. The funnel-heap of Brodal and Fagerberg [13] achieves ExtractMin logarithmic in the number of insertions performed since the element to be deleted was inserted. Elmasry [50] described a priority queue where ExtractMin takes time logarithmic in the number of elements inserted after the element to be deleted was inserted and are still present in the priority queue. Iacono and Langerman [87] introduced the Queap priority queue where ExtractMin takes time logarithmic in the number of elements inserted *before* the element to be deleted and still present in the priority queue, a property denoted "queueish". Elmasry *et al.* [54] describe a priority queue with a unified property covering both queueish and working set properties.

9 RAM Priority Queues

Priority queues storing non-negative integers and where the running time depends on the maximal possible value N stored in the priority queue were presented by van Emde Boas *et al.* [60,61], who achieved Insert and ExtractMin in time $O(\lg \lg N)$ using space $O(N \lg \lg N)$ and $O(N)$ in [61] and [60], respectively. Using dynamic perfect hashing, the Y-fast tries of Willard [124] reduces the space to $O(n)$, by making the time bounds amortized randomized $O(\lg \lg N)$.

Subsequent work initiated by the fusion trees of Fredman and Willard [78] has explored the power of the RAM model to develop algorithms with $o(\lg n)$ time priority queue operations and being independent of the word size w (assuming that elements stored are integers in the range $\{0, 1, \ldots, 2^w - 1\}$). Fusion trees achieve $O(\lg n / \lg \lg n)$ time INSERT and EXTRACTMIN using linear space.

Thorup [120] showed how to support INSERT and EXTRACTMIN in $O(\lg \lg n)$ time for w-bit integers on a RAM with word size w bits (using superlinear space or linear space using hashing). Linear space deterministic solutions using $O((\lg \lg n)^2)$ amortized and worst-case time were presented by Thorup [119] and Andersson and Thorup [3], respectively. Raman [112] presented a RAM priority queue supporting DECREASEKEY, resulting in an $O(m + n\sqrt{\lg n \lg \lg n})$ time implementation of Dijkstra's single source shortest path algorithm.

That priority queues can be used to sort n numbers is trivial. Thorup in [122] proved that the opposite direction also applies: Given a RAM algorithm that sorts n words in $O(n \cdot S(n))$ time, Thorup describes how to support INSERT and EXTRACTMIN in $O(S(n))$ time, i.e. proving the equivalence between sorting and priority queues. Using previous deterministic $O(n \lg \lg n)$ time and expected $O(n\sqrt{\lg \lg n})$ time RAM sorting algorithms by Han [82] and Han and Thorup [83], respectively, this implies deterministic and randomized priority queues with INSERT and EXTRACTMIN in $O(\lg \lg n)$ and expected $O(\sqrt{\lg \lg n})$ time, respectively. Thorup [121] presented a RAM priority queue supporting INSERT and DECREASEKEY in constant time and EXTRACTMIN in $O(\lg \lg n)$ time, resulting in an $O(m + n \lg \lg n)$ time implementation of Dijkstra's single source shortest path algorithm.

A general technique to convert non-meldable priority queues with INSERT operations taking more than constant time to a corresponding data structure with constant time INSERT operations was presented by Alstrup *et al.* [2]. A general technique was described by Mendelson *et al.* [100] to convert non-meldable priority queues without DECREASEKEY into a priority queue supporting MELD in constant time and with an additive $\alpha(n)$ cost in the time for the DELETE operation, i.e. the operation of deleting an arbitrary element given by a reference. Here α is the inverse of the Ackermann function.

Brodnik *et al.* studied the power of the RAMBO model (random access machine with byte overlap). In [18] they showed how to support INSERT and EXTRACTMIN in constant time (and in [19] they showed how to perform constant time queries and updates for the dynamic prefix sum problem).

10 Hierarchical Memory Models

Early work on algorithm design in the 60s and 70s made the (by then realistic) assumption that running time was bound by the number of instructions performed, and the goal was to construct algorithms minimizing the number of instructions performed. On modern computer architectures the running time of an algorithm implementation is often not dominated by the number of instructions performed, but by other factors such as the number of cache faults, page faults,

TLB misses, and branch mispredictions. This has lead to computational models such as the I/O-model of Aggarwal and Vitter [1] and the cache-oblivious model Frigo *et al.* [79], modeling that the bottleneck in a computation is the number of cache-line or disk-block transfers performed by an algorithm. The I/O-model assumes that the parameters M and B are known to the algorithm, where M and B are the capacity in elements of the memory and a disk block, respectively. In the cache-oblivious model the block and memory parameters are not known by the algorithm, with the consequence that a cache-oblivious algorithm with good I/O-performance automatically achieves good I/O-performance on several levels.

Fadel *et al.* [62] described an amortized I/O-optimal priority queue by adopting binary heaps to external memory by letting each node store $\Theta(M)$ elements and the degree of each node be $\Theta(M/B)$. An alternative solution with the same amortized performance was achieved by Arge [4] using a "buffer tree". An external memory priority queue with worst-case bounds matching the previous structures amortized bounds was presented in [15].

Cache-oblivious priority queues were presented by Arge *et al.* [5] and Brodal and Fagerberg [13]. External memory and cache oblivious priority queues supporting an adapted version of DecreaseKey to solve the single source shortest path problem on undirected graphs with $O(n + \frac{m}{B} \lg \frac{m}{B})$ I/Os were presented by Kumar and Swabe [97] and Brodal *et al.* [14], respectively.

Fischer and Paterson [64] introduced the Fishspear priority queue designed for sequential storage such as a constant number of stacks or tapes, and using amortized $O(\lg n)$ time per Insert and ExtractMin operation.

11 Priority Queues for Sorting with Limited Space

Since the seminal paper by Munro and Patterson [102] on sorting and selection for read-only input memory with a limited read-write working space (and write-only output memory for the case of sorting), a sequence of papers have presented priority queues for sorting in this model. Frederickson [72] achieved a time-space product of $O(n^2 \lg n)$ for sorting, and [107] and [7] achieved an $O(n^2)$ time-space product for a wide-range of working space sizes, which was proven to be optimal by Beame [9].

12 Empirical Investigations

Many experimental evaluations of priority queues have been done, e.g. [20,90,98,30,75,29,44]. The importance of cache misses were observed in [98], and an implementation adopted to be cache efficient based on merging sorted lists and making efficient use of registers was presented by Sanders [114].

Modern machines are complex and an efficient implementation is not necessarily an implementation performing the fewest possible instructions. As mentioned, other parameters that are important to reduce are e.g. the number of cache misses, number of TLB misses, number of branch mispredictions, and

number of branches performed. Memory hierarchy issues can be addressed on the algorithm design level, other issues such as branch mispredictions can be reduced by using special CPU instructions such as predicated instructions such as conditional move instruction (e.g. the CMOV instruction available on the Intel Pentium II and later processors), and exploiting parallelism using e.g. SIMD instructions. Some recent work considering priority queues in this context was done by Edelkamp *et al.* [42].

13 Concluding Remarks

As stated in the introduction, this paper lists some of the research done related to priority queues, but the list is not expected to be complete. A lot of branches of related work have not been discussed. Examples are: Work on discrete event simulation that makes heavy use of priority queues, and where a lot of work on specialized priority queues has been done; priority queues in parallel models, both practical and theoretical work; concurrency issues for parallel access to a priority queue; results on sorting based on priority queues; just to mention few.

Acknowledgment. The author would like to thank Rolf Fagerberg, Andy Brodnik, Jyrki Katajainen, Amr Elmasry, Jesper Asbjørn Sindahl Nielsen and the anonymous reviewers for valuable input.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Alstrup, S., Husfeldt, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues (note). ACM Trans. Algorithms 1(1), 102–106 (2005)
3. Andersson, A., Thorup, M.: Tight(er) worst-case bounds on dynamic searching and priority queues. In: Proc. 32nd ACM Symposium on Theory of Computing, pp. 335–342. ACM (2000)
4. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37, 1–24 (2003)
5. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM J. Comput. 36(6), 1672–1695 (2007)
6. Arvind, A., Rangan, C.P.: Symmetric min-max heap: A simpler data structure for double-ended priority queue. Inf. Process. Lett. 69(4), 197–199 (1999)
7. Asano, T., Elmasry, A., Katajainen, J.: Priority queues and sorting for read-only data. In: Chan, T.-H.H., Lau, L.C., Trevisan, L. (eds.) TAMC 2013. LNCS, vol. 7876, pp. 32–41. Springer, Heidelberg (2013)
8. Atkinson, M.D., Sack, J.R., Santoro, N., Strothotte, T.: Min-max heaps and generalized priority queues. Commun. ACM 29(10), 996–1000 (1986)
9. Beame, P.: A general sequential time-space tradeoff for finding unique elements. SIAM J. Comput. 20(2), 270–277 (1991)

10. Bollobás, B., Simon, I.: Repeated random insertion into a priority queue. J. Algorithms 6(4), 466–477 (1985)
11. Brodal, G.S.: Fast meldable priority queues. In: Sack, J.-R., Akl, S.G., Dehne, F., Santoro, N. (eds.) WADS 1995. LNCS, vol. 955, pp. 282–290. Springer, Heidelberg (1995)
12. Brodal, G.S.: Worst-case efficient priority queues. In: Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58. SIAM (1996)
13. Brodal, G.S., Fagerberg, R.: Funnel heap - a cache oblivious priority queue. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 219–228. Springer, Heidelberg (2002)
14. Brodal, G.S., Fagerberg, R., Meyer, U., Zeh, N.: Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 480–492. Springer, Heidelberg (2004)
15. Brodal, G.S., Katajainen, J.: Worst-case efficient external-memory priority queues. In: Arnborg, S. (ed.) SWAT 1998. LNCS, vol. 1432, pp. 107–118. Springer, Heidelberg (1998)
16. Brodal, G.S., Lagogiannis, G., Tarjan, R.E.: Strict Fibonacci heaps. In: Proc. 44th ACM Symposium on Theory of Computing, pp. 1177–1184. ACM (2012)
17. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. J. Funct. Program. 6(6), 839–857 (1996)
18. Brodnik, A., Carlsson, S., Fredman, M.L., Karlsson, J., Munro, J.I.: Worst case constant time priority queue. J. Systems and Software 78(3), 249–256 (2005)
19. Brodnik, A., Karlsson, J., Munro, J.I., Nilsson, A.: An $O(1)$ solution to the prefix sum problem on a specialized memory architecture. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) Fourth IFIP International Conference on Theoretical Computer Science, TCS 2006. IFIP, vol. 209, pp. 103–114. Springer, Boston (2006)
20. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM J. Comput. 7(3), 298–319 (1978)
21. Carlsson, S.: The deap - a double-ended heap to implement double-ended priority queues. Inf. Process. Lett. 26(1), 33–36 (1987)
22. Carlsson, S., Chen, J.: The complexity of heaps. In: Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms, pp. 393–402. SIAM (1992)
23. Carlsson, S., Chen, J., Strothotte, T.: A note on the construction of data structure "deap". Inf. Process. Lett. 31(6), 315–317 (1989)
24. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
25. Chan, T.M.: Quake heaps: a simple alternative to Fibonacci heaps. Manuscript (2009)
26. Chang, S.C., Du, M.W.: Diamond deque: A simple data structure for priority deques. Inf. Process. Lett. 46(5), 231–237 (1993)
27. Chazelle, B.: A minimum spanning tree algorithm with inverse-ackermann type complexity. J. ACM 47(6), 1028–1047 (2000)
28. Chazelle, B.: The soft heap: an approximate priority queue with optimal error rate. J. ACM 47(6), 1012–1027 (2000)
29. Cherkassky, B.V., Goldberg, A.V., Silverstein, C.: Buckets, heaps, lists, and monotone priority queues. SIAM J. Comput. 28(4), 1326–1346 (1999)
30. Cho, S., Sahni, S.: Weight-biased leftist trees and modified skip lists. ACM J. Experimental Algorithmics 3, 2 (1998)

31. Cho, S., Sahni, S.: Mergeable double-ended priority queues. Int. J. Found. Comput. Sci. 10(1), 1–18 (1999)
32. Chong, K., Sahni, S.: Correspondence-based data structures for double-ended priority queues. ACM J. Experimental Algorithmics 5, 2 (2000)
33. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Tech. Rep. Technical Report STAN-CS-77-606, Computer Science Department, Stanford University (1977)
34. Crane, C.A.: Linear lists and priority queues as balanced binary trees. Ph.D. thesis, Stanford University, Stanford, CA, USA (1972)
35. Ding, Y., Weiss, M.A.: The K-D heap: An efficient multi-dimensional priority queue. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1993. LNCS, vol. 709, pp. 302–313. Springer, Heidelberg (1993)
36. Ding, Y., Weiss, M.A.: The relaxed min-max heap. Acta Inf. 30(3), 215–231 (1993)
37. Ding, Y., Weiss, M.A.: On the complexity of building an interval heap. Inf. Process. Lett. 50(3), 143–144 (1994)
38. Doberkat, E.E.: Deleting the root of a heap. Acta Inf. 17, 245–265 (1982)
39. Doberkat, E.E.: An average case analysis of floyd's algorithm to construct heaps. Information and Control 61(2), 114–131 (1984)
40. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM 31(11), 1343–1354 (1988)
41. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
42. Edelkamp, S., Elmasry, A., Katajainen, J.: A catalogue of algorithms for building weak heaps. In: Smyth, B. (ed.) IWOCA 2012. LNCS, vol. 7643, pp. 249–262. Springer, Heidelberg (2012)
43. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap data structure: Variants and applications. J. Discrete Algorithms 16, 187–205 (2012)
44. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap family of priority queues in theory and praxis. In: Proc. 18th Computing: The Australasian Theory Symposium, CRPIT, vol. 128, pp. 103–112. Australian Computer Society (2012)
45. Edelkamp, S., Elmasry, A., Katajainen, J.: Ultimate binary heaps (submitted, 2013)
46. Edelkamp, S., Stiegeler, P.: Implementing HEAPSORT with $(n \log n - 0.9n)$ and QUICKSORT with $(n \log n + 0.2n)$ comparisons. ACM J. Experimental Algorithmics 7, 5–24 (2002)
47. Edelkamp, S., Wegener, I.: On the performance of weak-heapsort. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
48. Elmasry, A.: Layered heaps. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 212–222. Springer, Heidelberg (2004)
49. Elmasry, A.: Parameterized self-adjusting heaps. J. Algorithms 52(2), 103–119 (2004)
50. Elmasry, A.: A priority queue with the working-set property. Int. J. Found. Comput. Sci. 17(6), 1455–1466 (2006)
51. Elmasry, A.: Pairing heaps with $O(\log \log n)$ decrease cost. In: Proc. 20th ACM-SIAM Symposium on Discrete Algorithms, pp. 471–476. SIAM (2009)
52. Elmasry, A.: Pairing heaps with costless meld. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 183–193. Springer, Heidelberg (2010)
53. Elmasry, A.: The violation heap: A relaxed Fibonacci-like heap. In: Thai, M.T., Sahni, S. (eds.) COCOON 2010. LNCS, vol. 6196, pp. 479–488. Springer, Heidelberg (2010)

54. Elmasry, A., Farzan, A., Iacono, J.: A priority queue with the time-finger property. J. Discrete Algorithms 16, 206–212 (2012)
55. Elmasry, A., Jensen, C., Katajainen, J.: On the power of structural violations in priority queues. In: Proc. 13th Computing: The Australasian Theory Symposium, CRPIT, vol. 65, pp. 45–53. Australian Computer Society (2007)
56. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms 5(1) (2008)
57. Elmasry, A., Jensen, C., Katajainen, J.: Two new methods for constructing double-ended priority queues from priority queues. Computing 83(4), 193–204 (2008)
58. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Inf. 45(3), 193–210 (2008)
59. Elmasry, A., Katajainen, J.: Worst-case optimal priority queues via extended regular counters. In: Hirsch, E.A., Karhumäki, J., Lepistö, A., Prilutskii, M. (eds.) CSR 2012. LNCS, vol. 7353, pp. 125–137. Springer, Heidelberg (2012)
60. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. 6(3), 80–82 (1977)
61. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory 10, 99–127 (1977)
62. Fadel, R., Jakobsen, K.V., Katajainen, J., Teuhola, J.: Heaps and heapsort on secondary storage. Theoretical Computer Science 220(2), 345–362 (1999)
63. Fagerberg, R.: A generalization of binomial queues. Inf. Process. Lett. 57(2), 109–114 (1996)
64. Fischer, M.J., Paterson, M.: Fishspear: A priority queue algorithm. J. ACM 41(1), 3–30 (1994)
65. Floyd, R.W.: Algorithm 113: Treesort. Commun. ACM 5(8), 434 (1962)
66. Floyd, R.W.: Algorithm 245: Treesort3. Commun. ACM 7(12), 701 (1964)
67. Ford Jr., L.R., Johnson, S.M.: A tournament problem. The American Mathematical Monthly 66(5), 387–389 (1959)
68. Franceschini, G., Grossi, R.: Optimal worst-case operations for implicit cache-oblivious search trees. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 114–126. Springer, Heidelberg (2003)
69. Franceschini, G., Grossi, R., Munro, J.I., Pagli, L.: Implicit B-trees: a new data structure for the dictionary problem. J. Comput. Syst. Sci. 68(4), 788–807 (2004)
70. Franceschini, G., Munro, J.I.: Implicit dictionaries with $O(1)$ modifications per update and fast search. In: Proc. 17th ACM-SIAM Symposium on Discrete Algorithm, pp. 404–413. SIAM (2006)
71. Frederickson, G.N.: Implicit data structures for the dictionary problem. J. ACM 30(1), 80–94 (1983)
72. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J. Comput. Syst. Sci. 34(1), 19–26 (1987)
73. Frederickson, G.N.: An optimal algorithm for selection in a min-heap. Inf. Comput. 104(2), 197–214 (1993)
74. Fredman, M.L.: On the efficiency of pairing heaps and related data structures. J. ACM 46(4), 473–501 (1999)
75. Fredman, M.L.: A priority queue transform. In: Vitter, J.S., Zaroliagis, C.D. (eds.) WAE 1999. LNCS, vol. 1668, pp. 244–258. Springer, Heidelberg (1999)
76. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica 1(1), 111–129 (1986)
77. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34(3), 596–615 (1987)

78. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. 47(3), 424–436 (1993)
79. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Foundations of Computer Science, pp. 285–297. IEEE (1999)
80. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. 15(4), 964–971 (1986)
81. Haeupler, B., Sen, S., Tarjan, R.E.: Rank-pairing heaps. SIAM J. Comput. 40(6), 1463–1485 (2011)
82. Han, Y.: Deterministic sorting in $O(n \log\log n)$ time and linear space. In: Proc. 34th ACM Symposium on Theory of Computing, pp. 602–608. ACM (2002)
83. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proc. 43rd Foundations of Computer Science, pp. 135–144. IEEE (2002)
84. Harvey, N.J.A., Zatloukal, K.C.: The post-order heap. In: Proc. 3rd International Conference on Fun with Algorithms (2004)
85. Høyer, P.: A general technique for implementation of efficient priority queues. In: Proc. 3rd Israel Symposium on Theory of Computing and Systems, pp. 57–66. IEEE (1995)
86. Iacono, J.: Improved upper bounds for pairing heaps. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 32–45. Springer, Heidelberg (2000)
87. Iacono, J., Langerman, S.: Queaps. Algorithmica 42(1), 49–56 (2005)
88. Johnson, D.B.: Priority queues with update and finding minimum spanning trees. Inf. Process. Lett. 4(3), 53–57 (1975)
89. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM 24(1), 1–13 (1977)
90. Jones, D.W.: An empirical comparison of priority-queue and event-set implementations. Commun. ACM 29(4), 300–311 (1986)
91. Kaldewaij, A., Schoenmakers, B.: The derivation of a tighter bound for top-down skew heaps. Inf. Process. Lett. 37(5), 265–271 (1991)
92. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and boolean union-find. In: Proc. 34th ACM Symposium on Theory of Computing, pp. 573–582. ACM (2002)
93. Kaplan, H., Tarjan, R.E.: New heap data structures. Tech. Rep. TR-597-99, Department of Computer Science, Princeton University (1999)
94. Kaplan, H., Tarjan, R.E.: Thin heaps, thick heaps. ACM Trans. Algorithms 4(1) (2008)
95. Kaplan, H., Zwick, U.: A simpler implementation and analysis of chazelle's soft heaps. In: Proc. 20th ACM-SIAM Symposium on Discrete Algorithms, pp. 477–485. SIAM (2009)
96. Khoong, C.M., Leong, H.W.: Double-ended binomial queues. In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 128–137. Springer, Heidelberg (1993)
97. Kumar, V., Schwabe, E.J.: Improved algorithms and data structures for solving graph problems in external memory. In: Proc. 8th Symposium on Parallel and Distributed Processing, pp. 169–177. IEEE (1996)
98. LaMarca, A., Ladner, R.E.: The influence of caches on the performance of heaps. ACM J. Experimental Algorithmics 1, 4 (1996)
99. van Leeuwen, J., Wood, D.: Interval heaps. Comput. J. 36(3), 209–216 (1993)
100. Mendelson, R., Tarjan, R.E., Thorup, M., Zwick, U.: Melding priority queues. ACM Trans. Algorithms 2(4), 535–556 (2006)
101. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. J. Comput. Syst. Sci. 33(1), 66–74 (1986)

102. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)
103. Munro, J.I., Poblete, P.V.: Searchability in merging and implicit data structures. BIT 27(3), 324–329 (1987)
104. Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. J. Comput. Syst. Sci. 21(2), 236–250 (1980)
105. Okasaki, C.: Alternatives to two classic data structures. In: Proc. 36th SIGCSE Technical Symposium on Computer Science Education, pp. 162–165. ACM (2005)
106. Olariu, S., Overstreet, C.M., Wen, Z.: A mergeable double-ended priority queue. Comput. J. 34(5), 423–427 (1991)
107. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: Proc. 39th Foundations of Computer Science, pp. 264–268. IEEE (1998)
108. Peterson, G.L.: A balanced tree scheme for meldable heaps with updates. Tech. Rep. GIT-ICS-87-23, School of Informatics and Computer Science, Georgia Institute of Technology (1987)
109. Pettie, S.: Towards a final analysis of pairing heaps. In: Proc. 46th Foundations of Computer Science, pp. 174–183. IEEE (2005)
110. Pettie, S., Ramachandran, V.: An optimal minimum spanning tree algorithm. J. ACM 49(1), 16–34 (2002)
111. Porter, T., Simon, I.: Random insertion into a priority queue structure. IEEE Trans. Software Eng. 1(3), 292–298 (1975)
112. Raman, R.: Priority queues: Small, monotone and trans-dichotomous. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 121–137. Springer, Heidelberg (1996)
113. Sack, J.R., Strothotte, T.: An algorithm for merging heaps. Acta Inf. 22(2), 171–186 (1985)
114. Sanders, P.: Fast priority queues for cached memory. ACM J. Experimental Algorithmics 5, 7 (2000)
115. Schoenmakers, B.: A tight lower bound for top-down skew heaps. Inf. Process. Lett. 61(5), 279–284 (1997)
116. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
117. Sleator, D.D., Tarjan, R.E.: Self-adjusting heaps. SIAM J. Comput. 15(1), 52–69 (1986)
118. Stasko, J.T., Vitter, J.S.: Pairing heaps: experiments and analysis. Commun. ACM 30(3), 234–249 (1987)
119. Thorup, M.: Faster deterministic sorting and priority queues in linear space. In: Proc. 9th ACM-SIAM Symposium on Discrete Algorithms, pp. 550–555. SIAM (1998)
120. Thorup, M.: On RAM priority queues. SIAM J. Comput. 30(1), 86–109 (2000)
121. Thorup, M.: Integer priority queues with decrease key in constant time and the single source shortest paths problem. J. Comput. Syst. Sci. 69(3), 330–353 (2004)
122. Thorup, M.: Equivalence between priority queues and sorting. J. ACM 54(6) (2007)
123. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
124. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Inf. Process. Lett. 17(2), 81–84 (1983)
125. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)

On Generalized Comparison-Based Sorting Problems

Jean Cardinal and Samuel Fiorini

Université libre de Bruxelles (ULB)
{jcardin,sfiorini}@ulb.ac.be

Abstract. We survey recent results on comparison-based sorting problems involving partial orders. In particular, we outline recent algorithms for partial order production and sorting under partial information. We emphasize the complementarity of the two problems and the common aspects of the algorithms. We also include open questions on two other related problems, namely partial order identification and sorting with forbidden comparisons.

1 Introduction

Sorting by comparison is a cornerstone of algorithms theory, and thorough analyses of comparison-based sorting and its relatives, such as linear-time median finding, multiple selection, and binary search trees, have been carried out since the seventies. Such analyses typically use powerful tools from analytic combinatorics.

However, optimal algorithms for simple generalizations of comparison-based sorting are not always known, or not well understood. The generalizations we consider in this paper tackle the following questions:

- What if only a partial, or approximate, ordering is required?
- What if some a priori knowledge is available on the ordering?
- What if the input data has no underlying total order?
- What if not all comparisons are allowed?

When properly formalized, these questions give rise to challenging theoretical problems. We will concentrate on versions of these problems involving partial orders and graphs.

In Section 2, we define two complementary sorting problems, called respectively *partial order production* and *sorting under partial information*. In Section 3 we define the entropy of a graph and explain its relevance to these problems. Finally, in Section 4 we describe algorithms for the two problems that are (nearly) optimal with respect to the number of comparisons they perform, and run in polynomial time. This first part of the paper summarizes the work done by a superset of the authors in collaboration with Ian Munro, and first published in 2009 and 2010. We chose to explain the two contributions in parallel, in order

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 164–175, 2013.

to highlight the common features of the problems and their solutions. In fact, we believe that the two results are two facets of the same body of knowledge.

We consider another sorting problem, called *partial order identification*, in Section 5, and summarize known results about it. In contrast with the two previous ones, finding an efficient algorithm for this latter problem is still open, in a sense that we will make precise.

A fourth sorting problem, that we refer to as *sorting with forbidden comparisons*, is defined in Section 6. Although some special cases are well-studied, it seems that this question has received less attention than the ones above.

2 Sorting *to* and *from* a Partial Order

We now define our first two problems. In what follows, $e(P)$ denotes the number of linear extensions of a partially ordered set P.

2.1 Partial Order Production

In this problem, we are given a set P of n elements partially ordered by $\preceq$, and another set S of n elements with an underlying, unknown, total order $\leq$. We wish to find a bijection $f : P \mapsto S$, such that for every $x, y \in P$, we have $x \preceq y \Rightarrow f(x) \leq f(y)$. For this purpose, we are allowed to query the unknown total order $\leq$, and aim at minimizing the number of such queries.

Hence in this problem, the objective is to sort the data partially, by placing the items of S into bins, such that they obey the prescribed partial order relation $\preceq$ among the bins. Constructing a binary heap, for instance, amounts to placing items in nodes of a binary tree, so that the element assigned to a node is always smaller or equal to those assigned to the children of this node. The multiple selection problem can also be cast as the problem of partitioning the data into totally ordered bins of fixed sizes. Both problems are special cases of the partial order production problem.

The overall number of bijections is $n!$, but for the given partially ordered set P, we have $e(P)$ feasible bijections. Hence the information-theoretic worst-case lower bound on the number of comparisons for the partial order production problem is (logarithms are base 2):

$$\log n! - \log e(P).$$

Indeed, each query cuts out a part of the solution space that should contain, for the algorithm performing the sorting, half of the remaining bijections. One may stop querying when the remaining part of the solution space only contains feasible bijections. Because the solution space is initially of size $n!$ and the number of feasible bijections is $e(P)$, the number of queries one has to make is at least $\log n! - \log e(P)$ in the worst case.

The partial order production problem was first studied in 1976 by Schönhage [27], then successively by Aigner [1], Saks [26] and Bollobás and

Hell [3]. In his survey, Saks conjectured that the problem can be solved by performing a number of comparisons that is within a linear term of the lower bound in the worst case. Saks' conjecture was eventually proved in 1989 by Yao [30]. However, in the last section of his paper, Yao asked whether there exists an algorithm for the problem that is both query-optimal and runs in polynomial time.

2.2 Sorting with Partial Information

Here we are given a set P of n elements partially ordered by $\preceq$, and we seek an underlying, unknown, total order $\leq$ that extends $\preceq$. We wish to identify the total order by querying it, and aim at minimizing the number of such queries. Hence, this is the sorting problem in which the results of some comparisons are already known and given as input.

Since we wish to identify one of the $e(P)$ linear extensions of P, the information-theoretic lower bound on the worst-case number of comparisons for this problem is simply

$$\log e(P).$$

Note that the sum of the two lower bounds for the two problems is equal to $\log n!$, the lower bound for sorting. This is not surprising, since one can sort by first solving a partial order production problem for an arbitrary partial order, then solving an instance of sorting under partial information with the same partial order as input. In fact, this is exactly what *heapsort* does. In this algorithm, we first produce a partial order whose Hasse diagram is a binary tree, then sort the items in a total order using the partial information provided by the heap.

The problem of sorting under partial information was first posed by Fredman in 1976 [15], who showed that there exists an algorithm that performs $\log e(P) + 2n$ comparisons.

In 1984, Kahn and Saks [19] showed that there is a constant $\delta > 0$ such that every (non-totally ordered) poset has a query of the form "is $v_i < v_j$?" such that the fraction of linear extensions in which $v_i < v_j$ lies in the interval $[\delta, 1 - \delta]$. They proved this for the constant $\delta = 3/11$. The well-known 1/3-2/3 conjecture (which remains open), formulated independently by Fredman, Linial and Stanley (see [22]) asserts that the same result holds for $\delta = 1/3$ (which, if true, is tight). Kahn and Linial [18] later gave a simpler proof of the existence of such a δ (with smaller value of δ). Brightwell, Felsner and Trotter [5], and Brightwell [4] further improved the value of δ.

Iteratively choosing such a comparison yields an algorithm that performs $O(\log e(P))$ comparisons. However, we do not know of any polynomial-time algorithm for finding a balanced pair, hence this does not lead to an algorithm with polynomial overall complexity. Indeed, computing the proportion of linear extensions that place v_i before v_j is $\sharp$P-complete [6].

In 1995, Kahn and Kim [17] described a polynomial-time algorithm performing $O(\log e(P))$ comparisons. Their key insight is to relate $\log e(P)$ to the entropy of the incomparability graph of P. We describe this notion below, and outline a simplification of their result yielding a more practical algorithm.

3 The Role of Graph Entropy

It is known that computing the number of linear extensions of a partial order is ♯P-complete [6]. Hence just computing the exact value of the lower bounds given above is out of reach. This is where graph entropy comes to the rescue. This notion will help us not only in finding a polynomial-time computable lower bound, but also in designing efficient algorithms for both problems.

We will only define graph entropy for comparability graphs here, which is exactly what we need for tackling our sorting problems. Consider a set P of n elements partially ordered by $\preceq$. To each element v we associate an open interval $I(v) \subseteq (0,1)$ in such a way that whenever $v \preceq w$, interval $I(v)$ is entirely to the left of interval $I(w)$. We obtain a collection $\{I(v)\}_{v \in P}$ of intervals that is said to be *consistent* with $\preceq$. The *entropy* of P is then defined as

$$H(P) := \min_{\{I(v)\}_{v \in P}} \frac{1}{n} \sum_{v \in P} \log \frac{1}{\text{length}(I(v))},$$

where the minimum is taken over all collections of intervals consistent with $\preceq$.

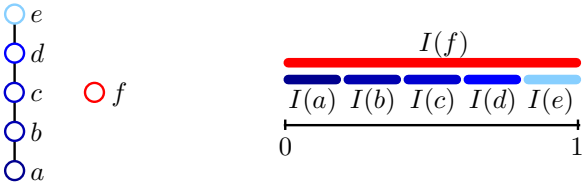


Fig. 1. A partially ordered set and its optimal consistent collection of intervals

The entropy of P in fact only depends on its comparability graph G, and coincides with the *graph entropy* $H(G)$, as first defined by Körner [21]. Because G is a perfect graph, the graph entropy of its complement $\bar{G}$ satisfies $H(\bar{G}) = \log n - H(G)$ (see Csiszár et al. [9]). This leads to defining

$$H(\bar{P}) := \log n - H(P),$$

which is exactly the graph entropy of the incomparability graph $\bar{G}$ of P.

This definition of $H(P)$ was instrumental in our work [7,8], although graph entropy admits several definitions and appears in other contexts [28].

Intuitively, $H(P)$ measures the average amount of information per element of P, and the optimal consistent collection of intervals tries to capture what the average linear extension of P looks like. Think of taking a random point in each interval; what we get as a result is a linear extension of P. Clearly, small intervals give us more information about the resulting linear extension than big intervals. For instance, in Figure 1, the contribution of f to the total amount of information is $\log 1 = 0$ and that of a is $\log 5 \approx 2.32$.

In case P is an antichain, $H(P) = 0$. In case P is a chain, $H(P) = \log n$. Between these two extreme cases, $H(P)$ is monotonic in the sense that adding comparabilities to P can never decrease $H(P)$. In particular, we always have $0 \leq H(P) \leq \log n$. For example, when P is formed of a chain of $n-1$ elements plus one element that is incomparable to all others, $H(P) = \frac{n-1}{n} \log(n-1) = \log n - \frac{1}{n} \log n - \Theta\left(\frac{1}{n}\right)$, see Figure 1 for an illustration for $n = 6$.

Similarly, $H(\bar{P})$ measures the average amount of uncertainty per element of P. For instance $H(\bar{P}) = 0$ in case P is a chain, $H(\bar{P}) = \log n$ in case P is an antichain. We have $0 \leq H(\bar{P}) \leq \log n$ and adding comparabilities to P can never increase $H(\bar{P})$.

The quantities that we will use are $nH(P)$ and $nH(\bar{P})$. These quantities measure the total amount of information and uncertainty in P, respectively. They yield lower bounds for the partial order production problem and the problem of sorting under partial information, respectively. The intuition is that each query produces information or equivalently, reduces the uncertainty, about the underlying unknown linear order. For producing P, we have to create at least as much information as contained in P. For sorting with partial information P, we have to reduce the uncertainty from that inherent in P to 0.

Theorem 1. *Let P be a partially ordered set of n elements. Then,*

$$nH(P) \leq \log n! - \log e(P) + \log e \cdot n, \tag{1}$$

$$nH(\bar{P}) \leq 2 \log e(P). \tag{2}$$

Eq. (1) is based on a simple volume argument using the so-called order polytope, see [7]. Kahn and Kim [17] proved that $nH(\bar{P}) \leq c \log e(P)$ with a constant c slightly larger than 11. Eq.(2) was proved in [8], and is tight (take P to be an antichain of two elements).

Now, we sketch a simple proof of the weaker inequality $nH(\bar{P}) \leq 4 \log e(P)$ that crucially relies on our interval-based definition of the entropy of partially ordered sets. The reader is referred to [8] for more details. The whole argument boils down to the following game: a player picks two incomparable elements a and b in P and gives them to an oracle. The oracle decides which of the two comparabilities $a \prec b$ or $b \prec a$ should be added to P. The goal for the player is to pick a and b in such a way that the entropy of the resulting partially ordered set P' always satisfies $nH(P') \leq nH(P) + 4$.

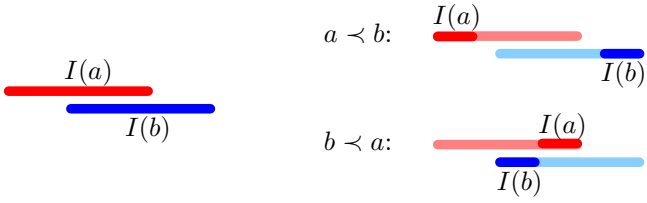


Fig. 2. The proof idea: take quarters of intervals

A winning strategy for the player is to compute the optimal consistent collection of intervals $\{I(v)\}_{v \in P}$ for P and then pick a and b such that the length of $I(a)$ is maximum and $I(b)$ contains the midpoint of $I(a)$. If the oracle answers $a \prec b$, the player changes the collection of intervals by replacing $I(a)$ by its first quarter and $I(b)$ by its last quarter. Otherwise, the oracle answers $b \prec a$ and the players replaces $I(a)$ by its last quarter and $I(b)$ by its first quarter. This is illustrated in Fig 2. In both cases, the player's changes to the collection of intervals causes an increase of $2 \log 4 = 4$ in the sum

$$\sum_v \log \frac{1}{\text{length}(I(v))}.$$

Hence, $nH(P') \leq nH(P) + 4$ in both cases.

4 Approximating the Entropy and Efficient Algorithms

At the heart of our method lies a greedy algorithm for coloring the comparability or incomparability graph of a partially ordered set P. This algorithm simply iteratively finds a maximum size independent set (or stable set), makes it a new color class and removes it from the graph, until no vertex is left. When the greedy coloring algorithm is applied to the comparability graph of P, we obtain a greedy antichain decomposition of P. When it is applied to the incomparability graph of P, we obtain a greedy chain decomposition of P. This is illustrated in Figure 3.

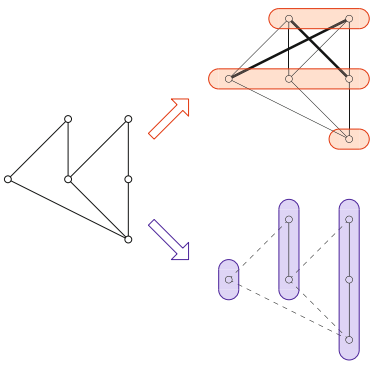


Fig. 3. Greedy antichain and chain decompositions

Any coloring of G has an *entropy*, defined as the Shannon entropy of the size distribution of its color classes. If k is the number of color classes and p_i denotes the fraction of vertices in the ith color class, then the entropy of the coloring is $\sum_{i=1}^{k} p_i \log \frac{1}{p_i}$. This actually equals the graph entropy of the graph obtained

from G by adding all edges between any two color classes. The key fact is that the entropy of this new graph is not much bigger than that of G, provided that G is perfect.

Theorem 2 ([7]). *Let G be a perfect graph on n vertices and denote by g the entropy of a greedy coloring of G. Then*

$$g \leq H(G) + \log H(G) + O(1). \tag{3}$$

We now explain how to design algorithms for the two problems that rely on greedy colorings. For the sake of a concise and readable exposition, we deliberately skip many important details.

The main idea is to replace the target or partial information P by a partially ordered set Q with a simpler structure, such that solving the problem with P replaced by Q also solves the problem for P. This may force the algorithm to perform more comparisons, but not much more, thanks to Theorem 2. Thus we trade a few extra comparisons for a major gain in terms of structure. The transformations for the two problems are illustrated on Figure 3.

4.1 From Partial Order Production to Multiple Selection

For the partial order production problem, we add comparabilities to P to transform it into a layered partially ordered set, where the elements in each layer are mutually incomparable. This is achieved by a greedy antichain decomposition of the order, as illustrated on top of Figure 3. Now the entropy of the layered order is equal to that of the greedy coloring of the comparability graph and Theorem 2 applies. (Note that it is well possible that applying the greedy coloring algorithm blindly can yield a collection of antichains that is not totally ordered. We can solve this issue by applying an uncrossing step that preserves the value of the entropy, or by applying the greedy coloring algorithm twice. Details are given in the original paper [7].)

When P is a layered partial ordered set, the partial order production problem is essentially equivalent to the multiple selection problem, for which Kaligosi, Mehlhorn, Munro and Sanders [19] give an algorithm that requires only at most $nH(P)+o(nH(P))+O(n)$ comparisons. Combining this with Theorems 1 and 2, we obtain a polynomial-time algorithm for the partial order production problem performing at most $\log n! - \log e(P) + o(\log n! - \log e(P)) + O(n)$ comparisons. Since there exists a simple $\Omega(n)$ adversarial lower bound for the problem, provided that the comparability graph of P is connected, our algorithm is query-optimal.

4.2 From Sorting under Partial Information to Multiple Merging

For the problem of sorting under partial information, we remove comparabilities from P and turn it into a union of chains. We then sort this collection of chains by iteratively picking the two minimum size chains and merging them, following

a known strategy proposed by Frazer and Bennett [14]. This exactly mimics the construction of a Huffman tree, and therefore the query complexity is easily seen to be related to the entropy of the coloring. Combining again the bounds of Theorems 1 and 2, this leads to a $O(n^{2.5})$ time algorithm performing at most $(1+\varepsilon)\log e(P) + O_\varepsilon(n)$ comparisons.

Clearly, the information-theoretic lower bound can be sublinear for this problem. For instance, think of a chain with $n-1$ elements with 1 extra element incomparable to all others as in Figure 1. There, the lower bound is $\log e(P) = \log n$ and $nH(\bar{P}) = \log n + \Theta(1)$. To obtain a query-optimal algorithm, extra work is needed. Putting aside the largest chain in the greedy chain decomposition and carefully merging this chain with the rest in a last step, remembering the comparabilities between the two parts, leads to a query-optimal algorithm. The interested reader is again referred to the original paper [8].

For both of the problems, we have query-optimal algorithms whose work can be divided in two phases: a first costly phase where P is carefully analyzed, that takes $O(n^{2.5})$ time but where no comparison is performed; a second phase where comparisons are done that takes $O(B)+O(n)$ time, where B is the corresponding information theory lower bound [7,8].

5 Partial Order Identification

We now consider the problem of identifying a partial order. Given a set P of n elements, with an unknown, underlying partial order $\preceq$, we wish to identify this partial order by querying it, and aim at minimizing the number of such queries. We suppose that we have access to an oracle that, given two elements a and b, tells us that either $a \preceq b$, $b \preceq a$, or a and b are incomparable. An algorithm for a given input size n can therefore be modeled as a ternary decision tree.

Partial order identification was studied in particular by Faigle and Turán in 1988 [13]. In this contribution, they present a number of other related problems, and a number of algorithms for the identification problem. The problem was also tackled by Dubhashi et al. [11], and much more recently by Daskalakis et al. [10].

The problem has a trivial $\Omega(n^2)$ lower bound in terms of number of comparisons, as $\binom{n}{2}$ comparisons are necessary to identify an empty order, in which all pairs are incomparable. In order to provide more interesting lower bounds, we introduce two parameters associated with a partial order P. The first parameter is the number N of downsets of P, where an downset is a subset of P that is closed for the relation $\preceq$. Hence D is an downset if and only for every $x \in D$, all elements $y \preceq x$ are also in D. The second parameter is w, the width of the order, that is, the size of a largest antichain in P. Note that while a largest antichain can be found in polynomial time, counting the number of downsets is again ♯P-complete [24].

5.1 Partial Order Identification Using Central Elements

The following lower bound was proved by Dubhashi et al. [11]:

Theorem 3. *The worst-case number of comparisons for solving the poset identification problem, given that the poset $(P, \preceq)$ is guaranteed to have at most N downsets, is $\Omega(n \log N)$ (where $n = |P|$).*

This lower bound comes from the fact that the number of partial orders with n elements and at most N downsets is exponential in $n \log N$. It appears to be achievable via a generalization of insertion sort: insert the elements one at a time in the poset induced by the previous elements. This can be carried out using a number of queries proportional to $\log N$, thanks to the existence of a so-called *central element*, a classical result due to Linial and Saks [23].

Theorem 4. *In every poset, there exists an element such that the fraction of downsets containing this element is between δ and $1 - \delta$, for a certain universal constant δ (at least 0.17).*

This algorithm was proposed by Faigle and Turán [13], although they do not seem to have noticed the optimality of the algorithm. For each element x, it performs two binary searches, one for identifying the downset consisting of elements smaller than x, and another one for the elements greater than x. The remaining elements must be those that are incomparable. The binary searches use the central elements as pivots. Here is a more detailed outline of the algorithm that, given a poset $(P, \preceq)$ and a new element $x \notin P$, identifies the downset of elements of P smaller than x. We use the notations $D_P(y) := \{z \in P : z \preceq y\}$ and $U_P(y) := \{z \in P : z \succeq y\}$.

1. **Bisect** (x, P)
2. if $P = \emptyset$, return $\emptyset$
3. choose a central element $y \in P$
4. if $x \succ y$:
   (a) let $P' := P \setminus D_P(y)$
   (b) let $Q :=$ Bisect (x, P')
   (c) return $D_P(y) \cup Q$
5. else:
   (a) let $P' := P \setminus U_P(y)$
   (b) return Bisect (x, P')

It can be checked that the number of remaining downsets after each comparison is cut down by a constant factor. Indeed, in the case where $x \succ y$, there cannot be more downsets in $P \setminus D_P(y)$ than downsets in P that contain y. Otherwise, all the downsets in $P \setminus U_P(y)$ are in one-to-one correspondence with downsets of P that do not contain y. In both cases, this number is a constant fraction of N.

However, the bisection, and therefore the whole insertion sort algorithm, is not known to admit a polynomial-time implementation. This is because we are so far not able to identify a central element in polynomial time. Indeed, any polynomial-time probabilistic algorithm for finding a δ-central element of a poset would yield a polynomial-time algorithm for estimating its number of downsets to within an arbitrary small factor with high probability [11].

Finding a 1/4-central element can be done in polynomial time in a 2-dimensional poset, as shown by Steiner [29]. Faigle et al. [12] study other special cases of posets, and give polynomial-time algorithms for finding central elements in interval and series-parallel orders. Hence identifying 2-dimensional, interval, and series-parallel orders can be done optimally in terms of queries, and the algorithm is polynomial.

5.2 Partial Order Identification Using Chain Decompositions

An alternative algorithm was proposed by Faigle and Turán [13] to implement the insertion sort procedure. For each inserted element x, this algorithm computes a chain decomposition of the current poset, and performs a bisection search for x in each of the chains. This algorithm has query complexity $O(nw \log(n/w))$, where w is the width of the poset. This is straightforward from the fact that there exists a chain decomposition into w chains (Dilworth's Theorem), which can be found in polynomial time. However, this is not tight with respect to the following lower bound as a function of w, proved by Daskalakis et al. [10].

Theorem 5. *The worst-case number of comparisons for solving the poset identification problem, given that the poset $(P, \preceq)$ is guaranteed to have width at most w, is $\Omega(n(\log n + w))$.*

Note that the lower bound with respect to w is not comparable to the lower bound with respect to N, since the sets of possible inputs are distinct. Daskalakis et al. [10] also give an algorithm that reaches the $\Omega(n(\log n + w))$ lower bound. However, again, this algorithm is not polynomial. In particular, it requires the exact computation of the number of linear extensions (under some constraints) of the current poset. They also propose a practical implementation of Faigle and Turán's suboptimal chain decomposition strategy.

Hence, to the best of our knowledge, the following question is still open: find a query-optimal polynomial-time algorithm for partial order identification, where query-optimality is with respect to either the width w or the number of downsets N.

6 Sorting with Forbidden Comparisons

In this last section, we briefly summarize previous results on sorting problems involving *forbidden comparisons*, that is, in which some designated pairs of elements cannot be compared.

A well-known special case of this problem is the so-called *nuts and bolts* problem. Let $B = \{b_1, b_2, \ldots, b_n\}$ (the bolts) be a set of totally ordered elements (say, real numbers), and let $S = \{s_1, s_2, \ldots, s_n\}$ (the nuts) be a permutation of B. The numbers correspond to the widths of the nuts and bolts, and the goal is to match them. The only comparison operations that are allowed are between nuts and bolts, that is, we are only allowed to test whether $s_i \leq b_j$ for some pair $i, j \in [n]$. It is not difficult to find a randomized Quicksort-like algorithm that

runs in expected $O(n \log n)$ time [25]. In 1994, Alon et al. [2] proposed a deterministic algorithm running in time $O(n(\log n)^{O(1)})$. Komlós, Ma and Szemerédi gave a deterministic $O(n \log n)$ algorithm in 1996 [20].

The problem of sorting with forbidden comparisons generalizes the nuts and bolts problem. We are given a graph $G = (V, E)$. The set V is the set of elements to sort, and E is the set of pairs for which comparison is allowed. When we compare a pair $(u, v) \in E$, we are given the orientation of the edge uv in G corresponding to the order of the elements u, v. The goal is to unveil the underlying total order on V. We are also promised that probing all edges effectively yields a total order. Hence G has a Hamiltonian path, and the orientation of the edges that are not in this path are implied by transitivity.

In 2011, Huang et al. [16] proposed an algorithm performing $O(n^{3/2} \log n)$ queries. This algorithm maintains a likeliness information about the orientation of each edge, and implements a two-way strategy that at each step identifies an edge to probe, or finds the orientation of the edges incident to a subset of $O(\sqrt{n})$ vertices in $O(n \log n)$ time. A subroutine of the algorithm involves estimating the average ranks of the elements by sampling the order polytope, a technique reminiscent of the ones used for sorting under partial information. There is not any interesting lower bound for this problem, however, apart from the standard $\Omega(n \log n)$ bound for sorting. Increasing this lower bound would definitely be an important progress.

Acknowledgments. This work is supported by the ARC Convention AUWB-2012-12/17-ULB2 (COPHYMA project), and the ESF EUROCORES programme EuroGIGA, CRP ComPoSe, F.R.S.-FNRS Grant R70.01.11F. We thank our coauthors on [7,8], Gwenaël Joret for proofreading a draft of this manuscript, as well as the anonymous referees for their careful reading and many important precisions.

References

1. Aigner, M.: Producing posets. Discrete Math. 35, 1–15 (1981)
2. Alon, N., Blum, M., Fiat, A., Kannan, S., Naor, M., Ostrovsky, R.: Matching nuts and bolts. In: SODA, pp. 690–696 (1994)
3. Bollobás, B., Hell, P.: Sorting and graphs. In: Graphs and Order, Banff, Alta, 1984, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., vol. 147, pp. 169–184. Reidel, Dordrecht (1985)
4. Brightwell, G.R.: Balanced pairs in partial orders. Discrete Mathematics 201(1-3), 25–52 (1999)
5. Brightwell, G.R., Felsner, S., Trotter, W.T.: Balancing pairs and the cross product conjecture. Order 2(4), 327–349 (1995)
6. Brightwell, G.R., Winkler, P.: Counting linear extensions. Order 8(3), 225–242 (1991)
7. Cardinal, J., Fiorini, S., Joret, G., Jungers, R.M., Munro, J.I.: An efficient algorithm for partial order production. SIAM J. Comput. 39(7), 2927–2940 (2010)
8. Cardinal, J., Fiorini, S., Joret, G., Jungers, R.M., Munro, J.I.: Sorting under partial information (without the ellipsoid algorithm). In: STOC, pp. 359–368 (2010); To appear in Combinatorica

9. Csiszár, I., Körner, J., Lovász, L., Marton, K., Simonyi, G.: Entropy splitting for antiblocking corners and perfect graphs. Combinatorica 10(1), 27–40 (1990)
10. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S., Verbin, E.: Sorting and selection in posets. SIAM J. Comput. 40(3), 597–622 (2011)
11. Dubhashi, D.P., Mehlhorn, K., Ranjan, D., Thiel, C.: Searching, sorting and randomised algorithms for central elements and ideal counting in posets. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 436–443. Springer, Heidelberg (1993)
12. Faigle, U., Lovász, L., Schrader, R., Turán, G.: Searching in trees, series-parallel and interval orders. SIAM J. Comput. 15(4), 1075–1084 (1986)
13. Faigle, U., Turán, G.: Sorting and recognition problems for ordered sets. SIAM J. Comput. 17(1), 100–113 (1988)
14. Frazer, W.D., Bennett, B.T.: Bounds on optimal merge performance, and a strategy for optimality. J. ACM 19(4), 641–648 (1972)
15. Fredman, M.L.: How good is the information theory bound in sorting? Theor. Comput. Sci. 1(4), 355–361 (1976)
16. Huang, Z., Kannan, S., Khanna, S.: Algorithms for the generalized sorting problem. In: FOCS, pp. 738–747 (2011)
17. Kahn, J., Kim, J.H.: Entropy and sorting. J. Comput. Syst. Sci. 51(3), 390–399 (1995)
18. Kahn, J., Linial, N.: Balancing extensions via Brunn-Minkowski. Combinatorica 11, 363–368 (1991)
19. Kaligosi, K., Mehlhorn, K., Munro, J.I., Sanders, P.: Towards optimal multiple selection. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 103–114. Springer, Heidelberg (2005)
20. Komlós, J., Ma, Y., Szemerédi, E.: Matching nuts and bolts in o(n log n) time. SIAM J. Discrete Math. 11(3), 347–372 (1998)
21. Körner, J.: Coding of an information source having ambiguous alphabet and the entropy of graphs. In: Transactions of the 6th Prague Conference on Information Theory, pp. 411–425 (1973)
22. Linial, N.: The information-theoretic bound is good for merging. SIAM J. Comput. 13(4), 795–801 (1984)
23. Linial, N., Saks, M.: Every poset has a central element. J. Comb. Theory, Ser. A 40(2), 195–210 (1985)
24. Provan, J.S., Ball, M.O.: The complexity of counting cuts and of computing the probability that a graph is connected. SIAM J. Comput. 12(4), 777–788 (1983)
25. Rawlins, G.J.E.: Compared to what? - an introduction to the analysis of algorithms. Principles of computer science series. Computer Science Press (1992)
26. Saks, M.E.: The information theoretic bound for problems on ordered sets and graphs. In: Graphs and order, Banff, Alta, 1984, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci, vol. 147, pp. 137–168. Reidel, Dordrecht (1985)
27. Schönhage, A.: The production of partial orders. In: Journées Algorithmiques, École Norm. Sup., Paris, 1975, pp. 229–246, Astérisque, No. 38–39. Soc. Math., France (1976)
28. Simonyi, G.: Graph entropy: a survey. In: Combinatorial Optimization (New Brunswick, NJ, 1992–1993. DIMACS Ser. Discrete Math. Theoret. Comput. Sci, vol. 20, pp. 399–441. Amer. Math. Soc., Providence (1995)
29. Steiner, G.: Searching in 2-dimensional partial orders. J. Algorithms 8(1), 95–105 (1987)
30. Yao, A.C.: On the complexity of partial order productions. SIAM J. Comput. 18(4), 679–689 (1989)

A Survey of the Game "Lights Out!"

Rudolf Fleischer[1,2] and Jiajin Yu[3]

[1] IIPL and SCS, Fudan University, Shanghai, China
[2] AIT Dept., GUtech, Muscat, Oman
rudolf.fleischer@gmail.com
[3] College of Computing, Georgia Institute of Technology, USA
jiajyu@gmail.com

Abstract. LIGHTS OUT! is an electrical game played on a 5×5-grid where each cell has a button and an indicator light. Pressing the button will change the light of the cell and the lights of its rectilinear adjacent neighbors. Given an initial configuration of lights, some on and some off, the goal of the game is to switch all lights off. The game can be generalized to arbitrary graphs instead of a grid. LIGHTS OUT! has been studied independently by three different communities, graph theoreticians, gamers, and algorithmicists. In this paper, we survey the game and present the results in a unified framework.

1 Introduction

The solitaire game LIGHTS OUT! is played on a 5×5-grid where each cell has a button and an indicator light. Pressing the button will change the light and the lights of all its rectilinear adjacent neighbors from on to off or vice versa. Given an arbitrary initial on/off pattern of lights, the goal of the game is to switch all lights off by pressing a subset of the buttons. The game can naturally be generalized to arbitrary graphs instead of a grid.

This game was first studied by Sutner [67,68] in the context of cellular automata (he tried to characterize configurations without predecessors, so-called *Garden-of-Eden* configurations). He showed that we can, for any graph, turn all lights off if initially all lights are on. Later, several simple proofs of this theorem were given or rediscovered [19,35,45,47] and variants of the game were studied [11,38,39,68]. Linear algebra, combinatorics, and the theory of cellular automata were employed to derive the results. After the all-on-to-all-off problem was was fairly well understood, Amin and Slater [5] and Goldwasser et al. [45,47] focused on universally solvable graphs. Furthermore, Armin and Slater [5] and Conlon et al. [34] studied the complexity of determining the minimum number of steps to turn off all lights. Goldwasser [44] studied maximization variants of the game.

This paper is organized as follows. We present the game and some linear algebra background in Section 2. In Section 3, we present three proofs for the all-on to all-off game, in particular we propose a new greedy algorithm to compute a solution. We continue with a characterization of completely solvable instances of various graph classes in Section 4. In Section 5 we turn to the complexity of

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 176–198, 2013.

optimization variants of the game and we show that it is NP-hard to find the minimum number of steps to turn off all lights when we start with all lights on. We also study the complexity of the game on special graph classes. At the end of each section, we give a short summary of previous results.

2 Preliminaries

2.1 The Game

First, we give a formal definition of the game Lights Out!. We are given an undirected graph $G = (V, E)$ with n nodes, where each node $v \in V$ has a *state* (light) $C_v \in \{0, 1\}$. We say v is *off* if $C_v = 0$, and *on* if $C_v = 1$.

A *configuration* of the game is a binary vector C of length n, where C_v is the state of node v. One step of the game consists of choosing a node v and flipping the states of v and all its neighbors in G from on to off or vice versa. We call this step an *activation* of node v. We call the configuration where all nodes are on (off) the *all-on (all-off) configuration*. The goal of the game is to reach the all-off configuration from the given initial configuration by a finite sequence of activations, in which case we say the initial configuration is *solvable*. We say a graph is *universally solvable* if every initial configuration is solvable.

Clearly, the order of activations is not important and it is not necessary to select a node more than once. Thus the game reduces to the question whether we can find a subset X of the nodes, called an *activation set*, such that activating all nodes in X will turn off all nodes in G. A *minumum solution* of the game is an activation set of minimum cardinality. The *characteristic vector* $\boldsymbol{x}$ of X is a vector of dimension n with a 1 entry at all positions indexed by nodes in X, and 0 entries otherwise.

We can generalize the game by introducing a *neighborhood vector* F of dimension n. For each node v, if $F_v = 1$ then activating node v will also flip the state of v (i.e., this is the model we used above), while $F_v = 0$ means that the state of v will not change, only the states of its neighbors change.

In this paper, we follow the notation by Sutner [68] and let σ^+ denote the game **1**-Lights Out! (using closed neighborhoods), while σ denotes the game **0**-Lights Out! (using open neighborhoods). More generally, σ^F denotes the game F-Lights Out!, or Lights Out! with *F-neighborhood constraints*.

2.2 The Math

Let $G = (V, E)$ be an undirected graph with $n = |V|$ nodes and $m = |E|$ edges. For a node $v \in V$, the *open neighborhood* $N(v)$ of v is the set of nodes adjacent to v, i.e., $N(v) = \{u \mid (u, v) \in E\}$. The *closed neighborhood* $N[v]$ is the open neighborhood plus v itself. We denote by A_G the adjacency matrix of G where all entries on the main diagonal are 1, i.e., A_G represents the closed neighborhood relation of the nodes in V.

A subset W of V is a *dominating set*, or *node cover*, of G if every node in V contains at least one node of W in its closed neighborhood. The *domination*

number $\gamma(G)$ of G is the size of a minimum dominating set. A *perfect cover* (or *efficient cover*) covers each node exactly once. More generally, an *odd (even) cover* W of G is a cover such that the closed neighborhood of each node contains an odd (even) number of nodes in W. For example, the emtpy set is an even cover of any graph G, and any perfect cover is an odd cover.

We can further generalize this concept. Let D be a binary vector of dimension n, called a *parity vector*. We say a node v is *even* if $D(v) = 0$, and *odd* if $D(v) = 1$. A *D-parity cover* of G is a nonempty subset W of V such that the closed neighborhood of each even (odd) node contains an even (odd) number of nodes in W. For example, V is a D-parity cover for the parity vector D which is defined by $D(v) = 0$ if $deg(v)$ is odd and $D(v) = 1$ if $deg(v)$ is even.

We say a graph is *APR (all parity realizable)* [4] if there exists a D-parity cover for any parity vector D. If we interpret D as an initial configuration for σ^+, then D-parity covers are exactly the activation sets solving the game. We thus obtain the following theorem relating LIGHTS OUT! and odd covers which can be found in most early papers on the subject.

Theorem 1. *An undirected graph is universally solvable if and only if it is APR. In particular, minimum σ^+ solutions for an initial configuration D are exactly the minimum cardinality D-parity covers of G.* □

We can extend the definitions above to open neighborhoods, or even a mixture of open and closed neighborhoods. Let F be a binary vector of dimension n, called the *neighborhood vector*. An *F-neighborhood cover* of G is a node cover where we use in the definition of "cover" the open neighborhood of a node v if $F(v) = 0$ and its closed neighborhood if $F(v) = 1$. If not specified, we always assume that $F(v) = 1$ for all nodes v, i.e., the default is to use closed neighborhoods. Note that F-neighborhood covers are exactly the activation sets for σ^+. The F-neighborhood relation can be described by the *F-adjacency matrix* A_G^F of G which is the adjacency matrix of G with F on the main diagonal.

We will also consider some special graph classes. P_n is a *path* of n nodes, C_n a *cycle* of n nodes, and $G_{n,m} = P_n \times P_m$ is the *complete $n \times m$ grid graph* with n columns and m rows. Any subgraph of some $G_{n,m}$ is a grid graph. K_n is the complete graph on n nodes, and $K_{n,m}$ is the complete bipartite graph on n and m nodes, respectively. A *caterpillar* consists of a path, called the *spine*, and an arbitrary number of nodes attached to the spine nodes, called *feet*. The nodes on the spine are called *segments*. A uniform caterpillar $C_{n,m}$ has a spine of length n and m feet attached to each segment.

We assume some familiarity with basic concepts from linear algebra. Throughout the paper, we do arithmetic in $GF(2)$ unless explicitly stated otherwise, i.e., addition is usually the binary XOR operation. Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are *orthogonal* if $\boldsymbol{x} \cdot \boldsymbol{y} = 0$. The *weight* $wt(\boldsymbol{x})$ of a binary vector is the number of ones in $\boldsymbol{x}$.

We denote by $diag(A)$ the main diagonal vector of A. The *image* of a matrix A is the set of all vectors $A\boldsymbol{x}$, for all $\boldsymbol{X}$. The *kernel*, or *nullspace*, $\ker A$ of A is the set of vectors $\boldsymbol{x}$ such $A\boldsymbol{x} = \boldsymbol{0}$, i.e., the kernel is orthogonal to A, and its dimension is the *nullity* of A. Amin et al. [3] called the nullity the *parity dimension* $PD(G)$ of G.

The *range* $R(A)$ of A is the set of vectors $A\boldsymbol{x}$, for all $\boldsymbol{x}$, its dimension is the *rank* of A. Note that rank and nullity add up to the number of columns of A. If $\boldsymbol{y}$ is orthogonal to $\ker A$, then $A\boldsymbol{x} = \boldsymbol{y}$ has a solution over $GF(2)$. An $n \times n$-matrix A is invertible if and only if $\ker A = \{\mathbf{0}\}$, i.e., the nullity of A is 0 and A has rank n.

2.3 Historical Review

LIGHTS OUT!. The commercial game LIGHTS OUT! is played on $G_{5,5}$ [38]. We can find many links and other materials on LIGHTS OUT! on Jaap's Puzzle Page [65]. Lotto [58] analysed the game *Quinto* which is equivalent to finding odd covers of a 5×5 grid. The commercial game `Orbix` is the open neighborhood version of LIGHTS OUT!, played on an icosahedron [38]. `Merlin` is LIGHTS OUT! played on a certain directed graph with nine nodes (a directed 3×3-grid with diagonals), see [62].

Fraenkel [40] studied a two-player variant of LIGHTS OUT! where the two players start from some initial configuration, and the first player switching off all lights wins.

Odd Covers. Sutner [67,68] observed that finding an odd cover for an undirected graph G is equivalent to solving $A_g \cdot \boldsymbol{x} = \mathbf{1}$ over $GF(2)$. This immediately implies an $O(n^3)$ algorithm to compute odd covers. Surprisingly, every graph has an odd cover. Sutner [67,68] gave an proof based on σ-automata. Caro [19] gave a simpler proof based on the following lemma [2]: $A\boldsymbol{x} = \mathbf{1}$ has no solution over $GF(2)$ if and only if A has an odd number of rows whose sum is zero (these rows would induce an odd-cardinality subgraph where all nodes have odd degree, which is impossible).

The number of different odd covers is 2^d, where d is the dimension of the nullspace (or kernel) of A_G, and any even cover must have even cardinality [3,67,68]. Dawes [36] observed that two odd covers differ by an element in the nullspace and that trees can have exponentially many odd covers. Amin and Salter [5] showed that for any parity vector D the number of different D-parity covers is the same.

$G_{4,4}$ has 16 odd covers (any subset of the nodes in the top row can be extended to an odd cover), $G_{5,5}$ has four odd covers, and $G_{6,6}$ has only one odd cover. P_n has one odd cover if $n \equiv 0, 1 \bmod 3$, and two odd covers if $n \equiv 2 \bmod 3$. C_n has one odd cover if $n \not\equiv 0 \bmod 3$, and four odd covers otherwise.

Galvin [41] proposed an algorithm to find an odd cover of trees in linear time. Dawes [36] proposed an $O(n \log n)$ time algorithm to find a minimum odd cover of trees, which was later extended by Amin and Slater to series-parallel graphs [4] and graphs of bounded treewidth by Gassner and Hatzl [42].

Cellular Automata. Sutner [67] studied non-uniform binary cellular automata on directed graphs, so-called σ-automata, whose behaviour is equivalent to playing LIGHTS OUT!, which is equivalent to determining odd covers. He distinguished between σ (open neighborhood) and σ^+ (closed neighborhood) games.

Conlon et al. [34] studied *neighborhood inversions* in graphs, which is equivalent to playing LIGHTS OUT!.

Domination Numbers. For an old survey on graph domination see Cockayne and Hedetniemi [33]. MacGillivray and Seyffarth [59] showed that the domination number of planar graphs of diameter two is at most three, and that of diameter three is at most 10; it can be unboundend for planar graphs of diameter four and non-planar graphs of diameter two. Goddard et al. [43] showed that there is only one unique planar diameter-two graph with domination number two; if the diameter is three and the radius is two then the domination number is at most six; large diamater-three planar graphs have domination number at most seven.

Jacobson et al. [51] presented closed formulas for $\gamma(G_{n,k})$ for $k = 1, \ldots, 4$: $\gamma(G_{n,2}) = \lceil \frac{n+1}{2} \rceil$, $\gamma(G_{n,3}) = n - \lfloor \frac{n-1}{4} \rfloor$, and $\gamma(G_{n,4}) = n + 1$ if and only if $n = 1, 2, 3, 4, 5, 9$ and $\gamma(G_{n,4}) = n$ otherwise. Further, they showed that $\lim_{m,n\to\infty} \frac{\gamma(G_{n,m})}{mn} = \frac{1}{5}$ and that $\gamma(G \times H) \geq \gamma(G) \cdot \gamma(H)$ if G is connected, of order $2n$, $\gamma(G) = n$, and $G \neq C_4$ (Cockayne had conjectured this inequality for all graphs). Cockayne et al. [31] proposed similar results, in particular they gave nearly tight lower and upper bounds for $\gamma(G_{n,n})$ (it is between $\frac{n^2+n-3}{5}$ and approximately $\frac{n^2+4n-16}{5}$). Chang et al. gave closed formulas for $k = 5, 6$ [23] and explicitly constructed minimum dominating sets for $k = 1, \ldots, 10$ [23,24].

Hare et al. [49] gave a linear time algorithm to compute $\gamma(G_{n,k})$ for fixed k, but the runtime is exponential in k. They observed that there are 2^k ways to choose nodes on the top row, and $\frac{(1+\sqrt{2})^{k+1}+(1-\sqrt{2})^{k+1}}{2}$ ways to cover or not cover the top row. Livingston and Stout [56] gave a constant time algorithm for fixed k that can also compute minimum or perfect dominating sets; the algorithm can be extended to graphs $G \times P_n$, where P_n can be a path, a cycle, or a complete binary tree. Recently, Goncalves et al. [48] settled the problem and determined $\gamma(G_{n,m})$.

Berman et al. [13] and Clark et al. [27] showed that it is NP-complete to compute the domination number of grid graphs and unit disk graphs.

Kikuno et al. [53] gave a linear time algorithm to determine the domination number of series-parallel graphs. Pfaff et al. [63] extended this algorithm to compute the *total domination number* (open neighborhood) and the *independent domination number*, i.e., the size of a smallest dominating independent set, or the size of a smallest clique in $\bar{G}$ [14] (dynamic program with sixteen labels).

Cockayne et al. [31] compared $\gamma(G)$ with the *irredundance number* $ir(G)$ (the smallest maximal non-redundant subset of nodes, where a node is redundant if its closed neighborhood is already covered) and the independent domination number $i(G)$. In [32], they proved $ir(G) \leq \gamma(G) \leq i(G)$. In [17], they showed that $i(G) \leq \gamma(G)(k-1)-(k-2)$ if G does not contain an induced $K_{1,k+1}$, generalizing an earlier result by Allan and Laskar [1] who had shown that $\gamma(G) = i(G)$ if G does not contain an induced $K_{1,3}$, and that $ir(G) > \frac{\gamma(G)}{2}$. Beyer et al. [14] conjectured that determining $i(G)$ is NP-complete, and they gave a linear time algorithm for trees (dynamic program with three labels).

Clark et al. [28,29,30] studied domination number of graphs with degree constraints. Caro et al. [22] studied connected odd covers and showed that it is NP-complete to decide whether such a cover exists. Caro et al. [20,21] studied more general modulo based domination problems.

Perfect Covers. Perfect d-covers were introduced by Biggs [16,15] to study perfect d-error correcting codes. Later, Bange et al. [10] characterized trees with one or two disjoint perfect 1-covers and called them *efficient dominating sets*. They showed that all perfect covers have the same cardinality and proposed a linear time algorithm to construct them on trees. They also showed that it is in general NP-complete to decide the existence of a perfect 1-cover (reduction from 3SAT), even in three-regular planar graphs (this result was attributed to Mike Fellows). In [9], they computed efficient near-dominating sets (minimizing the number of uncovered nodes) in all $G_{n,m}$.

Masters et al. [61] showed that a graph has at most one perfect dominating set if its adjacency matrix is invertible.

Livingston and Stout [55] gave a linear time algorithm to compute perfect d-covers of trees and series-parallel graphs. They also characterized graphs with perfect d-covers for several classes of graphs (paths, cycles, trees, grids, hypercubes, cube connected cycles, and de Bruijn graphs). Jacobson and Peters [52] showed that determining $\gamma_k(G)$ (cover each node at least k times) is NP-complete, and gave linear time algorithms for trees and seriell-parallel graphs (recursive algorithm, with 4 labels for the tree). For recent results on perfect covers see, for example, Brandstädt et al. [18].

3 Solving All-On Configurations

In this section we give three proofs for the amazing fact that the all-on configuration can be solved for any graph. The first proof uses graph-theory, the second one linear algebra, and the third one is algorithmic.

Theorem 2 ([67]). *The all-on configuration is solvable for any undirected graph.*

3.1 A Graph Theoretical Proof

Cowen et al. [35] and Eriksson et al. [39] proposed a proof by induction on n, the number of nodes in G.

Proof (Theorem 2). If $n = 1$, then there is only one node in G. Selecting this node will flip its state from on to off.

Assume the claim is true for graphs with at most n nodes. Consider a graph G with $n + 1$ nodes. For any node v in G, let X_v be an activation set for the n-nodes graph $G - \{v\}$. If there exists a node v such that X_v is also a solution for G, then we are done. Otherwise, each X_v will turn all nodes off except node v.

If n is odd, then we apply all the activitation sets X_v, for all nodes v. Each node v will not change its state when we apply X_v, and it will change its state an odd number of times when we apply the union of all sets X_w, for all $w \neq v$. Since there is an odd number of such nodes w, v will change its state and turn off.

If n is even, then at least one node v of G has even degree (the sum of all node degrees in G equals twice the number of edges, i.e., it is even). We first activate v. As a result, all nodes in $N[v]$ will turn off. Then we apply the activation sets

X_u, for all $u \in G - N[v]$. Since $n+1$ and $|N[v]|$ are odd, there is an even number of such nodes u. Thus, the nodes in $N[v]$ are not affected and will remain off because they will change their state an even number of times, while the nodes in $G - N[v]$ will change their state an odd number of times (X_w does not change the state of w), i.e., they will also turn off. □

Note that the proof implies an algorithm to compute an activation set. However, the running time is exponential in n (we must basically compute activation sets for all subgraphs of G). We will see more efficient algorithms in the next subsections.

3.2 An Algebraic Proof

We now state a more general result that was first reported by Dodis and Winkler [38], generalizing a proof by Goldwasser et al. [45,47]. It follows directly from the following theorem from linear algebra. The special case where all entries on the main diagonal are 1 was also studied by Sutner [67], Lotto [58], and Cowen et al. [35].

Theorem 3 ([45,47]). *For any symmetric binary matrix A, the linear equation $A\boldsymbol{x} = diag(A)$ has a solution over $GF(2)$.*

Proof. Let $F = diag(A)$. Consider a vector $\boldsymbol{y}$ in $\ker A$, i.e., $A\boldsymbol{y} = \mathbf{0}$. Let B be the submatrix of A where we delete all rows and columns v with $F_v y_v = 0$. Note that B is still symmetric and $diag(B) = \mathbf{1}$. The corresponding subvector of $\boldsymbol{y}$ is the vector $\mathbf{1}$. Thus, $B\mathbf{1} = \mathbf{0}$. This means that every row of B contains an even number of ones. Since B is symmetric, this is only possible if B has an even number of rows. This implies $F\boldsymbol{x} = 0$ (because $F_v y_v = 1$ for an even number of indices v), i.e., F is orthogonal to the kernel of A. But then F is in the image of A, i.e., there exists a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = F$. □

For a closed neighborhood adjacency matrix A_G, where the diagonal vector is $\mathbf{1}$, Theorem 3 says that the equation $A_G \cdot \boldsymbol{x} = \mathbf{1}$ always has a solution over $GF(2)$. The next theorem generalizes Theorem 2 to arbitrary F-neighborhood configurations.

Theorem 4 ([38]). *Let G be an undirected graph and F an arbitrary neighborhood vector. Then there exists an F-neighborhood F-parity cover of G, i.e., the initial configuration F is solvable for σ^F.*

Proof. The initial configuration is F. Consider a set X of nodes with characteristic vector $\boldsymbol{x}$. After activating X, each node v switches to state

$$F_v + \sum_{u \in N(v) \cap X} x_u + F_v x_v = F_v + (A_G^F \boldsymbol{x})_v \,,$$

where F_v is the state of v before the activation, the sum describes the effect of activating all nodes in the open neighborhood of v, and $F_v x_v$ is the possible contribution of activating v itself. Since $diag(A_G^F) = F$, we can apply Theorem 3 and conclude that the equation $A\boldsymbol{x} = F$, or equivalently $A\boldsymbol{x} + F = \mathbf{0}$, has a solution, i.e., there is an activation set X solving the initial configuration F with F-neighborhood constraints. □

3.3 A Greedy Algorithm

We can find an activation set for σ^F by solving the system of linear equations $A_G^F \boldsymbol{x} = F$, for example by Gaussian elimination. Since the matrix A_G^F is the adjacency matrx of the underlying graph G, we can interpret the steps of the Gaussian elimination as graph operations in G. This leads to the following greedy algorithm for σ^F.

Consider a graph $G = (V, E)$ with initial configuration F and F-neighborhood constraints. The algorithm runs in two phases. In Phase 1, we repeatedly select an arbitrary node v that is switched on. For all nodes $w \in N(v)$, we flip the state of w and the neighborhood constraint F_w. Thus, at any time, the state of a node is the same as its neighborhood constraint.

Then we modify the graph by replacing $N[v]$ by the complement of $N(v)$. To be more precise, we delete all edges connecting two nodes in $N(v)$ and we add edges between nodes in $N(v)$ that were unconnected. We call this step a *neighborhood inverson.* Then we delete v and its incident edges from the graph.

Phase 1 ends when all remaining nodes are switched off. We compute an activiation set W for this graph. For example, we could choose $W = \emptyset$. In Phase 2, we add back all deleted nodes in reverse order of deletion in Phase 1, each time undoing the neighborhood inversions. When we restore node v, we add v to W if $N(v) \cap W$ contains an even number of nodes (here $N(v)$ is the open neighborhood of v in the current graph after restoring v). At the end of Phase 2, we return W as activation set for G. Note that we do not need to track changes of states and neighborhood constraints in Phase 2 (although we do that in the proof below to show correctness of the algorithm).

Theorem 5 ([8,25]). *The greedy algorithm computes an activation set in time* $O(n^3)$.

Proof. Each step in Phase 1 needs time linear in the size of the neighborhood of the selected node. Since this neighborhood can contain at most $O(n)$ nodes, the total running time of Phase 1 is bounded by $O(n^3)$. Phase 2 reverses the operations of Phase 1 and has therefore the same running time.

To see the correctness of the algorithm, we claim that in Phase 2, at any time, the current set W is an activation set for the current graph. This is true initially because all nodes are off and W is empty. Now assume we have reached current graph H with activation set X. By restoring the next node v, we change the states and neighborhood constraints of nodes in $N(v)$ and we invert the neighborhood relation in $N(v)$. If the size of $N(v) \cap X$ is even, we add v to X.

Let p denote the parity of $|N(v) \cap X|$ (i.e., $p = 0$ if the number of nodes is even, and $p = 1$ if the number is odd). We select v if $p = 0$. Thus, choosing or not choosing v contributes $1 - p$ to the cover count (the cumulative effect of all its activated neighbors) of each node in $N(v)$.

For any node w in $N(v)$, inverting the neighborhood of v changes the connectivity of w to any other node in $N(v)$. If $w \notin X$, then we must add p to the cover count of w. Since we also flip the state of all nodes in $N(v)$, the state of w changes by $(1-p)+p+1 = 0$, i.e., w will still be off after activating all nodes in

$X \cup \{w\}$. If $w \in X$, then we must add $p-1$ to the cover count of w. However, we also flip the neighborhood constraint of w, i.e., we change whether w can influence its own state. Thus, the state of w changes by $(1-p)+(p-1)+1+1=0$, i.e., w will still be off after activating all nodes in $X \cup \{w\}$. □

Conlon et al. [34] studied graph classes where it is easy to determine activation sets. For a clique, we can activate an arbitrary node to switch all states. If all nodes of a graph have even degree, then the set of all nodes is an activation set. This is for example the case for Eulerian graphs. Interestingly, we can reverse this statement.

Lemma 6 ([34]). *If X is an activation set for the all-on configuration of a graph G, then X induces an Eulerian subgraph of G.* □

Corollary 7 ([34]). *If X is an activation set for the all-on configuration of a tree, then no two nodes of X are adjacent.* □

3.4 Historical Review

Sutner [67] showed for σ^+ that the all-on configuration can be solved for any graph. Later, Caro proposed a simpler proof based on linear algebra [19] and extended the ideas to mod$k$ domination problems [20]. Another linear algebra proof was suggested by Goldwasser et al. [45,47] and Anderson and Feil [6]. Dodis and Winkler [38] proposed the generalization to σ^F, see Theorem 3.

Cowen et al. [35] gave an elegant graph theoretical proof of Theorem 2 and generalized it to infinite graphs where every node has finite degree. Eriksson et al. [39] gave a similar graph theoretical proof. Our proof in Subsection 3.1 combines ideas from both proofs.

Algebraic proofs for special cases were given by Sutner [67], Lotto [58], and Cowen et al. [35]. The general case was proved by Goldwasser et al. [45,47], Anderson and Feil [6], and Dodis and Winkler [38].

Losada [57] rediscovered several of the techniques to solve Lights Out! on grids and generalized them to grids embedded on surfaces of higher genus. Scherpuis [65] and Martin-Sanchez and Pareja-Flores [60] summarize some of the algebraic solutions for Lights Out! on grids.

Arya et al. [8] and Chen and Gu [25] independently proposed the greedy algorithm from Subsection 3.3. Conlon et al. [34] studied graph classes where it is easy to determine activation sets.

4 Universally Solvable Graphs

Theorem 8 ([3,5,67,68]). *If G has a D-parity cover for some parity vector D, then there exist exactly $2^{PD(G)}$ distinct D-parity covers. In particular, there always exist $2^{PD(G)}$ distinct odd covers.* □

Theorem 9 ([45,47,67,68]). *A graph is universally solvable if and only if its adjacency matrix is invertible.*

Proof. The proof follows basically from the previous theorem. Let A be an $n \times n$-matrix. We have seen in Subsection 3.2 that an initial configuration C is solvable if the linear equation $A\boldsymbol{x} = C$ has a solution over $GF(2)$. Since there are 2^n different initial configurations, the image of A has dimension n (and nullity 0), i.e., A is invertible. □

So we need to study the kernel of the adjacency matrix if we want to show universal solvability. In the rest of this section we will do this for a few simple graph classes. We begin with universally solvable grid graphs.

4.1 LIGHTS OUT! on Grid Graphs

$G_{n,m}$ has nm nodes in m rows and n columns. We denote the node in column i and row j by (i, j). The neighbors of (i, j) are $\{(i-1, j), (i, j+1), (i+1, j), (i, j-1)\}$ (not including those pairs where at least one component is zero).

Goldwasser et al. [45,47] observed that the structure of the kernel of the adjacency matrix of a grid can be analysed using Fibonacci polynomials. For $i \geq 0$, the ith *Fibonacci polynomial* f_i is recursively defined as

$$\begin{aligned} f_0(x) &= 1 \\ f_1(x) &= x \\ f_k(x) &= x f_{k-1}(x) + f_{k-2}(x) \text{ for } k \geq 2 \end{aligned}$$

For example, $f_{i-1}(1) = F_i$, the ith Fibonacci number, for $i \geq 1$. The adjacency matrix $A_{n,m}$ of $G_{n,m}$ can be written as

$$A_{n,m} = \begin{bmatrix} B_m & I_m & 0 & 0 & 0 \cdots & 0 & 0 \\ I_m & B_m & I_m & 0 & 0 \cdots & 0 & 0 \\ 0 & I_m & B_m & I_m & 0 \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 \cdots & I_m & B_m \end{bmatrix}$$

where I_m is the $m \times m$ identity matrix and B_m is an $m \times m$ binary matrix. For the σ game, we use B_m^0 which is defined as $B_m^0[i, j] = 1$ if $|i - j| = 1$, and $B_m^0[i, j] = 0$ otherwise. For the σ^+ game, we use $B_m^+ = B_m^0 + I_m$. We use B_m for both B_m^0 and B_m^+ if the context is clear.

Let $\boldsymbol{x}$ be a vector of dimension mn satisfying $A\boldsymbol{x} = \boldsymbol{0}$. We split $\boldsymbol{x}$ into n subvectors $(\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n})$, each of length m, and define $\boldsymbol{x_0} = \boldsymbol{x_{m+1}} = 0$. Then, clearly $\boldsymbol{x_i} = B_m \boldsymbol{x_{i-1}} + \boldsymbol{x_{i-2}}$, for $i = 2, \ldots, n+1$, or, $\boldsymbol{x_i} = f_{i-1}(B_m) \cdot \boldsymbol{x_1}$, for $i = 1, \ldots, n+1$. It is now straightforward, though tedious and technical, to prove a few properties of the Fibonacci polynomials applied to the matrix B_m.

Lemma 10 ([11,45,47])

(a) For $n > m$, $f_n(x) = f_{n-m}(x) f_m(x) + f_{n-m-1}(x) f_{m-1}(x)$.
(b) $A_{n,m}$, $f_n(B_m)$, and $f_m(B_n)$ have the same nullity.
(c) $f_{m-1}(B_m^0)$ is invertible.

(d) $f_m(x)$ *is the characteristic polynomial of* B_m^0*, i.e.,* $f_m(B_m^0) = 0$.
(e) $f_m(x+1)$ *is the minimal polynomial of* B_m^+*, i.e.,* $f_m(B_m^+ + I_m) = 0$. □

Theorem 11 ([11,45,47])

(a) The nullity of $A_{n,m}^0$ *is* $\gcd(n+1, m+1) - 1$*. In particular, the* σ *game on* $G_{n,m}$ *is universally solvable if and only if* $n+1$ *and* $m+1$ *are relatively prime.*
(b) The nullity of $A_{n,m}^+$ *is the degree of* $\gcd(f_n(x+1), f_m(x))$*. In particular, the* σ^+ *game on* $G_{n,m}$ *is universally solvable if and only if* $f_n(x+1)$ *and* $f_m(x)$ *are relatively prime.*

Proof (Sketch)

(a) By Lemma 10(b), we only need to determine the nullity δ of

$$\begin{aligned} f_n(B_m^0) &= f_{n-m}(B_m^0) f_m(B_m^0) + f_{n-m-1}(B_m^0) f_{m-1}(B_m^0) \\ &= f_{n-m-1}(B_m^0) f_{m-1}(B_m^0)\,, \end{aligned}$$

which is the same as the nullity of $f_{n-m-1}(B_m)$ because $f_{m-1}(B_m)$ is invertible by Lemma 10(c).

If $n+1$ is a multiple of $m+1$, we can repeatedly apply Theorem 10(a) until we see that δ is equal to the nullity of $f_m(B_m^0)$. Since $f_m(B_m^0) = 0$, $\delta = \dim(\ker f_m(B_m))) = m$. If $m+1$ does not divide $n+1$, we can similarly show that $\delta = \gcd(n+1, m+1) - 1$.

(b) If $P(x) = \gcd(f_m(x+1), f_n(x))$, then there exists $u(x)$ and $v(x)$ such that $P(x) = f_m(x+1)u(x) + f_n(x)v(x)$. Thus, $P(B_m^+) = f_n(B_m^+)v(B_m^+)$ by Lemma 10(e), which implies $\ker f_n(B_m^+) \subseteq \ker P(B_m^+)$. On the other hand, $P(x)$ is a factor of $f_n(x)$, which implies $\ker P(B_m^+) \subseteq \ker f_n(B_m^+)$. Thus, $\ker f_n(B_m^+) = \ker P(B_m^+)$. Using the Primary Decomposition Theorem, we can show that the dimension of the kernel is equal to the degree of $P(x)$. □

4.2 Lights Out! on Other Graph Classes

The techniques of the previous subsection are very specific for grid graphs and do not generalize to other graph classes. We start with a few easy observations.

Theorem 12 ([4]). *A graph is universally solvable, or APR, if and only if the empty set is the only even cover.*

Proof. Consider a graph with adjacency matrix A. If S is an even conver, then $A\boldsymbol{s} = \boldsymbol{0}$, where $\boldsymbol{s}$ is the characteristic vector of S. The theorem follows now from Theorem 9. □

Corollary 13 ([4]). *A graph where all nodes have odd degree is not universally solvable.*

Proof. If all nodes have odd degree, then the set of all nodes is an even cover. □

Amin and Slater [5] characterized several APR graph classes. We first consider paths.

Theorem 14 ([5]). *A path P_n is universally solvable if and only if $n \neq 3k+2$, for $k \geq 0$*

Proof. It is easy to see that P_1 is APR and P_2 is non-APR. For $n \geq 3$, assume there is a non-empty even cover S of P_n. If some node does not belong to S, then both its neighbors must belong to S. If a node belongs to S, then it has exactly one neighbor in S. Both endpoints of P_n must belong to S. Thus, $n = 3k+2$ for some $k \geq 1$. □

An *n-leg spider* $T = (v, P_1, P_2, \ldots, P_n)$ is a tree where n paths $P_1, P_2, \ldots, P_n$ are connected to the root v. 1-leg and 2-leg spiders are paths and are therefore covered by the previous theorem.

Theorem 15 ([5]). *Let T be an n-leg spider, for some $n \geq 3$. For $i = 0, 1, 2$, let t_i denote the number of legs of length $3k+i$, for some $k \geq 0$. T is not universally solvable if and only if $t_2 = 0$ and t_1 is odd, or $t_2 > 0$ and t_2 is even.*

Proof. Assume T has a non-empty even cover S. If the root v is not in S, then S induces non-empty even covers for all legs. In that case, all legs contribute to t_2 by Theorem 14 and all their endpoints are in S. Since v must be covered by an even number of nodes, T has an even number of legs, i.e., t_2 is even.

If v is in S, then there must be an odd number of legs whose first node (adjacent to the root) is in S. For these legs, the second node cannot be in S, while S induces an even cover for the rest of the path. Thus, by Theorem 14, these legs contribute to t_1. The other legs all contribute to t_0 because S induces an even cover for the leg minus its first node. Thus, $t_2 = 0$ and t_1 is odd.

Conversely, if $t_2 = 0$ and t_1 is odd, or if $t_2 > 0$ and t_2 is even, then we can construct in the same way a non-empty even cover for T. □

We now consider caterpillars. If a caterpillar has only nodes of odd degree, then every internal segment must have an odd number of feet, while the two end segments must each have an even number of feet. It is difficult to characterize the properties of APR caterpillars, we can only give an inductive construction of non-APR caterpillars. The *composition* of two caterpillars T_1 and T_2 is the caterpillar $T_1 \circ x \circ T_2$, where x is a new node with an arbitrary number of feet which is connected to one end segment of T_1 and T_2.

Theorem 16 ([5]). *The set $\mathcal{T}$ of all non-APR caterpillars can be defined by*

1. *each caterpillar with only odd-degree nodes is in $\mathcal{T}$;*
2. *if T_1 and T_2 are in $\mathcal{T}$, then their composition is also in $\mathcal{T}$.*

Proof. We first show that all caterpillars in $\mathcal{T}$ are non-APR. If all nodes have odd degree, then the caterpillar is non-APR by Corollary 13. If T_1 and T_2 are non-APR caterpillars with non-empty even covers S_1 and S_2, respectively, then the S_i must contain the end segments. Thus, $S_1 \cup S_2$ is a non-empty even cover of the composition of T_1 and T_2.

We now show that $\mathcal{T}$ contains all non-APR caterpillars. Consider a non-APR caterpillar T which has at least one even-degree node. Let S be a non-empty even cover of T, If a segment belongs to S, then so do all its feet. Since $S \neq V$ (this would imply that all nodes have odd degree), there must be a segment x which is not in S. This node cannot be an end segment, so it splits the spine into two parts, i.e., two non-APR caterpillars whose composition is T. □

We now consider APR trees. We know from Section 4 that a graph is universally solvable if and only if there is a bijection between initial configurations and activation sets. Let (T_i, x_i), for $i = 1, 2$, be an APR tree T_i with a designated node $x_i \in T_i$. Let X_i be the activation solving the initial configuration $C_i = \{x_i\}$. If $x_1 \notin X_1$, then we can combine T_1 and T_2 in a *Type 1* operation by adding the edge (x_1, x_2).

If we have an even number of APR trees (T_i, x_i), for $i = 1, \ldots, 2k$, with designated nodes $x_i \in T_i$ and all $x_i \in X_i$, then we can combine these trees into a single tree by creating a new node v and connecting it with all the x_i. It is easy to see that Type 1 and Type 2 operations again create APR trees.

Theorem 17 ([5]). *A tree is universally solvable if and only if it is K_1 or it can be obtained from a set of universally solvable trees by a Type 1 or Type 2 operation.* □

Theorem 18 ([5]). *If a tree contains exactly one vertex of even degree, then the tree is APR.* □

4.3 Historical Reviews

Pelletier [62] showed that `Merlin` is APR because its adjacency matrix is invertible. After computing the inverted adjacency matrix, it is easy to read off an solution for a given parity vector in quadratic time. Consider the submatrix of all columns whose corresponding nodes have odd parity; choose node i if row i has an odd number of non-zero entries in this submatrix.

Sutner [67,68] showed that a directed (or undirected) graph is APR if and only if its corresponding σ-automaton is reversible if and only if the nullspace of A_G has dimension 0. In this case, any solution is unique. The same result for undirected graphs was found by Amin and Slator [4]. They observed that a graph cannot be APR if all nodes have odd degree. On the other hand, a tree is APR if it has exactly one vertex of even degree [5]. They also gave other characterizations of APR trees [5]. Amin et al. [3] studied the parity dimension of various graph classes like cycles, paths, trees, and random graphs.

Amin and Slater [4] showed how to decide whether a series-parallel graph is APR and find a minimum D-cover (if it exists) using a dynamic program with 16 labels on each node. They further showed that K_n is not APR, for $n \geq 2$. For P_n, the nullspace has dimension $d = 0$ if $n \equiv 0, 1 \bmod 3$, and $d = 1$ if $n \equiv 2 \bmod 3$ [4,67,68]. For open neighborhoods, it is $d = 0$ if n is even and $d = 1$ if n is odd [68]. In this case, the nullspace of $G_{n,m}$ has dimension $gcd(n + 1, m + 1) - 1$. $G_{n,n}$ is invertible iff n is even. For closed neighborhoods, Sutner claimed that for any n

there is an $m \geq n$ such that the nullspace of $G_{n,m}$ has dimension n. He gave a table of the dimensions of $G_{n,n}$, for $n = 1, \ldots, 100$. C_n is APR iff $n \not\equiv 0 \bmod 3$ [67,4]. $G_{n,1}$ and $G_{n,3}$ are APR iff $n \not\equiv 2 \bmod 3$, $G_{n,2}$ is APR iff n is even, $G_{n,5}$ is APR iff $n \not\equiv 4 \bmod 5$, and $G_{2n+1,3k+2}$ is not APR for $k \geq 0$ and $n \geq 1$ [4].

$G + H$ is APR iff G and H are APR and G or H has an odd cover of even cardinality [4]. In particular, a *fan* $F_n = K_1 + P_n$ is APR iff $n \equiv 0, 4 \bmod 6$, a *wheel* $W_n = K_1 + C_n$ is APR iff $n \equiv 2, 4 \bmod 6$, and $K_{n,m}$ is APR iff nm is even. Amin et al. [5] studied APR trees: A *spider* (or *star*) is APR iff one ray has length $\equiv 2 \bmod 3$ or there is no such ray but an even number of rays of length $\equiv 1 \bmod 3$; if k is even then a k-ary tree is APR; a complete k-ary tree is APR iff it has exactly one node of even degree; they also constructively characterized all APR caterpillars and all APR trees.

Sutner [68] studied solvability of directed graphs with open and closed neighborhoods in the context of σ-automata. Finding a D-cover for directed graph G is equivalent to solving $A_G \cdot x = D$ over $GF(2)$ (or $(A_g - I) \cdot x = D$ in case of open neighborhoods). The number of different solutions (if there is a D-cover) is 2^d, where d is the dimension of the nullspace of A_G.

Sutner [70] showed that the reachability and predecessor problems are undecidable for 1-dimensional infinite cellular automata. For finite 1-dimensional cellular automata, the predecessor problem can be solved in linear time. For 2-dimensional cellular automata, there exists a rule such that the predecessor problem is NP-complete. In [69], he gave a quadratic algorithm to decide surjectivity of a linear cellular automaton. In [71], he gave a quadratic time algorithm to test whether the global map is injective, m-to-one, or surjective.

Goldwasser et al. [45,47] first used divisibility properties of Fibonacci polynomials to characterize APR grid graphs for σ^+. In particular, they obtained an $O(n \log^2 n)$ time algorithm to decide whether $G_{n,m}$ is APR, where $n \geq m$. Sutner [68] obtained similar results for σ and σ^+ on grid graphs studying the same recursive polynomials which he called *Chebychev polynomials*. Later, Barua and Ramakrishnan [11] simplified this technique, and Sutner [72] extended the results. Sutner [73] also used the Fibonacci polynomials to analyze the cycle structure of σ-automata.

Ware [76] computed all factors of the first one hundred Fibonacci polynomials and give a list of all solvable grids $G_{n,n}$, for $n \leq 1,000$. Goldwasser et al. [46] showed that Fibonacci polynomials can also be used to analyze *even covers*. In 2002, Klostermeyer [54] presented a survey on parity domination in grid graphs.

Dodis and Winkler [38] studied F-neighborhood covers. For fixed F, they called the parity vector $D = F$ a *universal configuration* because it is the only parity vector that guarantees the existence of a D-parity F-neighborhood cover in any graph. Note that the greedy algorithm in Section 3.3 always moves from one universal configuration to another in a smaller graph (and stops when it reaches the all-zero parity vector which has a trivial empty set solution).

5 Optimization Problems

5.1 Minimum Odd Covers

We have seen in Section 3 that all undirected graphs have an activation set for the all-on initial configuration. It is natural to ask for a smallest such activation set. In this section we study this and other optimization problems. Let k-AllOff denote $\{(G = (V, E), k) \mid G$ has an activation set of size at most k for the all-on configuration$\}$.

Theorem 19 ([66]). k-AllOff *is* NP*-complete.*

Proof. k-AllOff is clearly in NP. To prove NP-hardness, we give a reduction from 3SAT. Let F be a formula in 3CNF with n variables and m clauses. We now construct a graph G. For each variable x_i, the variable gadget consists of a triangle, where two of the nodes correspond to the positive and negative literal of the variable, respectively.[1] For each clause, the clause gadget has three a-nodes and seven b-nodes, see Fig. 1. The b-nodes form a clique K_7 (these edges are omitted in the figure). The a-nodes represent the three literals in the clause, and are connected to their corresponding literal nodes in the variable gadgets.

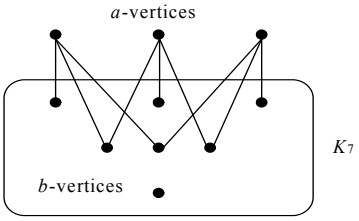


Fig. 1. A clause gadget in the reduction from 3SAT to k-AllOff

We now show that F is satisfiable if and only if G has an activation set of size $n + m$. Let X be an activation set for G. X must contain at least one node of each variable gadget and at least one one of the b-nodes of each clause gadget. Since X has size $n + m$, it must contain exactly one node from each gadget. We can therefore define a truth-assignment α for the variables by $\alpha(x_i) = 1$ if and only if $x_i \in X$. Since no b-node can cover all three a-nodes in its clause gadget, at least one of the a-nodes must be covered by a variable gadget, i.e., each clause is satisfied by α and F is true.

On the other hand, if we have a satisfying assignment α for F, we select the x_i in the varibale gadget for the activation set if $\alpha(x_i) = 1$, and $\bar{x}_i$ if $\alpha(x_i) = 0$. This covers at least one a-node in each of the clause gadgets. The other two

[1] Sutner [66] only used two nodes connected by an edge as variable gadget, but then the correctness argument becomes more involved because it is seemingly possible that a-nodes could be used to cover the variable nodes.

nodes can then be covered by choosing an appropriate b-node. This gives us an activation set of size $m + n$. □

Conlon et al. [34] studied graph classes where it is easy to compute the size of a minimum activation set, which we denote by $\alpha(G)$. Recall that the characteristic vector $\boldsymbol{x}$ of an activation set is a solution to $A\boldsymbol{x} = \mathbf{1}$, and every node of G must have an odd number of activated nodes in its closed neighborhood. Also, if two non-adjacent nodes have the same open neighborhood, then they must either be both activated or both not activated.

Theorem 20 ([34])

(a) $\alpha(K_n) = 1$.

(b) $\alpha(P_n) = \lceil \frac{n}{3} \rceil$.

(c) $\alpha(C_n) = \begin{cases} \frac{n}{3} & \text{if } n \equiv 0 \bmod 3 \\ n & \text{otherwise} \end{cases}$

(d) $\alpha(K_{n,m}) = \begin{cases} \min(m,n) & \text{if both } m \text{ and } n \text{ are odd} \\ m & \text{if only } m \text{ is odd} \\ n & \text{if only } n \text{ is odd} \\ m+n & \text{if } m \text{ and } n \text{ are even} \end{cases}$

(e) $\alpha(C_{n,m}) = \begin{cases} \lceil \frac{n}{3} \rceil + m\lfloor \frac{2n}{3} \rfloor & \text{if } m \text{ is even} \\ nm & \text{if } m \text{ is odd and } n \text{ is even} \\ \frac{1}{2}(n-1)(m+1)+1 & \text{if } m \text{ and } n \text{ are odd} \end{cases}$

(f) $\alpha(G_{n,2}) = \begin{cases} n & \text{if } n \equiv 0 \bmod 4 \\ n+2 & \text{if } n \equiv 2 \bmod 4 \\ \frac{1}{2}(n+1) & \text{if } n \text{ is odd} \end{cases}$

(g) $\alpha(G_{n,3}) = \begin{cases} \frac{5n}{3} & \text{if } n \equiv 0 \bmod 6 \\ n & \text{if } n \equiv 1 \bmod 6 \\ n+2 & \text{if } n \equiv 2,3 \bmod 6 \\ 10\lceil \frac{n}{6} \rceil & \equiv 4 \bmod 6 \\ n+1 & \text{if } n \equiv 5 \bmod 6 \end{cases}$

(h) $\alpha(G_{n,4}) = \begin{cases} n & \text{if } n \equiv 0 \bmod 5 \\ 6\lceil \frac{n}{5} \rceil + 2 & \text{if } n \equiv 1 \bmod 5 \\ 6\lceil \frac{n}{5} \rceil + 4 & \text{if } n \equiv 2 \bmod 5 \\ 2(n+2) & \text{if } n \equiv 3 \bmod 5 \\ 6\lceil \frac{n}{5} \rceil & \text{if } n \equiv 4 \bmod 5 \end{cases}$

(i) If T_h *is a full binary tree of height* $h \geq 0$*, then* $\alpha(T_h) = \frac{1}{3}(4^{\lfloor h/2 \rfloor + 1} - 1)$.

Proof. (Sketch)

(a) We can activate any node.

(b) By Lemma 7, every consecutive pair of activated nodes must be separated by a pair of non-activated nodes.

(c) If no two activated nodes are adjacent, then every consecutive pair of activated nodes must be separated by a pair of non-activated nodes. Otherwise, all nodes must be activated.

(d) Let V_m and V_n be the two partitions of $K_{n,m}$. If m or n are odd, it suffices to activate the smallest odd partition, otherwise we must activate all nodes.

(e) Since all feet of a segment must either be all activated or all not activated, and the segment is activated if the feet are not activated, we try to activate as many segments as possible. If m is even, we find the minimum activation for the spine, which is a P_n, and activate all feet connected to non-activated segments. If m and n are odd, we must activate every second segment starting with an end segment. If m is odd and n is even, we must activate all feet and no segment.

(f) If $m \equiv 2 \bmod 4$, we must alternate 4-cycles of activated and unactivated nodes. Similarly for $m \equiv 0 \bmod 4$. If m is odd, we can use a perfect star cover with the activated nodes arranged in a knights move pattern.

(g) Extensive case analysis.

(h) Extensive case analysis.

(i) We must activate the root and all nodes on even levels of the tree. □

Amin and Slater [4] proposed a linear-time algorithm for series-parallel graphs based on dynamic programming. Gassner and Hatzl [42] generalized this algorithm to graphs of bounded treewidth and distance-hereditary graphs.

5.2 Maximizing Off Nodes

We have seen that for some graphs not all initial configurations are solvable. It is natural to ask how difficiult it is to switch off as many nodes as possible. Goldwasser et al. [44] called this problem MOS (Maximizing Off Switches) and showed that it is NP-complete via a gap-preserving reduction from a variant of MAX-3SAT where each variable appears in exactly five clauses [7].

Theorem 21 ([44]). MOS *is* NP*-complete. Further, there exists a constant* $\varepsilon > 0$ *such that no polynomial time algorithm for MOS can achieve an approximation ratio better than* $1 + \varepsilon$*, unless* P=NP. □

Note that it is easy to achieve an approximation ratio of $2 - \lfloor \frac{2}{n} \rfloor$. We know from Section 3 that there always exists an odd cover that changes the state of every node in the graph. If more than half of the nodes are on in the initial configuration, we can use this odd cover to turn more than half of the nodes off.

We now turn to the fixed parameter variant of MOS. Let G be a graph and $c \geq 0$ a constant. We denote by c-MOS the problem to decide whether we can

reach a configuration with at most c on nodes from any initial configuration [44]. We attack this problem using techniques from coding theory. We have seen in Theorem 9 that $\ker A_G$ is orthogonal to the range of A. Let $\boldsymbol{y_1}, \boldsymbol{y_2}, \cdots, \boldsymbol{y_k}$ be a basis of $\ker A_G$. Let H be the $k \times n$-matrix with y_i as row i, for $i = 1, \ldots, k$. Every solvable initial configuration $\boldsymbol{x}C$ satisfies $H = \mathbf{0}$.

This means all the solvable configurations form a *linear code* $\mathcal{C}$ with binary codewords of length n. The matrix H is the *parity check matrix* of the code $\mathcal{C}$, i.e., $H\boldsymbol{c} = 0$ for all $\boldsymbol{c} \in \mathcal{C}$. For any word $\boldsymbol{w}$ of length n, the k-dimensional vector $\boldsymbol{s} = H\boldsymbol{w}$ is called the *syndrome* of $\boldsymbol{w}$.

Lemma 22 ([44]). *The coset $\boldsymbol{w}+\mathcal{C}$ is equal to the set of all configurations that can be reached from initial configuration $\boldsymbol{w}$ via activations.*

Proof. For every $\boldsymbol{c} \in \mathcal{C}$ there exists a vector $\boldsymbol{x}$ representing a $\boldsymbol{c}$-parity activation set, i.e., $A\boldsymbol{x} = \boldsymbol{c}$. Thus, for every word $\boldsymbol{u}$ in the coset $\boldsymbol{w} + \mathcal{C}$ there exists a codeword $\boldsymbol{c} \in \mathcal{C}$ and a vector $\boldsymbol{x}$ such that $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{c} = \boldsymbol{w} + A\boldsymbol{x}$. Thus, u can be obtained from the initial configuration $\boldsymbol{w}$ via the activation set $\boldsymbol{x}$. □

Observe that all $\boldsymbol{u}$ in the coset $\boldsymbol{w}+\mathcal{C}$ have the same syndrome since there exists a codeword $\boldsymbol{c}$ such that $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{c}$ and thus $H\boldsymbol{u} = H(\boldsymbol{w} + \boldsymbol{c}) = H\boldsymbol{w} = \boldsymbol{s}$. The *weight* of a coset is the minimum weight of any vector in the coset. The *covering radius* of the code is the maximum weight of any coset.

Lemma 23 ([44]). *The weight of a coset $\boldsymbol{w}+\mathcal{C}$ is equal to the smallest number of columns of H whose sum is the syndrome $H\boldsymbol{w}$ of the coset.*

Proof. The syndrome $\boldsymbol{s} = H\boldsymbol{w}$ is the sum of some columns of H, encoded by the positions of the ones in $\boldsymbol{w}$. Since every word u in the coset $w + \mathcal{C}$ has the same syndrome $\boldsymbol{s}$, the weight of the coset is the smallest number of columns in H whose sum equals $\boldsymbol{s}$. □

In c-MOS, we want from any initial configuration reach a configuration with at most c on nodes. This means, the vectors in the range of A can contain at most c ones. The weight of a coset $\boldsymbol{w}+\mathcal{C}$ is the minimum number of ones of any vector in the coset. If the weight of any coset is not greater than c, then we can turn off at least $n - c$ nodes for every initial configuration. So all we need to do is to compute the covering radius of $\mathcal{C}$, i.e., check whether every syndrome $\boldsymbol{s}$ can be formed by a sum of at most c columns of H.

5.3 Bounds for Trees

Wang and Wu [75] recently proposed a tight bound for the minimum number of on nodes we can guarantee for trees. For any tree and any initial configuration, we can always find an activation set that leaves at most $\lceil \frac{\ell}{2} \rceil$ nodes on, where ℓ is the number of leaves.

We first introduce some notation. We denote the set $\{1, 2, \ldots, n\}$ by $[n]$. For an $m \times n$ matrix A, if I and J are subsets of $[m]$ and $[n]$, respectively, then we denote by $M(I, J)$ the submatrix indexed by the rows I and columns J. The *covering radius* of a matrix A is defines as $\rho(M) = \max_{\boldsymbol{x}} \min_{\boldsymbol{y} \in R(M)} wt(\boldsymbol{x} - \boldsymbol{y})$.

Lemma 24 ([75]). *Let A be an $m \times n$ matrix. Let I and J be disjoint subsets of $[n]$ such that $A([m], I \cup \{j\})$ is of full column rank for all $j \in J$. Then, for any $\boldsymbol{c}$, there exists $\boldsymbol{y} \in R(A)$ with $wt(\boldsymbol{c}, \boldsymbol{y}) \leq n - |I| - \lceil \frac{|J|}{2} \rceil$. In particular, $\rho(A) \leq n - |I| - \lceil \frac{|J|}{2} \rceil$.* □

Theorem 25 ([75]). *Let T be a tree with ℓ leaves. For any initial configuration, there exists an activation set that leaves at most $\lfloor \frac{\ell}{2} \rfloor$ nodes on, and these nodes are leaves.*

Proof. Let A_T be the adjacency matrix of T. Let I be the column index set corresponding to internal nodes of T and J the column index set corresponding to leaf nodes. It is not difficult to see that I and J satisfy the conditions in Lemma 24, so we conclude

$$\rho(A) \leq n - |I| - \lceil \frac{|J|}{2} \rceil = n - r - \lceil \frac{l}{2} \rceil = \lfloor \frac{l}{2} \rfloor$$

□

5.4 Historical Review

Minimum Covers. Sutner [66] gave the first NP-hardness proof for finding minimum odd covers. Later several special graph classes were studied. Conlon et al. [34] derived closed formulas for simple graph classes like paths, cycles, caterpillars, and narrow grids. Galvin [41], Dawes [36], and Chen et al. [26] proposed to first enumerate all feasible solutions for a tree and then use some linear algebra techniques to find the minimum solution. Amin and Slate [5] proposed linear time dynamic programs to compute minimum odd covers for trees and series-parallel graphs.

Heck [50] showed how to compute a minimum solution for Nine-Tails, which is Lights Out! on $G_{3,3}$, using dynamic programming in time $\Theta(2^n)$ (it is actually BFS in the game graph going backwards from the end configuration). Delahan et al. [37] proposed a simpler algorithm. For each node v, he precomputed the (unique) D_v-cover, where $D_v(w) = 1$ exactly if $w = v$. Then we can compute a minimum D-cover (they did not mention that any D-cover must be unique, though) in time $O(n^2)$ by bitwise addition of those D_v where $D(v) = 1$.

Goldwasser and Klostermeyer [44] introduced MOS and proved it is NP-hard to find an optimal solution and a $(1 + \varepsilon)$-approximation. They also used coding theory to construct a polynomial time algorithm to solve the fixed parameter variant c-MOS and gave tight bounds for grid graphs. Wang and Wu [75] gave tight bounds for trees.

Codes. The following problems are NP-complete [64, p. 739 ff]:

1. Weight of Error (or Maximum Likelihood Decoding): Does $H\boldsymbol{y} = \boldsymbol{s}$ have a solution $\boldsymbol{y}$ with weight at most w? (Reduction from 3D-Matching [12], the matrix is non-symmetric.)

2. `Minimal Weight`: Does $H\boldsymbol{y} = \mathbf{0}$ have a non-zero solution $\boldsymbol{y}$ of weight at most w? (Reduction from `Weight of Error` [74])
   This is equivalent to `Even Vertex Set`: For a given graph, are there at most w nodes (but more than zero nodes) that form an even cover?
3. `Maximal Weight` (alphabet size two or three): Does $H\boldsymbol{y} = \mathbf{0}$ have a non-zero solution $\boldsymbol{y}$ of weight at least w? (Reduction from `Max-Cut`)
4. `Weight Distribution`: Does $H\boldsymbol{y} = \mathbf{0}$ have a non-zero solution $\boldsymbol{y}$ of weight w? (Reduction from `3D-Matching` [12])

Acknowledgements. We thank the following colleagues for their helpful discussions at HKUST during an early stage of this paper: Sunil Arya, Siu-Wing Cheng, Mordecai Golin, Torleiv Kløve, Stefan Langermann, Yiu Cho Leung, Hyeon-Suk Na, Sheung Hung Poon, Gerhard Trippen, Ho Man Tsui, Antoine Vigneron, and Joseph Zhen Zhou.

References

1. Allan, R.B., Laskar, R.: On domination and some related topics in graph theory. In: Proceedings of the 9th Southeastern Conference on Combinatorics, Graph Theory and Computing, pp. 43–56. Utilitas Mathematica, Winnipeg (1978)
2. Alon, N., Caro, Y.: On three zero-sum Ramsey-type problems. Journal of Graph Theory 17, 177–192 (1993)
3. Amin, A., Clark, L., Slater, P.: Parity dimension for graphs. Discrete Mathematics 187, 1–17 (1998)
4. Amin, A.T., Slater, P.J.: Neighborhood domination with parity restrictions in graphs. Congressus Numerantium 91, 19–30 (1992)
5. Amin, A.T., Slater, P.J.: All parity realizable trees. Journal of Combinatorial Mathematics and Combinatorial Computing 20, 53–63 (1996)
6. Anderson, M., Feil, T.: Turning lights out with linear algebra. Mathematical Magazine 71(4), 300–303 (1998)
7. Arora, S., Lund, C.: Hardness of approximations. In: Hochbaum, D.S. (ed.) Approximation Algorithms for NP-Hard Problems, pp. 399–446. PWS Publishing Company, Boston (1997)
8. Arya, S., Cheng, S.-W., Fleischer, R., Golin, M., Kløve, T., Langermann, S., Leung, Y.C., Na, H.-S., Poon, S.H., Trippen, G., Tsui, H.M., Vigneron, A., Zhou, Z.: Fiver (2002) (manuscript)
9. Bange, D.W., Barkauskas, A.E., Host, L.H., Slater, P.J.: Efficient near-dominating of grid graphs. Congressus Numerantium 58, 83–92 (1987)
10. Bange, D.W., Barkauskas, A.E., Slater, P.J.: Efficient dominating sets in graphs. In: Ringeisen, R.D., Roberts, F.S. (eds.) Applications of Discrete Mathematics, pp. 189–199. Society of Industrial and Applied Mathematics, Philadelphia (1988)
11. Barua, R., Ramakrishnan, S.: σ-game, σ^+-game and two-dimensional addititive cellular automata. Theoretical Computer Science 154(2), 349–366 (1996)
12. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the inherent intractability of certain coding problems. IEEE Transactions on Information Theory IT-24, 384–386 (1978)

13. Berman, F., Leighton, F.T., Shor, P., Snyder, L.: Generalized planar matching. Technical Report MIT/LCS/TM-273. MIT (April 1985)
14. Beyer, T., Proskurowski, A., Hedetniemi, S., Mitchell, S.: Independent domination in trees. In: Proceedings of the 8th Southeastern Conference on Combinatorics, Graph Theory and Computing, pp. 321–328 (1977)
15. Biggs, N.: Perfect codes and distance transitive graphs. In: McDonough, F.P., Mavron, V.C. (eds.) Proceedings of the 3rd British Combinatorial Conference (Combinatorics). London Mathematical Society Lecture Notes Series, vol. 13, pp. 1–8 (1973)
16. Biggs, N.: Perfect codes in graphs. Journal of Combinatorial Theory, Series B 15, 289–296 (1973)
17. Bollobás, B., Cockayne, E.J.: Graph-theoretic parameters concerning domination, independence, and irredundance. Journal of Graph Theory 3, 241–249 (1979)
18. Brandstädt, A., Leitert, A., Rautenbach, D.: Efficient dominating and edge dominating sets for graphs and hypergraphs. arXiv:1207.0953v2[cs.DM] (2012)
19. Caro, Y.: Simple proofs to three parity theorems. Ars Combinatoria 42, 175–180 (1996)
20. Caro, Y., Jacobson, M.S.: On non-$z(\mathrm{mod}k)$ dominating sets. Discussiones Mathematicae Graph Theory 23, 89–199 (2003)
21. Caro, Y., Klostermeyer, W.: The odd domination number of a graph. Journal of Combinatorial Mathematics and Combinatorial Computing 44, 65–84 (2003)
22. Caro, Y., Klostermeyer, W.F., Yuster, R.: Connected odd dominating sets in graphs (2011), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.2320`
23. Chang, T.Y., Clark, W.E.: The domination numbers of the $5 \times n$ and the $6 \times n$ grid graphs. Journal of Graph Theory 17(1), 81–107 (1993)
24. Chang, T.Y., Clark, W.E., Hare, E.O.: Domination numbers of grid graphs, i. Ars Combinatoria 38, 97–111 (1994)
25. Chen, W.Y.C., Gu, N.S.S.: Loop deletion for the lamp lighting problem (2011) (manuscript), `http://www.billchen.org/preprint/lamp/lamp.htm`
26. Chen, W.Y.C., Li, X., Wang, C., Zhang, X.: The minimum all-ones problem for trees. SIAM Journal on Computing 33(2), 379–392 (2004)
27. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit disk graphs. Discrete Mathematics 86(1-3), 165–177 (1990)
28. Clark, W.E., Dunning, L.A.: Tight upper bounds for the domination numbers of graphs with given order and minimum degree. The Electronic Journal of Combinatorics 4(1, R26), 1–25 (1997)
29. Clark, W.E., Ismail, M.E.H., Suen, S.: Application of upper and lower bounds for the domination number to Vizing's conjecture. Ars Combinatoria (2003)
30. Clark, W.E., Suen, S., Dunning, L.A.: Tight upper bounds for the domination numbers of graphs with given order and minimum degree, II. The Electronic Journal of Combinatorics 7(1, R58), 1–19 (2000)
31. Cockayne, E.J., Hare, E.O., Hedetniemi, S.T., Wimer, T.V.: Bounds for the domination number of grid graphs. Congressus Numerantium 47, 217–228 (1985)
32. Cockayne, E.J., Hedetniemi, S.T.: Independence graphs. In: Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory and Computing, pp. 471–491. Utilitas Mathematica, Winnipeg (1974)
33. Cockayne, E.J., Hedetniemi, S.T.: Towards a theory of domination in graphs. Networks 7, 247–261 (1977)
34. Conlon, M.M., Falidas, M., Forde, M.J., Kennedy, J.W., McIlwaine, S., Stern, J.: Inversion numbers of graphs. Graph Theory Notes of New York 37, 42–48 (1999)

35. Cowen, R., Hechler, S.H., Kennedy, J.W., Ryba, A.: Inversion and neighborhood inversion in graphs. Graph Theory Notes of New York 37, 37–41 (1999)
36. Dawes, R.W.: Minimum odd neighborhood covers for trees. In: Sherwani, N.A., Kapenga, J.A., de Doncker, E. (eds.) Great Lakes CS Conference 1989. LNCS, vol. 507, pp. 161–169. Springer, Heidelberg (1991)
37. Delahan, F., Klostermeyer, W.F., Trapp, G.: Another way to solve Nine-Tails. ACM SIGCSE Bulletin 27(4), 27–28 (1995)
38. Dodis, Y., Winkler, P.: Universal configurations in light-flipping games. In: Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), pp. 926–927 (2001)
39. Eriksson, H., Eriksson, K., Sjostrand, J.: Note on the lamp lighting problem. Advances in Applied Mathematics 27, 357–366 (2004)
40. Fraenkel, A.S.: Two-player games on cellular automata. In: Proceedings of the 2000 MSRI Workshop on Combinatorial Games — More Games of No Chance. Mathematical Sciences Research Institute Publications, vol. 42, pp. 279–306. Cambridge University Press, Cambridge (2002)
41. Galvin, F.: Solution. The Mathematical Intelligencer 11(2), 32 (1989)
42. Gassner, E., Hatzl, J.: A parity domination problem in graphs with bounded treewidth and distance-hereditary graphs. Journal of Computing 82(2), 171–187 (2008)
43. Goddard, W., Henning, M.A.: Domination in planar graphs with small diameter. Journal of Graph Theory 40(1), 1–25 (2002)
44. Goldwasser, J., Klostermeyer, W.: Maximization versions of "Lights Out" games in grids and graphs. Congressus Numerantium 126, 99–111 (1997)
45. Goldwasser, J., Klostermeyer, W., Trapp, G.: Characterizing switch-setting problems. Linear and Multilinear Algebra 43(1-3), 121–136 (1997)
46. Goldwasser, J., Klostermeyer, W., Ware, H.: Fibonacci polynomials and parity domination in grid graphs. Graphs and Combinatorics 18(2), 271–283 (2002)
47. Goldwasser, J., Klostermeyer, W.F., Trapp, G.E., Zhang, C.Q.: Setting switches in a grid. Technical Report TR-95-20, Department of Statistics and Computer Science, West Viginia University (1995), `http://ww.cs.wvu.edu/usr05/wfk/.public-html/switch.ps`
48. Goncalves, D., Pinlou, A., Rao, M., Rhomassé, S.: The domination number of grids. SIAM Journal on Discrete Mathematics 25(3), 1443–1453 (2011)
49. Hare, E.O., Hedetniemi, S.T., Hare, W.R.: Algorithms for computing the domination number of $k \times n$ complete grid graphs. Congressus Numerantium 55, 81–92 (1986)
50. Heck, P.: Dynamic programming for pennies a day. ACM SIGCSE Bulletin 26(1), 213–217 (1994)
51. Jacobson, M.S., Kinch, L.F.: On the domination number of products of graphs: I. Ars Combinatoria 18, 33–44 (1983)
52. Jacobson, M.S., Peters, K.: Complexity questions for n-domination and related parameters. Congressus Numerantium 68, 7–22 (1989)
53. Kikuno, T., Yoshida, N., Kakuda, Y.: A linear algorithm for the domination number of a series-parallel graph. Discrete Applied Mathematics 5, 299–311 (1983)
54. Klostermeyer, W.: Lights Out!: A survey of parity domination in grid graphs (2002), `http://citeseer.nj.nec.com/498805.html`
55. Livingston, M., Stout, Q.F.: Perfect dominating sets. Congressus Numerantium 79, 187–203 (1990)
56. Livingston, M.L., Stout, Q.F.: Constant time computation of minimum dominating sets. Congressus Numerantium 105, 116–128 (1994)

57. Losada, R.: All Lights and Lights Out (in Spanish). In: SUMA, vol. 40 (2002), English version http://centros5.pntic.mec.es/~antoni48/Lights.doc
58. Lotto, B.: It all adds up to elegance and power: A computer puzzle as a paradigm for doing mathematics. Vasser Quarterly 93(1), 19–23 (1996)
59. MacGillivray, G., Seyffarth, K.: Domination numbers of planar graphs. Journal of Graph Theory 22(3), 213–229 (1996)
60. Martin-Sánchez, Ó., Pareja-Flores, C.: Two reflected analyzes of Lights Out. Mathematical Magazine 74(4), 295–304 (2001)
61. Masters, J.D., Stout, Q.F., van Wieren, D.M.: Unique domination in cross-product graphs. Congressus Numerantium 118, 49–71 (1996)
62. Pelletier, D.: Merlin's magic square. American Mathematical Monthly 94, 143–150 (1987)
63. Pfaff, J., Laskar, R., Hedetniemi, S.T.: Linear algorithms for independent domination and total domination in series-parallel graphs. Congressus Numerantium 45, 71–82 (1984)
64. Pless, V., Huffman, W. (eds.): Handbook of Coding Theory, vol. 1. Elsevier Science Publishing Co., New York (1998)
65. Scherphuis, J.: Jaap's puzzle page (2013), http://www.jaapsch.net/puzzles/lights.htm
66. Sutner, K.: Additive automata on graphs. Complex Systems 2(6), 649–661 (1988)
67. Sutner, K.: Linear cellular automata and the Garden-of-Eden. The Mathematical Intelligencer 11(2), 49–53 (1989)
68. Sutner, K.: The σ-game and cellular automata. American Mathematical Monthly 97(1), 24–34 (1990)
69. Sutner, K.: De Bruijn graphs and linear cellular automata. Complex Systems 5(1), 19–30 (1991)
70. Sutner, K.: On the computational complexity of finite cellular automata. Journal of Computer and System Sciences 50(1), 87–97 (1995)
71. Sutner, K.: Linear cellular automata and de Bruijn automata. In: Delorme, M., Mazoyer, J. (eds.) Cellular Automata: A Parallel Model. Mathematics and its Applications, vol. 460. Kluwer, Boston (1999)
72. Sutner, K.: Sigma-automata and Chebyshev polynomials. Theoretical Computer Science 230(1-2), 49–73 (2000)
73. Sutner, K.: Decomposition of additive cellular automata. Complex Systems 13(3), 245–270 (2001)
74. Vardy, A.: The intractability of computing the minimum distance of a code. IEEE Transactions on Information Theory 43(6), 1757–1766 (1997)
75. Wang, X., Wu, Y.: σ-game on trees: covering radius and tree order (2007)
76. Ware, H.: Divisibility properties of Fibonacci polynomials over GF(2). Master's thesis, Department of Statistics and Computer Science, West Virginia University (1997), http://www.csee.wvu.edu/~java/etd/ware.pdf

Random Access to High-Order Entropy Compressed Text

Roberto Grossi

Dipartimento di Informatica, Università di Pisa, Italy
grossi@di.unipi.it

Abstract. This paper is a survey on the problem of storing a string in compressed format, so that (*a*) the resulting space is close to the high-order empirical entropy of the string, which is a lower bound on the compression achievable with text compressors based on contexts, and (*b*) constant-time access is still provided to the string as if it was uncompressed. This is obviously better than decompressing (a large portion of) the whole string each time a random access to one of its substrings is needed. A storage scheme that satisfies these requirements can thus replace the trivial explicit representation of a text in any data structure that requires random access to it, alleviating the algorithmic designer from the task of compressing it.

1 Introduction

Massive textual data and text with markup are the formats of choice for documents, data interchange, document databases, data backup, log analysis, and configuration files. These forms of unstructured, semi-structured, and replicated data are gaining increasing popularity. Most of their processing is in the main memory in computing clusters where space is an important issue. At the same time, they are often highly compressible when considered as strings.

While most sequential processing methods decompress on the fly and scan the content of these compressed strings (e.g. using tools such as `zcat` or `bzcat`), there are many situations in which fast *random* access to some selected contiguous segments (substrings) of these compressed strings is needed. The decompression of the whole strings is too expensive because the accessed substrings may be relatively few and small, and potentially scattered through the memory.

Problem Statement. The above scenario motivates the introduction of a *compressed string storage scheme* (CSSS), which stores a string $S[1,n]$ from the alphabet $[\sigma] = \{1,\ldots,\sigma\}$ in compressed form, while allowing random access to the string via the operation:

$\mathsf{Access}(i,m)$**:** return the substring $S[i,i+m-1]$ for $m \geq 1$ and $1 \leq i \leq n-m+1$.

The *dynamic* version of a CSSS supports also update operations on S, namely, insertions, deletions or substitutions of symbols in S.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 199–215, 2013.

Model and Complexity. We adopt the standard RAM model with a word size of $\Theta(\lg n)$ bits. We also assume that each symbol of the alphabet $[\sigma]$ is encoded by $\lceil \lg \sigma \rceil$ bits[1] This means that the uncompressed string S requires $n\lceil \lg \sigma \rceil$ bits to be stored in *raw* form, and Access to any substring of m symbols can be optimally (and trivially) performed in $\Theta(1 + m \lg \sigma / \lg n)$ time as each word stores $\Theta(\lg n / \lg \sigma)$ symbols.[2]

When S is highly compressible, which is the case in almost all large-scale texts, the compressed representation of S can reach its information theoretic minimum, called entropy. For each $c \in [\sigma]$, let $p_c = n_c/n$ be its empirical probability of occurring in S, where n_c is the number of occurrences of symbol c in S. The *zeroth order empirical entropy* of S is defined as $H_0(S) = -\sum_{c=1}^{\sigma} p_c \lg p_c$, where $H_0(S) \leq \lg \sigma$. Popular compressors such as those based in Huffman coding and Arithmetic coding are able to store S using $nH_0(S)$ bits plus some lower order term. Note that this compressed representation is effective when $H_0(S) < \lg \sigma$. But this is not theoretically the best for a compressible text.

We can exploit the fact that a certain portion of S, say ω, is often followed by a symbol c in S. In other words, the number of occurrence of ω and ωc are very close. The *conditional* probability of finding c after reading ω in S is therefore very high, potentially much larger than p_c. This translates into a better entropy notion. For any string ω of length k, let ω_S be the string of single symbols following the occurrences of ω in S, taken from left to right, and $|\omega_S|$ be the length of ω_S. The *kth order empirical entropy* of S is defined as $H_k(S) = \frac{1}{n} \sum_{\omega \in [\sigma]^k} |\omega_S| \, H_0(\omega_S)$. Not surprisingly, for any $k \geq 0$ we have $H_k(S) \geq H_{k+1}(S)$. The value of $nH_k(S)$ bits is a lower bound to the output size of any compressor that encodes each symbol of S only considering the symbol itself and the k immediately preceding symbols [24].

Requirements on Time and Space Bounds. Based on the above discussion, we require that a CSSS for string S fulfills the following conditions:

- It supports Access optimally in $\Theta(1 + m \lg \sigma / \lg n)$ time for decoding any substring of S of length m.
- It takes $n\,(H_k(S) + \rho(k, \sigma, n))$ bits, where $\rho(k, \sigma, n)$ is the *redundancy* per symbol, or any additional space required to support random access.

The rationale for the above two conditions is that a CSSS for S can replace S itself for any RAM algorithm A having S as input. This replacement does not penalize the asymptotic time complexity of A, and its space complexity is not worsened when $H_k(S) + \rho(k, \sigma, n) \leq \lceil \lg \sigma \rceil$. For highly compressible S, it is actually $H_k(S) + \rho(k, \sigma, n) = o(\lg \sigma)$, thus showing the importance of using a CSSS for massive texts.

The redundancy $\rho(k, \sigma, n)$ is a quantity of significant fundamental interest, particularly for lower bounds (see [28] and references therein), and it is critical

[1] The logarithms are to the base 2 unless otherwise specified, and $\sigma \leq n$ is customarily taken to be a (usually slowly-growing) function of n.

[2] Several authors prefer to use $\lg_\sigma n$ in place of $\lg n / \lg \sigma$.

in practice. We anticipate that the best redundancy is currently $\rho(k,\sigma,n) = O\left(\frac{\lg\sigma}{\lg n}(k\lg\sigma + \lg\lg n)\right)$, which holds *simultaneously* for all $0 \leq k \leq \lg n/\lg\sigma$. As k increases, the H_k term decreases, but $\rho(k,\sigma,n)$ increases. However, as long as $k = o(\lg n/\lg\sigma)$, we have that the term $n\rho(k,\sigma,n) = o(n\lg\sigma)$ is asymptotically smaller than the space required by S in raw form. Interestingly, no non-trivial lower bounds on the redundancy $\rho(k,\sigma,n)$ are known. The rest of the paper describes the state of the art on this topic.

Impact on Succinct Data Structures. CSSS is a natural fit for *systematic* data structures for texts, also known as *succinct indexes* for texts, where the indexing data structure is separated from the input string S. This concept was born for proving lower bounds [7,13,14], and rapidly extended to to analyze the upper bounds [33,1] on the space required to encode some data structures. Non-systematic data structures instead encode both S and the index data structure together, with no clear separation between the two objects. Some of the advantages of systematic data structures are pointed out in [1] and rephrased here in terms of CSSS.

(1) A systematic data structure does not make assumptions on S, which can therefore be replaced by a CSSS for S thus providing high-order entropy bounds, namely, $n\,(H_k(S) + \rho(k,\sigma,n))$ bits plus the space required by the succinct index. Note that a non-systematic data structure requires instead S to be stored in a specific format.

(2) The same CSSS for S can be shared among several systematic data structures. Here, we can build several succinct indexes on the same CSSS for S without introducing replication. Note that multiple non-systematic data structures must instead replicate S (or its equivalent format) internally.

These features can be made effective by showing how to obtain high-order entropy bounds with static systematic data structures on texts.

Lemma 1 (Barbay et al. [1], Sadakane and Grossi [33]). *Given a* CSSS *for a string S of length n over the alphabet $[\sigma]$, let $n\,(H_k(S) + \rho(k,\sigma,n))$ be its number of required bits. Let $I_1, I_2, \ldots, I_d$ be static systematic data structures defined on the same S, where each I_j uses $n\,\tau_j(\sigma,n)$ bits, without accounting for the storage of S. It is possible to build a static succinct data structure G supporting all the functionalities of $I_1, I_2, \ldots, I_d$ in their same asymptotic time costs, namely, if a functionality takes $t(n)$ time in some I_j, it now still takes $O(t(n))$ time in G. The overall number of bits for G is*

$$n\left(H_k(S) + \rho(k,\sigma,n) + \sum_{j=1}^{d}\tau_j(\sigma,n)\right).$$

The global succinct data structure G in Lemma 1 strips the representation of S from each I_j, and uses a shared CSSS for S. When the algorithms for the functionalities of $I_1, I_2, \ldots, I_d$ need access to S, simply G calls Access on the CSSS for S. The functionalities have only a constant slow-down factor in their

time complexities, while the space occupancy greatly reduces from representing d times the same S in some (unknown) format versus a single and optimal entropy-encoded representation of S by the CSSS. In many applications it is often the case that $\rho(k, \sigma, n) + \sum_{j=1}^{d} \tau_j(\sigma, n) = o(\lg \sigma)$, making the simple idea behind Lemma 1 very useful.

Paper Organization. Section 2 describes the basic scheme that is shared by the static CSSS presented in Section 3 and the dynamic CSSS presented in Section 4. Finally, some further discussion and conclusions are given in Section 5.

2 Basic Scheme

As widely used in practice, the basic approach employs a compressor C of choice. It partitions S into disjoint substrings called *blocks* $B_1, B_2, \ldots, B_r$, and stores S as the sequence $Z = C(B_1) \cdot C(B_2) \cdots C(B_r)$ obtained by concatenating the output of C on the blocks. To perform $\mathsf{Access}(i, m)$, it has to find the index j of the block B_j of S that contains position i, and decode $C(B_j), C(B_{j+1}), \ldots$, until it gets the wanted m symbols. For the sake of discussion, suppose that all blocks in S have the same size w, so that finding j is simple arithmetics, namely, $j - 1 = \lfloor (i-1)/w \rfloor$.

When considering the above approach in terms of the requirements on time and space of a CSSS described in Section 1, we run into a trade-off that can work in some practical cases but it surely causes some theoretical drawbacks. One one hand, if w is much larger than m, then Access has no guarantee to take $\Theta(1 + m \lg \sigma / \lg n)$ time. On the other hand, if w is too small, then Z has not guarantee to use $n\,(H_k(S) + \rho(k, \sigma, n))$ bits of space; for instance, some repetitions inside S of the form xx for some substring x, could be split among two or more blocks $B_j, B_{j+1}, \ldots, B_{j+\ell}$ such that the size of $C(B_j) \cdot C(B_{j+1}) \cdots C(B_{j+\ell})$ is significantly larger than the output size of compressing their concatenation $C(B_j \cdot B_{j+1} \cdots B_{j+\ell})$.

Nevertheless this basic scheme is still good if we replace the black-box compressor C with a pool of suitable succinct data structures. Summing up, we need the following ingredients to implement the basic scheme in a better way, where points 1–2 indicate how to partition S into blocks, and points 3–5 indicate how to encode the sequence of blocks of S (see Fig. 1).

1. *Partition into blocks*: Strategy to partition S into blocks $B_1, B_2, \ldots, B_r$, which can have fixed size w or variable sizes.
2. *Identify the target block*: Succinct data structure(s) or rule to find in $O(1)$ time the index j of the block B_j containing position i, as required by the implementation of $\mathsf{Access}(i, m)$.
3. *Encode a block*: Strategy to assign an encoding $E(B_j)$ to each block B_j, for $1 \leq j \leq r$.
4. *Retrieve a block from its encoding*: Succinct data structure(s) to store, and retrieve in $O(1)$ time, each block B_j from its encoding $E(B_j)$, for $1 \leq j \leq r$.

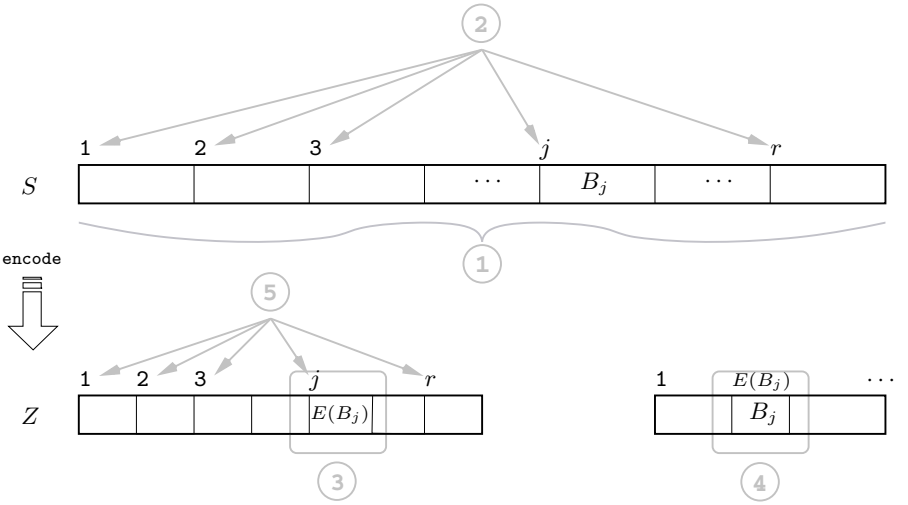


Fig. 1. An illustration of points 1–5 in the basic scheme

5. *Find the encoding of the target block*: Succinct data structure(s) to store $Z = E(B_1) \cdot E(B_2) \cdots E(B_r)$, the concatenation the encodings of the blocks, and access each $E(B_j)$ in $O(1)$ time.

Using the scheme of points 1–5, we can implement $\mathsf{Access}(i, m)$ as follows. We find the index j by points 1–2 and retrieve $E(B_j)$ by point 5. We return B_j by points 3–4. We repeat this for B_{j+1}, B_{j+2}, ..., until we decode the wanted m symbols. As we will see, this implementation requires $O(1)$ time.

In the following we describe, in chronological order, some approaches that follow the above scheme using high-order entropy bounds for the space complexity, employing some basic succinct representation of bitvectors and binary trees that are surveyed in other chapters of this book ([30].). For example, point 3 is easy arithmetics when the blocks have the same size.

When the blocks are of variable sizes, a *fully indexable dictionary* (*FID*) [29] can be employed to mark with a `1` the position in S at the beginning of each block and with `0`s the rest of the positions in S. The resulting bitvector X contains r `1`s and $n-r$ `0`s, and can be stored in the FID using $\lceil \lg \binom{n}{r} \rceil + O(n \lg\lg n / \lg n) = O(r \lg(n/r) + n \lg\lg n / \lg n)$ bits. We remark that this space bound is often negligible when compared to that taken by the other succinct data structures employed in the CSSS.

We recall that a FID supports two basic constant-time operations: $\mathsf{Rank}_b(i)$ returns the number of occurrences of bit $b \in \{0, 1\}$ in the first i positions in X; $\mathsf{Select}_b(i)$ returns the position of the ith occurrence of bit b in X. We can thus locate the block index j, corresponding to the block B_j in S that contains position i as mentioned in point 2, by computing $j = \mathsf{Rank}_1(i)$.

3 Static Schemes

We describe the family of CSSS where only Access operation is supported on input string S, which does not change during its lifetime. The first scheme has variable-size blocks in the partition of S (point 1 in Section 2) but produces fixed-size encodings for them (point 3); the last two schemes have fixed-size blocks for S but produce variable-size encodings for them. Table 1 summarizes some distinguishing features for the three static schemes discussed in this section.

Table 1. Summary of discussed results on static CSSS

redundancy $\rho(k,\sigma,n)$	context size	block size	ref.
$O(\frac{\lg\sigma}{\lg n}((k+1)\lg\sigma+\lg\lg n))$	any k	variable	§3.1
$O(\frac{\lg\sigma}{\lg n}(k\lg\sigma+\lg\lg n))$	fixed $k<(1-\epsilon)\lg n/\lg\sigma$	$(1/2)\lg n/\lg\sigma$	§3.2
$O(\frac{\lg\sigma}{\lg n}(k\lg\sigma+\lg\lg n))$	$k=o(\lg n/\lg\sigma)$	$(1/2)\lg n/\lg\sigma$	§3.3

The three schemes share the same optimal time cost of $\Theta(1+m\lg\sigma/\lg n)$ for Access, and $O(n\lg\sigma)$ construction time. Space is $n\,(H_k(S)+\rho(k,\sigma,n))$ bits, where the redundancy $\rho(k,\sigma,n)$ per symbol is reported in the table and is slightly higher (a term $(k+1)$ instead of k) for the first scheme. The first scheme is oblivious with respect to the value of k (which is not part of the input but appears only in the analysis) and allows for potentially long blocks, even though the range of values for $k=\Omega(\lg n/\lg\sigma)$ gives a too large redundancy $\rho(k,\sigma,n)=\Omega(\lg\sigma)$. The second method requires a specific choice of $k<(1-\epsilon)\lg n/\lg\sigma$, for any $0<\epsilon<1$, and all blocks must contain $(1/2)\lg n/\lg\sigma$ symbols. Interestingly, it also supports append operations to add symbols at the end of S in constant amortized time per symbol. The third scheme works for all $k=o(\lg n/\lg\sigma)$ simultaneously, and all blocks must contain $(1/2)\lg n/\lg\sigma$ symbols.

3.1 LZ78 Parsing and Encoding

Sadakane and Grossi [33] introduced the notion here called CSSS, meeting the time and space requirements described in Section 1 with redundancy $\rho(k,\sigma,n)=O(\frac{\lg\sigma}{\lg n}((k+1)\lg\sigma+\lg\lg n))$ simultaneously for any k.[3] We give a simplified description of the ideas in [34], following the basic scheme of Section 2.

As for point 1, the partitioning of the input string S produces r blocks of variable sizes using first the Ziv-Lempel compression algorithm [38], also known as LZ78 parsing, and then a greedy post-processing.

The LZ78 parsing works as follows. First we initialize a trie T as empty, the current position $p=1$ in S. Then, we parse S into blocks from left to right,

[3] The authors of [15] pointed out a mistake in the smaller redundancy originally reported in [34] that we fix here in Lemma 2.

finding the longest string $t \in T$ that appears as a prefix of $S[p, n]$ (where t is the empty string when T is empty). Thus we obtain the block $S[p, p+|t|] \equiv t \cdot S[p+|t|]$ to be inserted into T. We set $p = p+|t|+1$, and repeat the parsing to discover the next block. The resulting trie T is called an *LZ-trie*, and r' is the final number of blocks generated by algorithm LZ78 (and thus the number of nodes in T). We use Lemma 2.3 from [20] for bounding r' in terms of the kth-order empirical entropy $H_k(S)$ of the string S.

Lemma 2 (Kosaraju and Manzini [20]). *Let r' be the number of blocks produced by any parsing of the string S, such that each block appears at most M times. For any $k > 0$,*

$$r' \lg r' \leq nH_k(S) + r' \lg \frac{n}{r'} + r' \lg M + \Theta(kr' \lg \sigma)$$

The *greedy post-processing* of the LZ78 parsing with window size w works as follows, for a parameter[4] $w = \frac{1}{2} \lg n / \lg \sigma$. We define a block *short* if it contains less than w symbols, and *long* otherwise. We perform a left-to-right scan of the r' blocks found by the LZ78 parsing, tagging each block as either short or long. However, during this scan, we cluster together maximal runs of consecutive *short* blocks, so that the resulting substring, called *dense block*, is not longer than w symbols: a dense block replaces the short blocks that it contains. Moreover, we impose that any two consecutive dense regions are always separated by a (short or long) block.

This greedy post-processing thus partitions S into r blocks, where $r \leq r'$, satisfying the following conditions.

- The blocks are pairwise disjoint and tagged as either long, short, or dense: by construction, no two consecutive short blocks or dense blocks can exist.
- Any two consecutive blocks contains more than w symbols in total.

As a result, any substring of S of length m overlaps with $O(1 + m/w) = O(1 + m \lg \sigma / \lg n)$ blocks. Retrieving each such block in constant time provides the claimed bound for Access. We thus discuss how to encode the sequence of blocks so that each of them can be retrieved in constant time (see points 3–5 in the basic scheme of Section 2, as we use a FID for point 2 as already discussed).

We consider the r blocks as a set of r strings, which are stored in a fast compressed trie F. Note that we do not specify the details on how to store F since there are many ways described in the chapter of this book dealing with succinct trees [30]. Conceptually F is a refinement of the LZ-trie produced during the LZ78 parsing, augmented with the dense blocks. However, since the dense blocks are of length at most w, there cannot be more than $\sigma^w = O(\sqrt{n})$ distinct ones, so they increment the size of the LZ-trie by $o(n)$ bits when obtaining F.

As a result, for each block B_j of the partition of S we get a unique identifier in $[r]$ using F and vice versa. This identifier is the encoding $E[B_j]$. The theoretical implementation of F so that given $E[B_j]$, we can retrieve B_j in constant time

[4] In practical situations it is more convenient to fix a larger value of w.

is quite complex (see [34]) but practical non-constant implementations can be adapted to this goal using string dictionaries (e.g. [4,27]).

Using the above identifiers, each block B_j is encoded by the $b = \lceil \lg r \rceil$ bits of $E(B_j)$, and the sequence $Z = E(B_1) \cdot E(B_2) \cdots E(B_r)$ is the encoding of the sequence of blocks.[5] In order to retrieve the jth block B_j, we access the jth b-bit integer $E(B_j)$ in Z by simple arithmetics, and use this integer as the identifier for the wanted block. This is given to F as an input query, and the outcome is B_j as explained above.

When computing the space bound, we need $O(r \lg(n/r) + n \lg\lg n / \lg n)$ bits for the FID in point 2, $O(n \lg\sigma \lg\lg n / \lg n)$ bits for storing F in a succinct way (point 4) given the choice of w, and $r\lfloor \lg r \rfloor \leq r \lg r + r$ bits for Z (point 5). Observing that $r \leq r'$, we can apply Lemma 2 on r, and using the fact that $r \leq 2n/w$ and $M \leq \sigma$ by construction, we can thus bound the space of Z as

$$r \lg r + r \leq nH_k(S) + \frac{2n}{w}(1 + \lg w) + \frac{n}{w} \lg \sigma + \Theta\left(\frac{nk \lg \sigma}{w}\right)$$

The total redundancy $\rho(k, \sigma, n)$ is therefore

$$O\left(\frac{\lg\lg n}{\lg n / \lg \sigma} + \frac{1}{w} \lg w + \frac{1}{w} \lg \sigma + \frac{k \lg \sigma}{w}\right) = O\left(\frac{\lg \sigma}{\lg n}((k+1) \lg \sigma + \lg\lg n)\right)$$

using the choice $w = \frac{1}{2} \lg n / \lg \sigma$.

Theorem 1 (Sadakane and Grossi [34]). *A* CSSS *using LZ78 parsing and encoding can be implemented with redundancy of* $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}((k+1) \lg \sigma + \lg\lg n))$ *simultaneously for any* k.

Note that this CSSS actually works with *any* parsing (not only LZ78) that guarantees a high-order entropy bound as stated in Lemma 2. The choice of LZ78 is motivated by the property that the space requirement of the LZ-trie dictionary (and so of the dictionary F) can be shown to be a lower-order term.

3.2 Statistical Encoding

González and Navarro [15] observed that an alternative and simpler CSSS can be obtained by using a semi-static kth-order modeling plus statistical encoding, yielding a redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ for any fixed $k < (1 - \epsilon) \lg n / \lg \sigma$ and any constant $0 < \epsilon < 1$.

A semi-static kth-order modeler applied to a string S produces the empirical conditional probability of finding a symbol c after reading substring ω in S (see Model and complexity in Section 1). Formally, $q_1, q_2, \ldots, q_n$ are the empirical probabilities such that $q_i = n_S^c / |\omega_S|$ for $k + 1 \leq i \leq n$, where ω_S is the string of single symbols following all the occurrences of substring $\omega \equiv S[i - k, i - 1]$ in S,

[5] A smarter encoding from [8] can be employed but the final redundancy does not change asymptotically.

and n_S^c is the number of occurrence of symbol $c \equiv S[i]$ in ω_s. As a result, it is noted in [15] that $\sum_{i=k+1}^{n} -q_i \lg q_i = nH_k(S)$. Hence, any statistical encoder E, such as Arithmetic coding [37], that encodes the ith symbol of S with $-q_i \lg q_i$ bits, has output size $|E(S)| = nH_k(S) + O(k \lg n)$ bits where the additive term is an upper bound for the first k symbols of S.

The above considerations are exploited in [15], as described below by following our basic scheme of Section 2. The input string S is partitioned in blocks of fixed size $w = \frac{1}{2} \lg n / \lg \sigma$. Thus there are $r = \lceil n/w \rceil$ blocks and any block can be located in constant time by simple arithmetics (points 1–2). We therefore discuss how to encode the sequence of blocks so that each of them can be retrieved in constant time (points 3–5).

As for point 3, the semi-static statistical encoder E is applied to the individual blocks using the statistics of their immediately previous k symbols. Namely, for any given block B_j of S, $j > 1$, let κ_j denote the kth-order context of B_j, defined as the last k symbols of B_{j-1} preceding B_j in S. Then, the encoding $E(B_j)$ is obtained using the conditional probabilities of the semi-static kth-order modeler applied to the concatenated sequence $\kappa_j \cdot B_j$. It is worth noting that only the symbols of B_j are encoded, and decoding $E(B_j)$ needs the knowledge of its context κ_j.

As for point 4, we store the mapping between each block B_j and its encoding $E(B_j)$, for $1 \leq j \leq r$, using a two-dimensional table T of w-long strings, such that $T[\kappa_j, E(B_j)] = B_j$, for $1 \leq j \leq r$. (Here, κ_1 is the empty string.)

It remains to describe the succinct data structures for storing and accessing the sequence Z of the statistical encodings of the blocks (point 5).

- Define $Z_j = E(B_j)$ if the number of bits $|E(B_j)| \leq (1/2) \lg n$, or $Z_j = B_j$ otherwise, for $1 \leq j \leq r$: store $Z = Z_1 \cdot Z_2 \cdots Z_r$ along with a two-level index [25] to record the lengths $|E(B_j)|$ and mark the starting position of each Z_j inside Z.
- Store a FID D such that $D[j] = 1$ if and only if $Z_j = B_j$, for $1 \leq j \leq r$. This marks the blocks B_j in S that are stored verbatim because their statistical encodings $E(B_j)$ are too large.
- Store the concatenation of contexts $K = \kappa_1 \cdot \kappa_2 \cdots \kappa_r$ as a long string.

By construction, any substring of S of length m overlaps with $O(1 + m/w) = O(1 + m \lg \sigma / \lg n)$ blocks. Retrieving each such block B_j in constant time provides the claimed bound for Access. To this end, we retrieve Z_j from Z using its two-level index. We check whether $D[j] = 1$ and, if so, we merely return Z_j as $Z_j = B_j$. Otherwise, $Z_j = E(B_j)$: we extract κ_j from K by simple arithmetics, and return $T[\kappa_j, E(B_j)]$ as B_j.

The space requirement of this CSSS can be computed as follows. The two-dimensional table T has size upper bounded by $\sigma^k \times 2^{(1/2) \lg n} \times (1/2) \lg n = O(\sigma^k \sqrt{n} \lg n)$, which is $O(n^{1-\epsilon})$ when $k < (1/2 - \epsilon) \lg n / \lg \sigma$. Playing with the multiplicative constant in the choice of w, we can allow for $k < (1 - \epsilon) \lg n / \lg \sigma$. The storage of D requires $O(r) = O(n \lg \sigma / \lg n)$ bits and that of K takes $O(nk \lg^2 \sigma / \lg n)$ bits. Finally, using Arithmetic coding as encoder E,

the storage of Z takes $nH_k(S)+O(k\lg n+r)$ bits, plus $O(n\lg\sigma\lg\lg n/\lg n)$ bits for its two-level index.

Theorem 2 (González and Navarro [15]). *A* CSSS *using semi-static kth-order modeling plus statistical encoding can be implemented with redundancy of* $\rho(k,\sigma,n) = O(\frac{\lg\sigma}{\lg n}(k\lg\sigma+\lg\lg n))$ *for any fixed* $k < (1-\epsilon)\lg n/\lg\sigma$ *and any constant* $0<\epsilon<1$.

Interestingly, this CSSS can also support append operations, where the input string S is extended as $S\cdot c_1\cdots c_2\cdots c_g$ by appending symbols $c_1, c_2, \ldots, c_g$. Using the logarithmic method and the global rebuilding technique to dynamize static data structures, the amortized cost is $O(1)$ per appended symbol.

3.3 Frequency Encoding

Ferragina and Venturini [9] coined the term CSSS and described a simplification that avoids the use of LZ-based or statistical compressors, still guaranteeing a redundancy of $\rho(k,\sigma,n) = O(\frac{\lg\sigma}{\lg n}(k\lg\sigma+\lg\lg n))$ simultaneously for all $k = o(\lg n/\lg\sigma)$.

We follow our basic scheme of Section 2 to describe this approach. As done in Section 3.2, the input string S is partitioned into blocks of fixed size $w = \frac{1}{2}\lg n/\lg\sigma$. Thus there are $r = \lceil n/w\rceil$ blocks and each block can be located in constant time by simple arithmetics (points 1–2).

The main idea is how to encode the blocks (points 3–5). We observe that while there are $r = n/w$ blocks, the number of distinct blocks is at most $\sigma^w = \sqrt{n}$. These distinct blocks are sorted in non-increasing order of frequency, namely, according to the number of times each distinct block appears in the partition of S (hence, the total sum of frequencies is r). The distinct blocks thus sorted are then assigned codewords in lexicographic order, starting form the empty string, $\epsilon, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \ldots$, so that less frequent blocks cannot get assigned shorter codewords than more frequent blocks. This is a crucial fact, as we will see. For each block B_j of S, then its encoding $E(B_j)$ is simply the codeword assigned to B_j (as a distinct block) in this way. This completes point 3.

As for point 4, we store the mapping between each block B_j and its encoding $E(B_j)$, for $1\le j\le r$, using a lookup table T such that $T[E(B_j)] = B_j$, for $1\le j\le r$.

Finally, we store $Z = E(B_1)\cdot E(B_2)\cdots E(B_r)$, the concatenation the encodings of the blocks, along with a two-level index [25] to mark the starting position of each $E(B_j)$ inside Z (point 5).

By construction, any substring of S of length m overlaps with $O(1+m/w) = O(1+m\lg\sigma/\lg n)$ blocks. Retrieving each such block B_j in constant time provides the claimed time bound for Access: retrieve $E(B_j)$ from Z using its two-level index and return $T[E(B_j)]$ as B_j.

The space requirement of this CSSS can be computed as follows. Table T requires $O(\sqrt{n}\lg n)$ bits. The two-level index for Z requires $O(\frac{n\lg\sigma\lg\lg n}{\lg n})$ bits. It is interesting to analyze the space requirement for Z as argued next. Let S_w

be the sequence of blocks as they appear in S. A relevant property is that the 0th-order entropy encoding of S_w gives the kth-order entropy encoding of S.

Lemma 3 (Ferragina and Venturini [9]). *For any $1 \leq w \leq n$, it holds $rH_0(S_w) \leq nH_k(S) + O(rk \lg \sigma)$, simultaneously over all $k \leq w$.*

The codewords $E(B_j)$ assigned to the blocks in S_w attain the 0th-order entropy according to the *golden rule* of data compression: assign shorter codewords to more frequent symbols. The crucial property is therefore that the $\sum_{j=1}^{r} |E(B_j)|$ bits in Z cannot be larger than the 0th-order encoding of S_w.

Theorem 3 (Ferragina and Venturini [9]). *A* CSSS *using frequency encoding can be implemented with redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

Fredriksson and Nikitin [12] employed a similar idea of frequency encoding using other forms of codewords but their redundancy is $\rho(k, \sigma, n) = 1 + o(H_k(S) + 1)$, which can be larger than $O(\frac{n \lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$. Among others, they adopted the Fibonacci codes [10]: any positive integer x has a unique representation as a sum Fibonacci numbers, such that no two of them are consecutive. Thus the code for x uses $\lg_\phi n + 1$ bits, where ϕ is the golden ratio: since the bit in the last position is `1`, another `1` can be appended so that this is the only position where two consecutive `1`s appear. This fact is useful to locate each $E(B_j)$ inside Z by looking at their unique pattern of consecutive pair of `1`s.

4 Dynamic Schemes

The dynamic version of CSSS is responsive to the changes of the input string S without recomputing from scratch the entire scheme after each update to S. The goal is to guarantee optimal time for Access and still high-order entropy bounds for the space in this dynamic setting. Jansson et al. [19] define the following two variants of dynamic CSSS.

The *compressed random access memory* (*CRAM*) supports Access and

Replace(i, c)**:** replace $S[i]$ by a symbol $c \in [\sigma]$.

The *extended compressed random access memory* (*ECRAM*) supports Access and

Insert(i, c)**:** insert the symbol c into S between positions $i - 1$ (if it exists) and i, and make S one symbol longer.
Delete(i)**:** delete $S[i]$ and make S one symbol shorter.

Table 2 reports the bounds achieved by the known dynamic schemes for CSSS. The space bounds are $n\,(H_k(S) + \rho(k, \sigma, n))$ bits and, for brevity, we use the notation ρ_{stat} in the table as a shorthand for the redundancy bound seen in Section 3.3 for the the static case, namely, $O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously

Table 2. Summary of discussed results on dynamic CSSS

Access(i, m)	Replace	redundancy $\rho(k, \sigma, n)$	ref.
$\Theta(1 + m \lg \sigma / \lg n)$	$O\left(\min\left\{\frac{\lg n}{\lg \sigma}, \frac{(k+1)\lg n}{\lg \lg n}\right\}\right)$	$O(\rho_{\text{stat}})$	§4.1
$\Theta(1 + m \lg \sigma / \lg n)$	$O(1/\epsilon)$	$O(\epsilon(k+1)\lg \sigma + \rho_{\text{stat}})$	§4.1
$\Theta(1 + m \lg \sigma / \lg n)$	$O(1)$	$O(\rho_{\text{stat}})$	§4.2
Access(i, m)	Insert/Delete	redundancy $\rho(k, \sigma, n)$	ref.
$\Theta(\lg n / \lg \lg n + m \lg \sigma / \lg n)$	$\Theta(\lg n / \lg \lg n)$	$O(k \lg \lg n / \lg n + \rho_{\text{stat}})$	§4.1
$\Theta(\lg n / \lg \lg n + m \lg \sigma / \lg n)$	$\Theta(\lg n / \lg \lg n)$	$O(\lg \sigma \lg \lg n / \lg n)$	§4.2

for all $k = o(\lg n / \lg \sigma)$. The reason is that these dynamic schemes use the frequency coding framework presented in Section 3.3 as a baseline.

The top part of Table 2 provides the bounds for the CRAM, and it shows that the CRAM can be implemented with the same optimal Access time and high-order entropy bound of the static CSSS seen in Section 3.

The bottom part of Table 2 provides the bounds for the extended CRAM, noting that Replace can be theoretically simulated by Delete followed by Insert. It is observed in [19] that the Access and Insert/Delete bounds are optimal as the extended CRAM solves the *list representation* problem, for which there is a cell probe lower bound of $\Omega(\lg n / \lg \lg n)$ [11]. There is no dependency on ρ_{stat} and k in the redundancy in the last line because Access has a larger complexity than that in the static case. Finally, note that the scheme in Section 3.2 supports append operations in constant amortized time.

4.1 Managing CRAM and Extended CRAM

Jansson et al. [19] started out from the frequency coding of Ferragina and Venturini [9] (Section 3.3) as a baseline to design a dynamic version of their CSSS. We review the basic scheme of Section 2 to highlight the difficulties of this dynamization process.

First of all, the partition of the input string S into fixed-size blocks (points 1–2) can be handled with some standard techniques for dynamic data structures. Since the block size is $w = (1/2) \lg n / \lg \sigma$, when n grows or shrinks by a constant factor, we rebuilt all the storage scheme using an updated value of w. This has to be incrementally deamortized to provide the worst-case bounds discussed before.

The real difficulty comes with the frequency coding of blocks (points 3–5). The golden rule behind frequency coding is that shorter codewords are assigned to more frequent blocks. When a single symbol is changed in a block, the frequency of the old block should be decreased by one and that of the new block should be either initialized to one (if this is the first time that the block appears) or increased by one. However, it is inefficient to update the lookup table T and recode the entire Z described in Section 3.3, as it could take nearly $O(r)$ time.

The lifetime of the CSSS is therefore divided into phases, where at the end of each phase there is a reconstruction (and deamortization applies too). Two fundamental ideas are employed during a phase.

The first idea is to take a snapshot F_0 of the frequencies of blocks at the beginning of the phase, and maintain an updated version F_1 of the frequencies during the phase. However, F_0 is the one employed to encode the blocks during the phase, while F_1 becomes the new F_0 only in the next phase. This approach makes sense because the high-order empirical entropy of a string does not change too much after a small change to the string.

Lemma 4 (Jansson et al. [19]). *For any two strings T and T' that differ in a single symbol from $[\sigma]$, let $t = \max\{|T|, |T'|\}$. It holds $t\,|H_k(T) - H_k(T')| = O((k+1)(\lg t + \lg \sigma))$.*

We can therefore use F_0 and delay the update of the encodings of the blocks, changing only part of the data structures. However, after some updates, we have to face the costly process of re-encoding. This requires changing Z heavily, which is expensive for several reasons. One of them is that the encoding of the blocks is of variable size, and so we cannot simply maintain Z as the concatenation of the encodings.

Consequently, the second idea is to use a memory-manager data structure to store a set of r variable-length codewords that represent the content of Z. The codewords can change length, up to $\lg r$ bits, while r cannot change during the phase. We want to access the ith codeword and reallocate it when it changes, in constant time, while keeping the wasted space small.

Lemma 5 (Jansson et al. [19]). *Let z be the total number of bits in the encoding of Z. Its r codewords can be stored using a memory manager that occupies $z + O(\lg^4 z + r \lg\lg z)$ bits while supporting the access and the reallocation operations in constant time each.*

Other data structures and invariants are described in [19] to support Access and Replace, while more sophisticated solutions are needed to support Insert and Delete. The CRAM has also a practical implementation [32].

Theorem 4 (Jansson et al. [19]). *The CRAM can be implemented so that* Access *takes optimal $\Theta(1 + m \lg \sigma / \lg n)$ time and* Replace *takes $O(1/\epsilon)$ time for any $\epsilon > 0$, with a redundancy of $\rho(k, \sigma, n) = O(\epsilon(k+1) \lg \sigma + \frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$. The redundancy can be reduced to $O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ by increasing the cost of* Replace *to $O\left(\min\left\{\frac{\lg n}{\lg \sigma}, \frac{(k+1)\lg n}{\lg\lg n}\right\}\right)$ time.*

Theorem 5 (Jansson et al. [19]). *The extended CRAM can be implemented so that* Access *takes optimal $\Theta(\lg n / \lg\lg n + m \lg \sigma / \lg n)$ time, and* Insert *and* Delete *take $\Theta(\lg n / \lg\lg n)$ time, with a redundancy of $\rho(k, \sigma, n) = O(k \lg\lg n / \lg n + \frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

4.2 Multiple Encodings of Blocks

Grossi et al. [17] conceptually represented S as a sequence S_w of r macro-symbols over the macro-alphabet $[\sigma^w]$, where each macro-symbol is a block of S. They exploited the property in Lemma 3 stating that the 0th-order entropy encoding of S_w gives the kth-order entropy encoding of S.

This implies that if we can maintain a dynamic compressed representation of S_w in $rH_0(S_w)+O(r \lg \lg n)$ bits, we obtain a dynamic compressed representation of S in $nH_k(S) + O(n\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ bits as $r = n/w$. Hence, the plan for the frequency coding of blocks (points 3–5) is to use Lemma 5 with $z = rH_0(S_w) + O(r)$ to store the codewords in Z.

We can therefore focus on dynamically maintaining the encoding of the macro-symbols in $[\sigma^w]$ to obtain a 0th-order entropy encoding of S_w. We divide the whole set of assigned codewords into $O(\lg r)$ classes C_j. In the ideal static situation, each macro-symbol y of frequency f_y is assigned a codeword from the class C_j such that $\frac{r}{2^j} < f_y \leq \frac{r}{2^{j+1}}$. Recalling that $\sum_{y \in [\sigma^w]} f_y = r$, the 0th-order entropy plus a lower-order term is achieved for S_w when the codeword for y is $\lg(r/f_y) = j + O(1)$ bits long. This implies that $|C_j| \leq 2^{j+O(1)}$. In the dynamic setting, we need flexibility and so we assign more than one class to y, under the requirement that y has at most one codeword assigned from each such class.

When a symbol of S is replaced, the frequency of a macro-symbol can change. Thus, macro-symbols may move to different classes in the lifetime of the data structure. Once a macro-symbol enters a class for the first time, it is assigned an available codeword e of that class; the next time it will re-enter that class, it will reuse the same codeword e. Since the number of available codewords in any class is limited, it may happen that the last available codeword is consumed in this way (i.e., $|C_j| = 2^{j+O(1)}$). If so, it is shown in [17] that $\Omega(r)$ Replaces have been done, and so we can amortized the cost (which can be deamortized with an incremental rebuilding technique).

This mechanism causes no significant waste of bits in the 0th-order empirical entropy of S_w, since the extra space is a lower-order term: the waste due to multiple codewords (from distinct classes) for the same y is just $O(f_y)$ bits.

Lemma 6 (Grossi et al. [17]). *For any macro-symbol $y \in [\sigma^w]$, the overall space required by the codewords of y in the encoding of S_w is $f_y \lg \frac{r}{f_y} + O(f_y)$ bits.*

The implication of Lemma 6 is that the encoding of S_w takes $\sum_{y \in [\sigma^w]} f_y \lg \frac{r}{f_y} + O(f_y) = rH_0(S_w) + O(r)$ bits.

Other properties and data structures are employed to obtain the bounds for the CRAM and the extended CRAM. In the latter, a variable-size partition of S is maintained.

Theorem 6 (Grossi et al. [17]). *The CRAM can be implemented so that* Access *takes $\Theta(1 + m \lg \sigma / \lg n)$ time and* Replace *takes $O(1)$ time, with a redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

Theorem 7 (Grossi et al. [17]). *The extended CRAM can be implemented so that* Access *takes* $\Theta(\lg n/\lg\lg n + m\lg\sigma/\lg n)$ *time, and* Insert *and* Delete *take* $\Theta(\lg n/\lg\lg n)$ *time, with a redundancy of* $\rho(k,\sigma,n) = O(\lg\sigma\lg\lg n/\lg n)$.

5 Further Discussion and Conclusions

This paper described a survey on compressed string storage schemes (CSSS) that have optimal access time, as if the text was uncompressed, and squeeze the text of n symbols over alphabet $[\sigma]$ to reach a space bound in bits that is close to the kth-order empirical entropy $nH_k + o(n\lg\sigma)$ (see Table 1). Time bounds for replacing, inserting and deleting individual text symbols are also optimal (see Table 2). Interestingly, there are currently no lower bounds on the redundancy $o(n\lg\alpha)$ when a CSSS has optimal access time. From the practical point of view, some of the proposed methods have been implemented but still the research in this direction has not been fully explored all the directions. For example, one main limitation in practice is that the required value of the block size $w = (1/2)\lg n/\lg\sigma$ is quite small, even for ASCII text. New results in this direction could bring also fresh theoretical questions to investigate.

When removing some of the optimality constraints on a CSSS, such as the constant-time bound for accessing $O(w)$ symbols of the text or the high-order entropy H_k, there are a plethora of ideas and solutions. Indeed, random access to compressed data is a basic problem in many applications on massive data sets, and there are too many practical and effective solutions to be mentioned in this paper (e.g. see the book [36]).

Compressed text indexing is a good source of ideas and sophisticated tools (e.g. see the surveys [16,18,26]) that can achieve high-order H_k entropy bounds. For example, several solutions are based on storing the Burrows-Wheeler transform [5] in some 0th-order compressed data structures, but recovering a substring of the text is suboptimal when compared to the optimal access cost of CSSS. On the other hand, these solutions have indexing and searching functionalities while CSSS can only support Access.

Another field that is steadily growing is that of grammar compressed texts (e.g. see the survey [22] and a practical algorithm [21]). Some elegant results can decode a substring of length m in $O(m + \lg n)$ time [2], where n is the length of the uncompressed text, and there is a matching lower bound for the logarithmic cost [35]. In general, these methods can potentially compress better than the CSSS described in this survey, but finding the optimal grammar is NP-hard. The problem admits a logarithmic approximation factor [6,31], where the measure is the number of rules rather than the number of bits to represent the grammar.

Several other kinds of encodings have the property that each substring of the input text is translated into a substring of the corresponding encoded text (e.g. [3,23]), but they do not achieve high order entropy but still compress well in practice. One theoretical question is related to the encoding in [8] that shows how to store optimally a sequence of integers with constant-time random access to read and change one element. In our notation, the stated bound is $\lceil n\lg\sigma\rceil + O(1)$

bits, which can save $\Theta(n)$ bits over the raw representation in $n\lceil \lg \sigma \rceil$ bits. When comparing it to the bound of $nH_k + o(n \lg \sigma)$ for CSSS, the extra space is just $O(1)$ bits instead of $o(n \lg \sigma)$, but H_k can be much lower than $\lg \sigma$ for compressible text (and so $nH_k = o(n \lg \sigma)$). An interesting open problem is whether it is possible to combine the best of the two choices in some tradeoff that can blend the two opposite situations mentioned above.

Acknowledgments. I am in debt with Giuseppe Ottaviano for reading and commenting a preliminary version of this paper, and with Philip Bille, Gonzalo Navarro, Kunihiko Sadakane, Rossano Venturini for answering clarification requests, and Rossano for pointing out the open question mentioned at the end of the conclusions.

References

1. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
2. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: SODA, pp. 373–389 (2011)
3. Brisaboa, N.R., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
4. Brisaboa, N.R., Cánovas, R., Claude, F., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 136–147. Springer, Heidelberg (2011)
5. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
6. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
7. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. J. Algorithms 48(1), 2–15 (2003)
8. Dodis, Y., Patrascu, M., Thorup, M.: Changing base without losing space. In: STOC, pp. 593–602 (2010)
9. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. 372(1), 115–121 (2007)
10. Fraenkel, A.S., Kleinb, S.T.: Robust universal complete codes for transmission and compression. Discrete Applied Mathematics 64(1), 31–55 (1996)
11. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: STOC, pp. 345–354 (1989)
12. Fredriksson, K., Nikitin, F.: Simple random access compression. Fundam. Inform. 92(1-2), 63–81 (2009)
13. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379, 405–417 (2007)
14. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comput. Sci. 387, 348–359 (2007)
15. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 294–305. Springer, Heidelberg (2006)

16. Grossi, R.: A quick tour on suffix arrays and compressed suffix arrays. Theor. Comput. Sci. 412(27), 2964–2973 (2011)
17. Grossi, R., Raman, R., Rao, S.S., Venturini, R.: Dynamic compressed strings with random access. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 504–515. Springer, Heidelberg (2013)
18. Hon, W.-K., Shah, R., Vitter, J.S.: Compression, indexing, and retrieval for massive string data. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 260–274. Springer, Heidelberg (2010)
19. Jansson, J., Sadakane, K., Sung, W.K.: Cram: Compressed random access memory. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 510–521. Springer, Heidelberg (2012)
20. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM Journal of Computing 29(3), 893–911 (1999)
21. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Data Compression Conference, pp. 296–305 (1999)
22. Lohrey, M.: Algorithmics on slp-compressed strings: A survey. Groups Complexity Cryptology 4(2), 241–299 (2012)
23. Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inf. Syst. 15(2), 124–136 (1997)
24. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3), 407–430 (2001)
25. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
26. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39(1) (2007)
27. Ottaviano, G., Grossi, R.: Fast compressed tries through path decompositions. In: ALENEX, pp. 65–74 (2012)
28. Patrascu, M., Viola, E.: Cell-probe lower bounds for succinct partial sums. In: Charikar, M. (ed.) SODA, pp. 117–122. SIAM (2010)
29. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
30. Raman, R., Rao, S.S.: Succinct representations of ordinal trees. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Munro Festschrift 2013. LNCS, vol. 8066, pp. 319–332. Springer, Heidelberg (2013)
31. Rytter, W.: Application of lempel-ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)
32. Sadakane, K.: Personal communication (2012)
33. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proc. of the 17th ACM-SIAM SODA, pp. 1230–1239 (2006)
34. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: SODA, pp. 1230–1239. ACM Press (2006)
35. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 247–258. Springer, Heidelberg (2013)
36. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers (1999)
37. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM 30(6), 520–540 (1987)
38. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Succinct and Implicit Data Structures for Computational Geometry

Meng He

Faculty of Computer Science, Dalhousie University, Halifax, NS, B3H 4R2, Canada
mhe@cs.dal.ca

Abstract. Many classic data structures have been proposed to support geometric queries, such as range search, point location and nearest neighbor search. For a two-dimensional geometric data set consisting of n elements, these structures typically require $O(n)$, close to $O(n)$ or $O(n \lg n)$ words of space; while they support efficient queries, their storage costs are often much larger than the space required to encode the given data. As modern applications often process very large geometric data sets, it is often not practical to construct and store these data structures.

This article surveys research that addresses this issue by designing space-efficient geometric data structures. In particular, two different but closely related lines of research will be considered: *succinct geometric data structures* and *implicit geometric data structures*. The space usage of succinct geometric data structures is equal to the information-theoretic minimum space required to encode the given geometric data set plus a lower order term, and these structures also answer queries efficiently. Implicit geometric data structures are encoded as permutations of elements in the data sets, and only zero or $O(1)$ words of extra space is required to support queries. The succinct and implicit data structures surveyed in this article support several fundamental geometric queries and their variants.

1 Introduction

Many applications such as spatial databases, computer graphics and geographic information systems store and process geometric data sets that typically consist of point coordinates. In other applications such as relational databases and data mining applications, the given data are essentially sets of records whose fields are values of different properties, and thus can be modeled as geometric data in multidimensional space. Thus the study of geometric data structures which can potentially be used to preprocess these data sets so that various queries can be performed quickly is critical to the design of a large number of efficient software systems.

Researchers have studied many different geometric queries. Among them, the following three geometric query problems are perhaps the most fundamental:

- *Range Search*: Preprocess a point set, so that information regarding points inside a query region, e.g., the number of points in this region, can be efficiently computed;

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 216–235, 2013.

- *Point Location*: Preprocess a subdivision of space into a set of cells, so that the cell that contains a query point can be located quickly;
- *Nearest Neighbor*: Preprocess a point set, so that the point closest to a query point can be found efficiently.

Extensive work has been done to design data structures that provide fast support for these queries. Take planar point location for example. A number of different data structures have been designed to support point location in 2D, achieving $O(\lg n)$[1] optimal query time using linear space, including the classic data structures proposed by several different groups of researchers in the 70's and 80's [53,52,32,28,68]. Such work has yielded a large number of solutions to geometric query problems, and invented many data structure design techniques.

Most geometric data structures designed to manipulate data sets in two-dimensional space use linear, almost linear, or $O(n \lg n)$ words of space. Asymptotically, $O(n)$-space structures occupy less space than other solutions, but the constants hidden in the asymptotic space bounds are however usually large. As modern applications often process large data sets measured in terabytes or even petabytes, it has become undesirable or even infeasible to construct these data structures. Even for smaller data sets of several gigabytes, the storage cost of these structures often makes it impossible to load them into internal memory of computer systems, and performance is sacrificed if they have to be stored in external memory which is orders of magnitude slower. Thus, during the past decade, researchers have been developing data structure techniques that can be used to reduce storage cost drastically.

Succinct data structures have been used to design more space-efficient solutions to geometric query problems. Initially proposed by Jacobson [50] to represent combinatorial objects such as bit vectors, trees and graphs, the research in succinct data structures aims at designing data structures that can represent the given data using space close to the information-theoretic lower bound, and support efficient navigational operations in them. Numerous results have been achieved to represent combinatorial objects and text indexes succinctly [61,66,10,64,35,67,62], which save a lot of space compared with standard data structures.

The study of *implicit data structures* is another line of research that is focused on improving space efficiency of data structures, and it has also been applied to computational geometry. The term implicit means that structural information is encoded in the relative ordering of data elements, rather than explicit pointers. Thus an implicit data structure is stored as a permutation of its data elements, and with zero or a constant number of words of additional space, it can support operations. The standard binary heap is one of the earliest and most well-known data structures that are implicit. Later, researchers have designed implicit data structures for a number of problems such as partial match on a set of multi-key records [59,40,6] and dynamic predecessor search [60,39].

The aim of this article is to survey research work that has been conducted to design succinct geometric data structures and implicit geometric data structures,

[1] $\lg n$ denotes $\log_2 n$.

in order to help researchers understand and follow research work in this area. The main ideas of the most relevant work are summarized, and main results are presented in theorems. To help readers evaluate these contributions, some background information of each geometric query discussed in this article is also given, though a thorough review of all the related work which is not necessarily focused on space efficiency issues is out of the scope of this survey. In the rest of this article, we devote one section to each of the three fundamental geometric queries listed in this section, to discuss succinct and implicit data structures designed for these queries and their variants. We present some concluding remarks and give some open problems in the last section.

2 Range Search

In this section, we survey succinct and implicit data structures designed for range search. As there are many variants of range search queries and they may be studied under different models, we further divide this section into subsections. In each subsection, before we describe succinct or implicit solutions, we define each range query and give some background information.

2.1 Implicit Data Structures for Orthogonal Range Reporting

Orthogonal range reporting is a fundamental range search problem. In this problem, we preprocess a set, N, consisting of n points in d-dimensional space, so that given a d-dimensional axis-aligned query box P, the points in $N \cap P$ can be reported efficiently. Thus in the two-dimensional case, P is an orthogonal query rectangle. We follow the convention and let k denote the number of points to be reported.

This problem has been studied extensively. The classic kd-tree proposed by Bentley [11] is a linear-space data structure that can answer orthogonal range reporting queries in $O(n^{1-1/d} + k)$ time. Thus in the two-dimensional case, the query time is $O(\sqrt{n} + k)$. By allowing penalties for each point to be reported, different query time can be achieved; in two-dimensional space, the linear-space data structure of Chazelle [25] can answer queries in $O(\lg n + k \lg^{\epsilon} n)$ time, for any positive constant ϵ. However, to achieve the optimal $O(\lg n + k)$ query time, more space is required. The data structure of Chazelle [25] uses $O(n \lg^{\epsilon} n)$ words of space to answer queries in optimal time, for any positive constant ϵ. For more recent results that aim at improving query time in higher dimensions using more space under the RAM model, see the work of Chan *et al.* [22]. For similar work under the pointer machine model, we refer to results presented by Afshani *et al.* [1,2].

The first implicit data structure for range reporting is an implicit representation of kd-tree proposed by Munro in 1979 [59]. This is the only implicit or succinct geometric structure that was not designed within the past decade. Originally, this structure was introduced as a structure storing multi-key data records to support partial matching, but its description can be easily rewritten to support geometric queries as follows. To construct this data structure, let $C[0..n-1]$

be the array in which each element stores the coordinates of a point in N, and the construction algorithm reorders the elements of C to encode the kd-tree implicitly. Assume that dimensions are numbered $0, 1, \ldots, d-1$. The construction algorithm consists of $\lceil \lg n \rceil$ stages. Initially, before stage 0, we treat the entire array C as one segment containing all the array entries; in each stage, we essentially divide each segment of C produced in the previous stage into two halves and reorder elements accordingly. More precisely, in stage i, for each segment S produced in the previous stage, we perform the following three steps:

1. Look for the point whose coordinate in dimension $(j = i \bmod d)$ is the median among the coordinates of all the points in S in this dimension, and swap it with the point stored in the middle of S. Let m be this median value.
2. Move the points in S whose coordinates in dimension j are smaller than m to the first half of S, and those with large coordinates to the second half.
3. Let the first half and the median element be one new segment, and the second half the other new segment.

At the end of the last stage, each segment will store exactly one point, and the content of array C becomes the implicit structure.

Readers who are familiar with kd-trees can now see that after this construction algorithm, the array C encodes a kd-tree implicitly. If we number the levels of a kd-tree starting from the root level as levels $0, 1, \ldots$, then a segment produced in the ith stage in this algorithm corresponds to a node at the ith level of the kd-tree. Furthermore, the region represented by each node can be inferred during a top-down traversal by checking the median values. Thus we can modify algorithms over kd-trees to support orthogonal range reporting: Start from the segment that contains all the points. Each time we investigate a segment produced in stage i, we check if the region corresponding to this segment is entirely contained in the query box. If it is, then report all the points stored in this segment. Otherwise, perform the algorithm recursively on the two child segments produced from this segment. The analysis on the original kd-tree can also be used to show the following theorem:

Theorem 1 ([59]). *Given a set of n points in d-dimensional space, there exists an implicit data structure that can answer orthogonal range reporting queries over this point set in $O(n^{1-1/d}+k)$ time, where k is the number of points reported. This data structure can be constructed in $O(n \lg n)$ time.*

Arroyuelo *et al.* [8] designed adaptive data structures for two-dimensional orthogonal range reporting, including an implicit version. Their data structures are adaptive in the sense that they take advantage of the inherent sortedness of given data to improve search efficiency. In their work, they say that a set of points form a monotonic chain if, when listed from left to right, the y-coordinates of these points are either monotonically increasing or monotonically decreasing; the sortedness of a given point set is then measured in terms of the minimum number, m, of monotonic chains that the points in this set can be decomposed into. Their linear-space data structure can support two-dimensional orthogonal

range counting in $O(m + \lg n + k)$ time. For any point set, m is bounded by $O(\sqrt{n})$ (this is an obvious consequence of the Erdős-Szekeres theorem), so the query performance matches that of the *kd*-tree in the worst case, and can be significantly better if m is small. Though it is NP-hard to compute m, there is an $O(n^3)$-time approximation algorithm that can achieve constant approximation ratio [38].

We briefly summarize the main idea of constructing the implicit version of this data structure. Observe that, if we decompose the point set into m monotonic chains and store each chain in a separate array, then we can perform binary search m times to answer a query. If we concatenate these arrays into a single array storing all point coordinates, we need encode the starting position of each sub-array, which requires $O(m \lg n) = O(\sqrt{n} \lg n)$ bits. This can be encoded using a standard technique: in each sub-array, we group points into pairs. If we swap the points in a pair, then we use this to encode a 1 bit; otherwise, a 0 bit is encoded. If we know whether the chain this pair is in is monotonically increasing or decreasing, then we can decode this bit by comparing this pair of coordinates. Extra care has to be taken to address the case in which chains contain odd numbers of points, so that the encoded information can be decoded correctly. Note that the query performance of this implicit structure is slightly worse than their linear-space structure; the techniques used in the latter to speed up query requires the storing of duplicate copies of some points which is not allowed in the design of implicit data structures. We summarize the space and time cost of their implicit data structure in the following theorem:

Theorem 2 ([8]). *Given a set of n points in two-dimensional space, there exists an implicit data structure that can answer orthogonal range reporting queries over this point set in $O(m \lg n + k)$ time, where k is the number of points reported, and $m = O(\sqrt{n})$ is the minimum number of monotonic chains that this set can be decomposed into. This data structure can be constructed in $O(n^3)$ time.*

2.2 Succinct Data Structures for Orthogonal Range Counting and Reporting

Another fundamental range query is *orthogonal range counting*. Here we focus on the two-dimensional case, as this is what the succinct data structures that we survey in this section are designed for. In the two-dimensional range counting problem, we are to preprocess a set, N, of n points in the plane, so that given an axis-aligned query rectangle, P, the number of points in $N \cap P$ can be computed efficiently. Among linear-space data structures for this problem, the structure of Chazelle [25] supports range counting in $O(\lg n)$ time. When point coordinates are integers, JáJá *et al.* [51] designed a linear-space structure that answers queries in $O(\lg n / \lg \lg n)$ time. This matches the lower bound on query time proved by Pătraşcu [63] under the cell probe model for data structures occupying $O(n \lg^{O(1)} n)$ words of space.

In a special case, the point sets are in rank space, i.e., they are on an $n \times n$ grid. The general orthogonal range counting and reporting problems can be reduced to this using a well-known technique [42], and Pătraşcu's lower bound mentioned in the previous paragraph also applies to this special case. Some range search structures [42,25,5] for the more general case were achieved by first considering rank space. Indeed, this reduction allowed Chazelle [25] to use operations under RAM to achieve the first linear-space solution that supports range counting in $O(\lg n)$ time, which is more space-efficient than the classic range tree [12] which uses $O(n \lg n)$ space to provide the same query support. In addition, the study of this problem is crucial to the design of several space-efficient text indexing structures [54,14].

To describe succinct data structures designed for this problem, some background knowledge is required. In particular, there is a key structure which is also used in most other succinct data structures: bit vectors. Given a bit vector $B[1..n]$ storing 0 and 1 bits, the following three operations are considered:

- $\texttt{access}(B, i)$ which returns $B[i]$;
- $\texttt{rank}_\alpha(B, i)$ which returns the number of times the bit α occurs in $B[1..i]$, for $\alpha \in \{0, 1\}$;
- $\texttt{select}_\alpha(B, i)$ which returns the position of the ith occurrence of α in B, for $\alpha \in \{0, 1\}$.

Jacobson [50] considered this problem under the bit probe model; later Clark and Munro [27] showed how to represent a bit vector succinctly using $n + o(n)$ bits to support these three operations in $O(1)$ time under the word RAM model with word size of $O(\lg n)$ bits (this is the model that almost all succinct data structures assume; unless otherwise specified, succinct data structure results surveyed in the rest of this article assume this model). We refer to the work of Pătraşcu [64] for the most recent result on this extensively studied fundamental problem.

The definitions of `rank` and `select` operations on bit vectors can be generalized to a string $S[1..n]$ over alphabet $[\sigma]$[2], by letting α take any value in $[\sigma]$. The first succinct data structure designed for this problem is the wavelet tree of Grossi *et al.* [44]. In a wavelet tree representing string S, each node, v, represents a range of alphabet symbols $[a..b]$. Let S_v be the subsequence of S (not necessarily a contiguous subsequence) consisting of all the characters of S in $[a..b]$, and let $n_v = |S_v|$. Then a bit vector B_v of length n_v is constructed for the node v, in which $B_v[i] = 0$ iff $S_v[i] \in [a..\lfloor (a+b)/2 \rfloor]$. Thus the 0 bits correspond to the characters in the smaller half of the range of alphabet symbols represented by v, and 1 bits correspond to the greater half. Node v has two children v_1 and v_2, corresponding to the two subsequences of S_v that consist of all the characters of S_v in $[a..\lfloor (a+b)/2 \rfloor]$ and $[\lfloor (a+b)/2 \rfloor + 1..b]$, respectively. In other words, v_1 and v_2 correspond to the 0 and 1 bits stored in B_v, respectively. Bit vectors for these children and their descendants are defined in a recursive manner. To construct a wavelet tree for S, the root represents the range $[1..\sigma]$, and nodes at

[2] $[\sigma]$ denotes the set $\{1, 2, \ldots, \sigma\}$.

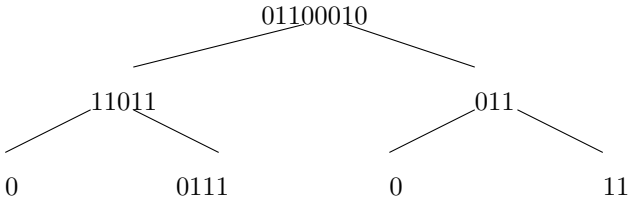


Fig. 1. A wavelet tree constructed for the string 35841484 over an alphabet of size 8. This is also a wavelet tree constructed for the following point set on an 8 by 8 grid: $\{1,3\},\{2,5\},\{3,8\},\{4,4\},\{5,1\},\{6,4\},\{7,8\},\{8,4\}$.

each successive level are created recursively as mentioned previously. Each leaf represents a range of size 2. Thus the wavelet tree has $\lceil \lg \sigma \rceil$ levels. The wavelet tree is not stored explicitly. Instead, for each level, we visit the nodes from left to right, and concatenate the bit vectors created for these nodes. The concatenated bit vector is represented using a bit vector structure supporting `rank` and `select`. Thus the wavelet tree is stored as $\lceil \lg \sigma \rceil$ bit vectors of the same length n, which occupy $(n + o(n))\lceil \lg \sigma \rceil$ bits in total. See Figure 1 for an example.

No additional information is required to navigate in the wavelet tree; during a top-down traversal, the `rank` operation over each length-n bit vector is sufficient to identify the starting and ending positions of the bit vector B_v which corresponds to a node v. By taking advantage of this, one can design algorithms that support `access`, `rank` and `select` operations over S. These algorithms perform a constant number of rank/select operations over the bit vector constructed for each level of the tree, and hence their running time is $O(\lg \sigma)$.

Mäkinen *et al.* [54] showed that a wavelet tree can be used to support orthogonal range counting and reporting for n points on an $n \times n$ grid, when the points have distinct x-coordinates (this restriction can be removed through a reduction [14]). To construct a wavelet tree for such a point set, we conceptually treat it as a string S of length n over alphabet $[n]$, in which $S[i]$ stores the y-coordinate of the point whose x-coordinate is i. Thus Figure 1 can also be considered as a wavelet tree constructed for 8 points on an 8 by 8 grid, and this point set is given in the caption of the figure. To show how to support range search, take range counting for example. Suppose that the given query rectangle P is $[x_1..x_2] \times [y_1..y_2]$. To count the number of points in P, we perform a top-down traversal of the wavelet tree, and use a variable c (initially 0) to record the number of points that we have identified to be inside P during this traversal. At each node v, if the range $[a,b]$ represented by v is a subset of $[y_1..y_2]$, then the entries of S_v that correspond to points inside P form a contiguous substring of S_v. The starting and ending positions of this substring in S_v can be computed by performing rank queries over bit vectors, during the top-down traversal. With these positions, we can increased the value of c by the size of this substring. If range $[a,b]$ intersects $[y_1..y_2]$ but is not its subset, then we visit the children of v whose ranges intersect $[y_1..y_2]$ and perform the above process recursively.

This algorithms visits at most two nodes and performs a constant number of rank/select operations over bit vectors at each level of the wavelet tree, and thus range counting can be supported in $O(\lg n)$ time. Range reporting can be supported in a similar manner, in $O((k+1)\lg n)$ time.

There is a similarity between wavelet trees and Chazelle [25]'s data structure for orthogonal range search in rank space. During the construction of Chazelle's structure for n points on an $n \times n$ grid, a set of $\lceil \lg n \rceil$ conceptual bit vectors of length n is also defined. Unlike bit vectors in wavelet trees, these vectors encode the process of performing mergesort on y-coordinates when points are pre-sorted by x-coordinate. Bits in these conceptual vectors are then organized in a set of non-succinct data structures to facilitate queries; these non-succinct structures could be replaced by succinct bit vectors designed after Chazelle's work to reduce space cost. Similar algorithms can be performed on wavelet trees and Chazelle's structure to support range search. The exact content of these bit vectors are however not the same. The underlaying tree structures are also different in these two structures: The bit vectors corresponding to the nodes at the same level of a wavelet tree may be of different lengths, while this is not the case in Chazelle's structure due to the nature of mergesort. The difference in layout allows wavelet trees to directly encode strings succinctly to support rank/select when the string length is much larger than the alphabet size.

To speed up rank/select operations on strings, Ferragina *et al.* [37] designed a data structure called generalized wavelet tree. The main difference between this structure and the original wavelet tree is that each node, v, in the generalized wavelet tree has $t = \lg^{\epsilon} n$ children for a positive constant ϵ less than 1. Instead of constructing a bit vector for v, a string over alphabet $[t]$ is constructed, and each character in $[t]$ corresponds to a subrange represented by a child of v. They then designed a succinct representation of strings over an alphabet of size t that can support `access`, `rank` and `select` in constant time. Thus a generalized wavelet tree has $O((\lg \sigma / \lg \lg n) + 1)$ levels, and operations at each level can be supported in constant time. This can be used to support `access`, `rank` and `select` in $O((\lg \sigma / \lg \lg n) + 1)$ time. Note that the idea of increasing the fanout of a tree structure to speed up query time by an $O(\lg \lg n)$ factor is a standard strategy under the word RAM model, and was also used by JáJá *et al.* [51] for range search. New auxiliary structures, however, had to be designed to guarantee that the resulting data structure is still succinct.

To further use a generalized wavelet tree to provide better support for range search, we need a succinct representation of n points on a narrow grid, i.e., an $n \times O(\lg^{\epsilon} n)$ grid, that supports range counting and reporting in constant time. The simultaneous work of two groups of researchers, Bose *et al.* [14] and Yu *et al.* [70] designed such data structures. These data structures are essentially equivalent and they provide the same support for queries. The following theorem summarizes the support for range search provided by a generalized wavelet tree:

Theorem 3 ([14,70]). *A set of n points on an $n \times n$ grid can be represented using $n \lg n + o(n \lg n)$ bits to support orthogonal range counting in $O(\lg n / \lg \lg n)$ time, and orthogonal range reporting in $O((k+1)\lg n / \lg \lg n)$ time, where k is*

the number of points reported. This data structure can be constructed in $O(n \lg n)$ *time.*

Bose *et al.* [14] further used this to improve previous results on designing succinct data structures for integer sequences and text indexes. One text index they designed is an improvement over previous results on *position-restricted text search*, which looks for the occurrences of a given query string within a given substring of the text.

2.3 Other Range Queries

2D 3-Sided Orthogonal Range Reporting. In the *2D 3-sided orthogonal range reporting* problem, we are given a set, P, of n points in the plane, and the query is to report the points in 3-sided query ranges of the form $[x_1, x_2] \times (-\infty, y_2]$. The classic priority search tree of McCreight [58] can answer such a query in $O(\lg n + k)$ time, where k is the number of points to be reported. Each node in a priority search tree stores one point from P; the entire tree can be viewed as a min-heap over the y-coordinates of points in P, and a binary search tree over x-coordinates. To facilitate the navigation in the tree by x-coordinate, each node also stores the median value of the x-coordinates of the points stored in its descendants, as this value is used to distribute points to subtrees rooted at the left or right child of this node. Brönnimann *et al.* [15] designed a variant of priority search trees to avoid storing this extra median x-coordinate in each node, and this variant can be made implicit. To lay out points from P in an array, they store the point, p_0, with the smallest y-coordinate among points in P at the head of the array, and the point, p_1, with the median x-coordinate among points in $P \setminus \{p_0\}$ in the next entry. Then they divided all the points in $P \setminus \{p_0, p_1\}$ into two halves by this median x-coordinate, and recurse in these two subarrays. The query algorithm for the original priority search tree can be easily adapted to this implicit variant to answer a 3-sided orthogonal range reporting query in $O(\lg n + k)$ time.

If we would like to use this implicit priority search tree to support range reporting for both query ranges of the form $[x_1, x_2] \times (-\infty, y_2]$ and ranges of the form $[x_1, x_2] \times [y_1, \infty)$, then two trees have to be constructed. To avoid the duplication of point coordinates, De *et al.* [29] designed a min-max priority search tree which stores one copy of point coordinates to answer queries of both forms. They also showed that their structure can be made implicit, answering queries in $O(\lg n + k)$ time.

Simplex, Halfspace and Ball Range Reporting. In the *simplex range reporting* problem, the query region is a simplex, while in *halfspace range reporting*, it is a halfspace. For these two problems, Brönnimann *et al.* [15] showed how to lay out, in an array of coordinates, the variants of partition trees proposed by Matousek [55,56]. In d-dimensional space, one variant of their implicit partition tree can answer simplex range reporting in $O(n^{1-1/d+\epsilon} + k)$ time, where k is the number of points to be reported and ϵ is a positive constant that can be

arbitrarily small. The query performance is very close to $O(n^{1-1/d} + k)$ which is believed to be the optimal query time using linear-space structures [57,20]. They designed another implicit partition tree that can support halfspace range reporting in $O(n^{1-1/\lfloor d/2 \rfloor + \epsilon} + k)$ time. For discussions of better trade-offs for halfspace range reporting in different cases using linear-space structures, see [20]. Both implicit data structure can be constructed in $O(n \lg n)$ expected time.

In the unpublished full version of [15], Brönnimann *et al.* further showed how to speed up the support for simplex range reporting at the cost of increased preprocessing time. More precisely, the n^ϵ factor in the query time can be replaced by polylog n, and the preprocessing time is increased to $O(n \text{ polylog } n)$. They also stated that that their implicit data structures can support *simplex range counting*, i.e., counting the number of points in the query simplex, without the $O(k)$ additive term (this applies to both trade-offs).

The d-dimensional *ball range reporting* problem, in which the query region is a ball, can be reduced to $(d+1)$-dimensional halfspace range reporting by the standard lifting transformation that maps points in $\mathbb{R}^d$ to points on the unit paraboloid in $\mathbb{R}^{d+1}$. Thus it follows from [15] that there is an implicit data structure supporting ball range reporting in $\mathbb{R}^d$ in $O(n^{1-1/\lfloor (d+1)/2 \rfloor} \text{ polylog } n + k)$ time.

3 Point Location

In the *planar point location query* problem, we preprocess a planar subdivision with n vertices, so that the face containing a given query point can be located quickly. In a special case of this problem, the planar subdivision is a planar triangulation, i.e., a planar subdivision in which each face is a triangle. When the data set is static, the general problem can be reduced to this special case by triangulating all the faces of the planar subdivision. As mentioned in Section 1, a number of linear-space classic solutions were proposed to support point location in the optimal $O(\lg n)$ time based on different techniques [53,52,32,28,68].

Succinct data structures that represent planar triangulations and planar maps [17,18,9] using $O(n)$ bits support queries regarding connectivity information such as adjacency test, but they cannot be directly combined with point location structures without using additional space of $O(n)$ words or $O(n \lg n)$ bits. Thus, to design space-efficient solutions to point location, Bose *et al.* [13] proposed to design data structures called *succinct geometric indexes*. These data structures occupy $o(n)$ bits, excluding the space needed for the input array, which stores the coordinates of the n vertices of the subdivision. The n vertices may be permuted in the array. Hence $o(n)$ bits of space is the only extra storage required to support queries, in addition to the storage cost required of the given data.

They first designed a succinct geometric index that supports point location in planar triangulations. To construct this index, they use graph separators twice to decompose the given planar triangulation T. More precisely, they first, in the top-level partition, apply the t-separator theorem [3] on the dual graph of T, choosing $t = \lg^3 f/f$, where f is the number of faces of T. By doing so, they partition

T into a separator consisting of $O(n/\lg^{3/2} n)$ faces and $O(n/\lg^{3/2} n)$ subgraphs called *regions*; each region consists of $O(\lg^3 n)$ vertices and corresponds to a connected component of the dual graph after removing the separator. They further, in the bottom-level partition, apply the separator theorem on each region to create subregions consisting of $O(\lg n)$ vertices each. The reason why they perform two levels of partition is that they intend to create one point location structure for the top-level partition (any linear-space solution supporting query in logarithmic time will be sufficient) and a point location structure for each region to answer queries. The structure for the top-level partition is constructed by first triangulating the graph consisting of the outer face and the separator for the top-level partition, and then building a structure to answer point location. This structure will either report that the query point is in a separator face and thus terminate, or locate the region containing the query point. Since the size of this graph is $O(n/\lg^{3/2} n)$, $O(n/\lg^{1/2} n) = o(n)$ bits would be sufficient. Then, they construct a similar point location structure for each region to tell whether the given query point is in a separator face of this region, or in a particular subregion. They hope that, since there are $O(\lg^3 n)$ vertices in each region, $O(\lg\lg n)$ bits would be sufficient to identify each vertex and to encode each pointer in these point location structures. This guarantees that all these point location structures occupy $o(n)$ bits in total. Finally, if the point is in a subregion, they check each face of the subregion to compute the result. If this idea works, then the query time would be $O(\lg n)$.

The main challenge for this to work is that after applying the separator theorem, each vertex could appear in multiple regions and/or subregions, so that we can not simply assign an $O(\lg\lg n)$-bit identifier for each vertex when constructing the point location structure of a region. To overcome this difficulty, they use the following strategy to assign identifiers at three different levels for each vertex. First, for each subregion that a vertex is in, a *subregion-label* of $O(\lg\lg n)$ bits is assigned by applying the approach of Denny and Sohler [30] that permutes the vertex set to encode the graph structure of a planar triangulation. For each subregion, the rank of a vertex in the permuted sequence for this subregion becomes its subregion-label for its occurrence in this subregion. Next, for each region, a *region-label* is assigned to each of its vertices as follows: Visit the subregions in this region in an arbitrary order, and for each subregion visited, visit its vertices by subregion-label. During this traversal, they incrementally assign region-labels (starting from 1) for each vertex in the order in which it is first visited; thus, even if a vertex appears in multiple subregions, it has a distinct region-label. Finally, each vertex is assigned a distinct *graph-label* over the entire triangulation. Graph-labels are constructed from region-labels in a way similar to the way in which region-labels are constructed from subregion-labels. Point coordinates are then stored in an array, indexed by graph-label. Given a subregion (or region) and a subregion-label (or region-label) of a vertex, the graph-label of this vertex can be computed in constant time using succinct sparse bit vectors [66] and other data structures; readers with background in succinct data structures can attempt to design an $o(n)$-bit structure achieving

this on their own. With this, a vertex in the point location structure constructed for a region can be identified using its $O(\lg \lg n)$-bit region-label, to guarantee that the succinct index constructed occupies $o(n)$ bits only. Graph-labels are also used as point coordinates in the point location structure constructed for the top-level partition, so that no point coordinates are duplicated.

To further construct a succinct index for a general planar subdivision, they partition each large face into smaller faces and assign identifiers at three different levels to each face. Their main result can then be summarized in the following theorem:

Theorem 4 ([13]). *Given a planar subdivision of n vertices, there exists an $o(n)$-bit succinct geometric index that supports point location in $O(\lg n)$ time. This index can be constructed in $O(n)$ time.*

Three variants of the succinct geometric index for planar triangulations were also designed, to match the query efficiency of data structures with improved query time under various assumptions. The first index supports point location using $\lg n + 2\sqrt{\lg n} + O(\lg^{1/4} n)$ point-line comparisons, which matches the query efficiency of the linear-space structure of Seidel and Adamy [69]. The second addresses the case in which the query distribution is known. In this case, let p_i denote the probability of a query point residing in face p_i, and the entropy of the distribution is $H = \sum_{i=1}^{f}(p_i \lg \frac{1}{p_i})$, where f is the number of faces. They designed a succinct index supporting point location in $O(H+1)$ expected time, which matches the query time of the linear-space structure of Iacono [49]. The third variant assumes that the point coordinates are integers bounded by $U \leq 2^w$, where w is the number of bits in a word, and it supports queries in $O(\min\{\lg n / \lg \lg n, \sqrt{\lg U / \lg \lg U}\})$ time. This matches the query efficiency of the linear-space structure of Chan and Pătraşcu [23][3]. These three succinct geometric indexes can be constructed in linear time.

The succinct geometric index can be further used to design implicit data structures for point location. The main idea is to adopt the standard approach of encoding one bit of information by swapping one pair of points, in order to encode the $o(n)$-bit geometric index in the permuted sequence of point coordinates. This requires several modifications to the succinct index, including labeling schemes. The support for queries becomes slower, as $O(\lg n)$ time is required to decode one word of information. The implicit structure is summarized in the following theorem:

Theorem 5 ([13]). *Given a planar subdivision of n vertices, there exists an implicit data structure that supports point location in $O(\lg^2 n)$ time. This data structure can be constructed in $O(n)$ time.*

[3] The query time of this variant of succinct geometric index was stated as $O(\min\{\lg n / \lg \lg n, \sqrt{\lg U}\})$ in [13], and this was because a preliminary version of the structure in [23] was used to prove the query time. It is trivial to apply the main result of [23] to achieve the query time stated here.

Bose *et al.* [13] also showed how to design succinct geometric indexes and implicit data structures for a related problem called *vertical ray shooting*. In this problem, a set of n disjoint line segments is given, and the query returns the line segment immediately below a given query point. The succinct index and implicit data structure support this query in $O(\lg n)$ and $O(\lg^2 n)$ time, respectively. They can be constructed in $O(n \lg n)$ time.

He *et al.* [48] considered the problem of maintaining a dynamic planar subdivision to support point location. The update operations they consider include

- Inserting a new vertex v by replacing the edge between two existing vertices u_1 and u_2 with two new edges (v, u_1) and (v, u_2);
- Deleting a node of degree 2 by replacing its two incident edges with a single edge connecting its two neighbors if they were not adjacent;
- Inserting an edge between two existing vertices across a face whose boundary contains these two vertices, preserving planarity;
- Deleting an edge between two vertices of degrees greater than 2.

To design a succinct geometric index for this problem, they designed a succinct version of the P-tree proposed by Aleksandrov and Djidjev [4] which maintains the partition of a planar subdivision with constant face size under the same update operations; these operations can be used to transform any connected planar subdivision to any other connected planar subdivision. He *et al.* then applied two-level partitioning on the given subdivision using a succinct P-tree. Combined with linear-space data structures for dynamic point location [26,7], they designed succinct geometric indexes to match the query times of previous results, though the update times are slightly slower:

Theorem 6 ([48]). *Let G be a planar subdivision of n vertices in which faces are of constant size and vertices have coordinates that can be encoded in $M = \Theta(\lg n)$ bits. Under the word RAM model with $\Theta(\lg n)$-bit word size, there exists a data structure that can represent G in $nM + o(n)$ bits to supports, for any positive constant ϵ,*

- *point location in $O(\lg n)$ time and updates in $O(\lg^{3+\epsilon} n)$ amortized time[4];*
- *point location in $O(\lg^2 n)$ time and updates in $O(\lg^{2+\epsilon} n)$ worst-case time.*

4 Nearest Neighbor Search

Given a set, N, of n points in the plane, the two-dimensional nearest neighbor query returns the point that is closest (in terms of Euclidean distance) to a given query point. It is well-known that this problem can be reduced to planar point location: Construct the Voronoi diagram. Then the answer is the point whose Voronoi cell contains the query point. This however cannot be used directly to design implicit structures for nearest neighbor search, as the reduction requires $O(n)$ extra space.

[4] This tradeoff is from the unpublished full version of He *et al.* [48].

To design an implicit data structure for two-dimensional nearest neighbor search, Brönnimann *et al.* [15] applied a separator theorem (such as the t-separator theorem [3][5]) on T to partition the Voronoi diagram into a separator of $O(n/\lg n)$ cells and clusters of $O(\lg^2 n)$ cells each. Thus a point location structure of $O(n)$ bits can be constructed to tell which separator cell or cluster contains the query point. By choosing an appropriate parameter for the separator, this point location structure can be represented using at most $n/2$ bits, which can be encoded using the standard techniques of encoding bits by swapping pairs of points. This yields an implicit data structure supporting nearest neighbor search in $O(\lg^2 n)$ time.

Brönnimann *et al.* [15] showed that a similar approach can be used to construct implicit data structures for a set of n halfspaces in 3-dimensional space in $O(n \lg n)$ time, which answer *ray shooting* and *linear programming* queries in the intersection of halfspaces in $O(\lg^2 n)$ time. These two queries are defined as follows: A ray shooting query determines the first halfspace hit by a query ray. In a linear programming query, linear constraints are represented as halfspaces, and the query returns a point that minimizes the value of a query linear function while satisfying all these constraints. Note that the linear-space structure of Dobkin and Kirkpatrick [31] can answer both queries in $O(\lg n)$ time in $\mathbb{R}^3$.

To further improve the query efficiency for nearest neighbor search, Chan and Chen [21] designed a recursive structure. They organize points in an array recursively in a generalization of the van Emde Boas layout [34]: Apply the separator theorem to partition the Voronoi diagram into a separator consisting of $O(\sqrt{bn})$ cells and $O(b)$ clusters each consisting of $O(n/b)$ cells; the parameter b is to be determined by the recursive formula for the query time. According to this decomposition, they reorder the array of points, so that points corresponding to the separator are stored in the first segment of the array, and points corresponding to each cluster are stored in a subsequent segment. They then apply this strategy recursively to the separator as well as each cluster, reordering points in a separator or cluster in each recursion.

Their query algorithm over this structure is also recursive: First determine the cluster containing the cell whose corresponding point is nearest to the given query point among all the points whose cells are not in the separator. Then, perform this algorithm recursively in both this cluster and the separator, to find two points that are candidates of the nearest neighbor. Between these two points, the one that is closer to the query point is the answer. The challenging part is how to locate this cluster, and they proved geometric properties between the Voronoi diagrams of a point set and a subset of it, and designed additional recursive structures. As with the van Emde Boas tree, the query time of their structure is also determined by a recursive function. With the choice of parameter $b = n^{1/3}$, the main recurrences in the critical cases are of the form $Q(n) = 2Q(n^{2/3}) + O(\lg^c n)$, where c is a constant number depending on the particular

[5] They actually applied the separator theorem of Frederickson [41] which requires $O(n \lg n)$ time. However, the t-separator theorem would work as well, and the advantage is that a t-separator can be computed in $O(n)$ time.

case. Thus the query time can be shown to be $O(\lg^{\log_{3/2} 2} n \lg\lg n) = O(\lg^{1.71} n)$, and their main result can be summarized in the following theorem:

Theorem 7 ([21]). *Given a set of n points in two-dimensional space, there exists an implicit data structure that supports nearest neighbor search in $O(\lg^{1.71} n)$ time. This data structure can be constructed in $O(n \lg n)$ time.*

Chan [19] considered the approximate nearest neighbor search problem in constant-dimensional space. Here the word "approximate" means that for any fixed positive constant ϵ, the distance between the query point q and the point returned as the answer is guaranteed to be within a factor of $1 + \epsilon$ from the minimum distance to q. When point coordinates are integers, they designed a simple strategy of laying out coordinates in an array based on shifting and sorting. Surprisingly, it can be proved that this guarantees an approximation ratio of $1 + \epsilon$. As random choices are made by the preprocessing algorithm, its query time is expected.

Theorem 8 ([19]). *Given a set of n points with integer coordinates in constant-dimensional space, there exists an implicit data structure that supports approximate nearest neighbor search in $O(\lg n)$ expected time. This data structure can be constructed in $O(n \lg n)$ time.*

We finally mention that when polylogarithmic query time is not required, there is an implicit data structure for points in $\mathbb{R}^d$ supporting nearest neighbor search in $O(n^{1-1/\lfloor (d+1)/2 \rfloor}$ polylog $n)$ time. This again uses the implicit halfspace range reporting structure [15] summarized in Section 2.3, via lifting transformation. For ray shooting and linear programming queries in intersections of halfspaces in $\mathbb{R}^d$ where $d \geq 4$, Brönnimann *et al.* [15] designed an implicit structure that can answer these queries in $O(n^{1-1/\lfloor d/2 \rfloor}$ polylog $n)$ time, which is also based on their structure for halfspace range search.

5 Conclusion

In this article, we have surveyed previous results on designing succinct and implicit data structures for geometric query problems. Research in these directions developed new algorithmic approaches for computational geometry, succinct data structures and implicit data structures. As more and more applications process large geometric data structures, we also expect that such research will have great impact in the engineering of modern software systems.

There has been some recent development in the design of solutions to geometric query problems that make use of succinct data structures. Unlike the work surveyed in this article that focus on designing succinct solutions, the key strategy is to use succinct data structures to either achieve improvement upon previous results in terms of running time, or to reduce space usage by non-constant factors. Note that some of Chazelle [25]'s structures already used tricks under word RAM which happen to be useful for succinct data structures as well, though these tricks do not include techniques particularly developed later for succinct

data structures. Among the works that focus on using succinct structures to improve running time, the work of He and Munro [45] on dynamic two-dimensional orthogonal range counting problem is perhaps the most relevant to this survey. In this problem, in addition to supporting queries, the insertion/deletion of a point into/from the given point set is also considered. The structure of He and Munro occupies $O(n)$ words of space, answers queries in $O((\lg n/\lg\lg n)^2)$ time, and performs updates in $O((\lg n/\lg\lg n)^2)$ amortized time. This is currently the most efficient linear-space solution to this problem. Succinct data structures are also extensively used in the design of data structures occupying linear or near-linear space for dynamic range median [46], range majority [33], path queries on weighted trees [47], orthogonal range maxima [36] and adaptive and approximate range counting [24]. The use of succinct data structure techniques is crucial to achieving these results.

We end our article by giving several important open problems in the design of succinct and implicit geometric data structures:

- Can implicit partitions trees be further improved to match the performance of Chan's optimal partition trees [20]?
- Can the implicit data structures that achieve polylogarithmic query times, including the structures for 2D point location and nearest neighbor search, be further improved? The $O(\lg^{1.71} n)$ and $O(\lg^2 n)$ query times are slower than logarithmic query times of linear-space data structures for the same problems.
- The implicit geometric data structures that we have surveyed are all designed for static data sets, and the only dynamic succinct geometric structure is the structure of He *et al.* [48] for point location. Brönnimann *et al.* [15] considered semi-dynamization of their implicit geometric data structures which only allows the insertion of points without supporting deletions. Thus designing fully dynamic versions of most of the succinct and implicit structures surveyed here remains open. Many dynamic succinct and implicit data structures have been already designed for bit vectors [65], strings [43,62], trees [67], graphs [16] and partial search [60,39], and thus we expect that progress can be made regarding this open problem.
- Even though a number of data structures have been presented here, there are many other geometric query problems that do not have succinct or implicit solutions. In fact, for each geometric query problem that has a linear-space solution, we can ask the following questions: Can we construct a succinct or implicit solution to this query problem? If the answer is negative, how to give a related lower bound proof?

References

1. Afshani, P., Arge, L., Larsen, K.D.: Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In: Proceedings of the 26th Annual ACM Symposium on Computational Geometry, pp. 240–246 (2010)

2. Afshani, P., Arge, L., Larsen, K.G.: Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In: Proceedings of the 28th Annual ACM Symposium on Computational Geometry, pp. 323–332 (2012)
3. Aleksandrov, L., Djidjev, H.: Linear algorithms for partitioning embedded graphs of bounded genus. SIAM Journal on Discrete Mathematics 9(1), 129–150 (1996)
4. Aleksandrov, L.G., Djidjev, H.N.: A dynamic algorithm for maintaining graph partitions. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 71–82. Springer, Heidelberg (2000)
5. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: Proceedings of the 41st IEEE Symposium on Foundations of Computer Science, pp. 198–207 (2000)
6. Alt, H., Mehlhorn, K., Munro, J.I.: Partial match retrieval in implicit data structures. Information Processing Letters 19(2), 61–65 (1984)
7. Arge, L., Brodal, G.S., Georgiadis, L.: Improved dynamic planar point location. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 305–314 (2006)
8. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search. Theoretical Computer Science 412(32), 4200–4211 (2011)
9. Barbay, J., Castelli Aleardi, L., He, M., Munro, J.I.: Succinct representation of labeled graphs. Algorithmica 62(1-2), 224–257 (2012)
10. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52:1–52:27 (2011)
11. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
12. Bentley, J.L.: Decomposable searching problems. Information Processing Letters 8(5), 244–251 (1979)
13. Bose, P., Chen, E.Y., He, M., Maheshwari, A., Morin, P.: Succinct geometric indexes supporting point location queries. ACM Transactions on Algorithms 8(2), 10:1–10:26 (2012)
14. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
15. Brönnimann, H., Chan, T.M., Chen, E.Y.: Towards in-place geometric algorithms and data structures. In: Symposium on Computational Geometry, pp. 239–246 (2004)
16. Castelli Aleardi, L., Devillers, O., Schaeffer, G.: Dynamic updates of succinct triangulations. In: Proceedings of the 17th Canadian Conference on Computational Geometry, pp. 134–137 (2005)
17. Castelli Aleardi, L., Devillers, O., Schaeffer, G.: Succinct representation of triangulations with a boundary. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 134–145. Springer, Heidelberg (2005)
18. Castelli Aleardi, L., Devillers, O., Schaeffer, G.: Succinct representations of planar maps. Theoretical Computer Science 408(2-3), 174–187 (2008)
19. Chan, T.M.: A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions (2006) (unpublished manuscript)
20. Chan, T.M.: Optimal partition trees. Discrete & Computational Geometry 47(4), 661–690 (2012)

21. Chan, T.M., Chen, E.Y.: In-place 2-d nearest neighbor search. In: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 904–911 (2008)
22. Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: Proceedings of the 27th ACM Symposium on Computational Geometry, pp. 1–10 (2011)
23. Chan, T.M., Pătraşcu, M.: Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. SIAM Journal on Computing 39(2), 703–729 (2009)
24. Chan, T.M., Wilkinson, B.T.: Adaptive and approximate orthogonal range counting. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 241–251 (2013)
25. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)
26. Cheng, S.W., Janardan, R.: New results on dynamic planar point location. SIAM Journal on Computing 21(5), 972–999 (1992)
27. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 383–391 (1996)
28. Cole, R.: Searching and storing similar lists. Journal of Algorithms 7(2), 202–220 (1986)
29. De, M., Maheshwari, A., Nandy, S.C., Smid, M.H.M.: An in-place min-max priority search tree. Computational Geometry: Theory and Applications 46(3), 310–327 (2013)
30. Denny, M., Sohler, C.: Encoding a triangulation as a permutation of its point set. In: Proceedings of the 9th Canadian Conference on Computational Geometry (1997)
31. Dobkin, D.P., Kirkpatrick, D.G.: Determining the separation of preprocessed polyhedra - a unified approach. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 400–413. Springer, Heidelberg (1990)
32. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM Journal on Computing 15(2), 317–340 (1986)
33. Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 150–159. Springer, Heidelberg (2011)
34. Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: Proceedings of the 16th Annual Symposium on Foundations of Computer Science, pp. 75–84 (1975)
35. Farzan, A., Munro, J.I.: Succinct representations of arbitrary graphs. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 393–404. Springer, Heidelberg (2008)
36. Farzan, A., Munro, J.I., Raman, R.: Succinct indices for range queries with applications to orthogonal range maxima. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 327–338. Springer, Heidelberg (2012)
37. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2), 20:1–20:24 (2007)
38. Fomin, F.V., Kratsch, D., Novelli, J.C.: Approximating minimum cocolorings. Information Processing Letters 84(5), 285–290 (2002)

39. Franceschini, G., Grossi, R.: Optimal worst-case operations for implicit cache-oblivious search trees. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 114–126. Springer, Heidelberg (2003)
40. Frederickson, G.N.: Implicit data structures for the dictionary problem. Journal of the ACM 30(1), 80–94 (1983)
41. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM Journal on Computing 16(6), 1004–1022 (1987)
42. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proceedings of the 16th Annual ACM Symposium on Theory of Computing, pp. 135–143 (1984)
43. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410(43), 4414–4422 (2009)
44. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
45. He, M., Munro, J.I.: Space efficient data structures for dynamic orthogonal range counting. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 500–511. Springer, Heidelberg (2011)
46. He, M., Munro, J.I., Nicholson, P.K.: Dynamic range selection in linear space. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 160–169. Springer, Heidelberg (2011)
47. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 575–586. Springer, Heidelberg (2012)
48. He, M., Nicholson, P.K., Zeh, N.: A space-efficient framework for dynamic point location. In: Chao, K.-M., Hsu, T.-S., Lee, D.-T. (eds.) ISAAC 2012. LNCS, vol. 7676, pp. 548–557. Springer, Heidelberg (2012)
49. Iacono, J.: Expected asymptotically optimal planar point location. Computational Geometry 29(1), 19–22 (2004)
50. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 549–554 (1989)
51. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
52. Kirkpatrick, D.G.: Optimal search in planar subdivisions. SIAM Journal on Computing 12(1), 28–35 (1983)
53. Lipton, R.J., Tarjan, R.E.: Application of a planar separator theorem. In: Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science, pp. 162–170 (1977)
54. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3), 332–347 (2007)
55. Matousek, J.: Efficient partition trees. Discrete & Computational Geometry 8, 315–334 (1992)
56. Matousek, J.: Reporting points in halfspaces. Computational Geometry: Theory and Applications 2, 169–186 (1992)
57. Matousek, J.: Range searching with efficient hiearchical cutting. Discrete & Computational Geometry 10, 157–182 (1993)
58. McCreight, E.M.: Priority search trees. SIAM Journal on Computing 14(2), 257–276 (1985)
59. Munro, J.I.: A multikey search problem. In: Proceedings of the 17th Allerton Conference on Communication, Control and Computing, pp. 241–244 (1979)

60. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. Journal of Computer and System Sciences 33(1), 66–74 (1986)
61. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
62. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 865–876 (2013)
63. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing, pp. 40–46 (2007)
64. Pătraşcu, M.: Succincter. In: Proceedings of 49th IEEE Annual Symposium on Foundations of Computer Science, pp. 305–313 (2008)
65. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)
66. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 43 (2007)
67. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 134–149 (2010)
68. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Communications of the ACM 29(7), 669–679 (1986)
69. Seidel, R., Adamy, U.: On the exact worst case query complexity of planar point location. Journal of Algorithms 37(1), 189–217 (2000)
70. Yu, C.C., Hon, W.K., Wang, B.F.: Improved data structures for the orthogonal range successor problem. Computational Geometry: Theory and Applications 44(3), 148–159 (2011)

In Pursuit of the Dynamic Optimality Conjecture

John Iacono*

Polytechnic Institute of New York University, Brooklyn, New York, USA

Abstract. In 1985, Sleator and Tarjan introduced the splay tree, a self-adjusting binary search tree algorithm. Splay trees were conjectured to perform within a constant factor as any offline rotation-based search tree algorithm on every sufficiently long sequence—any binary search tree algorithm that has this property is said to be dynamically optimal. However, currently neither splay trees nor any other tree algorithm is known to be dynamically optimal. Here we survey the progress that has been made in the almost thirty years since the conjecture was first formulated, and present a binary search tree algorithm that is dynamically optimal if any binary search tree algorithm is dynamically optimal.

1 Introduction

A *binary search tree (BST)* is a classic structure of computer science. A binary search tree stores a totally ordered set of data and supports the operations of insert, delete, and predecessor queries. Here we focus only on predecessor queries which we call *searches*. Since there are no insertions and deletions, we can assume the tree contains the integers from 1 to n.

To execute each search in the BST model, there is a single pointer that starts at the root of the tree, and at unit cost can move the pointer to the left child, right child, parent, or perform a rotation with the parent (we call these *BST unit-cost primitives*). In order to properly execute the search it is required that the result of the search be touched by the pointer in the course of executing the search. This model was formalized in [Wil89].

We consider search sequences X of length m: $X = x_1, x_2, \ldots x_m$. To avoid issues with small sequences and the initial state of the tree, we assume m is sufficiently long (often only $m = \Omega(n)$ is needed) and that the tree is in some canonical initial state. A BST-model algorithm is simply a way of choosing a sequence of BST unit cost primitives to execute each search. A BST-model algorithm is *online* if its choice of BST unit cost primitives to execute search x_i is a function of $x_1, \ldots x_i$. The online BST model is still very permissive, as only BST-model unit cost operations are counted, and unlimited computation could be done to determine these operations. What is normally thought of as a BST is an online BST model algorithm that can be implemented on a BST where

* Research supported by NSF grants CCF-1018370 and 0430849.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 236–250, 2013.

$O(\log n)$ bits of data can be augmented on every node, and where unit cost operations are chosen based on the current search, the contents of the node currently pointed to, including any augmented data, and $O(\log n)$ bits of global state. Such a BST algorithm is called a *real-world BST*, a term coined by [BCFL12]. We let $R_A(X)$ denote the cost in the BST model to execute X using some BST-model algorithm A.

Let $\mathrm{OPT}(X)$ be the fastest runtime of any BST that can execute X; that is $\mathrm{OPT}(X) = \min_A R_A(X)$. Given enough time (i.e. exponential in m), $\mathrm{OPT}(X)$ can be computed exactly, and an offline algorithm A such that $R_A(X) = \mathrm{OPT}(X)$ can be produced.

Splay trees are a BST structure introduced by Sleator and Tarjan [ST85b]. They use a simple restructuring heuristic, to move the searched item to the root. This heuristic has the following effect on nodes other than the one searched: if the node x is at depth d and l of the ancestors of x are on the search path, after the search x will be at depth $d + \frac{l}{2} + O(1)$. The work of [Sub96] explores a class of heuristics that have the same general properties of splay trees. Splay trees have been proven to have a number of amortized bounds on their performance, including such basic facts as $O(\log n)$ amortized runtime per search. However, the focus of this work is on the *dynamic optimality conjecture*:

Conjecture 1 (Dynamic Optimality Conjecture [ST85b]). $R_{\mathrm{splay}}(X) = O(\mathrm{OPT}(X))$

We refer to any BST algorithm A such that $R_A(X) = O(\mathrm{OPT}(X) + f(n))$ for some $f(n)$ as *dynamically optimal*. Rather then focus on splay trees themselves, we focus on whether there are any dynamically optimal BSTs. We present several different formulations of this, from weakest to strongest.

Offline Optimality. It is possible to compute in polynomial time (in, say, the RAM model) an algorithm A such that $R_A(X) = O(\mathrm{OPT}(X))$? As we have noted that computing such an algorithm is possible, given enough time, this question concerns only running time, and is the easiest of the questions presented. We believe that computing $\mathrm{OPT}(X)$ exactly is likely to be NP-complete. NP completeness for this very closely related problem was presented in [DHI+09]: instead of a sequence of single searches to be executed on a BST, a sequence of sets of items to be searched are provided and the algorithm can order the searches in each set in whatever manner is beneficial to it. Computing the exact optimal cost for executing such a sequence of sets of searches was shown to be NP-complete.

Online Optimality. Is there an online BST algorithm A such that $R_A(X) = O(\mathrm{OPT}(X))$? In this statement of the problem, A could do significant computation in order to determine which BST unit-cost operation to perform at every step, subject only to the requirement that it is online. This conjecture represents the claim that there is no asymptotic difference in power between online and offline algorithms in the BST model. Such equivalence in power between

online and offline power is generally not possible in more permissive models, and is typically only found in very strict models such as the optimality of the move-to-front rule for search in a linked list [ST85a]. In more permissive models like the RAM, an offline algorithm could fill an array A such that $A[i] = x_i$ and thus could trivially achieve offline performance that an online algorithm could never match.

Online Real-World Optimality. The end goal of this line of research is to obtain, not just an online BST algorithm A such that $R_A(X) = O(\mathrm{OPT}(X))$, but one where the runtime in the BST model dominates the runtime and which could be reasonably implemented. The real-world BST model gives a formalization of that goal, and data structures such as splay trees meet the definition of a real-world BST.

Our hope is that solving the offline optimality problem is the bottleneck, and that a solution to that could be transformed into an online real-world algorithm.

We begin this survey by reviewing a geometric view of the problem. We then summarize the results on the lower and upper bounds for $\mathrm{OPT}(X)$. Finally we present a conditional result whereby a concrete online algorithm is presented which is dynamically optimal if any online algorithm is dynamically optimal. Throughout the presentation we try to identify possible avenues for improvement in as well as the challenges of each of the approaches presented.

2 Geometric View

We now present a geometric view of a binary search tree algorithm A, first introduced in [DHI+09]. This geometric view was created in the hope that reasoning with this view would be significantly simpler than reasoning with rotation-based trees.

Suppose you have a BST algorithm executing some search sequence X. Let the set of points $G_X = \{(x_i, i)\}$ be the *geometric view* of this sequence. Consider the execution of a BST algorithm A on this sequence. Let $t_A(i)$ be all the nodes touched by the pointer in executing x_i. Then let $G_X^A = \{(j, i) | j \in t_A(i)\}$; that is, (i, j) is in G_X^A if algorithm A when executing x_i touches j. Observe that $G_X \subseteq G_X^A$ for any algorithm X, since the item to be searched must be touched in the execution of X by any valid algorithm. This thus gives G_X^A as a plot of everything touched in an execution, seemingly stripping away the specific pointer movements and rotations performed. Also, the runtime of A on X is $O(|G_X^A|)$.

Call two points p, q a set of points P *arborally satisfied (AS)* if there is at least one other point in P in or on the orthogonal rectangle they define. Call a set of points P an *arborally satisfied set (ASS)* if all pairs of points $p, q \in P$ that differ in both coordinates are AS. We have shown that all point sets G_X^A are ASS. Moreover, we have shown that given a ASS point set P where $G_X \subseteq P$, there is a BST algorithm A with runtime $O(|P|)$ such that $G_X^A = P$.

Thus, the following two problems are equivalent: (1) find an $O(1)$-competitive offline BST to execute X, and (2) find an $O(1)$-factor approximation of a minimal ASS superset of G_X^A.

One can observe a kind of duality between the time a key is searched and its value. Suppose X is a permutation of $1..n$. Then the transpose of G_X is also a permutation X and represents some sequence X^T where if $x_i = j$ in X then $x^T_j = i$ in X^T. As the horizontal and vertical coordinates are treated symmetrically in the definition of ASS, it follows that $\mathrm{OPT}(X) = \mathrm{OPT}(X^T)$.

Note that this statement of the geometric view is offline. An algorithm for computing an ASS superset P of G_X is said to be online if the points of P are only a function of the points of G_X with the same or smaller y-coordinate. Note that an online BST algorithm yields an online ASS superset algorithm directly from the preceding definitions. However, the conversion in the other direction does not preserve online-ness. The method used by Fox [Fox11] to turn a particular online ASS superset algorithm into an online BST algorithm could probably be adapted to turn any online ASS superset algorithm into an online BST algorithm.

While computing $\mathrm{OPT}(X)$ offline seems like a clean geometric optimization problem, a solution has remained elusive. The main stumbling block is that it is fairly easy to come up with a minimal superset P of G_X such that all pairs of points in G_X are AS with respect to P; the problem is that the points in $P \setminus G_X$ must all also be pairwise AS for the set P to be ASS.

3 Lower Bounds

One defining feature of this problem is the existence of non-trivial lower bounds on $\mathrm{OPT}(X)$ for particular fixed X, as opposed to lower bounds where X is drawn from a distribution. There are several known bounds, we present each of them separately.

Independent Rectangle Bound [DHI+09]. Given a set of rectangles R, each of which is defined by two points in P, we say R is an independent set of rectangles if no corner of one rectangle lies properly inside of another. Define $\mathrm{IRB}(P)$ to be the size of the maximum set of independent rectangles possible with respect to point set P. It has been shown that $\mathrm{OPT}(X) = \Omega(\mathrm{IRB}(P_X))$.

Computing a value which is $\Theta(\mathrm{IRB}(G_X))$ can be done with a simple sweep algorithm. We call an unsatisfied (i.e. not AS) rectangle defined by two points a +-type rectangle if its upper point is to the right of its lower point. Let $\mathrm{IRB}^+(G_X)$ to be the point set obtained from P by repeatedly looking at the lowest + type unsatisfied rectangle (with lowest upper coordinate), and adding a point to the set in the upper-right corner of this rectangle. This process is repeated until no more unsatisfied +-type rectangles remain. $\mathrm{IRB}^-(G_X)$ is defined in a symmetric way. See Figure 4 for an illustration of the result of this process. We have shown that $\mathrm{IRB}(P) = \Theta(|\mathrm{IRB}^-(G_X)| + |\mathrm{IRB}^+(G_X)|)$.

The independent rectangle bound is the best known lower bound on $\mathrm{OPT}(X)$, but there are two other bounds that predate it due to Wilber, which are interesting in their own right. They were both introduced at the same time in [Wil89] using the language of BSTs and we briefly state them here in the language of the geometric representation.

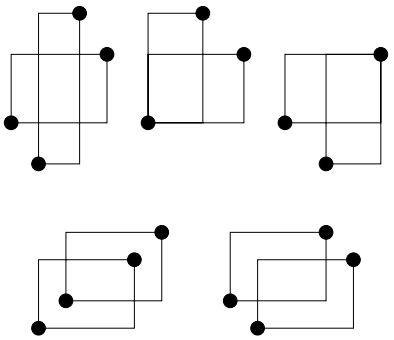


Fig. 1. Here we consider all combinatorial cases of overlapping +-rectangles are independent, and when they are not. The top three pairs of rectangles are independent; the bottom two are not.

Alternation Bound. The alternation bound can be computed using a the geometric view G_X as follows: pick a vertical line ℓ, and sweep in order of y-coordinate through the points of X, counting the number of times the points of G_X in this sweep alternate between the left and right side of the line ℓ. Split the point set G_X into two sets using the line and repeat the process recursively on each side. See Figure 2 for an illustration of this process. The lower bound is the total number of alternations in all levels of the recursion. In computing this bound, there is freedom to choose the vertical line at each step; classically the line at each step is chosen to go though the median x-coordinate, and we will call this lower bound $\mathrm{ALT}(X)$. While $\mathrm{OPT}(X) = \Omega(\mathrm{ALT}(X))$ it can be shown that it is not always tight; for all n there is a sequence X_n of size n such that $\mathrm{ALT}(X_n) = \Theta(\mathrm{OPT}(X_n) \log\log n)$. Such a sequence can be created randomly by picking $O(\log n)$ nodes that are to the left of the dividing line in all levels of the recursion but one and randomly searching only them. Each random search must take time $O(\log\log n)$ in expectation but will only contribute $O(1)$ to the lower bound calculation. There are several possible avenues for improving this bound: one would be to figure out how to choose the lines best (and [Har06] shows that you don't have to restrict the lines to vertical ones), or perhaps change them dynamically. Another avenue for improvement would be to somehow do another type of recursion that would directly reduce the gap exhibited above, possibly reducing it $O(\log\log\log n)$ or $O(\log^* n)$. The main interest in this lower bound is that is it the only bound that has been turned into an algorithm (see Tango trees below); thus improvements on this bound have a reasonable chance of being able to create a better algorithm than what is known.

Funnel Bound. Given a point $(x_i, i) \in G_X$, the *funnel* of x_i, $f(x_i)$ is the set of points below x_i that form unsatisfied rectangles with (x_i, i). For each funnel, look at the points in the funnel sorted by y coordinate, and count the number of alternations from the left to the right of x_i that occur; this is the amortized

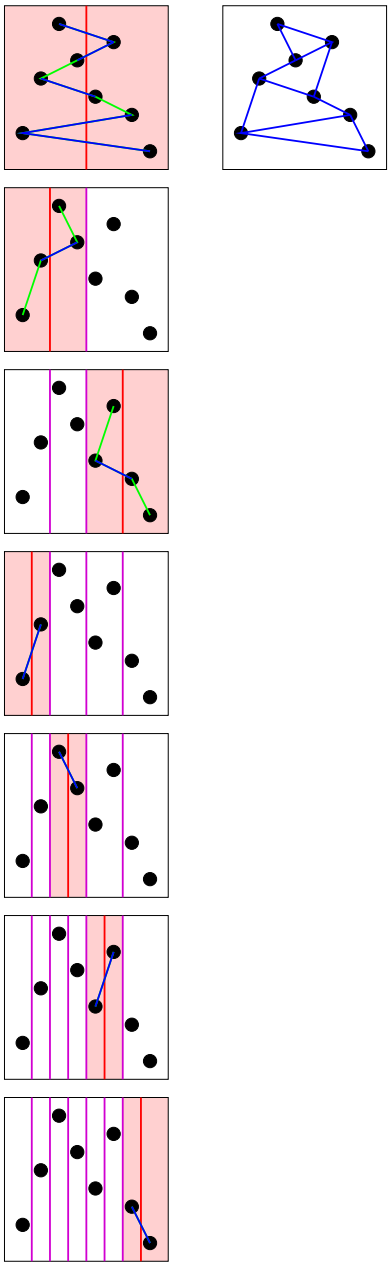


Fig. 2. The interleave bound. As each line is introduced we restrict out view to the cell bounded by previous lines in and containing the current one. In this cell, we connect the points, top to bottom, and a connection that crosses the current line is colored blue and contributes one unit to the lower bound. Green connections are do not cross the current dividing line and do not contribute to the lower bound. As all blue lines generated are distinct, we can visualize the entire bound using the right diagram.

lower bound for x_i; computing and summing this value for all x_i in X gives the lower bound. A different, tree-based view of this bound is as follows: execute X on a binary search tree, and perform a series of single rotations to bring x_i to the root; this BST algorithm was first proposed by Allen and Munro in [AM78]. The amortized lower bound for x_i is the number of turns on the path from the root to x_i. It is worth noting that this will maintain at each step a treap where the heap value of each x is the last i such that $x_i = x$, or equivalently the working set number of the item. This tree view gives an immediate idea for an algorithm—since only the turns in the tree contribute to the lower bound, is it possible to create a method to maintain a representation of the treap so that an item can be searched in a time proportional to the number of turns in the tree? The main obstacle to this approach is that there could be a linear number of items that are one turn away, thus some kind of amortization would be needed to show that that situation could not happen in each search.

Relationship among These Bounds. All of these three bounds are lower bounds, that is all of them compute values which are $O(\mathrm{OPT}(X))$. No relationship is known among the alternation bound and the funnel bound. However, the alternation bound and the funnel bound have been shown to correspond to sets of independent rectangles, and thus they are asymptotically implied by the independent rectangle bound [DHI$^+$09]. We have mentioned that the alternation bound is not known to be tight. However, while the independent rectangle bound asymptotically implies the funnel bound, the converse is unknown, and we conjecture that they are equivalent.

Improving the Bounds. In proving the independent rectangle bound, it was shown the need to put a point in each independent rectangle to satisfy the empty rectangles in the original set. Now, adding these points could cause new unsatisfied rectangles, including those that are defined entirely by added points. We call these problems *secondary effects* and it is easy to show that they occur. The question is, are these secondary effects asymptotically significant or not? If one believes that the independent rectangle bound is tight, they could try to show that if one were to put points to satisfy the rectangles in phases, where the points in each phase satisfy the unsatisfied rectangles in the previous phase, the number of problems would form some kind of exponential progression. On the other hand, to show the bound is not tight one would need an example where these secondary effects dominate.

4 Upper Bounds

In this section we survey the various progress towards the dynamic optimality conjecture by producing actual BSTs.

4.1 Concrete Bounds

One approach has been to come up with concrete closed-form bounds that express the runtime of a particular BST data structure. These bounds initially

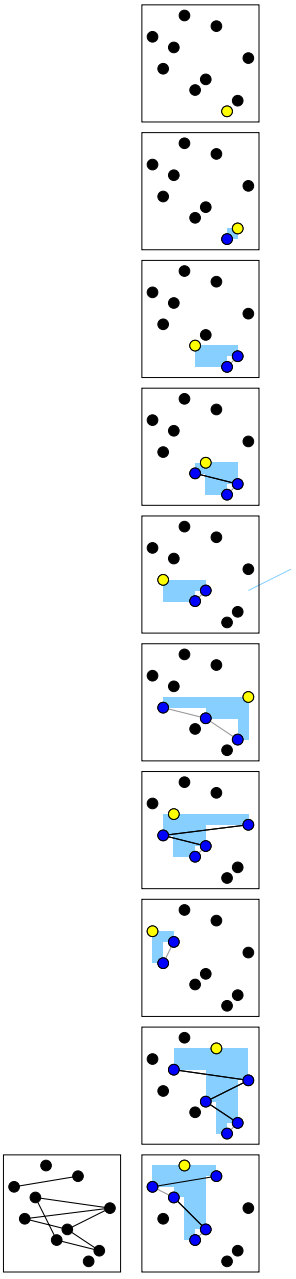


Fig. 3. The funnel bound. For each yellow node, the blue nodes are the nodes in its funnel, and the lines correspond to pairs in the funnel which cross the yellow node and yield one unit of lower bound.

began with the analysis of splay trees [ST85b], with the working set bound, which says a search is fast if it was searched recently, and the dynamic finger bound [Col00, CMSS00] which says that a search is fast if it is close in key value to the previous search. In [BCDI07], we proposed combining these bounds into one which bounds a search as being fast if it is close in key value to some search that has been searched frequently; a BST-model algorithm with this bound was claimed in [Der09] but may be buggy [Sle11]. These bounds all can easily be seen to not be tight, that is there are classes of sequences X such that they are $\omega(\mathrm{OPT}(X))$. Can refining bounds of this type have any hope of a closed form solution that is an asymptotically tight expression of $\mathrm{OPT}(X)$? For example, in the closely related problem of the runtime of the best static tree where each search beings where the previous one ended, a closed-form expression for the asymptotic runtime was obtained [BDIL13]. However, for optimal BST's with rotations, this approach seems difficult as it is easy to come up with search sequences which can be executed quickly on a BST but which all known concrete upper bounds are not tight and which seem to defy a simple formulaic characterization.

4.2 Tango Trees [DHIP07]

In the language of competitive analysis, the problem of finding a dynamically optimal binary search tree is to find one which is $O(1)$-competitive with the best offline tree. Any tree with $O(\log n)$ search time is trivially $O(\log n)$ competitive. Tango trees are a data structure that was created to have a non-trivial approximation factor of $O(\log \log n)$. Specifically, they are created to be within a $O(\log \log n)$ factor of the alternation lower bound.

We present another view of computing the alternation bound, one which leads easily to the central idea of the Tango tree construction. This computation is presented algorithmically. To compute the alternation bound, envision a *reference tree* which is a balanced binary containing $[1..n]$. Each nonleaf node in the tree has one of its children designed as the preferred child—the preferred child is designated based on which subtree of that node has had the most recent search. Now to compute the bound, go though the sequence X and execute each search on the reference tree. The process of executing a search involves starting at the root and following children, which are either preferred or not; at the end of the search the nodes where the search did not follow the preferred child are updated to reflect that the preferred child has changed and is now aligned with the search path; the sum of these preferred child changes is equivalent to the alternation lower bound. Given a node, call the preferred path the path in the reference tree obtained by following preferred children starting from that node until a leaf is reached. Due to the balanced nature of the reference tree, the size of any preferred path is $O(\log n)$. Given this setup, the idea behind the Tango tree is to store each preferred path in a balanced binary search tree of height $O(\log \log n)$. Thus a search in the reference tree that involved following $\log n$ nodes among k preferred paths (and this has a lower bound of $O(k)$) can be executed in time $O((k+1) \log \log n)$ time. Details of the Tango tree involve arranging the BST's

created from each preferred path into one BST, and using split and merge operations to maintain the changing of the preferred paths. However, this method is limited by the fact that the alternation bound is sometimes tight and sometimes off by a $\Theta(\log \log n)$ factor. Thus, no improvement in the competitive ratio is possible while still using the alternation bound as-is as the basis of the competitive ratio. Note that [DS09] improved Tango trees to have some additional desirable properties, such as $O(\log n)$ worst-case time per search, as opposed to $O(\log n \log \log n)$ in their original presentation.

4.3 Greedy

If one were to come up with an idea for candidates for the best possible offline method there is one greedy method that stands out. Search for the current item. Then for (asymptotically) free one can rearrange the nodes on the search path into any tree. The heuristic that makes the most sense would be to place the node to be searched next at the root, or if the node to be searched next is not on the search path, place the subtree that contains it as close to the root at possible. Then, recurse on the the remaining nodes on the search path. This method was first proposed by Lucas [Luc88].

In the geometric view, there is a natural greedy method to find an ASS superset of G_X: from bottom to top, add points on every row so that the point set is satisfied from that row down. See Figure 4 for an illustration of this process. It turns out that by applying the geometric-to-BST conversion to this method, Lucas's greedy tree method is obtained; thus the two greedy methods are in fact identical.

While this method seems intuitively to be a good idea, basic facts like $O(\log n)$ amortized time per search were not known until the work of Fox [Fox11], who also showed that there is an equivalent online BST to this method. Showing that this method which greedily looks into the future has an equivalent online method provides some support for the belief that the best online and offline BST algorithms have asymptotically the same runtime.

In the geometric view, recall that the independent rectangle lower bound can be computed by sweeping twice though R_X and satisfying the $+$ and $-$ unsatisfied rectangles separately. The greedy method is a single sweep though R_X satisfying both the $+$ and the $-$ rectangles at the same time (again, see Figure 4). Proving that the point sets obtained though these two methods are within a constant factor of each other would be enough to show the greedy method is a dynamically optimal binary search tree. We have spent much time coding and looking for ways to prove such a correspondence to no avail.

4.4 Combining Trees

In [DILÖ13], we have shown that given any constant number of online BST algorithms (subject to certain technical restrictions described in the paper), there is an online BST algorithm that performs asymptotically as well on any sequence as the best input BST algorithm. If the output algorithm did not have

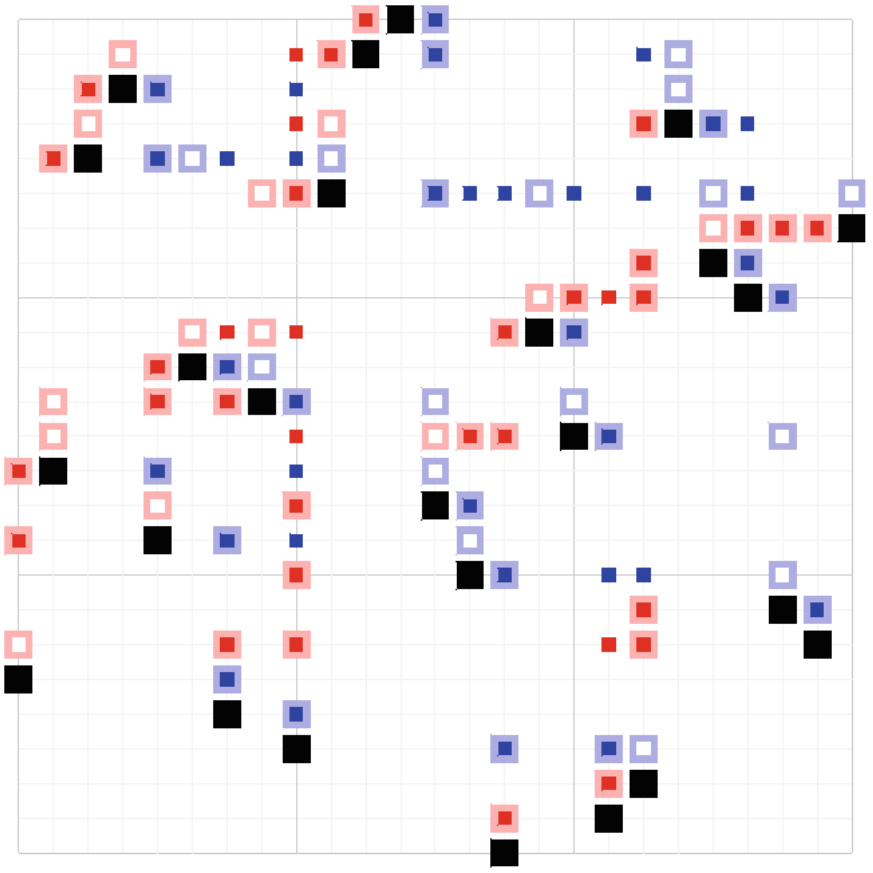


Fig. 4. The geometric view of binary search trees. A black dot at location (x, y) represents that we wish to search for key value x at time y; it is set set G_X. The black dots, combined with the solid blue and red represent an execution of Lucas's greedy future algorithm to execute this search sequence; a dot at (x, y) represents the greedy algorithm touching item with key x at time y. The solid dots are *satisfied*, that is for every two dots not in the same row or column, you can find a third one on the rectangle they define. Color simply represents being to the left or to the right of black. The □-shaped markers are a visual representation of a the incremental construction of the independent rectangle lower bound for the minimal satisfied superset of the black points. Showing Lucas's method is dynamically optimal is equivalent to showing the solid and □'s are always within a constant factor of each other for any such diagram.

to be in the BST model this would be trivial as the input algorithms could just be run in parallel; however the BST-model restriction makes this non-trivial. It is open whether or not it is possible to combine a superconstant number of BST algorithms. This would be of interest as a dynamically optimal BST could be viewed as combining *all* algorithms and taking the minimum. As the number of BST algorithms can be limited to be a function of n (see next section), this opens the possibility of having an algorithm with a runtime of $O(\mathrm{OPT}(X) + f(n))$, for some possibly huge function n.

4.5 Search Optimality

In [BCK03], the notion of *search optimality* was presented. The *search cost* of a search BST algorithm is simply the depth of the node to be searched. Any rotations or pointer movements off the search path are free; in this way the BST can be arbitrarily reconfigured between searches at zero cost. It was shown that there is a BST model algorithm for which the search cost is $O(\mathrm{OPT}(X))$. The general method was to use a machine learning approach to finding the best tree after each search based on the searches performed so far. If one were to try to adapt this method to the standard online BST model cost function, a reasonable starting point would be to try to determine if there is any cohesion of the trees produced by the method from one search to the next, and to try to figure out if one could use such cohesion to transform one tree to the next in time proportional the the search cost.

5 Online Optimality

In this section we present the only new result of this paper: we present the best possible online BST algorithm for sufficiently long search sequences. In particular, we prove the following:

Theorem 1. *There is an online BST algorithm OnOpt such that if there is an online algorithm A such that $R_A(X) = O(\mathrm{OPT}(X) + f(n))$ for some function $f(n)$, then there is an algorithm OnOpt and a function $g(n)$ such that $R_{OnOpt}(X) = O(\mathrm{OPT}(X) + g(n))$.*

The algorithm is decidedly not in the real-world BST model, and is a relatively straightforward application of known methods from learning theory.

We begin by summarizing a classic result from learning theory (see [AHK12] for a survey of its origins, variants and applications). The setup is that there are a sequence of events $Z = z_1, z_2, \ldots, z_\ell$ which are presented in an online manner—each event is an integer in the range $[1..\rho]$. Before each event is revealed, one of η *experts* numbered $[1..\eta]$ is chosen. After the event is chosen a *penalty* is determined based on an $\eta \times \rho$ table M which assigns penalties to each combination of event and expert; $M[a, z]$ is the penalty if expert a was chosen and event z happened.

Thus, if a single expert a were to be chosen for all events, the total penalty would be $\sum_{k=1}^{\ell} M[a, z_k]$. The main result we will use is that it is possible to pick online an expert before each event such that the total penalty is asymptotically that of the best expert:

Theorem 2 (Weighted Majority Algorithm). *For any $\epsilon > 0$, there is an online choice of expert $E = e_1, e_2, \ldots z_\ell$ such that*

$$\sum_{k=1}^{m} M[e_k, z_k] \leq \frac{\rho \ln \eta}{\epsilon} + (1+\epsilon) \min_a \sum_{k=1}^{m} M[a, z_k]$$

Now we apply this theorem to BSTs. Let $X = x_1, x_2, \ldots x_m$ be a sequence of searches in a BST-model data structure containing the integers $1..n$; for convenience we assume m is a multiple of $f(n)$, and we assume $f(n) \geq n$. We let the events be the $n^{f(n)}$ different search sequences of length $f(n)$; thus $\rho = n^{f(n)}$. We divide the search sequence into *epochs* of size $f(n)$, and denote the ith epoch as z_i. Note that each epoch z_i is an event.

How many BST-model algorithms are there to execute an epoch, assuming at the beginning and end of each epoch the BST is in a canonical state (e.g. a left path)? There are at most $n^{f(n)} 5^{O(nf(n))}$. This is because you can encode an algorithm by encoding the $O(nf(n))$ fundamental operations spent to execute each of the $n^{f(n)}$ possible epochs, and the 5 possible BST unit-cost operations at unit of time executing each epoch. We view the set of online BST epoch algorithms as the experts. Thus $e \leq n^{f(n)} 5^{O(nf(n))}$. This is a gross overestimate as this counts the offline algorithms, and does not cull those which do not properly execute each search. The cost $M[a, z_k]$ is simply the runtime of BST epoch algorithm a on epoch k.

Plugging this into Theorem gives:

Lemma 1. *There is a way to choose an algorithm A_k at each epoch such that:*

$$\sum_{k=1}^{m} M[A_k, x_k] = O\left(n^{f(n)} n f(n) + \min_a \sum_{k=1}^{m} M[a, x_k]\right)$$

Now, recall $\mathrm{OPT}(X)$ is the fastest any offline BST-model algorithm can execute the search sequence X.

Lemma 2. *Given a search sequence X of length m on a set of size n, let z_i be a search sequence of size n which is the ith epoch of S. Then for any $f(n) \geq n$*

$$\mathrm{OPT}(S) = \Theta\left(\sum_{i=1}^{m/f(n)} (\mathrm{OPT}(z_i) + f(n))\right)$$

Proof. Follows directly from the fact that any BST can be converted into any other in $O(n)$ time. Thus you can be forced into a canonical state every n searches and this only changes the optimal time by a constant.

These lemmas give a proof of Theorem 1. Specifically, if there is an unknown online BST algorithm with runtime $O(\mathrm{OPT}(X)+f(n))$, then using the weighted majority algorithm to pick an algorithm to run every $f(n)$ steps yields an online BST algorithm that runs in time $O(\mathrm{OPT}(X)+n^{f(n)}nf(n))$, which is $O(\mathrm{OPT}(X))$ for sufficiently long sequences X.

References

[AHK12] Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theory of Computing 8(1), 121–164 (2012)

[AM78] Allenand, B., Ian Munro, J.: Self-organizing binary search trees. J. ACM 25(4), 526–535 (1978)

[BCDI07] Badoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theor. Comput. Sci. 382(2), 86–96 (2007)

[BCFL12] Bose, P., Collette, S., Fagerberg, R., Langerman, S.: De-amortizing binary search trees. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 121–132. Springer, Heidelberg (2012)

[BCK03] Blum, A., Chawla, S., Kalai, A.: Static optimality and dynamic search-optimality in lists and trees. Algorithmica 36(3), 249–260 (2003)

[BDIL13] Bose, P., Douïeb, K., Iacono, J., Langerman, S.: The power and limitations of static binary search trees with lazy finger. CoRR, abs/1304.6897 (2013)

[CMSS00] Cole, R., Mishra, B., Schmidt, J.P., Siegel, A.: On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. SIAM J. Comput. 30(1), 1–43 (2000)

[Col00] Cole, R.: On the dynamic finger conjecture for splay trees. part ii: The proof. SIAM J. Comput. 30(1), 44–85 (2000)

[Der09] Derryberry, J.: Adaptive Binary Search Trees. PhD thesis, CMU (2009)

[DHI+09] Demaine, E.D., Harmon, D., Iacono, J., Kane, D.M., Patrascu, M.: The geometry of binary search trees. In: Mathieu, C. (ed.) SODA, pp. 496–505. SIAM (2009)

[DHIP07] Demaine, E.D., Harmon, D., Iacono, J., Patrascu, M.: Dynamic optimality - almost. SIAM J. Comput. 37(1), 240–251 (2007)

[DILÖ13] Demaine, E.D., Iacono, J., Langerman, S., Özkan, Ö.: Combining binary search trees. CoRR, abs/1304.7604 (2013)

[DS09] Derryberry, J.C., Sleator, D.D.: Skip-splay: Toward achieving the unified bound in the bst model. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 194–205. Springer, Heidelberg (2009)

[Fox11] Fox, K.: Upper bounds for maximally greedy binary search trees. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 411–422. Springer, Heidelberg (2011)

[Har06] Harmon, D.: New Bounds on Optimal Binary Search Trees. PhD thesis, MIT (2006)

[Luc88] Lucas, J.M.: Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University (1988)

[Sle11] Sleator, D.: Achieving the unified bound in the bst model. In: 5th Bertinoro Workshop on Algorithms and Data Structures. Talk (2011)

[ST85a] Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28(2), 202–208 (1985)

[ST85b] Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)

[Sub96] Subramanian, A.: An explanation of splaying. J. Algorithms 20(3), 512–525 (1996)

[Wil89] Wilber, R.E.: Lower bounds for accessing binary search trees with rotations. SIAM J. Comput. 18(1), 56–67 (1989)

A Survey of Algorithms and Models for List Update

Shahin Kamali and Alejandro López-Ortiz

School of Computer Science, University of Waterloo
Waterloo, Ont., N2L 3G1, Canada
{s3kamali,alopez-o}@uwaterloo.ca

Abstract. The list update problem was first studied by McCabe [47] more than 45 years ago under distributional analysis in the context of maintaining a sequential file. In 1985, Sleator and Tarjan [55] introduced the competitive ratio framework for the study of worst case behavior on list update algorithms. Since then, many deterministic and randomized online algorithms have been proposed and studied under this framework. The standard model as originally introduced has a peculiar cost function for the rearrangement of the list after each search operation. To address this, several variants have been introduced, chiefly *the MRM model* (Martínez and Roura, [46]; Munro, [49]), *the paid exchange model*, and *the compression model*. Additionally, the list update problem has been studied under locality of reference assumptions, and several models have been proposed to capture locality of input sequences. This survey gives a brief overview of the main list update algorithms, the main alternative cost models, and the related results for list update with locality of reference. Open problems and directions for future work are included.

1 Introduction

List update is a fundamental problem in the context of online computation. Consider an unsorted list of l items. The input to the algorithm is a sequence of n requests that must be served in an online manner. Let $\mathcal{A}$ be an arbitrary online list update algorithm. To serve a request to an item x, $\mathcal{A}$ linearly searches the list until it finds x. If x is the ith item in the list, $\mathcal{A}$ incurs a cost i to access x. Immediately after this access, $\mathcal{A}$ can move x to any position closer to the front of the list at no extra cost; this is called a *free exchange*. Also, $\mathcal{A}$ can exchange any two consecutive items at a cost of 1; these are called *paid exchanges*. An efficient algorithm can thus use free and paid exchanges to minimize the overall cost of serving a sequence. This model is called the *standard cost model* [7].

The competitive ratio, first introduced formally by Sleator and Tarjan [55], has served as a practical measure for the study and classification of online algorithms in general and list update algorithms in particular. An algorithm is said to be α-competitive (assuming a cost-minimization problem) if the cost of serving any specific request sequence never exceeds α times the optimal cost (up to some additive constant) of an *offline* algorithm which knows the entire request sequence in advance.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 251–266, 2013.

Notwithstanding its wide applicability, competitive analysis has some drawbacks. For certain problems, it gives unrealistically pessimistic performance ratios and fails to distinguish between algorithms that have vastly differing performance in practice. Such anomalies have led to the introduction of many alternatives to competitive analysis of online algorithms. For a comprehensive survey of alternative models, see [31].

As well, a common objection to competitive analysis is that it relies on an optimal offline algorithm, OPT, as a baseline for comparing online algorithms. While this may be convenient, it is rather indirect: One could argue that in comparing two online algorithms $\mathcal{A}$ and $\mathcal{B}$, all the information we should need is the cost incurred by the algorithms on each request sequence. For example, for some problems, OPT is too powerful, causing all online algorithms to seem equally bad. Certain alternative measures allow direct comparison of online algorithms, for example, the *Max-Max Ratio* [18], the *Relative Worst Order Analysis* [23,24], and the *Bijective Analysis* [12,13,14]. These measures have been applied mostly to the paging problem as well as some other online problems.

To the best of our knowledge, relative worst order analysis [33] and bijective analysis [13,14] are the only alternative measures which have already been applied to the list update problem. As mentioned above, both these measures directly compare two online algorithms $\mathcal{A}$ and $\mathcal{B}$. For a given sequence, the relative worst order analysis considers the worst ordering of the sequence for both $\mathcal{A}$ and $\mathcal{B}$ and compares their outcome on these orderings. Then, among all sequences, it considers the one that maximizes the worst case performance. For a precise definition, see [24]. Besides the list update problem, relative worst order analysis has been applied to other online problems including bin packing [23,36], paging [24,25], scheduling [35], and seat reservation [26].

Under bijective analysis, we say $\mathcal{A}$ is no worse than $\mathcal{B}$ if we can define a bijection f on the input sequences of the same length, such that the cost of $\mathcal{A}$ for any sequence σ is not more than that of $\mathcal{B}$ for $f(\sigma)$, where $f(\sigma)$ is the bijected sequence of σ. Bijective analysis proved successful in proving that LRU and MTF are the unique optimal algorithms for respectively paging and list update problems under a locality of reference assumption, while all other measures have failed to show this empirically observed separation.

The list update problem is closely related to the *paging problem*. For paging, there is a small memory (cache) of size k and a large memory of unbounded size. The input is a sequence of page requests. If a requested page a is already in the cache the algorithm pays no cost; otherwise, it pays a cost of one unit to bring a into the cache. On bringing a page to the cache, an algorithm might need to *evict* some pages from the cache to make room for the incoming page. Paging algorithms are usually described by their eviction strategy; for example, Least-Recently-Used (LRU) evicts the page in the cache that was least recently used, and First-In-First-Out (FIFO) evicts the page that was first brought to the cache. Paging can be seen as a type of list update problem with an alternative cost model [53,55]: Each page is equivalent to an item in the list and the access cost for the first k items (k being the size of the cash) is 0, while the cost for

other items is 1. The only difference between the two problems is that when a requested page is not in the cache, a paging algorithm should bring the page to the cache. This is optional for a list update algorithm.

1.1 Outline

In this survey, we give an overview of selected results regarding list update in the standard and alternative cost models. We do not aim to be exhaustive, but rather give highlights of the field. For other surveys on the list update problem, we refer the reader to [2,4,48]. In Section 2, we review a selection of existing results on list update for the standard model. In particular, we consider the classical deterministic and randomized algorithms for the problem. While list update algorithms with a better competitive ratio tend to have better performance in practice, the validity of the cost model has been debated, and a few other models have been proposed for the problem. In Section 3, we review the main results related to these alternative cost models which include the MRM cost model, the paid exchange model, the compression model, and the free exchange model. In Section 4, we discuss the list update problem with locality of reference and review the proposed models which measure this locality.

2 Algorithms

List update algorithms were among the first algorithms studied using competitive analysis. Three well-known deterministic online algorithms are *Move-To-Front* (MTF), *Transpose*, and *Frequency-Count* (FC). MTF moves the requested item to the front of the list; whereas Transpose exchanges the requested item with the item that immediately precedes it. FC maintains an access count for each item ensuring that the list always contains items in non-increasing order of frequency count. *Timestamp* is an efficient list update algorithm introduced by Albers [1]: After accessing an item x, Timestamp inserts x in front of the first item y that is before x in the list and was requested at most once since the last request for x. If there is no such item y, or if this is the first access to x, Timestamp does not reorganize the list. Sleator and Tarjan showed that MTF is 2-competitive, while Transpose and FC do not have constant competitive ratios [55]. Karp and Raghavan proved a lower bound of $2 - 2/(l + 1)$ (reported in [40]), and Irani proved that MTF gives a matching upper bound [40]. It is known that Timestamp is also optimum under competitive analysis [1].

Besides MTF and Timestamp which have optimum competitive ratio, El-Yaniv showed that there are infinitely many algorithms which have optimum ratio [34]. In doing so, he introduced a family of algorithms called Move-to-Recent-Item (MRI): A member of this family has an integer parameter $k \geq 1$, and it inserts an accessed item x just after the last item y in the list which precedes x and is requested at least $k+1$ times since the last request to x. If such item y does not exist, or if this is the first access to x, the algorithm moves x to front of the list. It is known that any member of MRI family of algorithms is 2-competitive [34].

Note that when $k = 0$, MRI is the same as Timestamp, and when k tends to infinity, it is the same as MTF. Schulz proposed another family of algorithm called Sort-By-Rank (SBR) [54], which is parameterized by a real value α where $0 \leq \alpha \leq 1$. The extreme values of α result in MTF (when $\alpha = 0$) and Timestamp (when $\alpha = 1$). It is known that any member of SBR family is also 2-competitive [54].

Classical list update algorithms have also been studied under relative worst order analysis [33]. It is known that MTF, Timestamp, and FC perform identically according to the relative worst order ratio, while Transpose is worse than all these algorithms. Note that these results are almost aligned with the results under competitive analysis.

In terms of the optimum offline algorithm for the list update problem, Manasse et al. presented an offline optimal algorithm which computes the optimal list ordering at any step in time $\Theta(n \times (l!)^2)$ [45], where n is the length of the request sequence. This time complexity was improved to $\Theta(n \times 2^l(l-1)!)$ by Reingold and Westbrook [51]. Hagerup proposed another offline algorithm which runs in time $O(2^l\ l!\ f(l) + l \times n)$, where $f(l) \leq l!\ 3^{l!}$ [38]. Note that the time complexity of these algorithms is incomparable to one another. As pointed out in [38]: "[Hagerup's algorithm] is probably the best algorithm known for $l = 4$, but it loses out rapidly for larger values of l due to the growth of $f(l)$." Another algorithm by Pietzrak is reported to run in time $\Theta(n\ l^3\ l!)$ [48]. It should be mentioned that Ambühl claims that the offline list update problem is NP-hard [8], although a full version of the proof remains to be published.

2.1 Paid Exchanges vs Free Exchanges

All the main existing online algorithms for the list update problem are *economical*, i.e., they only use free exchanges (look at Table 1). In fact, there is only one known non-trivial class of algorithms that uses paid exchanges [37]. This raises the question of how important it is to make use of paid exchanges. In their seminal paper, Sleator and Tarjan claimed that free exchanges are sufficient to achieve an optimal solution [55]. The following example by Reingold and Westbrook shows that this is not the case, and that paid exchanges are sometimes necessary [51].

Consider the initial list configuration (1,2,3) and the request sequence $\langle 3, 2, 2, 3 \rangle$. One can verify that any algorithm relying solely on free exchanges must incur a cost of at least 9. On the other hand, if we apply two paid exchanges (at a cost of 2) before any access is made and refashion the list into (2,3,1), the accesses $\langle 3, 2, 2, 3 \rangle$ can be served on that list at a cost of 2,1,1,2 respectively, for a total access cost of 6 and an overall cost of $2 + 6 = 8$ once we include the cost of the paid accesses.

Considering the above example, one might ask: what is the approximation ratio of the best (offline) list update algorithm which is restricted to only use free exchanges? This question is still open and remains to be answered. We conjecture this ratio to be strictly larger than 1 and smaller than or equal to 4/3.

Table 1. A review of online strategies for list update problem

Algorithm	Competitive Ratio	deterministic	Projective	Economical
MTF	2 [55,40]	✓	✓	✓
Transpose	non-constant [55]	✓	×	✓
Frequency Count	non-constant [55]	✓	✓	✓
Random-MTF (RMTF)	2 [21]	×	✓	✓
Move-Fraction (MF_k)	$2k$ [55]	✓	×	✓
Timestamp	2 [1]	✓	✓	✓
MRI family	2 [34]	✓	✓	✓
SBR family	2 [54]	✓	✓	✓
Timestamp family (randomized)	1.618 [1]	×	✓	✓
SPLIT	1.875 [40,42]	×	×	✓
BIT	1.75 [52]	×	✓	✓
Random Reset (RST)	1.732 [52]	×	✓	✓
COMB	1.60 [6]	×	✓	✓

Reingold and Westbrook showed that there are optimal offline algorithms which only make use of paid exchanges [51]. Note that this is the exact opposite of the situation as claimed by Sleator and Tarjan. It is still not clear how an online algorithm can make good use of paid exchanges. Almost all existing optimal online algorithms only use free exchanges, while there are optimal offline algorithms which only use paid exchanges. This calls into question the validity of the standard model (see Section 3.4).

2.2 Randomization

As mentioned earlier, any deterministic list update algorithm has a competitive ratio of at least $2 - 2/(l + 1)$. In order to go past this lower bound, a few randomized algorithms have been proposed. However, it is important to observe that the competitive ratio of a randomized algorithm is not a worst case measure, and in that sense, it is closer in nature to the *average* competitive ratio of a deterministic algorithm.

Randomized online algorithms are usually compared against an *oblivious* adversary which has no knowledge of the random bits used by the algorithm. To be more precise, an oblivious adversary generates a request sequence before the online algorithm starts serving it, and in doing so, it does not look at the random bits used by the algorithm. Note that the oblivious adversary knows the algorithm code. Two stronger types of adversaries are *adaptive online* and *adaptive offline* adversaries. An adaptive online adversary generates the tth requests of an input sequence by looking at the actions of the algorithm for serving the last $t - 1$ requests. An adaptive offline adversary is even more powerful and knows the random bits used by the algorithm, i.e., before giving the input sequence to the online algorithm, it can observe how the algorithm serves the sequence. The definition of the competitive ratio of an online algorithm is slightly different

when compared with different adversaries, and is based on the expectations over the random choices made by the online algorithm (and adaptive adversaries). For a precise definition of competitiveness for randomized algorithms, we refer the reader to [52].

Ben-David et al. proved that if there is a randomized algorithm which is c-competitive against an adaptive offline adversary, then there exist deterministic algorithms which are also c-competitive [19]. In this sense, randomization does not help to improve the competitive ratio of online algorithms against adaptive offline adversaries. In fact, for the list update problem, the adaptive online and adaptive offline adversaries are equally powerful, and the lower bound $2-2/(l+1)$ for deterministic algorithms extends to adaptive adversaries [50,52]. This implies that there is no randomized algorithm which achieves a competitive ratio better than $2-2/(l+1)$ when compared against adaptive adversaries. So, randomization can only help in obtaining a better competitive ratio when compared against an oblivious adversary. In the following review of randomized algorithms, by the notion of c-competitiveness, we mean c-competitiveness against an oblivious adversary.

Random-MTF (RMTF) is a simple randomized algorithm for the list update problem: after accessing an item, RMTF moves it to the front with probability 0.5. The competitive ratio of RMTF is 2 [21], which is no better than the best deterministic algorithms. The first randomized algorithm that beats the deterministic lower bound was introduced by Irani in 1991 [40]. In this algorithm, called SPLIT, each item x has a pointer to another item which precedes it in the list, and after each access to x, the algorithm moves x to either the front of the list or front of the item that x points to (we omit the details here). This randomized algorithm has a competitive ratio of 1.875 [40,42]. Reingold et al. proposed another randomized algorithm named BIT [52]: Before serving the sequence, the algorithm assigns a bit $b(x)$ to each item x which is randomly set to be 0 or 1. At the time of an access to an element x, the content of $b(x)$ is complemented. Then, if $b(x) = 1$, the algorithm moves x to the front; otherwise (when $b(x) = 0$), it does nothing. Note that BIT uses randomness only in the initialization phase, and after that it runs deterministically; in this sense the algorithm is *barely random*. It is known that BIT has a competitive ratio of 1.75 [52].

BIT is a member of a more generalized family of online algorithms called COUNTER[52]. A COUNTER algorithm has two parameters: an integer s and a fixed subset S of $\{0, 1, ..., s-1\}$. The algorithm keeps a counter modulo s for each item. With each access to an item x, the algorithm decrements the counter of x and moves it to the front of the list if the new value of the counter is a member of S. With good assignments of s and S, a COUNTER algorithm can be better than BIT. For example, with $s = 7, S = \{0, 2, 4\}$, the ratio will be 1.735, which is better than the 1.75 of BIT [52].

It is also known that a random reset policy can improve the ratio even further: The algorithm Random Reset maintains a counter $c(x)$ for each item x in the list. The counter is initially set randomly to be a number i between 0 and s

with probability π_i. When the requested item has a counter larger than 1, the algorithm makes no move and decrements the counter. If the counter is 1, it moves the item to front and resets the item counter to $i < s$ with probability π_i. Unlike COUNTER algorithms, RANDOM RESET algorithms are not barely random. The best values of s and D result in an algorithm with a competitive ratio of $\sqrt{3} \approx 1.732$ [52].

The deterministic Timestamp algorithm described earlier is indeed a special case of a family of randomized algorithms introduced by Albers [1]. A randomized Timestamp(p) algorithm has a parameter p. Upon a request to an item, the algorithm applies the MTF strategy with probability p and (deterministic) Timestamp with probability $1 - p$. The competitive ratio of Timestamp(p) is $\max\{2 - p, 1 + p(2 - p)\}$ which achieves its minimum when $p = (3 - \sqrt{5})/2$; this gives a competitive ratio of 1.618. Albers et al. proposed another hybrid algorithm which randomly chooses between two other algorithms [6]. This algorithm is called COMB. Upon a request to an item, the algorithm applies BIT strategy with probability 0.8 and (deterministic) Timestamp with probability 0.2. COMB has a competitive ratio of 1.6 [6], which is the best competitive ratio among existing randomized online algorithm for the list update problem.

There has been some research for finding lower bounds for competitive ratio of randomized list update algorithms against an oblivious adversary [40,52,56]. The best existing lower bound is 1.5 proven by Teia, assuming the list is sufficiently large [56]. Under the partial cost model, where an algorithm pays $i - 1$ units to access an item in the ith position, Ambühl et al. proved a lower bound of 1.50084 for a randomized online algorithm. Note that there is still a gap between the best upper and lower bounds [10].

Although randomized algorithms achieve better competitive ratios than deterministic algorithms, the validity of such comparisons is under question. As mentioned earlier, the competitive ratio of a randomized algorithm (against an oblivious adversary) is defined by the expectation over the random choices made by the online algorithm. In that sense, the competitive ratio does not capture the worst-case behavior of randomized algorithms, instead, it captures the expected cost over random bits for the worst sequence generated by the adversary. A better comparison would be to compare the worst cost over random bits, which is captured by adaptive adversaries. As mentioned earlier, randomized online algorithms cannot achieve improved ratios against adaptive adversaries. To conclude, the improved competitive ratios of randomized online algorithms are mostly due to the decreased power of adversary rather than enhanced power or smarts of online algorithms. There is empirical evidence supporting this, as in real life sequences (e.g., when there is locality of reference) deterministic algorithms outperform their randomized counterparts [15,13]. Besides randomized competitive analysis, randomized algorithms have also been studied under the relative worst order ratio. It is known that under this framework RMTF and BIT are not comparable [33].

2.3 Projective Property

Most of the existing algorithms for the list update problem satisfy the *projective property*. Intuitively, an algorithm is projective if the relative position of any two items x, y in the list maintained by the algorithm only depends on their relative position in the initial configuration and also the requests to x and y in the input sequence. The algorithms with the projective property are usually studied under the partial cost model, where an algorithm pays $i-1$ units to access an item in the ith position.

In order to achieve an upper bound for the competitive ratio of an algorithm $\mathcal{A}$ with the projective property, it suffices to compare the cost of $\mathcal{A}$ when applied to sequences of two items with the cost of an optimal algorithm OPT_2 for serving those sequences. Fortunately, the nature of OPT_2 is well-understood and there are efficient optimal offline algorithms for lists of size two [51]. This opens the door for deriving upper bounds for the competitive ratio of projective algorithms under the partial cost model, which also extend to the full cost model.

Ambühl et al. showed that COMB is the optimum randomized projective algorithm under competitive analysis [9,11]. Consequently, if one wants to improve on the randomized competitive ratio 1.6 of COMB, they should introduce a new algorithm which is not projective.

3 Alternative Models

The validity of the standard cost model for the list update has been debated, and a few other models have been proposed for this problem. In this section, we review these models and the main relevant results.

3.1 MRM Model

Martínez and Roura [46], and also Munro [49], independently addressed the drawbacks of the standard cost model. The result is a model that we refer to as the MRM model. The standard model is not realistic in some practical settings such as when the list is represented by an array or linked list. Martínez and Roura argued that, in a realistic setting, a complete rearrangement of all items in the list which precede the requested item at position i would require a cost proportional to i, while this has cost proportional to i^2 in the standard cost model. Munro provided the example of accessing the last item of the list of size l and then reversing the entire list. The real cost of this operation in an array or a linear link list should be $O(l)$, while it costs about $l^2/2$ in the standard cost model. As a consequence, their main objection to the standard model is that it prevents online algorithms from using their true power. They instead proposed the MRM model in which the cost of accessing the ith item of the list plus the cost of reorganizing the first i items is linear in i.

Surprisingly, it turns out that the offline optimum benefits substantially more from this realistic adjustment than the online algorithms do. Under the MRM

model, every online algorithm has an amortized cost of $\Theta(l)$ per access for some arbitrary long sequences, while there are optimal algorithms which incur a cost of $\Theta(\lg l)$ on every sequence. Hence, all online list update algorithms have a competitive ratio of $\Omega(l/\lg l)$. Among offline algorithms which have an amortized access cost of $\Theta(\lg l)$ per request, we might mention Order_By_Next_Request (OBNR) proposed by Munro [49]: After accessing an item x at position i, OBNR reorders the elements from position 1 to position p in order of next access (i.e., the items which will be requested earlier appear closer to front). Here p is the first position at or beyond i which is a power of 2, i.e., $p = 2^{\lceil \lg i \rceil}$.

We would like to observe that randomization does not help to improve the competitive ratio under the MRM model, and the same lower bound holds for randomized online algorithms, i.e., there is no randomized algorithm with a constant competitive ratio under the MRM model [46]. One may be tempted to argue that this is proof that the new model makes the offline optimum too powerful and hence this power should be removed; however, this is not correct as in real life online algorithms can rearrange items at the cost indicated. Note that the ineffectiveness of this power for improving the worst case competitive ratio does not preclude the possibility that under certain realistic input distributions (or other similar assumptions on the input) this power might be of use. Martínez and Roura observed this and posed the question [46]: "[An] important open question is whether there exist alternative ways to define competitiveness such that MTF and other good online algorithms for the list update problem would be competitive, even for the [modified] cost model". This question was answered by Angelopoulos et al. who showed that MTF is the unique optimal algorithm under bijective analysis for input sequences which have locality of reference [13,14].

Unlike the standard model, under which the offline problem is NP-hard, Golynski and López-Ortiz introduced an offline algorithm which computes the optimal arrangement in time $O(n^3)$ (n is the length of the input sequence) and serves any input sequence optimally under the MRM model [37]. It remains open to investigate whether an optimal offline algorithm with a better running time exists. Kamali et al. did an empirical study of the performance of the list update algorithms under the MRM model and observed that a *context-based* algorithm, initially applied for compression purposes, outperforms other algorithms [43].

3.2 Paid Exchange Model

Reingold et al. considered another variant of the standard model in which the access cost is similar to the standard model, but the cost for paid exchanges is scaled up by a value $d \geq 1$ [52]. As pointed out in [52], "This is a very natural extension, because there is no a priori reason to assume that the execution time of the program that swaps a pair of adjacent elements is the same as that of the program that does one iteration of the search loop". In the new model, the cost involved in a paid exchange is d, while free exchanges are not allowed. We refer to this model as the *d-paid exchange model*. Reingold et al. suggested a family of randomized COUNTER algorithms for this model [52]. Each algorithm in this family has a parameter s and maintains a modulo s counter for each item.

The counters are initially set independently and uniformly at random. When there is a request to an item x the counter for x is decremented. In case the counter becomes $s-1$, x is moved to front. The competitive ratio of these algorithms improves as d increases. For the best selection of s, the ratio will be 2.75 when $d=1$. This decreases to $(5+\sqrt{17})/4 \approx 2.28$ when d tends to infinity.

On the other hand, for any value of $d \geq 1$, no deterministic algorithm can be better than 3-competitive under the d-paid exchange model [52]. The proof is based on a cruel adversary which requests the last item in the list maintained by the online algorithm $\mathcal{A}$. If the cost paid by $\mathcal{A}$ for paid exchanges is more than half of the cost it pays for the accesses, the adversary takes the optimal static approach. (It sorts items in decreasing order of their frequencies in the sequence and does not move any item.) Otherwise, the adversary maintains a list which is the same list as the one maintained by the algorithm, but in reverse order. Hence, the total access cost that the adversary pays is n, compared to the $n \times l$ that the online algorithm pays (l being the size of the list), while the cost involved in exchanges remain the same in both. A precise analysis gives the lower bound of 3. This lower bound extends to randomized algorithms when compared against adaptive adversaries [52]. Westbrook showed that a deterministic version of the COUNTER family has a competitive ratio of at most $(5+\sqrt{17})/2 \approx 4.56$ (reported in [7]). These algorithms perform similar to randomized COUNTER algorithms, except that the counters are initially set to be 0. Note that there is still a gap between the best upper and lower bounds.

Sleator and Tarjan studied the list update problem under the standard model when no free exchanges are allowed [55]. We refer to this model as the *paid exchange model*. Note that this model is equivalent to the d-paid exchange model with $d=1$. Recall that almost all existing algorithms only make use of free exchanges. Under the paid exchange model, free exchanges can be replaced by a set of paid exchanges (paying an additional cost). For example, MTF pays almost twice for each request, since after accessing an element at index i, the algorithm pays another $i-1$ units to move the item to the front using paid exchanges. In fact, any algorithm with a competitive ratio i under the standard model has a competitive ratio of at most $2i$ under the paid exchange model. This holds because OPT pays the same cost under both the standard and paid exchange models. In particular, MTF has a competitive ratio of 4 under the paid exchange model [55]. So, there is a gap between the lower bound 3 and the upper bound of 4 given by MTF. To close this gap, one might consider the algorithm MTF-Every-Other-Access which moves a requested item to the front of the list on every even request for the item. Note that this is equal to (deterministic) COUNTER algorithm with $s=2$. A detailed analysis shows that this algorithm is 2-competitive under the standard model [21], and 3-competitive under the paid exchange model (we skip the details in this review). Hence, the algorithm is optimal under both models, and the lower bound 3 is tight for the paid exchange model.

3.3 Compression Model

An important application of the list update problem is in data compression. Such an application was first reported by Bentley et al. who suggested that an online list update algorithm can be used as a subroutine for a compression scheme [20]. Consider each character of a text as an item in the list, and the text as the input sequence. A compression algorithm writes an arbitrary initial configuration in the compressed file, as well as the access cost of ALG for serving each character in unary. Hence, the size of the compressed file is equal to the access cost of the list update algorithm. The initial scheme proposed in [20] used MTF as its subroutine. Albers and Mitzenmacher [5] used Timestamp and showed that in some cases it outperforms MTF. Bachrach et al. studied compressions schemes for a large family of list update algorithms which includes MTF and Timestamp [16]. In order to enhance the performance of the compression schemes, the Burrows-Wheeler Transform (BWT) can be applied to the input string to increase the amount of locality [27]. Dorrigiv et al. observed that after applying the BWT, the schemes which use MTF outperform other schemes in most cases [32].

All the above studies adopt the standard cost model for analysis of compressions schemes. More formally, when an item is accessed in the ith position of the list, the value of i is written in unary format on the compressed file. In practice, however, the value of i is written in binary format using $\Theta(\lg i)$ bits. Hence the true "cost" of the access is logarithmic in what the standard model assumes. This was first observed in the literature by Dorrigiv et al. in [32] where they proposed a new model for the list update problem which is more appropriate for compression purposes. We refer to this model as the *compression model*. Under this model, the cost of accessing an item in the ith position is $\Theta(\lg i)$. They observed that there is a meaningful difference between the standard model and compression model: Consider the Move-Fraction (MF) family of list update algorithms proposed by Sleator and Tarjan [55]. An algorithm in this family has a parameter k ($k \geq 2$) and upon a request to an item in the ith position, moves that item $\lceil i/k \rceil - 1$ positions towards the front. While MF_k is known to be $2k$-competitive under the standard model [55], it is not competitive under the compression model [32]. For example, MF_2 is 4-competitive under the standard model, and $\Omega(\lg l)$ competitive under the compression model. A precise analysis of MTF under the compression model shows that it has a competitive ratio of 2 under the compression model, which makes it an optimal algorithm under this model. We skip the details in this review.

A randomized algorithm can also be applied for text compression if the random bits used by the algorithm are included in the compressed file. The number of random bits does not change the size of the file dramatically, specially for barely random algorithms like BIT. So it is worthwhile to study these algorithms under the compression model.

3.4 Free Exchange Model

Recall that all well known existing online algorithms for the list update problem only use free exchanges, while an optimal offline can restrict itself to using paid

exchanges only. This suggests that it is more appropriate to consider a model which does not allow paid exchanges, i.e., after an access to an element an algorithm can only move the item to somewhere closer to the front. This is a natural variant of the standard model which we call the *free exchange model*. Since most of the existing algorithms for list update only use free exchanges, they have the same cost under this model. As mentioned earlier, under the standard model, OPT needs to use paid exchanges. Hence, restricting the model to only allow free exchanges decreases the power of OPT. However, the lower bound of 2 (the lower bound for the competitive ratio of any deterministic online algorithm under the standard model) can be extended to the free exchange model, i.e., under the free exchange model, any deterministic online algorithm has a competitive ratio of at least 2. The proof is straightforward and omitted in this review.

4 Locality of Reference

Another issue in the analysis of online algorithms is that "real-life" sequences usually exhibit *locality of reference*. Informally, this property suggests that the currently requested item is likely to be requested again in the near future. For the paging problem, several models for capturing locality of reference have been proposed [22,57,3,17].

Borodin et al. suggested the notion of *access graph* to model locality for paging [22]. This model assumes that input sequences are consistent with an access graph G, which is known to the online algorithm. Each page is associated to a vertex of G. In a consistent sequence, there is an edge between vertices associated with two consecutive requests. The quality of algorithms is then compared using competitive analysis [22,41,28] or any other measure, e.g., the relative worst order ratio [25]. It is known that an algorithm FAR, which brings the structure of the access graph into account, is a *uniformly competitive* algorithm, i.e., it is within a constant factor of the best attainable competitive ratio for any access graph [41]. In that sense, FAR outperforms both FIFO and LRU, which are k-competitive for some graph families (e.g., cycles of length $k + 1$). Among classical paging algorithms, most results show an advantage for LRU over other algorithms, in particular FIFO [28,25].

Other models defined for capturing locality of reference in paging include that of Karlin et al., which assumes the sequences are generated by a Markov chain [44], and that of Torng, which compares the average length of sequences in a window containing m different pages [57]. The model considered by Torng is related to the concept of *working set* defined by Denning [29,30]. At any time t, and for a time window of size τ, the working set $W(t, \tau)$ is the number of pages accessed by a process in the time window τ. Denning shows that in practical scenarios the size of the working set is a concave function of τ. Inspired by this, Albers et al. defined a model for locality, called *concave analysis*, in which the sequences are consistent with a concave function f so that the maximum number of distinct requests in a window of size τ is at most $f(\tau)$ [3]. Measuring

the performance of online algorithms by the page fault rate (the ratio between the number of faults and all requests), they showed that LRU is the optimal algorithm for sequences which are consistent with a concave function, while this is not the case for FIFO. Later, Angelopoulos et al. showed that under the same model of locality, LRU is the unique optimal algorithm for paging with respect to bijective analysis, when the quality is measured by the number of faults (rather than the fault rate) [12,14].

In practical scenarios, input sequences for the list update problem have a high degree of locality. This is particularly the case when list update is used for compression purposes after BWT (see Section 3.3). Hester and Hirschberg claim [39]: "Move-To-Front performs best when the list has a high degree of locality". Angelopoulos et al. formalized this claim by showing that MTF is the unique optimal solution under bijective analysis for sequences that have locality of reference with respect to concave analysis [13,14].

Albers and Lauer further studied the problem under locality of reference assumption [4]. They defined a new model which is based on the number of *runs* in input sequences: For an input sequence $\sigma_{x,y}$ involving two elements only, a run is a maximal subsequence of requests to the same item. A run is *long* if it contains at least two requests; otherwise, it is *short*. Consider the items in the list to be x and y. Consider a long run R of requests to x. Let R' denote the next long run which comes after R in the sequence. Note that there might be short runs between R and R'. If R' is formed by requests to y, then a *long run change* happens. Also, a single extra long run change happens when the first long run and the first request of the sequence reference the same item. For an arbitrary sequence σ, define the number of runs (resp. long run changes) to be the total number of runs (resp. long run changes) of the projected sequences over all pairs (x, y). Let $r(\sigma)$ and $l(\sigma)$ respectively denote the total number of runs and long run changes in σ. Define $\lambda = \frac{l(\sigma)}{r(\sigma)}$, i.e., λ represents the fraction of long run changes among all the runs. Note that we have $0 \leq \lambda \leq 1$. The larger values for λ imply a higher locality of the sequence, e.g., when all runs are long, we get $\lambda = 1$. Also, note that the length of long runs does not affect the value of λ. Using this notion of locality, the competitive ratio of MTF is at most $\frac{2}{1+\lambda}$, i.e., for sequences with high locality MTF is 1-competitive. The ratio of Timestamp does not improve on request sequences satisfying λ-locality, i.e., it remains 2-competitive. The same holds for algorithm COMB, i.e., it remains 1.6-competitive. However, for the algorithm BIT, the competitive ratio improves to $\min\{1.75, \frac{2+\lambda}{1+\lambda}\}$.

5 Concluding Remarks

The standard model for the list update problem has been found wanting for certain applications and alternative models have been proposed. These models are not yet fully-studied, and some aspects of them remain to be settled. In order to obtain meaningful results for new models, one might require alternative analysis methods which act as substitutes for competitive analysis. These methods can

also be applied for comparing deterministic online algorithms with randomized algorithms. As discussed earlier, such comparison is not valid under competitive analysis. There are also open questions regarding the list update problem with locality of reference, e.g., whether the access graph model can be applied for the list update problem, and in case it can, how one can devise a reasonable online algorithm which brings the graph structure into account. (Recall that such an algorithm exists for paging.)

References

1. Albers, S.: Improved randomized on-line algorithms for the list update problem. SIAM J. Comput. 27, 682–693 (1998)
2. Albers, S.: Online algorithms: a survey. Math. Program. 97(1-2), 3–26 (2003)
3. Albers, S., Favrholdt, L.M., Giel, O.: On paging with locality of reference. J. Comput. Systems Sci. 70(2), 145–175 (2005)
4. Albers, S., Lauer, S.: On list update with locality of reference. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 96–107. Springer, Heidelberg (2008)
5. Albers, S., Mitzenmacher, M.: Average case analyses of list update algorithms, with applications to data compression. Algorithmica 21(3), 312–329 (1998)
6. Albers, S., von Stengel, B., Werchner, R.: A combined BIT and TIMESTAMP algorithm for the list update problem. Inform. Process. Lett. 56, 135–139 (1995)
7. Albers, S., Westbrook, J.: Self-organizing data structures. In: Fiat, A. (ed.) Online Algorithms 1996. LNCS, vol. 1442, pp. 13–51. Springer, Heidelberg (1998)
8. Ambühl, C.: Offline list update is NP-hard. In: Paterson, M. (ed.) ESA 2000. LNCS, vol. 1879, pp. 42–51. Springer, Heidelberg (2000)
9. Ambühl, C., Gärtner, B., von Stengel, B.: Optimal projective algorithms for the list update problem. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 305–316. Springer, Heidelberg (2000)
10. Ambühl, C., Gärtner, B., von Stengel, B.: A new lower bound for the list update problem in the partial cost model. Theoret. Comput. Sci. 268, 3–16 (2001)
11. Ambühl, C., Gärtner, B., von Stengel, B.: Optimal projective algorithms for the list update problem. CoRR, abs/1002.2440 (2010)
12. Angelopoulos, S., Dorrigiv, R., López-Ortiz, A.: On the separation and equivalence of paging strategies. In: Proc. 18th Symp. on Discrete Algorithms (SODA), pp. 229–237 (2007)
13. Angelopoulos, S., Dorrigiv, R., López-Ortiz, A.: List update with locality of reference. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 399–410. Springer, Heidelberg (2008)
14. Angelopoulos, S., Schweitzer, P.: Paging and list update under bijective analysis. In: Proc. 20th Symp. on Discrete Algorithms (SODA), pp. 1136–1145 (2009)
15. Bachrach, R., El-Yaniv, R.: Online list accessing algorithms and their applications: Recent empirical evidence. In: Proc. 8th Symp. on Discrete Algorithms (SODA), pp. 53–62 (1997)
16. Bachrach, R., El-Yaniv, R., Reinstadtler, M.: On the competitive theory and practice of online list accessing algorithms. Algorithmica 32(2), 201–245 (2002)
17. Becchetti, L.: Modeling locality: A probabilistic analysis of LRU and FWF. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 98–109. Springer, Heidelberg (2004)

18. Ben-David, S., Borodin, A.: A new measure for the study of on-line algorithms. Algorithmica 11, 73–91 (1994)
19. Ben-David, S., Borodin, A., Karp, R.M., Tardos, G., Wigderson, A.: On the power of randomization in on-line algorithms. Algorithmica 11, 2–14 (1994)
20. Bentley, J.L., Sleator, D., Tarjan, R.E., Wei, V.K.: A locally adaptive data compression scheme. Commun. ACM 29, 320–330 (1986)
21. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
22. Borodin, A., Irani, S., Raghavan, P., Schieber, B.: Competitive paging with locality of reference. J. Comput. Systems Sci. 50, 244–258 (1995)
23. Boyar, J., Favrholdt, L.M.: The relative worst order ratio for online algorithms. ACM Trans. Algorithms 3(2) (2007)
24. Boyar, J., Favrholdt, L.M., Larsen, K.S.: The relative worst-order ratio applied to paging. J. Comput. Systems Sci. 73(5), 818–843 (2007)
25. Boyar, J., Gupta, S., Larsen, K.S.: Access graphs results for LRU versus FIFO under relative worst order analysis. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 328–339. Springer, Heidelberg (2012)
26. Boyar, J., Medvedev, P.: The relative worst order ratio applied to seat reservation. ACM Trans. Algorithms 4(4) (2008)
27. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, DEC SRC (1994)
28. Chrobak, M., Noga, J.: LRU is better than FIFO. Algorithmica 23(2), 180–185 (1999)
29. Denning, P.J.: The working set model for program behaviour. Commun. ACM 11(5), 323–333 (1968)
30. Denning, P.J.: Working sets past and present. IEEE Trans. Softw. Eng. 6, 64–84 (1980)
31. Dorrigiv, R., López-Ortiz, A.: A survey of performance measures for on-line algorithms. SIGACT News 36, 67–81 (2005)
32. Dorrigiv, R., López-Ortiz, A., Munro, J.I.: An application of self-organizing data structures to compression. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 137–148. Springer, Heidelberg (2009)
33. Ehmsen, M.R., Kohrt, J.S., Larsen, K.S.: List factoring and relative worst order analysis. Algorithmica 66(2), 287–309 (2013)
34. El-Yaniv, R.: There are infinitely many competitive-optimal online list accessing algorithms. Manuscript (1996)
35. Epstein, L., Favrholdt, L.M., Kohrt, J.S.: Separating online scheduling algorithms with the relative worst order ratio. J. Comb. Optim. 12(4), 363–386 (2006)
36. Epstein, L., Favrholdt, L.M., Kohrt, J.S.: Comparing online algorithms for bin packing problems. J. Sched. 15(1), 13–21 (2012)
37. Golynski, A., López-Ortiz, A.: Optimal strategies for the list update problem under the MRM alternative cost model. Inform. Process. Lett. 112(6), 218–222 (2012)
38. Hagerup, T.: Online and offline access to short lists. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 691–702. Springer, Heidelberg (2007)
39. Hester, J.H., Hirschberg, D.S.: Self-organizing linear search. ACM Computing Surveys 17, 295–312 (1985)
40. Irani, S.: Two results on the list update problem. Inform. Process. Lett. 38, 301–306 (1991)
41. Irani, S., Karlin, A.R., Phillips, S.: Strongly competitive algorithms for paging with locality of reference. SIAM J. Comput. 25, 477–497 (1996)

42. Irani, S., Reingold, N., Sleator, D., Westbrook, J.: Randomized competitive algorithms for the list update problem. In: Proc. 2nd Symp. on Discrete Algorithms (SODA), pp. 251–260 (1991)
43. Kamali, S., Ladra, S., López-Ortiz, A., Seco, D.: Context-based algorithms for the list-update problem under alternative cost models. In: Proc. Data Compression Conf., (DCC) (to appear, 2013)
44. Karlin, A., Phillips, S., Raghavan, P.: Markov paging. In: Proc. 33rd Symp. on Foundations of Computer Science (FOCS), pp. 208–217 (1992)
45. Manasse, M., McGeoch, L.A., Sleator, D.: Competitive algorithms for online problems. In: Proc. 20th Symp. on Theory of Computing (STOC), pp. 322–333 (1988)
46. Martínez, C., Roura, S.: On the competitiveness of the move-to-front rule. Theoret. Comput. Sci. 242(1-2), 313–325 (2000)
47. McCabe, J.: On serial files with relocatable records. Oper. Res. 12, 609–618 (1965)
48. Mohanty, R., Narayanaswamy, N.S.: Online algorithms for self-organizing sequential search - a survey. Elect. Coll. on Comput. Complexity 16, 97 (2009)
49. Munro, J.I.: On the competitiveness of linear search. In: Paterson, M. (ed.) ESA 2000. LNCS, vol. 1879, pp. 338–345. Springer, Heidelberg (2000)
50. Reingold, N., Westbrook, J.: Randomized algorithms for the list update problem. Technical Report YALEU/DCS/TR-804, Yale University (1990)
51. Reingold, N., Westbrook, J.: Off-line algorithms for the list update problem. Inform. Process. Lett. 60(2), 75–80 (1996)
52. Reingold, N., Westbrook, J., Sleator, D.D.: Randomized competitive algorithms for the list update problem. Algorithmica 11, 15–32 (1994)
53. Rivest, R.: On self-organizing sequential search heuristics. Commun. ACM 19, 63–67 (1976)
54. Schulz, F.: Two new families of list update algorithms. In: Chwa, K.-Y., Ibarra, O.H. (eds.) ISAAC 1998. LNCS, vol. 1533, pp. 99–108. Springer, Heidelberg (1998)
55. Sleator, D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28, 202–208 (1985)
56. Teia, B.: A lower bound for randomized list update algorithms. Inform. Process. Lett. 47, 5–9 (1993)
57. Torng, E.: A unified analysis of paging and caching. Algorithmica 20(2), 175–200 (1998)

Orthogonal Range Searching for Text Indexing

Moshe Lewenstein*

Bar-Ilan University
moshe@cs.biu.ac.il

Abstract. Text indexing, the problem in which one desires to preprocess a (usually large) text for future (shorter) queries, has been researched ever since the suffix tree was invented in the early 70's. With textual data continuing to increase and with changes in the way it is accessed, new data structures and new algorithmic methods are continuously required. Therefore, text indexing is of utmost importance and is a very active research domain.

Orthogonal range searching, classically associated with the computational geometry community, is one of the tools that has increasingly become important for various text indexing applications. Initially, in the mid 90's there were a couple of results recognizing this connection. In the last few years we have seen an increase in use of this method and are reaching a deeper understanding of the range searching uses for text indexing.

In this monograph we survey some of these results.

1 Introduction

The *text indexing* problem assumes a (usually very large) text that is to be preprocessed in a fashion that will allow efficient future queries of the following type. A query is a (significantly shorter) pattern. One wants to find all text locations that match the pattern in time proportional to the *pattern length and number of occurrences*.

Two classical data structures that are most widespread amongst all the data structures solving the text indexing problem are the *suffix tree* [104] and the *suffix array* [87] (see Section 2 for definitions, time and space usage).

While text indexing for exact matches is a well studied problem, many other text indexing related problems have become of interest as the field of text indexing expands. For example, one may desire to find matches within subranges of the text [86], or to find which documents of a collection contain a searched pattern [90], or one may want our text index compressed [93].

Also, the definition of a match may vary. We may be interested in a *parameterized* match [15,85], a *function* match [5], a *jumbled* match [4,14,25,33,89] etc.

* This paper was written while on Sabbatical in U. of Waterloo. This research was supported by the U. of Waterloo and BSF grant 2010437, a Google Research Award and GIF grant 1147/2011.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 267–302, 2013.

These examples are only a very few of the many different interesting ways that the field of text indexing has expanded.

New problems require more sophisticated ideas, new methods and new data structures. This indeed has happened in the realm of text indexing. New data structures have been created and known data structures from other domains have been incorporated for the use of text indexing data structures all mushrooming into an expanded, cohesive collection of text indexing methods. One of these incorporated methods is that of orthogonal range searching problems.

Orthogonal range searching refers to the preprocessing of a collection of points in d-dimensional space to allow queries on ranges defined by rectangles whose sides are aligned with the coordinate axes (orthogonal).

In the problems we consider here we assume that all input point sets are in rank space, i.e., they have coordinates on the integer grid $[n]^d = \{0, \ldots, n-1\}^d$. The rank-space assumption can easily be made less restrictive, but we do not dwell on this here as the rank-space assumption works well for most of the results here.

The set of problems one typically considers in range searching are queries on the range such as emptiness, reporting (all points in the range), report any (one) point, range minimum/maximum, closest point. In general, some function on the set of points in the range.

We will consider different range searching variants in the upcoming sections and will discuss the time and space complexity of each at the appropriate place. For those interested in further reading of orthogonal range searching problems we suggest starting with [1, 28].

Another set of orthogonal range searching problems is on arrays (not point sets[1]). We will lightly discuss this type of orthogonal range searching, specifically for Range Minimum Queries (RMQ).

In this monograph we take a look at some of the solutions to text indexing problems that have utilized range searching techniques. The reductions chosen are, purposely, quite straightforward with the intention of introducing the simplicity of the use of this method. Also, it took some time for the pattern matching community to adopt this technique into their repertoire. Now more sophisticated reductions are emerging and members of the community have also been contributing to better range searching solutions, reductions for hardness and more.

2 Problem Definitions and Preliminaries

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i, \ldots, j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j. The *suffix* S_i of S is the substring $S[i, \ldots, n]$.

[1] Nevertheless, the problems mentioned here can all be transformed to point sets using orthogonal range searching, if one so desired. This is done by adding a dimension and changing the values to be coordinates on the last dimension.

Suffix Tree. The *suffix tree* [48, 88, 101, 104] of a string S, denoted $\mathrm{ST}(S)$, is a compact trie of all the suffixes of $S\$$ (i.e., S concatenated with a delimiter symbol $\$ \notin \Sigma$, where Σ is the alphabet set, and for all $c \in \Sigma, \$ < c$). Each of its edges is labeled with a substring of S (actually, a representation of it, e.g., the start location and its length). The "compact" property is achieved by contracting nodes having a single child. The children of every node are sorted in the lexicographical order of the substrings on the edges leading to them. Consequently, each leaf of the suffix tree represents a suffix of S, and the leaves are sorted from left to right in the lexicographical order of the suffixes that they represent. $\mathrm{ST}(S)$ requires $O(n)$ space. The suffix tree can be prepared in $O(n + Sort(\Sigma))$, where n is the text size, Σ is the alphabet, and $Sort(Q)$ is the time required to sort the set Q [48]. For the suffix tree one can search an m-length pattern in $O(m + occ)$, where occ is the number of occurrences of the pattern. If the alphabet Σ is large this potentially increases to $O(m \log |\Sigma| + occ)$, as one need to find the correct edge exiting at every node. If randomization is allowed then one can introduce hash functions at the nodes to obtain $O(m + occ)$, even if the alphabet is large, without affecting the original $O(n + Sort(\Sigma))$ construction time.

Suffix Array. The *suffix array* [71, 87] of a string S, denoted $\mathrm{SA}(S)$, is a permutation of the indices $1, \ldots, n$ indicating the lexicographic ordering of the suffixes of S. For example, consider $S = mississippi$. The suffix array of S is $[11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$, that is $S_{11} = "i" < S_8 = "ippi" < S_5 = "issippi" < \ldots < S_3 = "ssissippi"$, where $<$ denotes less-than lexicographically. The construction time of a suffix array is $O(n + Sort(\Sigma))$ [71]. The time to answer an query P of length m on the suffix array is $O(m + \log n + occ)$ [87][2]. The $O(m + \log n)$ is required to find the range of suffixes (see Section 3.1 for details) which have P as a prefix and then since P appears as a prefix of suffix S_i it must appear at location i of the string S. Hence, with a scan of the range we can report all occurrences in additional $O(occ)$ time.

Relations between the Suffix Tree and Suffix Array. Let $S = s_1 s_2 \ldots s_n$ be a string. Let $\mathrm{SA} = \mathrm{SA}(S)$ be its suffix array and $\mathrm{ST} = \mathrm{ST}(S)$ its suffix tree. Consider ST's leaves. As these represent suffixes and they are in lexicographic ordering, ST is actually a tree over SA. In fact, one can even view ST as a search tree over SA.

Say we have a pattern P whose path from the root of ST ends on the edge entering node v in ST (the locus). Let $l(v)$ denote the leftmost leaf in the subtree of v and $r(v)$ denote the rightmost leaf in the subtree of v. Assume that i is the location of SA that corresponds to $l(v)$, i.e. the suffix $S_{\mathrm{SA}[i]}$ is associated with $l(v)$. Likewise assume j corresponds to $r(v)$. Then the range $[i, j]$ contains all the suffixes that begin with P and it is maximal in the sense that no other suffixes begin with P. We call this range the SA-*range of* P.

Consider the previous example $S = mississippi$ with suffix array $[11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$. For a query pattern $P = si$ we have that the SA-

[2] This requires LCP information. Details appear in Section 3.1.

range for P is $[8, 9]$, i.e. P is a common prefix of $\{s_{\mathrm{SA}[8]} \ldots s_n, s_{\mathrm{SA}[9]} \ldots s_n\} = \{sissippi, sippi\}$.

Beforehand, we pointed out that finding the SA-range for a given P takes $O(m + \log n)$ in the suffix array. However, given the relationship between a node in the suffix tree and the SA-range in the suffix array, if we so desire, we can use the suffix tree as a search tree for the suffix array and find the SA-range in $O(m)$ time. For simplification of results, throughout this paper we assume that indeed we find SA-ranges for strings of length m in $O(m)$ time.

Moreover, one can find for $P = p_1, \ldots, p_m$ all nodes in a suffix tree representing $p_i, \ldots, p_m$ for $1 \leq i \leq m$ in $O(m)$ time using suffix links. Hence, one can find *all* SA-ranges for $p_i, \ldots, p_m$ for $1 \leq i \leq m$ in $O(m)$ time.

3 1D Range Minimum Queries

While the rest of this paper contains results for orthogonal range searching in rank space, one cannot disregard a couple of important range searching results that are widely used in text indexing structures. The range searching we refer to is the *Range Minimum Query* (RMQ) problem on an array (not a point set). RMQ is defined as follows.

Let S be a set of linearly ordered elements whose elements can be compared (for $\leq$) in constant time.

d-Dimensional Range Minimum Query (d-RMQ)	
Input:	A d-dimensional array A over S of size $N = n_1 \cdot n_2 \cdot \ldots \cdot n_d$ where n_i is the size of dimension i.
Output:	A data structure over A supporting the following queries.
Query:	Return the minimum element in a range $q = [a_1..b_1] \times [a_2..b_2] \times \ldots \times [a_d..b_d]$ of A.

1-dimensional RMQ plays an important role in text indexing data structures. Hence, we give a bit of detail on results about RMQ data structure construction.

The 1-dimensional RMQ problem has been well studied. Initially, Gabow, Bentley and Tarjan [54] introduced the problem. They reduced the problem to the *Lowest Common Ancestor (LCA)* problem [61] on Cartesian Trees [103]. The *Cartesian Tree* is a binary tree defined on top of an array of n elements from a linear order. The root is the minimum element, say at location i of the array. The left subtree is recursively defined as the Cartesian tree of the sub-array of locations 1 to $i-1$ and the right subtree is defined likewise on the sub-array from $i+1$ to n. It is quite easy to see the connection between the RMQ problem and the Cartesian tree, which is what was utilized in [54], where the LCA problem was solved optimally in $O(n)$ time and $O(n)$ space while supporting $O(1)$ time queries. This, in turn, yielded the result of $O(n)$ preprocessing time and space for the 1D RMQ problem with answers in $O(1)$ time.

Sadakane [97] proposed a position-only solution, i.e. one that return the position of the minimum rather than the minimum itself, of $4n+o(n)$ *bits* space with $O(1)$ query time. Fischer and Heun [53] improved the space to $2n+o(n)$ bits and preprocessed in $O(n)$ time for subsequent $O(1)$ time queries. They also showed that the space must be of size $2n - O(\log n)$. Davoodi, Raman and Rao [42] showed how to achieve the same succinct representation in a different way with $o(n)$ working space, as opposed to the $n + o(n)$ working space in [53]. It turns out that there are two different models, the *encoding* model and the *indexing model*. The model difference was already noted in [44]. For more discussion on the modeling differences see [23]. In the encoding model we preprocess the array A to create a data structure *enc* and queries have to be answered using *enc* only, *without* access to A. In the indexing model, we create an index *idx* and are able to refer to A when answering queries. The result of Fischer and Heun [53] is the encoding model result. For the indexing model Brodal et al. [23] and Fischer and Heun [53], in parallel, showed that an index of size $O(n/g)$ *bits* is possible with query time $O(g)$. Brodal et al. [23] showed that this is an optimal tradeoff in the indexing model.

Range minimum queries on an array have been extended to 2D in [8, 13, 22, 23, 43, 56] and to higher dimension d in [13, 23, 31, 41].

3.1 The LCP Lemma

The Longest Common Prefix (LCP) of two strings plays a very important role in text indexing and other string matching problems. So, define as follows.

Definition 1. *Let x and y be two strings over an alphabet Σ. The* longest common prefix *of x and y, denoted $LCP(x, y)$, is the largest string z that is a prefix of both x and y. The length of $LCP(x, y)$ is denoted $|LCP(x, y)|$.*

The first sophisticated use of the LCP function for string matching was for string matching with errors in a paper by Landau and Vishkin [82]. An interesting and very central result to text indexing structures appears in the following lemma, which is not difficult to verify.

Lemma 1. [87] *Let $S^{[1]}, S^{[2]}, \ldots, S^{[n]}$ be a sequence of n lexicographically ordered strings. Then $|\mathrm{LCP}(S^{[i]}, S^{[j]})| = \min_{i \le h < j} |\mathrm{LCP}(S^{[h]}, S^{[h+1]})|$.*

This allows a data structure over the suffix array of size $O(n)$ that returns the LCP value of any two substrings in $O(1)$ time. This is done by building an RMQ data structure over the array containing the values of the LCP of lexicographically consecutive suffixes and using the lemma.

This result was implicitly[3] used in [87] to reduce the $O(m \log n)$ time for a search of an m-length pattern P in a suffix array indexing a text of length n to

[3] They did not actually use the RMQ data structure. Rather, since they know the path a binary search will follow, they know which interval one needs to (RMQ)query when consulting a given suffix array position (there is only one path towards it in the virtual binary search tree). So they directly store that range LCP value.

$O(m + \log n)$. The idea is as follows. Both find the SA-range for P based on a binary search of the pattern P on the suffixes of the suffix array. The $O(m \log n)$ time follows for a naive binary search because it takes $O(m)$ time to check if P is a prefix of a suffix and the $O(\log n)$ follows from the binary search.

Reducing to $O(m + \log n)$ is done as follows. The binary search is still used. Initially P is compared to the string in the center of the lexicographic ordering. This may take $O(m)$ time. However, at every stage of the binary search we maintain $|\mathrm{LCP}(P, T_i)|$ for the suffix T_i of T with the maximal $|\mathrm{LCP}(P, T_i)|$, over all the suffixes to which P has already been compared. When comparing P to the next suffix, say T_j, in the binary search, first $|\mathrm{LCP}(T_i, T_j)|$ is evaluated (in constant time) if $|\mathrm{LCP}(T_i, T_j)| \neq |\mathrm{LCP}(P, T_i)|$ we immediately know the value of $|\mathrm{LCP}(P, T_j)|$ - give it a moment of thought - and we can compare the character at location $|\mathrm{LCP}(P, T_j)|+1$ of P and T_j and continue the binary search from there. Otherwise, $|\mathrm{LCP}(T_i, T_j)| = |\mathrm{LCP}(P, T_i)|$ in which case we continue the comparison of P and T_j (but only) from the $|\mathrm{LCP}(P, T_j)|+1$-th character. Hence, one can claim, in an amortized sense, that the pattern is scanned only once. So, the search time is $O(m + \log n)$.

The dynamic version of this method is much more involved but has interesting applications, see [9].

3.2 Document Retrieval

The *Document Retrieval* problem is very close to the text indexing problem. Here we are given a collection of documents $D_1, \ldots, D_k$ and desire to preprocess them in order to answer document queries Q. A *document query* asks for the set of documents where Q appears.

The *generalized suffix array* (for generalized suffix *tree*, see [60]) is a suffix array for a collection of texts $T_1, \ldots, T_k$ and can be viewed as the suffix array for $T_1\$_1T_2\$_2 \ldots \$_{k-1}T_k$. However, we may remove, before finalizing the suffix array, all suffixes that start with a delimiter as they contain no interesting information. In order to solve the document retrieval problem one can build a generalized suffix array for $D_1, \ldots, D_k$. The problem is that when one seeks a query Q one will find all the occurrences of Q in all documents, whereas we desire to know only *in which* documents Q appears and are not interested in all match locations.

A really neat trick to solve this problem was proposed by Muthukrishnan [90]. Imagine the generalized suffix array for $D_1, \ldots, D_k$ of size $n = \sum_{1 \leq i \leq k} |D_i|$ and a document retrieval query Q of length m. In $O(m+\log n)$, or even in $O(m)$ time (as discussed in the end of Section 2) it is possible to find the SA-range for Q. Now we'd like to report all documents who have a suffix in this range. So, create a *document array* for the suffix array. The *document array* for $D_1, \ldots, D_k$ will be of length n and will contain at location i the document id d if $\mathrm{SA}[i]$ is a suffix beginning in document d. So, the former problem now becomes the problem of finding the unique id's in the SA-range of the document array.

Muthukrishnan [90] proposed a transformation to the RMQ problem in the following sense. Take the document array DA and generate, yet another, array

which we will call the *predecessor document array*. Let $\psi(i) = j$ if $j < i$, $DA(i) = DA(j)$ and for all $j < k < i, DA(k) \neq DA(i)$. $\psi(i) = -1$ if there is no such j. The predecessor document array has $\psi(i)$ at location i. The following observation now follows.

Lemma 2. *Let $D_1, \ldots, D_k$ be a collection of documents and let* SA *be their generalized suffix array. Let Q be a query and let $[i, j]$ be the* SA*-range of Q. There is a one-one mapping between the documents in range $[i, j]$ in the document array and the values $< i$ in range $[i, j]$ in the predecessor document array.*

Proof. Let $i_1 < i_2 < \ldots < i_r$ be all locations in $[i, j]$ where document id d appears in the document array. Then the i_1-th location of the predecessor document array will be $< i$. However, locations $i_2 < i_3 < \ldots < i_r$ will contain $i_1, i_2, \ldots, i_{r-1}$, all greater than or equal to i, in the predecessor document array. □

Hence, it is natural to consider an extended RMQ problem defined now.

Bounded RMQ	
Input:	An array A.
Output:	A data structure over A supporting the following *bounded RMQ* queries.
Query:	Given a range $[i, j]$ and a number b find all values in the range $[i, j]$ of value $< b$.

The bounded RMQ problem can be solved by recursively applying the known RMQ solution. Find an RMQ on $A[i, j]$, say it is at location r. If it is less than b then reiterate on $A[i, r-1]$ and $A[r+1, j]$. The preprocessing time and space are the same as those of the RMQ problem. The query time is $O(ans)$, where ans is the number of elements smaller than b.

This yields an $O(m+docc)$ solution for the document retrieval problem, where $docc$ is the number of documents in which the query Q appears.

4 Indexing with One Error

The problem of *approximate text indexing*, i.e. the text indexing problem where up to a given number of errors is allowed in a match is a much more difficult problem than text indexing. The problem is formally defined as follows.

Input: Text T of length n over alphabet Σ and an integer k.
Output: A data structure for T supporting *k-error queries*.
Query: A k-error query is a pattern $Q = q_1 q_2 \ldots q_m$ of length m over alphabet Σ for which we desire to find all locations i in T where Q matches with $\leq k$ errors.

We note that there are several definitions of errors. The *edit distance* allows for mismatches, insertions and deletions [84], the *Hamming distance* allows for mismatches only. For text indexing with k errors (for Hamming distance, Edit distance and more) Cole et al. [36] introduced a novel data structure which, for the Hamming distance version, uses space $(n \log^k n)$ (it is preprocessed within an $O(\log \log n)$ factor of the space complexity) and answers queries in $O(\log^k n + m + occ)$. See also [26, 100] for different space/time tradeoffs for the Hamming distance version.

Throughout the rest of this section we focus and discuss the special case of one error. Moreover, we will do so for the mismatch error, but a similar treatment will handle insertions and deletions. The reduction to range queries presented in this section was obtained in parallel by Amir et al. [10] and by Ferragina, Muthukrishnan and de Berg [51]. The goal of [51] was to show geometric data structures that solve certain methods in object oriented programming. They also used their data structure to solve the dictionary matching with one error. In [10] Amir et al. solved dictionary matching with one error and also solved the text indexing with one error. For the sake of simplicity, we will present the result of text indexing with one error from [10], but the reduction is the same for dictionary matching (see definition in [10]).

The algorithm that we will shortly describe combines a bidirectional construction of suffix trees, which had been known before. Specifically, it is similar to the data structure of [21]. However, in [21] a reduction to 2D range searching was not used.

4.1 Bidirectional Use of Suffix Arrays

For simplicity's sake we make the following assumption. *Assume that there are no* **exact** *matches of the pattern in the text.* We will relax this assumption later and show how to handle it in Section 4.3.

The Main Idea: Assume there is a pattern occurrence at text location i with a single mismatch in location $i + j - 1$. This means that $q_1 \dots q_{j-1}$ has an *exact match* at location i and $q_{j+1} \dots q_m$ has an exact match at location $i + j$.

The distance between location i and location $i + j$ is dependent on the mismatch location, and that is somewhat problematic. We therefore choose to "wrap" the pattern around the mismatch. In other words, if we stand exactly at location $i + j - 1$ of the text and look left, we see $q_{j-1} \dots q_1$. If we look right we see $q_{j+1} \dots q_m$. This leads to the following algorithm.

For the data structure supporting 1-mismatch queries construct a suffix array SA_T of text string T and a suffix array SA_{T^R} of the string T^R, where T^R is the reversed text $T^R = t_n \dots t_1$.

In order to reply to the 1-mismatch queries do as follows:

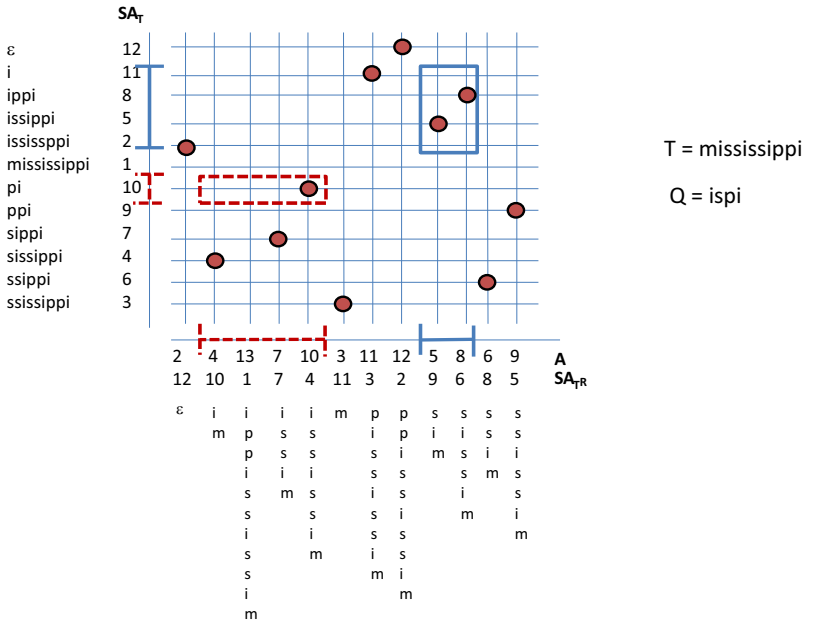


Fig. 1. A range query grid representing T and T^R. The dashed rectangle represents a mismatch at location 2 of Q and the solid rectangle represents a mismatch at location 3 of Q.

Query Reply:

For $l = 1, ..., m$ do

1. Find the maximal SA_T-range $I_l = [i_l, j_l]$ of $q_{l+1} \dots q_m$ in SA_T, if it is non-empty.
2. Find the maximal SA_{T^R}-range $RI_l = [ri_l, rj_l]$ of $q_{l-1} \dots q_1$ in SA_{T^R}, if it is non-empty.
3. If both I_l and RI_l are non-empty, then return the intersection of SA_T and SA_{T^R} on their respective ranges.

Steps 1 and 2 of the query reply can be done for the l's in *overall* linear time (see end of Section 2). Hence, we only need an efficient implementation of Step 3.

4.2 Set Intersection via Range Reporting

In Step 3, given SA-ranges $I_l = [i_l...j_l]$ and $RI_l = [ri_l...rj_l]$ we want to report the points in the intersection of SA_T and SA_{T^R} w.r.t. to the corresponding coordinates of the two ranges. We show that this quite straightforwardly reduces to the 2D range reporting problem.

Range Reporting in 2D (rank space)	
Input:	A point set $P = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subseteq [1, n] \times [1, n]$.
Output:	A data structure representing P that supports the following *range reporting* queries.
Query:	Given a range $R = [a, b] \times [c, d]$ report all points of P contained in R.

Since the arrays (the suffix array for T and the suffix array for T^R) are permutations, every number between 1 and $n + 1$ (we include suffix ϵ) appears precisely once in each array. The coordinates of every number i are (x_i, y_i), where $x_i = \mathrm{SA}_T^{-1}(i)$ and $y_i = SA_{T^R}^{-1}(n - i + 3)$ (the choice of $n - i + 3$ is to align the appropriate reverse suffix with suffix i — explanation: i becomes $n - i + 1$ when reversing the text. Then one needs to move over one to the mismatch location and one more to the next location). We define the point set to be $P = \{(x_2, y_2), \ldots, (x_{n+1}, y_{n+1})\}$ and construct a $2D$ range reporting structure for it (efficiency to be discussed in a moment). It is clear that the range elements intersection corresponds precisely with a $2D$ query $I_l \times RI_l$.

The current best range reporting data structures in 2D are as follows:

1. **Alstrup, Brodal and Rauhe [2]:** a data structure requiring $O(n \log^{\epsilon} n)$ space, for any constant ϵ, that can answer queries in $O(\log \log n + k)$, where k is the number of points reported.
2. **Chan, Larsen and Pătraşcu [28]:** a data structure requiring $O(n \log \log n)$ space that can answer queries in $O(\log \log n(1 + k))$.
3. **Chan, Larsen and Pătraşcu [28]:** a data structure requiring $O(n)$ space that can answer queries in $O(\log^{\epsilon} n(1+k))$. Other succinct results of interest appear in a footnote[4].

Therefore, we have the following.

Theorem 1. *Let $T = t_1 \ldots t_n$ and $Q = q_1 \ldots q_m$. When no exact match exists, indexing with one error can be solved with $O(s(n))$ space such that queries can be answered in $O(qt(n, m, occ))$ time, where:*

s(n) *= the space for a range reporting data structure and*
occ *= the number of occurrences of Q in T with one error, and*
qt(n,m,occ) *= the query time for the same range reporting data structure.*

Proof: Other than the range reporting data structure the space required is $O(n)$. Likewise, Steps 1 and 2 of the query response require total time $O(m)$ for $j = 1, ..., m$. Hence, the space and time are dominated by the range reporting data structure at hand, i.e. space $O(s(n))$ and query time $qt(n, m, occ)$. □

[4] Note that prior succinct solutions show a novel adaptation of the method of Chazelle [29] to Wavelet Trees [58], see [70], [86] and [16]. It is especially worth reading the chapter of "Application as Grids" in [91] for more results along this line.

4.3 Indexing with One Error When Exact Matches Exist

We assumed that the text contained no exact pattern occurrence in the text. In fact, the algorithm would also work for the case where there are exact pattern matches in the text, but its time complexity would suffer. Recall that the main idea of the algorithm was to pivot a pattern position and check, for every text location, whether the pattern to the left and to the right of the pivot were exact matches. However, if the pattern occurs as an exact match in the text then at that occurrence a match is announced for all m pivots. So, this means that every exact occurrence is reported m times. The worst case could end up being as bad as $O(m * occ)$ (for example if the text is a^n and the pattern is a^m then it would be $O(nm)$).

To handle the case of exact occurrences one can use the following idea. Add a third dimension to the range reporting structure representing the character in the text at the mismatch location. The desired intersection is of all suffix labels such that this character is *different from* the symbol at that respective pattern location. This leads to a specific variant of range searching.

3D 5-Sided Range Reporting (rank space)	
Input:	A point set $P = \{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}$ $\subseteq [1, n] \times [1, n] \times [1, n]$.
Output:	A data structure representing P that supports the following *range reporting* queries.
Query:	Given a range $R = [a, b] \times [c, d] \times [e, \infty]$ report all points of P contained in R.

3D 5-Sided Range Reporting can be solved with space $O(n \log^{O(\epsilon)} n)$ and query time $O(occ + \log \log n)$ [28].

Back to our problem. We need to update the preprocessing phase.

Preprocessing: Preprocess for 3-dimensional range queries on the matrix $[1, ..., n] \times [1, ..., n] \times \Sigma$. If Σ is unbounded, then use only the $O(n)$ symbols in T. The new geometric points are (x_i, y_i, z_i), where x_i and y_i will be the same as before and $z_i = t_{i-1}$ will be the text character that needs to mismatch. This will be added in the preprocessing stage.

The only necessary modification is for Step 3 of the query reply which becomes:

3. If I_l and RI_l both exist, then return all the points in $I_l \times R_l \times \Sigma$ for which the z-coordinate is not the respective pattern mismatch symbol.

The above step can be implemented by two 3D 5-sided range queries on the three dimensional range $I_l \times R_l \times [1, a-1]$ and $I_l \times R_l \times [a+1, |\Sigma|]$ where a is the current pattern symbol being examined. We assume that the alphabet symbols are numbered $1, ..., |\Sigma|$.

See Figure 2 depicting the 3D queries.

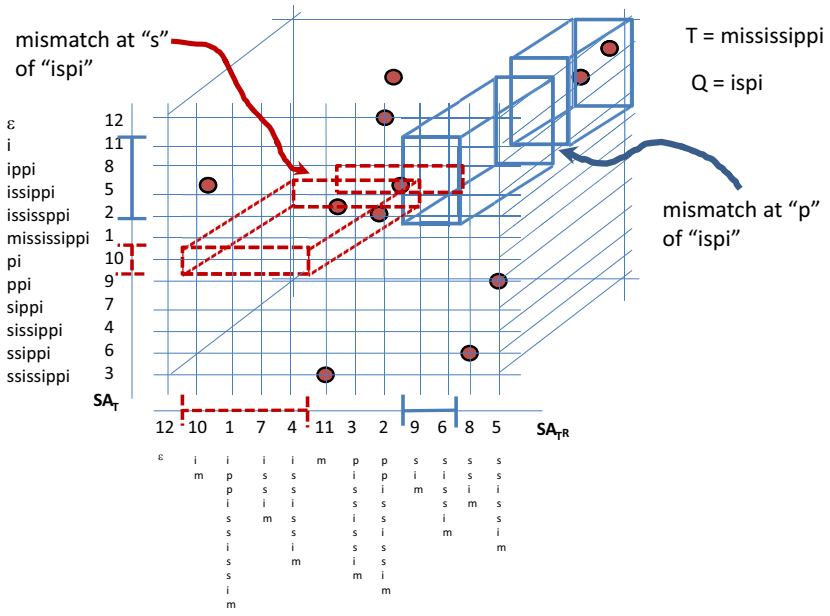


Fig. 2. The 3D 5-sided range queries on a 3D grid representing T and T^R and the appropriate "not to match" character. The dashed boxes represent a mismatch at location 2 of Q and the solid boxes represent a mismatch at location 3 of Q.

Theorem 2. *Let $T = t_1 \dots t_n$ and $Q = q_1 \dots q_m$. Indexing with one error can be solved with $O(n \log^{O(\epsilon)} n)$ space and $O(occ + m \log\log n)$ query time, where occ is the number of occurrences of the pattern in the text with at most one error.*

Proof: As in Theorem 1, the space and query time of the solution are dominated by the range searching structure. Hence, using the results of [28], the space is $O(n \log^{O(\epsilon)} n)$ and the query time is $O(occ + m \log\log n)$. □

4.4 Related Material

For a succinct index for dictionary matching the best result appears in [62].

A *wildcard* character is one that matches all other symbols. When used we denote them with ϕ.

Iliopoulos and Rahman [67] consider the problem of indexing a text T to answer queries of the form $Q = Q_1 \phi^d Q_2$, where d, $|Q_1|$ and $|Q_2|$ are known during the preprocessing. The is known as gap-indexing. Bille and Gørtz [19] consider the same problem. However, they required only d to be known in advance. The solution in [19] uses a reduction to range reporting. The reduction they apply is similar to the one presented in this chapter. In fact, it is easier because the gap's location within the query pattern is known. So, one does not need to check every position of the pattern as one does in the case of mismatch. Hence, the 2D

range reporting data structure is sufficient. One does need to adapt the search for a difference of d instead of 1. This requires setting $y_i = n - i + d + 2$.

5 Compressed Full-Text Indexes

Given that texts may be very large it makes sense to compress them - and there are many methods to do so. On the other hand, these are not constructed to allow text indexing. Doing both at once has been the center of a lot of research activity over the last decade. However, reaching this stage has happened in phases. Initially, pattern matching on compressed texts was considered starting by Amir et al. [6]. By pattern matching on compressed texts we mean that a compressed text, with some predefined compressor, and a pattern are given and the goal is to find the occurrences of the pattern efficiently without decompressing the text. The second phase was text indexing while maintaining a copy of the original text and augmenting it with some sublinear data structure (usually based on a compressor) that would allow text indexing, e.g. [72]. We will call this phase, the *intermediate phase*. Finally, compressed full-text indexing was achieved, that is text indexing without the original text and with a data structure with size depending on the compressibility of the string. See Section 3 for the discussion on the encoding model vs. the indexing model. Compressed full-text indexes of note are that of Ferragina and Manzini [50], the *FM-Index*, that of Grossi and Vitter [59], the *Compressed Suffix Array* and that of Grossi, Gupta and Vitter [58], the *Wavelet Tree*. For an extensive survey on compressed full-text indexes see [93].

In this section we will present two results, that of Kärkkäinen and Ukkonen [72] and that of Claude and Navarro [34]. The latter uses a central idea of the former. However, the former is from the intermediate phase. So, it maintains a text to reference. This certainly makes the indexing easier. The latter is a compressed full-text index. The former uses the more general LZ77 and the latter uses SLPs (Straight Line Programs). Both LZ77 and SLP compressions are defined in this section.

5.1 LZ77 Compressed Indexing

The Lempel-Ziv compression schemes are among the best known and most widely used. In this section (and in Section 7) we will be interested in the variant known as LZ77 [107]. For sake of completeness we describe the LZ77 scheme here.

An Overview of the Lempel-Ziv Algorithm. Given an input string S of length n, the algorithm encodes the string in a greedy manner from left to right. At each step of the algorithm, suppose that we have already encoded $S[1, \ldots, k-1]$ with $i-1$ phrases $\rho_1, \ldots, \rho_{i-1}$ (phrase - to be defined shortly). We search for the location t, such that $1 \leq t \leq k-1$, for which the longest common prefix of $S_k = S[k, \ldots, n]$ and the suffix S_t is maximal. Once we have found the desired location, suppose the aforementioned longest common prefix

is the substring $S[t,\ldots,r]$, a *phrase*, ρ_i, will be added to the output which will include F_i the encoding of the distance to the substring (i.e., the value $k-t$), L_i the length of the substring (i.e., the value $r-t+1$), and the next character $C_i = S[k+(r-t+1)]$. The algorithm continues by encoding $S_{k+(r-t+1)+1} = S[k+(r-t+1)+1,\ldots,n]$. The sequence of phrases is called the string's (LZ77) *parse* and is defined:

$$Z = \rho_1 = (F_1, L_1, C_1), \rho_2 = (F_2, L_2, C_2), \ldots, \rho_{c(n)} = (F_{c(n)}, L_{c(n)}, C_{c(n)})$$

We denote with $u(i) = \sum_{j=1}^{i-1}(L_j+1)+1$ the start location of ρ_i in the string S. Finally, we denote the output of the LZ77 algorithm on the input S as $\mathrm{LZ}(S)$.

Kärkkäinen and Ukkonen Method. Farach and Thorup [47], in their paper on search in LZ77 compressed texts, noted a neat, useful observation. Say T of length n is the text to be compressed and Z is its LZ parse containing $c(n)$ phrases.

Lemma 3. [47] *Let Q be a query pattern and let j be the smallest integer such that $Q = T[j\ldots j+|Q|-1]$. Then $u(i) \in [j, j+|Q|-1]$ for some $1 \le i \le c(n)$.*

In other words, the first appearance of Q in T cannot be contained in a single phrase. Kärkkäinen and Ukkonen [72] utilized this lemma to show how to augment the text with a sublinear text indexing data structure based on LZ77.

The idea is as follows. Say we search for a pattern $Q = q_1,\ldots,q_m$ in the text T which has been compressed by LZ77. The occurrences of Q in T are defined differently for those that intersect with more than one phrase (which must exist if there are any matches according to Lemma 3) and those that are completely contained in a single phrase. The former are called *primary occurrences* and the latter are called *secondary occurrences*. The algorithm proposed in [72] first finds primary occurrences and then uses the primary occurrences to find secondary occurrences. Both steps use orthogonal range searching schemes.

In order to find the primary occurrences they used a bi-directional scheme based on Lemma 3. Take every phrase $\rho_i = t_{u(i)},\ldots,t_{u(i+1)-1}$ and creates its reverse $\rho_i^R = t_{u(i+1)-1},\ldots,t_{u(i)}$. Now, consider a primary occurrence and the first phrase it starts in, say ρ_i. Then there must be a j such that $q_1,\ldots,q_j$ is a suffix of ρ_i and $q_{j+1},\ldots,q_m$ is a prefix of the suffix of T, $t_{u(i+1)},\ldots,t_n$. So, to find the primary occurrences it is sufficient to find k such that:

1. $q_j,\ldots,q_1$ is a prefix of ρ_i^R.
2. $q_{j+1},\ldots,q_m$ is a prefix of the suffix $t_{u(i+1)},\ldots,t_n$ of T.

Let π be the lexicographic sort of $\rho_1^R, \rho_2^R, \ldots, \rho_{c(n)}^R$, i.e. $\rho_{\pi(1)}^R < \rho_{\pi(2)}^R < \ldots < \rho_{\pi(c(n))}^R$. This leads to a range reporting scheme, similar to that of the previous section, of size $c(n)\times c(n)$, where the point set $P = \{(x(i), y(i) \mid x(i) = \mathrm{SA}_T^{-1}(u(i+1)), y(i) = \pi^{-1}(i)\}$.

Therefore, for every $1 \leq j \leq m$ we need to (1) find the range of π for which every $\rho^R_{\pi(i)}$ within has $q_j, \ldots, q_1$ as a prefix and (2) find the range of suffixes that have $q_{j+1}, \ldots, q_m$ as a prefix. Once we do so we revert to range reporting, as in the previous section. However, finding the ranges is not as simple as in the previous section. Recall that we want to maintain the auxiliary data in sublinear space. So, saving a suffix array for the text is not a possibility. Also, the reversed phrases need to be indexed for finding the range of relevant phrases.

To this end a sparse suffix tree was used in [73]. A *sparse suffix tree* is a compressed trie over a subset of the suffixes and is also known as a *Patricia Trie* over this suffix set. As it is a compressed trie it is of size O(subset size), in our case $O(c(n))$. Lately, in [18] it was shown how to construct a sparse suffix array in optimal space and near-optimal time.

The sparse suffix tree can be constructed for the suffixes $T_{u(i+1)}$ and, since we have the original text on hand, we can maintain the compressed suffix tree in $O(c(n))$ words. Navigation on this tree is the same as in a standard suffix tree. For the reversed phrases we can associate each with a prefix of the text. Reversing them gives a collection of suffixes of the reversed text. Now construct a sparse suffix tree for these reversed suffixes and this will allow finding the range in π (with the help of the existing text) as in the previous section. Hence,

Theorem 3. *Let T be a text and Z its LZ77 parse. One can construct a text indexing scheme which maintains T along with a data structure of size $O(|Z|)$ such that for a query Q we can find all its primary occurrences in $O(|Q|(|Q| + \log^\epsilon |Z|) + occ * \log^\epsilon |Z|)$, where occ is the number of primary occurrences.*

Proof. For the query Q, as described above, each pattern position is evaluated for primary occurrences. That is for each of the $|Q|$ pattern positions, we first traverse the auxiliary sparse suffix trees in $O(|Q|)$ time and once the ranges (for that pattern position is found) we perform a range query. The traversal will cost $O(|Q|^2)$ time.

Using the 2D succinct range reporting of Chan et al. [28] (see previous section) we have linear space, i.e. $O(|Z|)$, and $O(\log^\epsilon |Z|(1+k))$ query time where k is the number of points found, which is the same as the number of primary occurrences found.

Hence, over the $|Q|$ pattern positions the range querying will cost $\sum_{i=1}^{|Q|} \log^\epsilon |Z|(1 + occ_i)$ time, where occ_i is the number of primary occurrences when we split the pattern at position i. However, $\sum_{i=1}^{|Q|} \log^\epsilon |Z|(1 + occ_i) = \sum_{i=1}^{|Q|} \log^\epsilon |Z| + \sum_{i=1}^{|Q|} \log^\epsilon |Z| * occ_i = |Q| \log^\epsilon |Z| + \log^\epsilon |Z| occ$. Recalling that the traversal cost $O(|Q|^2)$ time yields the desired. □

We note that the secondary occurrences still need to be found. This is another interesting part of the paper and we refer the interested reader to [72].

See the following papers for more along the following line using Lempel Ziv compressors [12, 50, 81, 96].

5.2 SLP Text Indexing

Claude and Navarro [34] proposed a full-text indexing scheme based on *straight line programs* (*SLP*s). An *SLP* is a grammar based compression for a text T. The grammar produces exactly one word T and the rules are in Chomsky Normal Form, i.e. each rule is $A \to BC$, where A, B, C are variables of the grammar or $A \to a$, where A is a variable and a is a terminal (a character of T).

The text indexing scheme that they propose follows the previous idea [72] of finding primary and secondary occurrences. However, for SLPs things are slightly different. Consider the derivation tree for the text T, that is deriving the full word T by generating from the start symbol S as the root (if $S \to AB$ then A and B will be children of the root S in the derivation tree - from here the derivation tree is applied recursively until the full T is spelled out in the left-to-right order of the leaves). Every occurrence of a pattern Q in the text T has a unique lowest variable V which produces this occurrence, but its children do not. That is if the children of V are V_1 and V_2, i.e. there is a rule $V \to V_1V_2$, then V_1 produces $x, q_1, \ldots, q_j$ (where $x \in \Sigma^*$) and V_2 produces $q_{j+1}, \ldots, q_m, y$ (where $y \in \Sigma^*$) for some j. We say that V splits pattern Q at location j. An occurrence of Q is called a *primary occurrence* if for some V and j V splits this occurrence of Q. All other occurrences are *secondary occurrences*.

The format of the algorithm is to, once again, find the primary occurrences and then to deduce the occurrences of Q in the text therefrom. With the goal of finding the primary occurrences in mind, once again, our grid will be of size $c(n) \times c(n)$, where $c(n)$ is the size of the variable set of the grammar. Each side of the grid will have one coordinate for each variable. The range searching point set is defined per rule, $V \to V_1V_2$. The location (x, y) on the grid corresponding to V_1 (for x) on one side and V_2 (for y) on the other will have a point, V, on the grid. The ordering of the variables on either side of the grid follows from the desire to satisfy the following conditions.

1. $q_j, \ldots, q_1$ is a prefix of V_i^R.
2. $q_{j+1}, \ldots, q_m$ is a prefix of V_l.
3. There is a rule $V \to V_iV_l$.

It is easy to see that the desired ordering, as in the LZ77 scheme, has the phrases in the x-coordinate in reverse lexicographic ordering and has the phrases in the y-coordinate in lexicographic ordering. The challenge here is to actually find the range of variables where $q_j, \ldots, q_1$ is a prefix of V_i^R. This is because it is a full-text index and the text is not accessible any more. Nevertheless, this is doable in the SLP compression scheme using a suffix array type of search and comparing $q_j, \ldots, q_1$ with the variable at hand. This comparison is not trivial. However, the full scheme is out of scope of this survey and we refer the reader to the full paper [34]. The result achieved is as follows:

Theorem 4. *Let T be a text of size N represented by an SLP with n variables and height h. There is a representation using $n(\log N + 3\log n + O(\log|\Sigma| + \log h) + o(logn))$ bits such that Q of length can be found in $O((m(m+h) + hocc)\log n)$ query time.*

An extension of this idea to a general grammar can be found in [35], where the dependency on h was removed from the search time. There is also other work for different compressors. See [55] for one of the latest.

6 Weighted Ancestors

6.1 2-Sided Sorted Range Reporting in 2D

In this section we consider the *2-sided sorted range reporting* problem[5] which is defined now.

2-Sided Sorted Range Reporting in 2D	
Input:	A point set $P = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subseteq [1, U] \times [1, U]$.
Output:	A data structure representing P that supports the following *2-sided sorted range reporting* queries.
Query:	Given a range $R = [-\infty, a] \times [-\infty, b]$ report all points of P contained in R sorted by their y-coordinate (from highest to lowest).

Note that we deviate from the assumption that the points are in rank-space. This is important for the the application of this section. We now show a method to solve the 2-sided sorted range reporting. The idea is as follows.

Consider the dynamic predecessor problem in which we need to support the following operations (a) insertions/deletions of integers ($\in [1, U]$) and (b) predecessor queries. This is a classical problem and is solved with a van-Emde Boas tree [102] or with y-fast tries [105] in $O(n)$ space (n is the current number of integers) and $O(\log \log U)$ time for the operations, where $[1, U]$ is the domain of the elements.

Dietz and Raman [45] asked whether this could be made partially persistent[6] within the same query times. In other words can one create a data structure where insertions and deletions are supported on the current version but predecessor queries can be made on any of the versions (current or previous) of the data structure. Recently, Chan [27] accomplished this by constructing a partially persistent predecessor data structure with space $O(n)$ and operations time $O(\log \log U)$. Chan's result [27] is in fact more general, showing that the first predecessor can be found (in any previous version) in $O(\log \log U)$ time but the predecessor of the predecessor (etc.) can be found in $O(1)$ time. This yields a time of $O(\log \log U + k)$ to find the k previous elements in sorted order in a chosen version of the data structure.

[5] Results in Section 6.1 stem from wonderful research chats with Timothy Chan.

[6] Actually Dietz and Raman [45] asked about persistency in general, which may refer to full persistence or partial persistence. We stick to partial persistence as it is sufficient for our needs.

We utilize this for the 2-sided sorted range reporting by creating a data structure for P as follows[7]. Consider the sort of the x-coordinates of P, i.e. some permutation π for which $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$. Now we insert the y-coordinates into the data structure according to π. That is we insert $y_{\pi(1)}$ and then $y_{\pi(2)}$ until $y_{\pi(n)}$. Now, a 2-sided sorted range reporting query $R = [-\infty, a] \times [-\infty, b]$ is answered as follows; first use a predecessor query to find a within $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$ - that is find i such that $x_{\pi(i)} \leq a < x_{\pi(i+1)}$. Then we go to the i-th copy of the partially persistent data structure which contains the points $(x_{\pi(1)}, y_{\pi(1)}), (x_{\pi(2)}, y_{\pi(2)}), \ldots, (x_{\pi(i)}, y_{\pi(i)})$. Hence, the points are exactly the points that satisfy that their x-coordinate $\in [-\infty, a]$. Now to find the relevant points ($y \in [-\infty, b]$) in R sorted by their y-coordinate we need to apply the predecessor query. This yields an $O(\log\log U + occ)$ when using the data structure from [27]. Hence,

Theorem 5. *The 2-sided sorted range reporting problem on an n-point set over a $U \times U$ grid can be solved with $O(n)$ space and $O(\log\log U + occ)$ time.*

6.2 Weighted Ancestors to 2-Sided Range Successor in 2D

Consider the weighted ancestors problem on an edge-weighted tree introduced by Farach and Muthukrishnan [46] for the sake of obtaining a perfect-hash for substrings. An edge-weighted tree is a tree where each edge e has a weight $w(e) \in [1, U]$. Each node v is associated with a weight $w(v) = \sum_{e \in p_v} w(e)$, where p_v is the path from root-to-v. The *weighted ancestors* problem is defined as follows.

Input: An edge-weighed tree T with weight function w.
Ouput: A data structure supporting *weighted ancestor queries.*
Query: Given a node u and a threshold t find the ancestor v of u such that $w(v) \geq t$, but $w(p(v)) < t$, where $p(v)$ is the parent of v.

The weighted ancestor problem is a natural extension of the predecessor problem to trees. The application considered by [46] was on suffix trees. A suffix tree can be viewed as an edge-weighted tree with the edge weights denoting the length of the text with which the edge is marked. Now, say you are given indices i and j and want to find the locus of $T[i \ldots j]$ in the suffix tree. This can be done by going to the leaf representing i and asking a weighted ancestor query with threshold $j - i + 1$. The answer to the query is the locus of $T[i \ldots j]$.

In [46] a solution was given with $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log\log U)$ query time. Their solution is based on a heavy path decomposition in order to linearize the input tree. Each path of the heavy path decomposition is assigned a predecessor structure. The preprocessing time of $O(n \log n)$ can be improved to $O(n)$ and this has been pointed out in [11, 79]. It should be

[7] We point out that for our purposes, finding one successor, the results of Dietz and Raman [45] are sufficient because (a) we seek only one successor and (2) the insertions are done first and then the queries are asked.

mentioned that the authors of [46] were considering a PRAM model and hence the $O(n \log n)$ time is really $O(\log n)$ parallel time and $O(n)$ work. Lately, it was shown that if the depth of the answer is d in the tree then the query can be answered in $O(\log \log d)$ time [78].

We now present a solution for this problem using 2-sided sorted range reporting.

Consider the edge-weighted input tree T. We assume that every internal node in the tree has at least two children. Otherwise, create a dummy child with an arbitrary edge weight, say 1. Now consider the leaves $l_1, \ldots, l_k$ ordered in inorder. For every two adjacent leaves denote their lowest common ancestor with $LCA(l_i, l_{i+1})$ and $nw(i) = w(LCA(l_i, l_{i+1}))$. With one scan of the tree all these values are computable. Now generate an array of the nw values. Say we are given a weighted ancestor query, node u and threshold t. We may assume that u is a leaf. Otherwise, we simply choose a descendant leaf to represent u (the answer will be the same). Consider the node v which is the answer to the query and consider its parent $p(v)$. Since $p(v)$ has at least two children v has at least one sibling. Say, v has a sibling to its left (in inorder). Let $u = l_i$ then in the nw array the first location $j < i$ that satisfies $nw(j) < t$ is the node for which the $LCA(l_j, l_i) = p(v)$. To obtain this j we revert to 2-sided sorted range reporting. We set the points on the grid to be $P = (j, nw(j))$. The query is bounded by i in the x-coordinates and t in the y-coordinates. What we are looking for is the first answer, the element with the largest y-coordinate.

Note that once this is done it is still necessary to find v (we only obtained $p(v)$). This can be done with a predecessor structure for each node. That is, for each node $p(v)$ save the index h of the leftmost leaf l_h for each of $p(v)$'s children. A predecessor query with i will return the correct edge with child v, the weighted ancestor of u. Hence,

Theorem 6. *Let T be an n node edge-weighted tree with weights from $[1, U]$. Then using 2-sided sorted range reporting one can answer weighted ancestor queries in $O(\log \log U)$ time. The space required is $O(n)$.*

Note that for a suffix tree, the motivation in [46], the weights are from $[1, n]$. So, the query time for a suffix tree is $O(\log \log n)$.

Recall (from Section 3) the definition of an SA-range and its relation to the suffix tree. Hence, the method just described precisely finds the boundaries of the SA-range for a given suffix i (in the suffix array) and its prefix of length (threshold) t.

7 Compressed Substring Retrieval

In this section we are concerned with the *substring compression* problem. The *substring compression* problem was introduced in [37]. Some of the definitions and layout here are from [37]. The solution, specifically the reduction to *range successor queries*, is from [75].

In *substring compression* one is given a text to preprocess so that, upon request, a compressed substring is returned. The goal is to do so quickly, preferably in $O(c(s))$ time, where $c(s)$ is the size of the compressed substring. *Generalized substring compression* is the same with the following twist. The queries contain an additional context substring (or a collection of context substrings) and the answers are the substring in compressed format, where the context substring is used to make the compression more efficient.

The compressor of interest is, once again, LZ77. We use the terminology from Section 5. Some extra terminology is as follows. The string S may be encoded within the context of the string T. We denote this by $\mathrm{LZ}(S \mid T)$. The encoded result will be equivalent to the result when LZ77 is performed on the concatenated string $T\$S$, where \$ is a symbol that does not appear in either S or T. However, only the portion of $\mathrm{LZ}(T\$S)$ which represents the compression of S is output by the algorithm.

Formally, given a string S of length n, we wish to preprocess S in such a way that allows us to efficiently answer the following queries:

Substring Compression Query ($\mathrm{SCQ}(i,j)$**):** given any two indices i and j, such that $1 \le i \le j \le n$, we wish to output $\mathrm{LZ}(S[i,\ldots,j])$.

Generalized Substring Compression Query ($\mathrm{GSCQ}(i,j,\alpha,\beta)$**):** given any four indices i, j, α, and β, such that $1 \le i \le j \le n$ and $1 \le \alpha \le \beta \le n$, we wish to output $\mathrm{LZ}(S[i,\ldots,j] \mid S[\alpha,\ldots,\beta])$.

The goal is to do answer queries quickly. Query times for both of the above query types will strongly depend on the number of phrases actually encoded. We denote these as $C(i,j)$ and $C_{\alpha,\beta}(i,j)$ for SCQ and GSCQ, respectively.

7.1 SCQ to Range Successor in 2D

Recall the definition of LZ77 from Section 5. Imagine that we have already computed the phrases for $S[i \ldots k-1]$ and desire to compute the next phrase which is a prefix of $S[k \ldots j]$. In other words, we want to find the location $i \le t \le k-1$ for which the longest common prefix of $S[k,\ldots,j]$ and the suffix S_t is maximal. Consider the suffix S_k, which is an extension of $S[k \ldots j]$. Clearly, it is sufficient to find the suffix S_t for which $|\mathrm{LCP}(S_k, S_t)|$ is maximized (without necessarily computing the value $|\mathrm{LCP}(S_k, S_t)|$ at this stage). Therefore we have two steps: (1) finding the location t, and (2) computing $|\mathrm{LCP}(S_k, S_t)|$. Step (2) is easy since we assume that we have a full LCP data structure as described in Section 3. So, our goal is to solve Step (1). To do so we generalize our problem to the following.

Interval Longest Common Prefix ($\mathrm{ILCP}(k,l,r)$**):** given k, l, r, we look for location $l \le t \le r$ of S for which the suffix S_t has the longest common prefix with S_k.

Clearly, for us it is sufficient to compute $\mathrm{ILCP}(k,i,k-1)$.

To compute the Interval Longest Common Prefix (ILCP(k, l, r)) we use a reduction to the problem of *3-sided range successor query*. That is given a 2D rank-space input on an $n \times n$ grid, a 3-sided query $R = [a, b] \times [-\infty, c]$ seeks the point in R with the largest y-coordinate. The 3-sided range successor query problem was considered under a different guise in [38] where it was called the *range next value* problem. There it was considered as an array problem for which one desires to preprocess the array to allow queries that seek the largest value on a range less than a value v. This can be translated to a grid and vice versa.

7.2 4-Sided and 3-Sided Sorted Range Reporting

The 3-sided range successor query generalizes quite nicely to the *3-sided sorted range reporting in 2D* which we define now.

3-Sided Sorted Range Reporting in 2D (rank space)	
Input:	A point set $P = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subseteq [1, n] \times [1, n]$.
Output:	A data structure representing P that supports the following *3-sided sorted range reporting* queries.
Query:	Given a range $R = [a, b] \times [-\infty, c]$ report all points of P contained in R sorted by their y-coordinate (from highest to lowest).

A solution for the 3-sided range successor query problem was proposed in Lenhof and Smid [83][8], and modified in [76] (improved query times) to work in rank space, i.e. on an $[n] \times [n]$ grid for n values with queries supported in $O(\log \log n)$ worst-case time, using $O(n \log n)$ space. However, there are now better results which solve, not only the 3-sided range successor problem, but the more general 3-sided sorted range reporting. The 3-sided sorted range reporting generalizes the 3-sided range successor query because the solutions presented can report the first location and stop.

For the same reason the 3-sided solutions work just as well for the 4-sided sorted range reporting, as they can report all points until the y-coordinates surpasses the range boundary and then stop.

The current best range 3-sided sorted range reporting data structures in 2D are as follows:

1. **Navarro and Nekrich [95]:**
   (a): a data structure with $O(n)$ space where queries can be answered in $O(\log n(1 + k))$ where k is the number of points reported and
   (b): a data structure with $O(n \log \log n)$ space where queries can be answered in $O(\log \log n(1 + k))$ and
   (c): a data structure with $O(n \log^{\epsilon} n)$ space for any constant $\epsilon > 0$, where queries can be answered in $O(\log \log n + k)$,
2. **Crochemore et al. [38]:** a data structure that requires $O(n^{1+\epsilon})$ space for any constant $\epsilon > 0$ and can answer queries in $O(k)$ time.

[8] Note, they called it the *Range Searching for Minimum* problem.

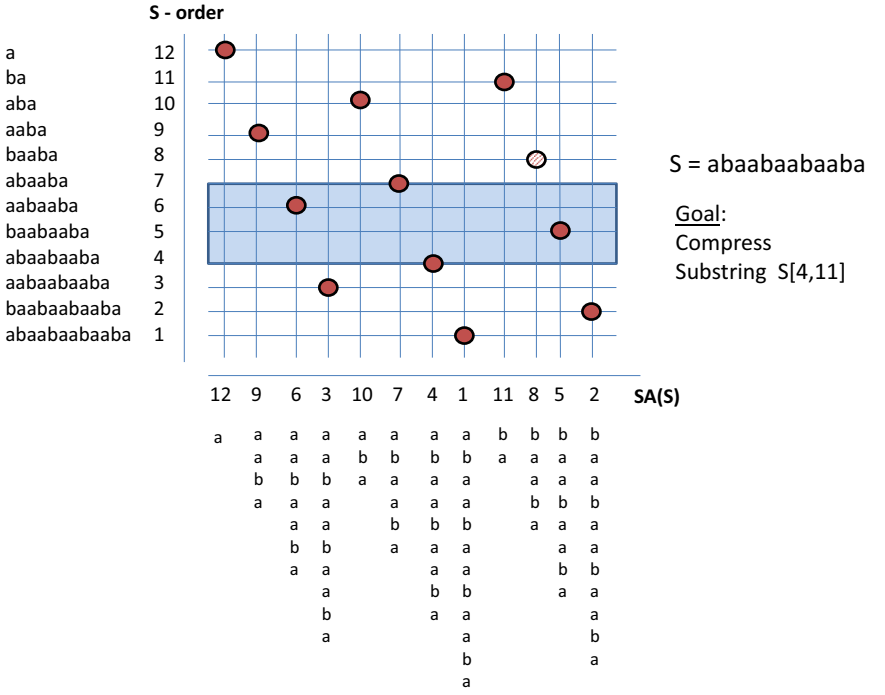


Fig. 3. The grid depicts P, the geometric representation of $S = abaabaabaaba$. The light point at $(8, 10)$ represents the suffix S_8 (and 10-th in the suffix array), as the substring $baab$, yet to be encoded, starts at position 8 in the string S. The grayed area of the grid represents the part of the substring which has already been encoded, that is $S[4, 7]$. When finding the start location t, we will be limited to using points found in the gray area.

7.3 The Interval Longest Common Prefix to 3-Sided Sorted Range Reporting

The reduction works as follows. Let $\mathrm{SA}(S)$ be the suffix array of our input string S of length n. We associate each suffix S_i with its string index i and with its lexicographic index $\mathrm{SA}^{-1}(i)$. From these two we generate a pair (x_i, y_i), where $x_i = i$ and $y_i = \mathrm{SA}^{-1}(i)$. We then preprocess the set $P = \{(x_i, y_i) \mid 1 \leq i \leq n\} \subseteq [1, n] \times [1, n]$ for 3-sided sorted range reporting queries. An example of the geometric representation of the scenario can be seen in Figure 3.

Computation of the ILCP. Consider the suffix S_k and the set of suffixes $\Gamma = \{S_l, \ldots, S_r\}$. Since $|\mathrm{LCP}(S_k, S_t)| = \max_{t' \in [l,r]} |\mathrm{LCP}(S_k, S_{t'})|$, S_t is in fact the suffix lexicographically closest to S_k, out of all the suffixes of the set Γ.

We will first assume that we are searching for a suffix S_{t_1}, such that the suffix S_{t_1} is *lexicographically smaller* than S_k. The process for the case where the suffix chosen is lexicographically greater than S_k is symmetric. Therefore, once both

are found all we will need to do is to choose the best of both, i.e., the option yielding the greater $|\mathrm{LCP}(S_k, S_t)|$ value.

Since we have assumed w.l.o.g. that S_{t_1} is lexicographically smaller than S_k, we have actually assumed that $y_{t_1} < y_k$, or equivalently, that t_1 appears to the *left* of k in the suffix array. Incorporating the lexicographical ranks of S_k and S_{t_1} into the expression, t_1 is actually the value which maximizes the expression $\max\{y_{t_1} \mid l \leq t_1 \leq r \text{ and } y_{t_1} < y_k\}$. Notice that $t_1 = x_{t_1}$.

Now consider the set $P = \{(x_i, y_i) \mid 1 \leq i \leq n\}$. Assuming that indeed $y_{t_1} < y_k$, we are interested in finding the maximal value y_{t_1}, such that $y_{t_1} < y_k$, and $l \leq x_{t_1} \leq r$. It immediately follows that the point $(x_{t_1}, y_{t_1}) \in P$ is the point in the range $[l, r] \times [-\infty, y_k - 1]$ having the maximal y-coordinate, and therefore can be obtained efficiently by obtaining the largest y-coordinate in the output of the 3-sided sorted range query. Once we have found the point (x_{t_1}, y_{t_1}), we have t_1, as $x_{t_1} = t_1$.

Equivalently, there exists t_2 such that S_{t_2} is the suffix lexicographically larger than S_k and closest to it. In other words, we assume $y_{t_2} > y_k$, or equivalently, that t_2 appears to the *right* of k in the suffix array. t_2 can be found using a symmetric procedure. An example of the queries performed can be seen in Figure 4.

Determining whether $t = t_1$ or $t = t_2$ is implemented by calculating both $|\mathrm{LCP}(S_k, S_{t_1})|$ and $|\mathrm{LCP}(S_k, S_{t_2})|$, and choosing the larger of the two. This gives us phrase ρ_i. To finish simply reiterate. Hence,

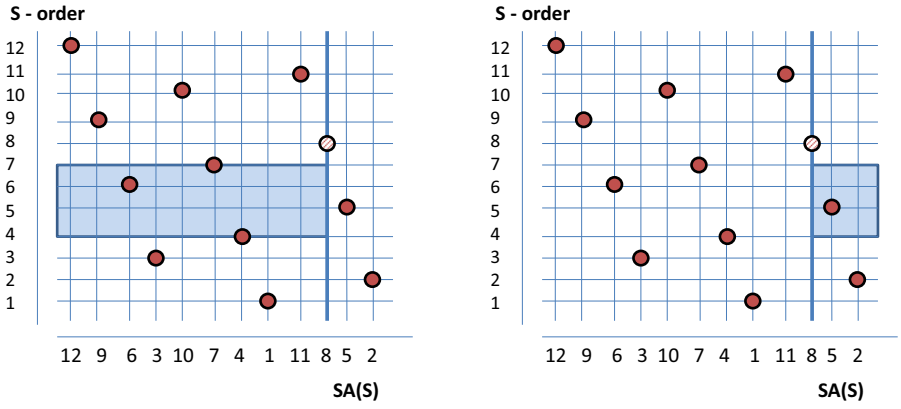


Fig. 4. Continued example from Figure 3 with string string $S = abaabaabaaba$. The grid on the right-hand side depicts the 3-sided range successor query for $[l, r] \times [-\infty, y_k - 1]$, where the grid on the left-hand side depicts the $[l, r] \times [y_k + 1, \infty]$ query. In both queries the values given for the example queries are: $k = 8$, $l = 4$ and $r = 7$. $S[l, r]$ is the substring that has already been encoded. The query $[l, r] \times [-\infty, y_k - 1]$ outputs $(4, 7)$ and the query on $[l, r] \times [y_k + 1, \infty]$ outputs $(5, 11)$. S_5 is chosen since $|\mathrm{LCP}S_8, S_5| > |\mathrm{LCP}S_8, S_4|$.

Theorem 7. *Let T be a text of length n. We can preprocess T in $O(s(n))$ space so that we can answer substring compression queries Q in $O(|\mathrm{LZ}(Q)|qt(n))$ time, where $s(n)$ and $qt(n)$ are the space and query times mentioned above ($occ = 1$).*

The GSCQ Problem. The *generalized substring compression* solution is more involved and uses binary searches on suffix trees applying range queries (3-sided range successor queries and emptiness queries) during the binary search. The interested reader should see [75].

Other Applications. The reduction from this section to range searching structures, i.e. the set of $P = \{(x_i, y_i) \mid 1 \leq i \leq n\} \subseteq [1, n] \times [1, n]$, defined by a suffix S_i with $x_i = i$ and $y(i) = \mathrm{SA}^{-1}(i)$ had been considered beforehand.

This reduction was first used, to the best of our knowledge, by Ferragina [49] as part of the scheme for searching in a dynamic text indexing scheme. The reduction and point set were used also in *position restricted substring search* [86]. However, in both *range reporting* was used.

To the best of our knowledge, the first use of 3-sided range successor queries for text indexing was for *range non-overlapping indexing and successive list indexing* [76] and in parallel for position restricted substring search in [39].

See Section 9 for more range-restricted string search problems.

8 Top-*k* Document Retrieval

The *Top k Document Retrieval* problem is an extension of the Document Retrieval problem described in Section 3.2. The extension is to find the *top k* documents in which a given pattern Q appears, under some relevance measure. Examples of such relevance measures are (a) *tf(Q,d)*, the number of times Q occurs in document d, (b) *mind(Q,d)*, the minimum distance between two occurrences of Q and d, (c) *docrank(d)*, an arbitrary static rank assigned to document d. In general, the type of relevance measures which which we shall discuss here are those that are defined by a function that assigns a numeric weight $w(S, d)$ to every substring S in document d, such that $w(S, d)$ depends only on the set of starting positions of occurrences of S in d. We call such a relevance measure a *positions based relevance measure*.

The following theorem is the culmination of the work of Hon, Shah and Vitter [65] and of Navarro and Nekrich [94].

Theorem 8. *Let D be a collection of strings (documents) of total length n, and let $w(S, d)$ be a positions based relevance measure for the documents $d \in D$. Then there exists an $O(n)$-word space data structure that, given a string Q and an integer k reports k documents d containing Q of highest relevance, i.e. with highest $w(Q, d)$ values, in decreasing order of $w(Q, d)$, in $O(|Q| + k)$ time.*

Hon, Shah and Vitter [65] reduced this to a problem on arrays and achieved query time of $O(|Q| + k \log k)$. We will outline their idea within this section. Navarro and Nekrich [94] then showed how to adapt their solution to a 3-sided 2 dimensional range searching problem on weighted points. The solution of the range searching problem given in [94] builds upon earlier work on *top k color queries for document retrieval* [74], another interesting result. We will describe the adaptation and range searching result shortly.

8.1 Flattening the Top-k Document Retrieval Suffix Tree

Consider a generalized suffix tree ST for the document collection D. The leaves have a one-one correspondence with the locations within the documents. If leaf l is associated with location i of document d, we say that it is a d-leaf. Let $l_1, l_2, \ldots, l_q$ be the set of d-leaves. Then a node is a d-node if (a) it is a d-leaf or if (b) it is an internal node v of ST such that it is the lowest common ancestor of adjacent d-leaves l_i and l_{i+1}. Let v be a d-node. If u is is the lowest common ancestor of v that is a d-node we say that u is v's d-parent (and that v is u's d-child). If there is no lowest common ancestor of v which is a d-node then the d-parent will be a dummy node which is the parent of the root. One can easily verify that the set of d-nodes form a tree, called a d-tree, and that an internal d-node has at least two d-children. It is also straightforward to verify that the lowest common ancestor of any two d-nodes is a d-node. Hence,

Lemma 4. *Let v be a node in the generalized suffix tree* ST *for the document collection D. For every document d for which the subtree of v contains a d-leaf there is exactly one d-node in the subtree of v that has a d-parent to an ancestor of v.*

Proof. 1. Every d-parent of a d-node in the subtree of v is either in the subtree or is an ancestor of v. 2. Assume, by contradiction, that there are two d-nodes x_1 and x_2 in the subtree of v each with a d-parent that is an ancestor of v. However, their lowest common ancestor, which must be a d-node, is no higher than v itself (since v is a common ancestor). Hence, their d-parents must be in v's subtree a contradiction.

Hence, since every d-node has a d-parent, there must be exactly one d-node with a d-parent to an ancestor of v. □

Corollary 1. *Let v be an arbitrary node in* ST *and let u be a descendant of v such that u is a d-node and its d-parent is an ancestor of v. Then all d-nodes u' which are descendants of v are also descendants of u.*

A node in ST may be a d-node for different d's, say for $d_{i_1}, \ldots, d_{i_r}$. Nevertheless, since every internal d-node has at least two d-children, the d-tree is linear in the number of d-leaves and, hence, the collection of all d-trees is linear in the size of the ST which is $O(n)$.

In light of this in [65] an array A was constructed by a pre-order traversal of the tree ST such that for each node v which is a d-node for $d \in \{d_{i_1}, \ldots, d_{i_r}\}$ indexes $j+1$ to $j+r$ are allocated in the array and contain the d_{i_1}-parent of v, ..., the d_{i_r}-parent of v. The integer interval $[l_v, r_v]$ denotes the interval bounded by the minimal and maximal indexes in A assigned to v or its descendants. Values l_v and r_v are stored in v.

The array A was used to obtain the query result of $O(|Q| + k \log k)$ in [65]. We now show how this was used in [94].

8.2 Solving with Weighted Range Searching

Let $j+t$ be the index in A associated with d_{i_t}-node v for $d_{i_t} \in \{d_{i_1}, \ldots, d_{i_r}\}$ and with its d_{i_t}-parent u_t. Let S be the string such that the locus of S is v. We generate a point $(j+t, \mathit{depth}(u_t))$, where *depth* denotes the depth of a node in the ST. The weight of the point p is $w(S, d_{i_t})$. Note that all points have different x-coordinates and are on an integer $n \times n$ grid.

It is still necessary to store a mapping from the x-coordinates of points to the document numbers. A global array of size $O(n)$ is sufficient for this task.

Queries. To answer a top-k query Q first find the locus v of Q (in $O(|Q|)$ time). Now, by Lemma 4 for each document d containing Q there is a unique d-node u which is a descendant of v with a d-parent who is an ancestor of v. By Corollary 1 $w(Q, d) = w(S, d)$, where S is the string with locus u, and $w(S, d)$ is the weight of the point corresponding to the pointer from u to its d-parent. So, there is a unique point (x, y) with $x \in [l_v, r_v]$ and $y \in [0, \mathit{depth}(v) - 1]$ for every document d that contains Q. Therefore, it is sufficient to report the k heaviest weight nodes in $[l_v, r_v] \times [0, \mathit{depth}(v) - 1]$. To do so Navarro and Nekrich [94] proposed the *three sided top-k range searching* problem.

Three sided top-k range searching	
Input:	A set of n weighted points on an $n \times n$ grid G.
Output:	A data structure over G supporting the following queries.
Query:	Given $1 \leq k, h \leq n$ and $1 \leq a \leq b \leq n$ return the k heaviest weighted points in the range $[a, b] \times [0, h]$.

In [94] a solution was given that uses $O(n)$-word space and $O(h+k)$ query time. This result is similar to that of [74]. It is easy to note that for the application of top-k document retrieval this yields an $O(\mathit{depth}(v) + k)$ query time, which is $O(|Q| + k)$, an optimal solution.

8.3 External Memory Top-k Document Retrieval

Lately, a new result for top-k document retrieval for the external memory model has appeared in [98]. The result is I/O optimal and uses $O(n \log^* n)$ space.

More on research in the vicinity of top-k document retrieval can be found in [92]. See also [17] to see how to add rank functionality to a suffix tree.

9 Range Restricted String Problems

Research inspired by the problem of applying string problems limited to ranges has been of interest in the pattern matching community from around 2005. Some of the results are general. Others focus on specific applications. One such application is the *substring compression* problem that was discussed in Section 7. These problems are natural candidates for range searching solutions and indeed many of them have been solved with these exact tools.

The first three results on range restricted variants of text indexing appeared almost in parallel. The results were for *property matching* (the conference version of [7]), *substring compression* [37] and *position-restricted substring searching* [86].

Property matching is the problem of generating a text index for a text and a collection of ranges over the text. The subsequent pattern queries asks for the locations where the text appears and are fully contained in some interval. The initial definition was motivated by *weighted matching*. In *weighted matching* a text is given with probabilities on each of the text symbols and each pattern occurrence in the text has *weight* which is the multiplication of the probabilities on the text symbols associated with that occurrence. Weighted matching was reduced to property matching. In [7] a solution was given using $O(n)$ space, where n is the text size, such that queries are answered in $O(|Q| + occ_\pi)$ time, where Q is the pattern query and occ_π is the number of appearances within the interval set π. The preprocessing time was near optimal and in a combination of a couple of papers was solved in optimal time [66,69]. See also [40]. In [77] property matching was solved in the dynamic case, where intervals can be inserted and removed. Formally, π denotes the collection of intervals and the operations are:

- Insert(s, f) - Insert a new interval (s, f) into π.
- Delete(s, f) - Delete the interval (s, f) from π.

In [77] it was shown how to maintain a data structure under interval deletions. Queries are answered in $O(Q| + occ_\pi)$ time and deletions take $O(f - s)$ time. If both insertions and deletions are allowed then the insertion/deletion time is $O(f - s + \log\log n)$, where n is the text length.

In [63] a succinct version was given that uses a compressed suffix array (CSA). The solution has a multiplicative logarithmic penalty for the query and update time.

Position-restricted substring searching is the problem where the goal is to preprocess an index to allow range-restricted queries. That is the query consists of a pattern query Q and a range described by text indices i and j. This is different from property matching because the interval is not given a-priori. On the other hand, it is one interval only. The queries considered in [86] are *position-restricted reporting* and *position-restricted counting*. Another two related queries also considered are *substring rank* and *substring select*, which are natural extensions of rank and select [24, 57, 68]. These are defined as follows.

1. **PRI-Report:** Preprocess text $T = t_1 \cdots t_n$ to answer queries Report($Q = q_1 \cdots q_m, i, j$), which reports all occurrences of Q in $t_i \ldots t_j$.
2. **PRI-Count:** Preprocess text $T = t_1 \cdots t_n$ to answer queries Count($Q = q_1 \cdots q_m, i, j$), which returns the number of occurrences of Q in $t_i \ldots t_j$.
3. **Substring Rank:** Preprocess text $T = t_1 \cdots t_n$ to answer queries SSR($Q = q_1 \cdots q_m, k$), which returns the number of occurrences of Q in $t_1 \cdots t_k$.
4. **Substring Select:** Preprocess text $T = t_1 \cdots t_n$ to answer queries SSS(Q, k), which returns the k^{th} occurrence of Q in T.

We note that substring rank and position-restricted counting reduce to each other. Also, position-restricted reporting can be obtained from applying one substring-rank and $occ_{i,j}$ substring-selects, where $occ_{i,j}$ is the number of pattern occurrences in $t_i \ldots t_j$. We leave it to the reader to verify the details.

Reporting: For the reporting problem Mäkinen and Navarro [86] reduced the problem to range reporting. The way to do so is to first find the SA-range of Q. Then this range and the position-restricted range i to j define a rectangle for which range reporting is used. One can use any of the data structures mentioned in Section 4. For example with $O(n \log^{\epsilon} n)$ space, for any constant ϵ, one can answer queries in $O(m + \log\log n + occ_{i,j})$ (we assume the alphabet is from $[1, n]$ otherwise if it is from $[1, U]$ then there is an extra additive factor of $\log\log U$). Crochemore et al. [39] noticed that a different type of reduction, namely range next value, could be more useful. The authors of [86] were more concerned with space issues. So, they also proposed a data structure which uses $n + o(n)$ space and reports in $O(m + \log n + occ_{i,j})$. The reporting time was improved by Bose et al. [20] to $O(m + \log n/\log\log n + occ_{i,j})$ with the same space constraints. Yu et al. [106] suggested a different algorithm with the same space and reporting time, but were able to report the occurrences in their original order.

Bille and Gørtz [19] went on to show that with $O(n(\log^{\epsilon} n + \log\log U))$ space the query time can be improved to $O(m + occ_{i,j})$, which is optimal. They also solved position-restricted reporting merged with property matching. The space and query time remain the same.

An interesting result for the reporting variant appeared in [64]. Specifically, it was shown that a succinct space polylog n time index for position-restricted substring searching is at least as hard as designing a linear space data structure for 3D range reporting in polylog n time.

Counting: In [86] the same data structure that uses $n + o(n)$ space and reports in $O(m + \log n)$ was used for counting[9]. Once again, Bose et al. [20] can improve the counting time to $O(m + \log n/\log\log n)$. Kopelowitz et al. [80] presented a counting data structure that uses $O(n(\log n/\log\log n))$ space and answers counting queries in time $O(m + \log\log|\Sigma|)$. Recently, in the upcoming journal version of [19] a similar result appears. The space is the same. However, the counting time is $O(m + \log\log n)$, which can be slightly worse.

[9] There is another result there that assumes faster query times that is flawed. See the introduction in [80] for an explanation.

Substring Select: In [86] a solution for indexing for substring select is given. The space is $O(nK \log \sigma / \log n)$, where K is an upper bound on the size of the queried patterns. The query time is $O(m \log \sigma / \log \log n)$. This was improved in [80] to allow for any length query with $O(n \log n / \log \log n)$ space and optimal $O(m)$ query time. The proposed solution in [80] uses persistent data structure which is a basic ingredient in most of the range searching solutions.

Substring compression has been expanded on in Section 7. It was introduced in [37] and improved upon in [75]. The results are detailed in Section 7. One of the problems that is of interest in substring compression is the $ILCP$ (interval longest common prefix) query (see Section 7). This inspired Amir et al. [3] to consider extensions to LCP range queries of different types.

Range Non-overlapping Indexing and Successive List Indexing [76]. In range non-overlapping indexing one wants to prepare an index so that when give a pattern query one can return a maximal set of occurrences so that the occurrences do not overlap. In successive list indexing one prepares an index to answer queries where a pattern is given along with a position i and one desires to find the first occurrence of the pattern after i. A reduction to range successor was used to solve this problem. Along with the results of sorted range reporting [95] one can solve the former with space $O(n \log^{\epsilon} n)$ space and $O(\log \log n + occ)$ query time. For the latter the query time is $O(\log \log n)$.

Range Successor in 2D solves several of the problems mentioned in this section. This has been discussed in Section 7 and is referred to in [38, 95, 106]. It is interesting that its generalization sorted range reporting [95] is a variant of range reporting that was considered in the community because of the unique range search problems that arise.

10 Lower Bounds on Text Indexing via Range Reporting

A novel use of range searching is its use to show lower bounds on text indexing via reductions from range reporting [32].

Theorem 9. *Let $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a set of n points in $[1, n] \times [1, n]$. We can construct a text T of length $O(n \log n)$ bits along with $O(n \log n)$ bits of auxiliary data such that we can answer range reporting queries on S with $O(\log^2 n)$ pattern match queries on T, each query is a pattern of length $O(\log n)$.*

We denote the set of x-coordinates, of S, $X = \{x_1, \ldots, x_n\}$ and the set of y-coordinates, of S, $Y = \{y_1, \ldots, y_n\}$.

The idea is as follows. Each point $(x, y) \in [1, n] \times [1, n]$. So, x and y both have binary representations of $\log n$ bits[10]. Denote with $b(w)$ the

[10] We assume that n is a power of 2. Otherwise, it will be $\lfloor \log n \rfloor + 1$.

binary representation of a number w and the reverse of the binary representation with $b^R(w)$. The text T (from the theorem) constructed is $b^R(x_1)\#b(y_1)\$b^R(x_2)\#b(y_2)\$\ldots\$b^R(x_1)\#b(y_1)$.

To obtain the result a collection of pattern queries on T is generated whose answers will yield an answer to the range reporting problem on the point set S. To this end, sort $b(y_1),\ldots,b(y_n)$ and $b^R(x_1),\ldots,b^R(x_n)$. Let π denote the ordering of the former and τ denote the ordering of the latter, i.e. $b(\pi(y_1)) < \ldots < b(\pi(y_n))$, where $<$ is a lexicographic-less than, and $b^R(\tau(x_1)) < \ldots < b^R(\tau(x_n))$. The n-length arrays $A =< b^R(\tau(x_1)),\ldots,b^R(\tau(x_n)) >$ and $B =< b(\pi(y_1)),\ldots,b(\pi(y_n)) >$ will be the basis of the search.

Over each of the arrays construct a binary search tree with each node representing a range of elements. Without loss of generality, consider the binary tree over B. The root represents all elements of Y. The left son is associated with one bit 0 and represents $R_0 = \{y \in Y | 0 \text{ is prefix of } b(y)\}$ and the right son represents $R_1 = \{y \in Y | 1 \text{ is prefix of } b(y)\}$ - each is a range over B - check. The left son of the left son of the root represents $R_{00} = \{y \in Y | 00 \text{ is prefix of } b(y)\}$, etc. In general, each node is associated with a binary string, say $b_0 \ldots b_i$, formed by the walk down from the root to the node and is also associated with a range, which we call a *node-range*, $R_{b_0\ldots b_i} = \{y \in Y | b_0 \ldots b_i \text{ is prefix of } b(y)\}$. The number of nodes in the binary tree and, hence, the number of ranges is $\leq 2n-1$. Each range can be represented as a pair of indexes to the array. Hence, the size of the auxiliary information is $O(n)$-words, or $O(n \log n)$ bits. We construct a complementary binary tree for A, with ranges RX.

An easy well known observation is that any range $R^{q,r} = \{y | 1 \leq q \leq y \leq r \leq n\}$ can be expressed as the disjoint union of at most $2 \log n$ node-ranges. The node-ranges of the disjoint union can be found by a traversal up and down the binary tree using the binary representations of q and r.

Now consider a range query on S, say $[x_{left}, x_{right}] \times [y_{bottom}, y_{top}]$. This can be seen as a query for all (x, y) such that $y \in R^{y_{bottom},y_{top}}$ and $x \in RX^{x_{left},x_{right}}$. By the previous observation this can be transformed into $O(\log^2 n)$ queries for all (x, y) such that x is in one of the node-ranges in the disjoint union expressing $R^{y_{bottom},y_{top}}$ and y is in one of the node-ranges in the disjoint union expressing $RX^{x_{left},x_{right}}$.

We show an a indexing query that searches for all (x, y) such that $x \in RX_c$ and $y \in R_d$ both node-ranges, the former for the binary string c over the array A and the latter for the binary string d over the array B. We define a pattern query $Q = c^R\#d$. We query the text index with Q. Every location where Q appears corresponds to a point that is in the desired range as c^R must align with the end of an $b^R(x_i)$, which is the same as c being a prefix of $b(x_i)$ and d is a prefix of (y_i), which is exactly the desired.

Chazelle [30] showed that in the pointer machine model an index supporting 2D range reporting in $O(\text{polylog}(n) + occ)$ query time, where occ is the number of occurrences, requires $\Omega(n(\log n/\log\log n))$ words of storage. Hence,

Theorem 10. *In the pointer machine model a text index on T of size n which returns locations of pattern occurrences in $O(\text{polylog}(n) + occ)$ time requires $\Omega(n(\log n/\log\log n))$ bits.*

More on This Result and Related Work. In [32] there are also very interesting results reducing text indexing to range searching. The reduction is known as a Geometric BWT, transforming a BWT into a point representation. The reductions in both directions show that obtaining improvements in space complexity of either will imply space complexity improvements on the other.

Another two lower bounds, along the lines of the lower bound described here, are for *position restricted substring search* [64], for *forbidden patterns* [52] and for *aligned pattern matching* [99].

Acknowledgement. I wanted to thank my numerous colleagues who were kind enough to provide insightful comments on an earlier version and pointers to work that I was unaware of. These people include (in alphabetical order) Phillip Bille, Timothy Chan, Francisco Claude, Pooya Davoodi, Johannes Fischer, Travis Gagie, Roberto Grossi, Orgad Keller, Tsvi Kopelowitz, Muthu Muthukrishnan, Gonzalo Navarro, Yakov Nekrich, Rahul Shah, Sharma Thankachan, Rajeev Raman, and Oren Weimann. Special thanks to Orgad, Rahul, Sharma and Yakov for numerous Skype conversations in which I learned more than can be contained within this monologue.

References

1. Agarwal, P.K.: Range searching. In: Handbook of Discrete and Computational Geometry, pp. 575–598. CRC Press, Inc. (1997)
2. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: Proc. of Foundations of Computer Science (FOCS), pp. 198–207 (2000)
3. Amir, A., Apostolico, A., Landau, G.M., Levy, A., Lewenstein, M., Porat, E.: Range LCP. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 683–692. Springer, Heidelberg (2011)
4. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. Journal of Discrete Algorithms 1(5-6), 409–421 (2003)
5. Amir, A., Aumann, Y., Lewenstein, M., Porat, E.: Function matching. SIAM Journal on Computing 35(5), 1007–1022 (2006)
6. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: Pattern matching in z-compressed files. Journal of Computer and System Sciences 52(2), 299–307 (1996)
7. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theoretical Computer Science 395(2-3), 298–310 (2008)
8. Amir, A., Fischer, J., Lewenstein, M.: Two-dimensional range minimum queries. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 286–294. Springer, Heidelberg (2007)

9. Amir, A., Francheschini, G., Grossi, R., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Managing unbounded-length keys in comparison-driven data structures with applications to on-line indexing. SIAM Journal on Computing (to appear, 2013)
10. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. Journal of Algorithms 37(2), 309–325 (2000)
11. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Transactions on Algorithms 3(2) (2007)
12. Arroyuelo, D., Navarro, G., Sadakane, K.: Stronger Lempel-Ziv based compressed text indexing. Algorithmica 62(1-2), 54–101 (2012)
13. Atallah, M.J., Yuan, H.: Data structures for range minimum queries in multidimensional arrays. In: Proc. of the Symposium on Discrete Algorithms (SODA), pp. 150–160 (2010)
14. Badkobeh, G., Fici, G., Kroon, S., Lipták, Z.: Binary jumbled string matching for highly run-length compressible texts. Information Processing Letters 113, 604–608 (2013)
15. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 52(1), 28–42 (1996)
16. Barbay, J., Claude, F., Navarro, G.: Compact binary relation representations with rich functionality. The Computing Research Repository (arXiv), abs/1201.3602 (2012)
17. Bialynicka-Birula, I., Grossi, R.: Rank-sensitive data structures. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 79–90. Springer, Heidelberg (2005)
18. Bille, P., Fischer, J., Gørtz, I.L., Kopelowitz, T., Sach, B., Vildhøj, H.W.: Sparse suffix tree construction in small space. In: Proc. of International Colloquium on Automata, Languages and Complexity, ICALP (2013)
19. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (2011)
20. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
21. Brodal, G.S., Gąsieniec, L.: Approximate dictionary queries. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 65–74. Springer, Heidelberg (1996)
22. Brodal, G.S., Davoodi, P., Lewenstein, M., Raman, R., Rao, S.S.: Two dimensional range minimum queries and Fibonacci lattices. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 217–228. Springer, Heidelberg (2012)
23. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica 63(4), 815–830 (2012)
24. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM Journal on Computing 28(5), 1627–1640 (1999)
25. Butman, A., Eres, R., Landau, G.M.: Scaled and permuted string matching. Information Processing Letters 92(6), 293–297 (2004)
26. Chan, H.-L., Lam, T.W., Sung, W.-K., Tam, S.-L., Wong, S.-S.: A linear size index for approximate pattern matching. Journal of Discrete Algorithms 9(4), 358–364 (2011)
27. Chan, T.M.: Persistent predecessor search and orthogonal point location on the word ram. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 1131–1145 (2011)

28. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. of the Symposium on Computational Geometry, SOCG (2011)
29. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)
30. Chazelle, B.: Lower bounds for orthogonal range searching: I. the reporting case. Journal of the ACM 37(2), 200–212 (1990)
31. Chazelle, B., Rosenberg, B.: The complexity of computing partial sums off-line. International Journal of Computational Geometry and Applications 1(1), 33–45 (1991)
32. Chien, Y.-F., Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: Geometric Burrows-Wheeler transform: Compressed text indexing via sparse suffixes and range searching. Algorithmica (to appear, 2013)
33. Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: Holub, J., Zdárek, J. (eds.) Proceedings of the Prague Stringology Conference (PSC), pp. 105–117 (2009)
34. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundamenta Informaticae 111(3), 313–337 (2011)
35. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 180–192. Springer, Heidelberg (2012)
36. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. of Symposium on Theory of Computing (STOC), pp. 91–100 (2004)
37. Cormode, G., Muthukrishnan, S.: Substring compression problems. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 321–330 (2005)
38. Crochemore, M., Iliopoulos, C.S., Kubica, M., Rahman, M.S., Tischler, G., Walen, T.: Improved algorithms for the range next value problem and applications. Theoretical Computer Science 434, 23–34 (2012)
39. Crochemore, M., Iliopoulos, C.S., Rahman, M.S.: Finding patterns in given intervals. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 645–656. Springer, Heidelberg (2007)
40. Crochemore, M., Kubica, M., Walen, T., Iliopoulos, C.S., Rahman, M.S.: Finding patterns in given intervals. Fundamenta Informaticae 101(3), 173–186 (2010)
41. Davoodi, P., Landau, G., Lewenstein, M.: Multi-dimensional range minimum queries (manuscript, 2013)
42. Davoodi, P., Raman, R., Satti, S.R.: Succinct representations of binary trees for range minimum queries. In: Gudmundsson, J., Mestre, J., Viglas, T. (eds.) COCOON 2012. LNCS, vol. 7434, pp. 396–407. Springer, Heidelberg (2012)
43. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 341–353. Springer, Heidelberg (2009)
44. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. Journal of Algorithms 48(1), 2–15 (2003)
45. Dietz, P.F., Raman, R.: Persistence, amortization and randomization. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 78–88 (1991)
46. Farach, M., Muthukrishnan, S.: Perfect hashing for strings: Formalization and algorithms. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 130–140. Springer, Heidelberg (1996)
47. Farach, M., Thorup, M.: String matching in Lempel-Ziv compressed strings. Algorithmica 20(4), 388–404 (1998)

48. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. Journal of the ACM 47(6), 987–1011 (2000)
49. Ferragina, P.: Dynamic text indexing under string updates. Journal of Algorithms 22(2), 296–328 (1997)
50. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
51. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: A geometric approach with applications to string matching problems. In: Proc. of Symposium on Theory of Computing (STOC), pp. 483–491 (1999)
52. Fischer, J., Gagie, T., Kopelowitz, T., Lewenstein, M., Mäkinen, V., Salmela, L., Välimäki, N.: Forbidden patterns. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 327–337. Springer, Heidelberg (2012)
53. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
54. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. of the Symposium on Theory of Computing (STOC), pp. 135–143 (1984)
55. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: A faster grammar-based self-index. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 240–251. Springer, Heidelberg (2012)
56. Golin, M., Iacono, J., Krizanc, D., Raman, R., Rao, S.S.: Encoding 2D range maximum queries. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 180–189. Springer, Heidelberg (2011)
57. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 368–373 (2006)
58. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003)
59. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing 35(2), 378–407 (2005)
60. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press (1997)
61. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
62. Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Compressed dictionary matching with one error. In: Proc. of the Data Compression Conference (DCC), pp. 113–122 (2011)
63. Hon, W.-K., Patil, M., Shah, R., Thankachan, S.V.: Compressed property suffix trees. In: Proc. of the Data Compression Conference (DCC), pp. 123–132 (2011)
64. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: On position restricted substring searching in succinct space. Journal of Discrete Algorithms 17, 109–114 (2012)
65. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: Proc. of Foundations of Computer Science (FOCS), pp. 713–722 (2009)
66. Iliopoulos, C.S., Rahman, M.S.: Faster index for property matching. Information Processing Letters 105(6), 218–223 (2008)
67. Iliopoulos, C.S., Rahman, M.S.: Indexing factors with gaps. Algorithmica 55(1), 60–70 (2009)

68. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554 (1989)
69. Juan, M.T., Liu, J.J., Wang, Y.L.: Errata for "faster index for property matching". Information Processing Letters 109(18), 1027–1029 (2009)
70. Kärkkäinen, J.: Repetition-Based Text Indexes. PhD thesis, University of Helsinki, Finland (1999)
71. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)
72. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proc. 3rd South American Workshop on String Processing (WSP). International Informatics Series 4, pp. 141–155. Carleton University Press (1996)
73. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 219–230. Springer, Heidelberg (1996)
74. Karpinski, M., Nekrich, Y.: Top-k color queries for document retrieval. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 401–411 (2011)
75. Keller, O., Kopelowitz, T., Landau, S., Lewenstein, M.: Generalized substring compression. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 26–38. Springer, Heidelberg (2009)
76. Keller, O., Kopelowitz, T., Lewenstein, M.: Range non-overlapping indexing and successive list indexing. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 625–636. Springer, Heidelberg (2007)
77. Kopelowitz, T.: The property suffix tree with dynamic properties. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 63–75. Springer, Heidelberg (2010)
78. Kopelowitz, T., Kucherov, G., Nekrich, Y., Starikovskaya, T.A.: Cross-document pattern matching. Journal of Discrete Algorithms (to appear, 2013)
79. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 565–574 (2007)
80. Kopelowitz, T., Lewenstein, M., Porat, E.: Persistency in suffix trees with applications to string interval problems. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 67–80. Springer, Heidelberg (2011)
81. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science 483, 115–133 (2013)
82. Landau, G.M., Vishkin, U.: Fast string matching with k differences. Journal of Computer and System Sciences 37(1), 63–78 (1988)
83. Lenhof, H.-P., Smid, M.H.M.: Using persistent data structures for adding range restrictions to searching problems. Theoretical Informatics and Applications (ITA) 28(1), 25–49 (1994)
84. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)
85. Lewenstein, M.: Parameterized matching. In: Encyclopedia of Algorithms (2008)
86. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
87. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
88. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2), 262–272 (1976)
89. Moosa, T.M., Rahman, M.S.: Indexing permutations for binary strings. Information Processing Letters 110(18-19), 795–798 (2010)

90. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. of the Symposium on Discrete Algorithms (SODA), pp. 657–666 (2002)
91. Navarro, G.: Wavelet trees for all. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 2–26. Springer, Heidelberg (2012)
92. Navarro, G.: Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. The Computing Research Repository (arXiv), abs/1304.6023 (2013)
93. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2 (2007)
94. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 1066–1077 (2012)
95. Nekrich, Y., Navarro, G.: Sorted range reporting. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 271–282. Springer, Heidelberg (2012)
96. Russo, L.M.S., Oliveira, A.L.: A compressed self-index using a Ziv-Lempel dictionary. Information Retrieval 11(4), 359–388 (2008)
97. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5(1), 12–22 (2007)
98. Shah, R., Sheng, C., Thankachan, S.V., Vitter, J.S.: On optimal top-k string retrieval. The Computing Research Repository (arXiv), abs/1207.2632 (2012)
99. Thankachan, S.V.: Compressed indexes for aligned pattern matching. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 410–419. Springer, Heidelberg (2011)
100. Tsur, D.: Fast index for approximate string matching. Journal of Discrete Algorithms 8(4), 339–345 (2010)
101. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
102. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Letters 6(3), 80–82 (1977)
103. Vuillemin, J.: A unifying look at data structures. Communications of the ACM 23(4), 229–239 (1980)
104. Weiner, P.: Linear pattern matching algorithm. In: Proc. of the Symposium on Switching and Automata Theory, pp. 1–11 (1973)
105. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\theta(n)$. Information Processing Letters 17(2), 81–84 (1983)
106. Yu, C.-C., Hon, W.-K., Wang, B.-F.: Improved data structures for the orthogonal range successor problem. Computational Geometry 44(3), 148–159 (2011)
107. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

A Survey of Data Structures in the Bitprobe Model

Patrick K. Nicholson[1], Venkatesh Raman[2], and S. Srinivasa Rao[3]

[1] Cheriton School of Computer Science, University of Waterloo, Canada
p3nichol@cs.uwaterloo.ca
[2] The Institute of Mathematical Sciences, Chennai, India
vraman@imsc.res.in
[3] School of Computer Science and Engineering,
Seoul National University, Republic of Korea
ssrao@cse.snu.ac.kr

1 Introduction

In this paper we survey data structures in the *bitprobe model* [31,15,50,30,11]. This model was introduced by Minsky and Papert in their book "Perceptrons" [31], studied later in the context of retrieval problems by Elias and Flower [15], and generalized by Yao [50] to the cell probe model. In the bitprobe model, we concern ourselves with the number of bit accesses or bit flips that occur during a computation. We wish to analyze the trade-off between the space occupied by a data structure, and the number of bits accesses that must be made to it in order to answer queries. Each bit access is referred to as a *probe* in this model. Furthermore, the amount of computation permitted by the query algorithm to determine which bits to probe is a secondary concern.

In Section 2 we discuss the *membership problem*, giving a brief history of the problem in other models of computation, and explain why this problem is fascinating in the bitprobe model. We survey the numerous recent results for this problem, mentioning the techniques used. In Section 3 we discuss the problem of maintaining an integer counter such that the operations increment and/or decrement can be executed efficiently. In Section 4 we briefly survey some of the other problems that have been considered in the bitprobe model, such as the greater-than problem, and various lower bounds. Finally, in Section 5, we conclude with a list of open problems.

2 The Membership Problem

The static *membership* problem, also known as the *dictionary* problem, is likely the first data structuring problem encountered in an introductory course on algorithms. Formally, the problem asks us to store a subset $\mathcal{N}$ of n elements from a universe $[1, m]$, such that we can efficiently determine, for any $x \in [1, m]$, "Is $x \in \mathcal{N}$?". In the *comparison model*, the cost of a query is equal to the number of comparisons that are made with the *query element* x to determine the yes or

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 303–318, 2013.

no answer. It is well-known that storing the set $\mathcal{N}$ in sorted order and performing a binary search over the elements is the best possible solution when comparisons with the query element are the only way to determine the relative order between elements [27, Sec. 6.2]. If we relax the constraint that $\mathcal{N}$ be stored in sorted order, and allow $\mathcal{N}$ to be stored in one of at most p permutations of sorted order, Alt, Mehlhorn, and Munro [2] proved a lower bound showing $\Omega(p^{1/n})$ comparisons are necessary for answering membership queries. To circumvent this lower bound, by further relaxing the model, Borodin et al. [5] mention (citing a communication by Feldman) that if comparisons between elements in $\mathcal{N}$ are also permitted— i.e., comparisons do not have to involve the query element— then it is possible to store $\mathcal{N}$ in one of $(n/4)!$ permutations of sorted order, described by *involutions*, while still supporting search in $\Theta(\lg n)$ comparisons.

In the word-RAM model, the cost of a query is measured in terms of the number of word operations that are carried out: i.e., arithmetic and read/write operations on an infinite array of w-bit cells called *words*. Note the connection between this model and the more powerful cell probe model, where *only* the read/write operations on cells are counted to measure cost. Since intermediate computation is not counted, lower bounds proved in the cell probe model hold in the word-RAM model, regardless of which arithmetic operations supported. The space usage of a data structure in the word-RAM model is the largest memory address that is used during a computation. When a bound on m is given, then a bit vector can be used to store $\mathcal{N}$ using m bits, and support queries (and even insertions and deletions) in constant time. However, the space usage of such a data structure can be prohibitive. A major breakthrough result for the membership problem was that of Fredman, Komlós and Szemerédi [19], now frequently referred to as *FKS-hashing*. They showed that on a word-RAM with $\Theta(\lg m)$-bit words, it is possible to answer membership queries using $O(1)$ word operations, using only $O(n \lg m)$ bits of space (a linear-in-n number of words).

The query time of FKS-hashing is clearly optimal (to within constant factors) in the word-RAM model, so we might wonder about the its space usage. The information theoretic lower bound indicates that a data structure solving this problem must occupy $\lg \binom{m}{n} = n \lg(m/n) + \Theta(n)$ bits of space. For sparse subsets $\mathcal{N}$, this shows that FKS-hashing is within a constant factor of optimal space usage, but when n is closer to m, there is room for improvement. These gaps were eventually closed by Brodnik and Munro [10], Pagh [36] and Pătraşcu [38], who showed that it was possible to achieve $O(1)$ time using $\lg \binom{m}{n} + o(\lg \binom{m}{n})$ bits of space.

Buhrman, Miltersen, Radhakrishnan, and Venkatesh [11] revived the study of the membership problem in the bitprobe model, where time and space are measured in bits. In this model, FKS-hashing uses $O(\lg m)$ probes to answer queries, and Buhrman et al. [11] asked whether this can be reduced. Recall that the trivial bit vector data structure, which stores m bits— each one corresponding to an element in the universe— can answer membership queries in one probe. Buhrman et al. showed, rather surprisingly, that if randomization and errors are allowed, one probe is sufficient to answer membership queries, using $O(n \lg m)$

bits of space[1]. Note that prior to the work of Buhrman et al., for the case where the query algorithm is allowed to make errors, there were existing data structures that improved upon the trivial bit vector, such as the Bloom filter [3] and its variants [9]. To represent a universe of size m, a Bloom filter uses a bit vector of size $m' < m$, together with a set of k different hash functions, $f_1, \ldots, f_k$, mapping $m \to m'$. To store an element x, we set the bits in locations $f_1(x), \ldots, f_k(x)$ to 1. To check if an element q is present, we probe locations $f_1(q), \ldots, f_k(q)$ and return that it is, if and only if all locations store a 1. Thus, even Bloom filters typically probe more than a single bit.

Consider the case where $n = 1$, and the query algorithm is *not* allowed to make an error. In this case, a trivial generalization of the bit vector can achieve $tm^{1/t}$ bits of space, and answer queries in t probes. The idea is to store t characteristic bit vectors; one for each $1/t$-th fraction of the bits of the sole element to be stored. As we shall see later, this strategy is space optimal, to within constant factors. We encourage the reader to try to generalize this strategy to the case where $n = 2$, to get a feel for the difficulties that arise in the bitprobe model. In particular, think about the case where $t = 2$, and try to come up with a data structure that occupies $O(\sqrt{m})$ bits[2].

Before discussing results on membership in the bitprobe model, we introduce some terminology.

2.1 Membership "Schemes"

Following the convention of prior work [11,40,41] we use the notation (n, m, s, t)-scheme to refer to a *storage scheme* that uses s bits to store any n element subset of a universe of size m, such that queries can always be answered using t probes. Thus, for a fixed set $\mathcal{N}$ the storage scheme is used to compute a data structure, $\Phi(\mathcal{N}) \in \{0,1\}^s$; i.e., a data structure that occupies s bits. Furthermore, the *query algorithm*, which is fixed once-and-for-all like the storage scheme, can perform membership queries for any element $q \in [1, m]$, by probing no more than t locations in $\Phi(\mathcal{N})$.

For example, a bit vector together with direct lookup is equivalent to a $(n, m, m, 1)$-scheme for the membership problem, whereas the FKS-hashing based data structures we discussed previously [10,36] are $(n, m, \lg \binom{m}{n} + o(\binom{m}{n}), \Theta(\lg m))$-schemes. There are several additional ways of categorizing a (n, m, s, t)-scheme, which we now define, following definitions of Buhrman et al. [11]:

Deterministic or Randomized: In a deterministic scheme the answer provided by the query algorithm must always be correct, and neither the storage scheme nor the query algorithm has access to random bits. In a randomized scheme the query algorithm may use random bits to decide which locations to

[1] The bound assumes an arbitrary constant error probability $\varepsilon > 0$. See Section 2.3.

[2] As we will see later this bound is not possible for $t = 2$, but it clearly illustrates one difficulty of the bitprobe model. Note that it is possible to achieve $O(\sqrt{m})$ bits for the case where $n = 2$, and $t = 3$.

probe in the data structure, and is also permitted to answer incorrectly with some failure probability, denoted by ε. However, in a randomized scheme the storage scheme (i.e., the method to compute the data structure $\Phi(\mathcal{N})$) must be deterministic. Randomized schemes are further subdivided into the following categories:

1. One-sided ε-error schemes: Schemes which never have false negatives— i.e., the query algorithm never returns a negative answer when the query element is in $\mathcal{N}$— but may return false positives with probability ε; i.e., the query algorithm returns a positive answer given a query element *not* in $\mathcal{N}$. Note that if we desire no false positives, rather than no false negatives, this is equivalent to storing a one-sided ε-error scheme on the complement of $\mathcal{N}$.
2. Two-sided ε-error schemes: Schemes which permit both false positives *and* false negatives with probability ε.

Non-adaptive or Adaptive: In *non-adaptive schemes* the locations of the data structure to be probed are fixed based only on the query. In contrast, in *adaptive schemes* the location of only the first probe is fixed based on the query, whereas the locations of subsequent probes can also depend upon the bits returned by prior probes. Thus, we can think of a (deterministic) adaptive scheme as a binary decision tree of depth t, where the root represents the first bit read, and the left/right children represent the location of which bit to read next, depending on whether the root represents a zero/one, respectively. Note that to be adaptive, the scheme need only have two nodes at the same level in the tree where the locations probed differ.

From the above discussion we note that any non-adaptive scheme can simulate an adaptive t-probe scheme, at the cost of doing at most an exponential $(2^t - 1)$ number of probes: simply probe all the locations in the decision tree and examine the bits to determine the path that would have been followed by the adaptive scheme.

Non-explicit or Explicit: A *non-explicit* scheme can be thought of as an existential proof. It essentially shows that there is a storage scheme that achieves the desired space bounds, such that membership queries can be answered in the desired number of probes, but does not provide intuition as to *how* to construct it efficiently, or how a query algorithm would compute which bits to probe. On the other hand, explicit schemes are divided into two categories:

1. *Explicit storage scheme*: an explicit storage scheme is one that, given $\mathcal{N}$, can compute the data structure of s bits in time polynomial in s.
2. *Explicit query scheme*: an explicit query scheme is a (n, m, s, t)-scheme where the locations of the probes to be performed by the query algorithm are computable in time polynomial in t and $\lg m$, for any query element $q \in [1, m]$.

We use the terminology *fully explicit scheme* to describe a scheme that has both of the above properties.

Note that the query algorithm is only aware of the values n, m, s, t, and the query element. So, if the scheme requires the use of prime numbers of size $\Theta(p)$ (e.g., to be used for describing a finite field, or a hash function) then these primes must be *acquired* by the query algorithm. They can be acquired in one of two ways. The first method is by computing the prime number using a predefined deterministic strategy that matches the one used by the data structure; presumably for the primes to be useful they have to be shared between the query algorithm and the data structure. The second method is for the query algorithm to read the prime number, via $\Theta(\lg p)$ probes, directly from the data structure. Indeed, there are schemes that use the second method, which we discuss later. However, if we choose the former strategy, we make the observation that, at present, all deterministic methods for computing a prime number of size $\Theta(p)$ that take $O(\texttt{polylog}(p))$ time rely on Cramér's Conjecture [47] (or similar conjectures). Without assuming any such conjecture the best time bound for a deterministic algorithm is $\tilde{O}(p^{0.525})$ [47]. Thus, unless we assume Cramér's conjecture, the query algorithm cannot make use of arbitrary prime numbers of size $\Theta(m^{\varepsilon})$, for any constant $\varepsilon > 0$, in a fully explicit scheme.

2.2 Deterministic Schemes

In this section we adopt the notation of Radhakrishnan, Shah, and Shannigrahi [41] and use $s_N(n, m, t)$ to denote the minimum space s such that there exists a deterministic non-adaptive (n, m, s, t)-scheme. We use $s_A(n, m, t)$ analogously for adaptive schemes. A summary of the results discussed in this section can be found in Table 1.

The first effort to address the worst-case behaviour of the membership problem in the bitprobe model was that of Buhrman et al. [11]. They showed that

$$\binom{m}{n} \leq \max_{i \leq nt} \binom{2s_A(n, m, t)}{i} \ . \tag{1}$$

As pointed out by Alon and Feige [1] this bound can be written as:

$$s_A(n, m, t) = \Omega(tn^{1-1/t}m^{1/t}) \ , \tag{2}$$

when $n \leq m^{1-\varepsilon}$, for some constant $\varepsilon > 0$. The bound is proved via an information theory argument. Note that this bound implies that the trivial $(n, m, m, 1)$-scheme is optimal to within constant factors. It also implies (Corollary 1.1 in [11]) that FKS-hashing makes an optimal number of probes (to within a constant factor) for schemes that use $\Theta(n \lg m)$ bits of space, when $n \leq m^{1-\varepsilon}$, for some constant $\varepsilon > 0$. Later, Pagh [37] improved the upper bound of FKS-hashing, describing a data structure that uses $\Theta(\lg(m/n) + 1)$ probes, and achieves space within a constant factor of the information theory lower bound. For the case when $n = \Theta(m)$, which is not covered by the lower bound of Equation 1, Viola [48] has shown that, for all sufficiently large m divisible by 3, $s_A(m/3, m, t) \geq \lg \binom{m}{m/3} + m/2^{O(t)} - \lg m$.

Table 1. Summary of results for deterministic membership schemes in the bitprobe model. A dagger (†) in the "Type" column indicates that the scheme is fully explicit.

Type	Bound	Constraints	Reference
Lower Bound	$\binom{m}{n} \leq \max_{i \leq nt} \binom{2^{s_A(n,m,t)}}{i}$		[11]
Lower Bound	$s_A(n,m,t) = \Omega(tn^{1-1/t}m^{1/t})$	$n \leq m^{1-\varepsilon}$ for $\varepsilon > 0$	[11,1]
Lower Bound	$s_A(m/3,m,t) \geq \lg \binom{m}{m/3} + m/2^{O(t)} - \lg m$	m divisible by 3	[48]
Lower Bound	$s_N(n,m,3) = \Omega(n^{1/2}m^{1/2}/\lg^{1/2} m)$	Valid when $n > 16 \lg m$	[1]
Upper Bound†	$s_A(n,m,\Theta(\lg m)) \leq \lg \binom{m}{n} + o(\lg \binom{m}{n})$		[10,36,38]
Upper Bound†	$s_A(n,m,\Theta(\lg(m/n)) \leq \Theta(\lg \binom{m}{n})$		[37]
Upper Bound†	$s_A(n,m,t) = O(nt'm^{1/t'})$	$t' = t - \Theta(\lg n + \lg\lg m)$ and $t > \Theta(\lg n + \lg\lg m)$	[11]
Upper Bound†	$s_A(n,m,t) = O\left(t^n m^{1/(t-n+1)}\right)$	$t > n \geq 2$	[41]
Upper Bound	$s_A(n,m,t) = O\left(ntm^{1/(t-(n-1)(t-1)2^{1-t})}\right)$	$t > n \geq 2$	[41]
Upper Bound†	$s_A(n,m,t) \leq (2^t - 1)m^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})}$	$t > 2\lfloor \lg n \rfloor$, $n \geq 3$	[29]
Upper Bound	$s_N(n,m,t) = O(ntm^{4/(t+1)})$	t is odd	[11]
Upper Bound†	$s_A(n,m,\lceil \lg\lg n \rceil + 2) = o(m)$	$n = O(m^{1/\lg\lg m})$	[40]
Upper Bound†	$s_A(n,m,\lceil \lg(n+1) \rceil + 1) \leq (n + \lceil \lg(n+1) \rceil)\sqrt{m}$		[40]
Upper Bound	$s_A(n,m,2) = O((mn\lg(\lg(m)/n))/\lg m)$	$n < \lg m$	[1]
Upper Bound	$s_A(n,m,3) = O(n^{1/3}m^{2/3})$		[1]
Upper Bound	$s_N(n,m,4) = O(n^{1/3}m^{2/3})$		[1]
Upper Bound†	$s_N(n,m,\Theta(\sqrt{n\lg n})) = O(\sqrt{mn\lg n})$		[44]

When both t and n are very small relative to m, the primary concern is with the exponent of m. In particular, the goal is to make the exponent as close to the Equation 2 lower bound of $1/t$ as possible, even though— as we shall see— it is not possible even when $n = 2$. For upper bounds, Buhrman et al. showed

$$s_N(n,m,t) = O(ntm^{4/(t+1)}) \text{ for odd } t, \text{ and} \tag{3}$$

$$s_A(n,m,t) = O(nt'm^{1/t'}) \ , \tag{4}$$

where $t' = t - \Theta(\lg n + \lg\lg m)$ and $t > \Theta(\lg n + \lg\lg m)$. The first bound is non-explicit and relies on the existence of a special kind of expander graph, whereas the second is fully explicit and generalizes FKS-hashing. The first bound shows that non-adaptivity is not too restrictive, since the exponent in the space bound for the non-adaptive scheme is only about four times larger than that of the lower bound in Equation 2. Compare this with our earlier discussion of simulating an adaptive scheme using a non-adaptive scheme that proved non-adaptivity was no more than exponentially worse. The exponent in the second bound matches the lower bound, after subtracting $\Theta(\lg n + \lg\lg m)$ probes. Note that the $\lg\lg m$ term comes from reading prime numbers stored by the data structure (see our earlier discussion regarding prime numbers).

Radhakrishnan, Raman, and Rao [40] focused on explicit deterministic constructions for small numbers of probes, describing a fully explicit scheme that, for $n = O(m^{1/\lg\lg m})$, shows $s_A(n,m,\lceil \lg\lg n \rceil + 2) = o(m)$. They also gave the following useful fully explicit bound: $s_A(n,m,\lceil \lg(n+1) \rceil + 1) \leq (n + \lceil \lg(n+1) \rceil)\sqrt{m}$. The idea is to divide the universe into buckets of size $\sqrt{m}$,

and assign $\lceil \lg(n+1) \rceil + 1$ bits to each bucket, indicating the rank of the largest element stored in the bucket. Rao [44] described several non-adaptive schemes. His main non-adaptive result is a fully explicit scheme[3] showing that: $s_N(n, m, \Theta(\sqrt{n \lg n})) = O(\sqrt{mn \lg n})$.

Later, Alon and Feige [1] improved the result of Radhakrishnan, Raman, and Rao (that we stated first), for the case when $n < \lg m$, by providing an explicit storage scheme showing $s_A(n, m, 2) = O((mn \lg(\lg(m)/n))/\lg m)$. The storage scheme makes use of an explicit construction of regular bipartite graphs of high girth, combined with an application of Hall's Theorem. They also describe non-explicit schemes showing that $s_A(n, m, 3) = O(n^{1/3}m^{2/3})$, and $s_N(n, m, 4) = O(n^{1/3}m^{2/3})$, based on similar graph and hypergraph theoretic techniques. They note that if the number of probes is increased to a larger constant, these non-explicit schemes can be converted into explicit storage schemes. Finally, they showed that if $n > 16 \lg m$, $s_N(n, m, 3) = \Omega(n^{1/2}m^{1/2}/\lg^{1/2} m)$. This lower bound is proved using techniques from extremal hypergraph theory[4].

Radhakrishnan, Shah, and Shannigrahi [41] presented two results. The first is a fully explicit scheme that, for $t > n \geq 2$, shows

$$s_A(n, m, t) = O\left(t^n m^{1/(t-n+1)}\right) \quad . \tag{5}$$

The second result is a non-explicit scheme that, for $t > n \geq 2$, shows

$$s_A(n, m, t) = O\left(ntm^{1/(t-(n-1)(t-1)2^{1-t})}\right) \quad . \tag{6}$$

Equation 5 is proved using a recursive unary encoding scheme, whereas Equation 6 is proved by a probabilistic argument. The probabilistic argument essentially sends all the elements in $\mathcal{N}$ into one table from a set of 2^{t-1} possible tables. Each table has its own hash function, which is assumed to assign elements to table entries uniformly at random. The argument shows that false positives can be avoided by deleting at most half the elements from the universe. Thus, by remapping to a universe of size $2m$, we can store a universe of size m using this scheme. Note that although both Equations 5 and 6 approach the optimal space bound exponent for fixed n and sufficiently large t, the second bound is significantly stronger when n is close to t.

Recently, Lewenstein, Munro, Nicholson and Raman [29] reduced the gap between between existing explicit and non-explicit schemes by describing a fully explicit scheme, for $t \geq 2\lfloor \lg n \rfloor + 1$, showing that:

$$s_A(n, m, t) \leq (2^t - 1)m^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})} \quad . \tag{7}$$

The scheme uses a recursive decomposition of the universe into buckets, assigning two bits for each bucket.

[3] Although some of the schemes presented by Rao [44] require Cramér's conjecture in order to be fully explicit, the main result (Corollary 4.2.7) does not.

[4] See also a survey by Blue [4] of the techniques used by Alon and Feige for non-adaptive upper and lower bounds.

Table 2. Summary of results for deterministic membership schemes in the bitprobe model, for the special case when $n = 2$. A dagger (†) indicates that the scheme is fully explicit.

Scheme	Lower Bound	Upper Bound	Constraints	Reference
$s_A(2, m, 1)$	$\Omega(m)$	$O(m)$†		[11]
$s_A(2, m, 2)$	$\Omega(m^{4/7})$	$O(m^{2/3})$†	$\Omega(m^{2/3})$ lower bound for a restricted class	[41,40]
$s_A(2, m, 3)$	$\Omega(m^{1/3})$	$O(m^{2/5})$†		[11,41,29]
$s_N(2, m, 2)$	$\Omega(m)$	$O(m)$†		[11]
$s_N(2, m, 3)$	$\Omega(\sqrt{m})$	$O(\sqrt{m})$†		[11,41]
$s_A(2, m, t)$	$\Omega(m^{1/t})$	$O(t^2 m^{1/(t-1)})$†		[41]
$s_A(2, m, t)$	$\Omega(m^{1/t})$	$O\left(tm^{1/(t-(t-1)2^{1-t})}\right)$		[41]
$s_A(2, m, t)$	$\Omega(m^{1/t})$	$(2^t - 1)m^{1/(t-2^{2-t})}$†		[29]

The First Non-trivial Special Case, When $n = 2$: If $n = 1$ then the scheme we discussed earlier can achieve the bound $s(1, m, t) = tm^{1/t}$, which matches the exponent-of-m in the lower bound of Equation 2. The idea is to store t characteristic bit vectors; one for each $1/t$-th fraction of the bits of the sole element to be stored. The first non-trivial special case that has been studied heavily, though is still not very well understood, is when $n = 2$. Equation 2 implies $s_A(2, m, 1) = \Omega(m)$, which raises the question of how the bound behaves for other values of t. We summarize the results for this special case in Table 2.

Buhrman et al. [11] showed that for non-adaptive schemes, adding a second probe does not improve the space bound over one probe, i.e., $s_N(2, m, 2) = \Omega(m)$. However, if adaptivity is permitted it is possible to get a $o(m)$ space bound. This shows a strict separation between the power of adaptive and non-adaptive probes. However, rather surprisingly, even the 2-probe case is still not completely settled! For upper bounds, Radhakrishnan, Raman, and Rao [40] showed that $s_A(2, m, 2) = O(m^{2/3})$ via a subtle fully explicit scheme. They also proved a matching lower bound for a restricted class of schemes, but could not show it in general. Later, Radhakrishnan, Shah, and Shannigrahi [41] showed that $s_A(2, m, 2) = \Omega(m^{4/7})$ by modelling the problem as a graph, and making a forbidden subgraph argument. Interestingly, this is the only lower bound for adaptive schemes that beats the bound of Equation 2 when $n \ll m$. They also conjectured that the true lower bound asymptotically matches the $O(m^{2/3})$ upper bound.

For $t = 3$, the complexity of non-adaptive schemes is settled [11,41] asymptotically: $s_N(2, m, 3) = \Theta(\sqrt{m})$. However, for adaptive schemes there are no lower bounds other than that of Equation 2. Plugging $n = 2$ and $t = 3$ into Equation 6 implies there is a non-explicit scheme with $s_A(2, m, 3) = O(m^{2/5})$, whereas the fully explicit scheme of [40, Theorem 1] only yields the bound $s_A(2, m, 3) = O(m^{1/2})$. For general values of $t \geq 3$ in the $n = 2$ case, the scheme of Equation 6 yields the following bound:

$$s_A(2, m, t) = O\left(tm^{1/(t-(t-1)2^{1-t})}\right) \quad . \tag{8}$$

Radhakrishnan, Shah, and Shannigrahi [41] pointed out that by using limited independence their scheme can be turned into explicit storage scheme. However, they left finding a fully explicit scheme that matches their non-explicit bound in this case as an open problem. This open problem was recently solved by Lewenstein, Munro, Nicholson and Raman [29], who describe a fully explicit scheme that matches Equation 8 when $t = 3$, and improves upon it when $t > 3$ and $2^t = o(m^\varepsilon)$, for any constant $\varepsilon > 0$. They show that:

$$s_A(2, m, t) \leq (2^t - 1)m^{1/(t-2^{2-t})} \quad . \tag{9}$$

The scheme is a natural generalization of the earlier 2-element 2-probe scheme of Radhakrishnan, Raman, and Rao [40], but also uses a property of modular arithmetic in a subtle way. The construction of a fully explicit scheme matching the bound of Equation 6 for $n \geq 3$ continues to be open.

2.3 Randomized Schemes

For randomized schemes, upper and lower bounds for the membership problem are better understood. As in the deterministic setting, the first results were those of Buhrman et al [11], who showed that there is a two-sided ε-error $(n, m, O((n \lg m)/\varepsilon^2), 1)$-scheme, for $0 < \varepsilon \leq 1/4$. The scheme is non-explicit and based on the existence of certain kinds of expander graphs, and a special kind of cover-free family of sets called a Nisan-Wigderson design [33]. They also show that the scheme is *almost* optimal, by proving that any two-sided ε-error scheme with $n/m^{1/3} \leq \varepsilon \leq 1/4$ must occupy $\Omega((n \lg m)/(\varepsilon \lg \varepsilon))$ bits of space. Using similar techniques, they also prove the existence of (i.e., show there is a non-explicit) one-sided ε-error $(n, m, O((n^2 \lg m)/\varepsilon^2), 1)$-scheme, for $0 < \varepsilon \leq 1/4$.

At the cost of increasing the space by a $(\lg m)$-factor they show how to make their one-sided error scheme fully explicit[5]. Furthermore, they show that this quadratic-in-n space term is necessary for one-probe scheme, by proving a lower bound of $\Omega(n^2/(\varepsilon^2 \lg(n/\varepsilon)))$ bits of space for any one-probe, one-sided ε-error scheme, where $n/m^{1/3} \leq \varepsilon \leq 1/4$. Both the one-sided and two-sided lower bounds are proved using lower bounds for cover-free families of sets. Finally, they show that if a constant $t > 1$ probes are allowed, there is a one-sided ε-error $(n, m, O(n^{1+\delta} \lg m), t)$-scheme, for any constant $\delta > 0$.

Later, Ta-Shma [46] gave an explicit construction of a two-sided ε-error $(n, m, n2^{O(((\lg \lg m)/\varepsilon)^3)}, 1)$-scheme. This improves upon the Buhrman et al.'s scheme when $\lg n = \Omega((\lg \lg m + \lg(1/\varepsilon))^3)$.

3 Integer Counters with Increment and Decrement

In this section, we consider the problem of representing integers using close to the optimal number of bits, while also supporting increment and decrement operations efficiently. If we use the standard binary representation of integers, an

[5] We note that their scheme uses finite fields of order $\Theta(n \lg m)$. Thus, if $n = \Omega(m^\varepsilon)$ for any $\varepsilon > 0$, then this scheme is only fully explicit assuming Cramér's Conjecture.

integer in the range $[0, \ldots, 2^n - 1]$ can be represented using n bits (which is optimal), but supporting increment and decrement operations in this representation requires reading and modifying $\Theta(n)$ bits in the worst case. Using Gray codes [25] to represent the integers, one can reduce the number of bits that need to be modified during each increment or decrement operation to 1, but it still requires $\Theta(n)$ bits to be read in the worst case. On the other hand, one can show that any deterministic scheme that supports increment or decrement operations on an n-bit representation of integers in the range $[0, \ldots, 2^n - 1]$ is required to read $\lg n + 1$ bits and modify $\Omega(1)$ bits in the worst case (see Section 3.2). In this section we survey the integer representations whose performance is close to these lower bounds. We also briefly discuss the representations supporting addition and subtraction operations efficiently.

Applications of representing integers using a *positional number system* (such as the standard binary representation) to *binomial queues* has been first studied by Vuillemin [49]. *Redundant binary counters* studied by Clancy and Knuth [12] and by various others have been used to making *purely functional* and *persistent* data structures more efficient. Their applications to data structures such as *heaps* and *random-access lists* are explored by Okasaki [34] (see also [16] for more applications). An application of maintaining a counter under increment/decrement operations in $O(1)$ amortized time in the bitprobe model to a space-efficient representation of a dynamic multiset (which in turn can be used to obtain an "optimal dynamic Bloom filter") has been described in [35].

3.1 Problem Definition and Notation

We define a *counter* as any data structure which represents integers modulo L, for any positive integer L, using d bits where $2^d \geq L$. We refer to d as the *dimension* of the counter. The data structure supports two operations called *increment* and *decrement* where increment (resp. decrement) refers to changing the counter to represent its next (resp. previous) value modulo L. The *space-efficiency* of a counter is the ratio $L/2^d$. A counter with space-efficiency equal to one is referred to as a *space-optimal counter*, while a counter with space-efficiency less than one as a *redundant counter*. For counters of dimension d with space-efficiency e, we define a (d, e, r, w)-*scheme* as a description of the increment and decrement operations which can be performed by reading r bits and writing w bits in the worst-case.

We define a *code* as any cyclic sequence of 2^d distinct (i.e., all possible) d-bit strings. We use $X = x_d x_{d-1} \ldots x_1$ to denote a bit string in a code. We use r and w to denote the number of bits read and written respectively during any (increment or decrement) operation. The average number of bits read (written) to perform increment/decrement is computed by adding the total number of bits read (written) to perform L increments/decrements starting from zero, and dividing this by L.

3.2 Counters with Increment and Decrement

The standard n-bit binary representation of integers in the range $[0, \ldots, 2^n - 1]$ gives rise to a code, which we refer to as the Standard Binary Code (SBC). It uses n bits $x_n x_{n-1} \ldots x_1$ to represent an integer in the range $[0, \ldots, 2^n - 1]$, and gives an $(n, 1, n, n)$-*scheme*. (Although its worst-case performance is bad, the average number of bits read and written to perform increment/decrement is only 2.) A Gray code is any code in which successive bit strings in the sequence differ in exactly one position. Gray codes have been studied extensively owing to their utility in digital circuits [45]. The problem of generating Gray codes has also been discussed by Knuth [28]. The Binary Reflected Gray Code (BRGC) [25] gives an $(n, 1, n, 1)$-*scheme* for increment/decrement. Bose et al. [6,26] developed a different Gray code called Recursive Partition Gray Code (RPGC) to improve the average-case read complexity. A counter of dimension n using RPGC requires on average $O(\lg n)$ reads to perform increment/decrement operations.

For redundant counters, Munro and Rahman [43] gave an $(n + 1, 1/2, \lg n + 4, 4)$-*scheme*, i.e., using one additional bit of space, the worst-case number of bits read is reduced from n to $\log n + 4$. This scheme represents the counter using an n-bit BRGC in addition to a bit indicating the "delayed bit flip". It partitions the BRGC into two parts consisting of the lower $\lg n$ bits and the upper $n - \lg n$ bits. During an increment or decrement operation, updates (i.e., bit flips) in the lower part are done immediately, but updates in the upper part are done in a delayed manner, using the additional bit to keep track of this information. For efficiency close to one, Bose et al. [6] describe an $(n + t \lg n, 1 - O(n^{-t}), O(t \lg n), 3)$-*scheme* for any parameter $t > 0$. Their scheme uses RPGC, and a technique to improve the space-efficiency at the cost of increased read complexity. Fredman [18] provided a redundant $(O(n), 1/2^{O(n)}, O(\lg n), 1)$-*scheme*, that with a constant factor space overhead supports increments with a logarithmic number of bit reads and a single bit write. Brodal et al. [7] improved some of these schemes, as summarized in Table 3. The main ingredients of the schemes of Brodal et al. [7] are: (i) using *BRGC or RPGC* to store the 'lower part' while using SBC to store the 'upper part'; (ii) using the least significant bits of the upper part to store the additional 'indicator bit', achieving a trade-off between the space efficiency and read complexity; and (iii) showing a one-bit trade-off between the read and write complexities.

Lower Bound. The increment/decrement algorithms in any (d, e, r, w)-*scheme* can be represented by a decision tree of height r in which the nodes are associated with the positions in the d-bit sequence representing the counter. At each leaf in the decision tree, we store the bits that are modified (written) along with their positions. Now, if we have a $(n, 1, r, w)$-*scheme* with $r \leq \lfloor \lg n \rfloor$, then the number of nodes in the decision tree is at most $n - 1$. Thus, at most $n - 1$ of the n bits in the counter representation are ever read during any increment (or decrement) operation. The bit(s) that is not read has to be modified (written) without reading its value – otherwise, the efficiency will be less than 1. Since without reading a bit, we can only set it to a fixed value (either 0 or 1), there will be at least two different configurations which will be modified to the same

Table 3. Summary of results for increment/decrement counters

Dimension	Space-efficiency	Bits read (r)		Bits written (w)	Inc. & Dec.	Ref.
		Average-case	Worst-case	Worst-case		
n	1	$2 - 2^{1-n}$	n	n	Y	Binary
		n		1	Y	[25]
		$6 \lg n$		1	Y	[6]
		$O(\lg^{(2c-1)} n)$		c	N	[6]
		$O(\lg n)$	$n-1$	3	Y	[7]
$n+1$	$1/2$	$O(1)$	$\lg n + 4$	4	Y	[43]
$n + \lg n$	$2/n - O(2^{-n})$	3	$\lg n + 1$	$\lg n + 1$	Y	[17]
$n + t \lg n$	$1 - O(n^{-t})$	$O(\lg^{(2c)} n)$	$O(t \lg n)$	$2c + 1$ $(c \geq 1)$	N	[6]
$O(n)$	$1/2^{O(n)}$	$O(\lg n)$	$O(\lg n)$	1	N	[18]
$n + t$	$\geq 1 - \frac{1}{2^t}$	$O(\log \log n)$	$\log n + t + 1$	3	N	[7]
			$\log n + t + 2$	2		
			$\log n + t + 3$	1		
			$\log n + t + 2$	3	Y	
			$\log n + t + 3$	2		

configuration after the operation, which violates the properties of a code (that each configuration has a unique predecessor/successor). Hence, any $(n, 1, r, w)$-*scheme* has $r \geq \lg n + 1$.

3.3 Supporting Addition and Subtraction

To add an integer M to an integer N, where M and N are represented in SBC or BRGC using m and n bits respectively, where $m \leq n$, we need $O(n)$ time in the worst-case. Munro and Rahman [43] gave a representation which uses $n + O(\log^2 n)$ bits to represent an integer in the range $[0, \ldots, 2^n - 1]$. Peforming addition or subtraction using this representation takes reading $O(m + \log n)$ bits and writing $O(m)$ bits in the worst-case. Brodal et al. [7] improved the efficiency of this scheme, and also gave other schemes exhibiting different trade-offs between the efficiency and the read complexity. Table 4 summarizes these results.

Table 4. Summary of results for addition and subtraction operations

Efficiency	Read	Write	Reference
1	$n + m$	n	Binary
$\Theta(1/n^{\log n})$	$O(m + \log n)$	$O(m)$	[43]
$\geq (1 - 1/2^t)^{\log n}$	$O(m + t \log n)$	$O(m)$	[7]
$\Omega(1/n)$	$O(m + \log n)$		
$\Theta(1)$	$O(m + \log n \log \log n)$		

4 Other Problems in the Bitprobe Model

Pătraşcu and Tarniţă [39] consider lower bounds in the bitprobe model for several problems, such as dynamic partial sums and dynamic connectivity. They also consider upper bounds for the *greater-than* problem. In the greater-than problem the algorithm receives a number $x_1 \in [1, m]$ as input, and after seeing it is allowed to flip t_u bits in memory. Later, the algorithm receives another number $x_2 \in [1, m]$ and is allowed to probe t_q bits of memory to decide whether $x_1 > x_2$. Fredman [20] had shown a lower bound of $t_u + t_q = \Omega(\lg m / \lg \lg m)$ for this problem. Pătraşcu and Tarniţă [39] show a matching upper bound of $t_q = t_u = O(\lg m / \lg \lg m)$ for a generalization of the greater-than problem called the *coloured predecessor problem*[6]. This problem is the same as the usual predecessor problem, except that each element stores a "colour" drawn from a fixed set of constant size. Instead of reporting the predecessor of an arbitrary position $x_1 \in [1, m]$, we are asked to report the colour of the predecessor, if one exists. Later, Mortensen, Pagh, and Pătraşcu [32] returned to the greater-than problem and proved tight trade-off bounds of: $t_u = \Theta(\log_{t_q} m)$ and $2^{t_q} = \Theta(\log_{t_u} m)$. Finally, Rahman [42] gave upper bounds for the coloured predecessor problem, showing that $t_q = O(k^2(\lg m / \lg \lg m)^{1/k})$ and $t_u = O(k \lg m / \lg \lg m)$ could be achieved simultaneously, for any positive integer k.

Elias and Flower [15], and, later, Miltersen [30] studied the so-called *full problem*, in which the storage scheme represents some element $x \in D$, and answers the query, "Is $x \in D_0$?", where D_0 is any subset of D. Yao and Yao [51] studied a variant of the membership problem that, given a query element q, decides whether there is an element stored in the data structure matching q, except with possibly 1 bit flipped. They showed this problem was solvable using $\Theta(\lg m \lg \lg n)$ probes into a data structure occupying $\Theta(n \lg m \lg n)$ bits of space. This result was later improved by Brodal and Venkatesh [8], who described a data structure that occupies $O(n \lg m \lg \lg m)$ bits and uses $\lg m$ probes to answer the query. Demaine and López-Ortiz [13] proved lower bounds on the size of indexing structures for text retrieval in the bitprobe model. Gál and Miltersen [21] examined the problem of evaluating polynomials in the bitprobe model. Viola [48] showed, using information theoretic arguments, that to represent m ternary values (trits) such that an individual trit can be read using t (adaptive) probes requires space $(\lg 3)m + m/2^{O(t)}$ bits. This matches an upper bound by Dodis, Pătraşcu and Thorup [14], up to the constant factor in the big-Oh.

Golynski [23] studied the problem of supporting *rank* and *select* queries on bit vectors. Rank queries ask to count the number of 1 (or 0) bits in an arbitrary prefix of the bit vector, and select queries ask for the position of the j-th left-to-right occurence of 1 (or 0) in the bit vector. Golynski proved lower bounds showing that, if the bit vector must be stored explicitly, the data structures stored in addition to the bit vector must occupy $\Omega(m \lg \lg m / \lg m)$ bits (for arbitrary n) in order to answer rank and select queries in $\Theta(\lg m)$ probes. *Density-sensitive* bounds (i.e., bounds that are parameterized in terms of n)

[6] Not to be confused with the coloured generalized union-split-find problem (c.f., [22]).

have also been proved [23,24]. Finally, the recent work of Lewenstein et al. [29] uses techniques from the membership problem to attack the problem of supporting rank queries, range counting queries (i.e., counting the number of 1 bits in an arbitrary range a bit vector), and range emptiness queries (i.e., returning whether a single one bit exists in an arbitrary range), on bit vectors where both n and t are small relative to m.

5 Open Problems

We conclude with a list of open problems:

1. Improve the lower bounds for the membership problem for the case of $n = 2$. In particular, is the $(n, m, 3m^{2/3}, 2)$-scheme of Radhakrishnan, Raman, and Rao [40] optimal? What can be said for higher values of t?
2. Close the gap between explicit schemes and non explicit schemes for membership for $n \geq 3$ elements. Can any of the non-explicit schemes (such as those of Radhakrishnan et al. [41]) be made fully explicit?
3. Improve the read complexity for space-optimal counters. The best known representation in terms of read complexity to support increment/decrement operations, by Brodal et al. [7], requires reading $n - 1$ bits. This representation is obtained a simple generalization of a $(4, 1, 3, 2)$-*scheme*, which in turn is obtained through an exhaustive search. Obtaining a representation with $O(1)$ efficiency that supports addition and subtraction operations with optimal read and write complexities is another interesting open problem.

Acknowledgements. We would like to thank the anonymous referees for their helpful comments and corrections to the preliminary version of this paper. The first author was supported in part by a David Cheriton Scholarship, and a Derick Wood Graduate Scholarship. The third author was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant number 2012-0008241).

References

1. Alon, N., Feige, U.: On the power of two, three and four probes. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, pp. 346–354. Society for Industrial and Applied Mathematics, Philadelphia (2009)
2. Alt, H., Mehlhorn, K., Munro, J.I.: Partial match retrieval in implicit data structures. Inf. Process. Lett. 19(2), 61–65 (1984)
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
4. Blue, R.: The Bit Probe Model for Membership Queries: Non-Adaptive Bit Queries. Master's thesis, University of Maryland (2009)

5. Borodin, A., Fich, F.E., Meyer auf der Heide, F., Upfal, E., Wigderson, A.: A tradeoff between search and update time for the implicit dictionary problem. Theor. Comput. Sci. 58, 57–68 (1988)
6. Bose, P., Carmi, P., Jansens, D., Maheshwari, A., Morin, P., Smid, M.: Improved methods for generating quasi-Gray codes. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 224–235. Springer, Heidelberg (2010)
7. Brodal, G.S., Greve, M., Pandey, V., Rao, S.S.: Integer representations towards efficient counting in the bit probe model. In: Ogihara, M., Tarui, J. (eds.) TAMC 2011. LNCS, vol. 6648, pp. 206–217. Springer, Heidelberg (2011)
8. Brodal, G.S., Venkatesh, S.: Improved bounds for dictionary look-up with one error. Information Processing Letters 75(1), 57–59 (2000)
9. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2004)
10. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM Journal on Computing 28(5), 1627–1640 (1999)
11. Buhrman, H., Miltersen, P.B., Radhakrishnan, J., Venkatesh, S.: Are bitvectors optimal? SIAM Journal on Computing 31(6), 1723–1744 (2002)
12. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Tech. rep., Stanford, CA, USA (1977)
13. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. Journal of Algorithms 48(1), 2–15 (2003)
14. Dodis, Y., Pătraşcu, M., Thorup, M.: Changing base without losing space. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, pp. 593–602. ACM (2010)
15. Elias, P., Flower, R.A.: The complexity of some simple retrieval problems. Journal of the ACM (JACM) 22(3), 367–379 (1975)
16. Elmasry, A., Jensen, C., Katajainen, J.: Two skew-binary numeral systems and one application. Theory Comput. Syst. 50(1), 185–211 (2012)
17. Frandsen, G.S., Miltersen, P.B., Skyum, S.: Dynamic word problems. J. ACM 44(2), 257–271 (1997)
18. Fredman, M.L.: Observations on the complexity of generating quasi-Gray codes. SIAM Journal on Computing 7(2), 134–146 (1978)
19. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. Journal of the ACM (JACM) 31(3), 538–544 (1984)
20. Fredman, M.: The complexity of maintaining an array and computing its partial sums. J. ACM 29(1), 250–260 (1982)
21. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379(3), 405–417 (2007)
22. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Transactions on Algorithms 5(3), 28:1–28:51 (2009)
23. Golynski, A.: Optimal lower bounds for rank and select indexes. Theoretical Computer Science 387(3), 348–359 (2007)
24. Golynski, A., Orlandi, A., Raman, R., Rao, S.S.: Optimal indexes for sparse bit vectors. Algorithmica, 1–19 (2011)
25. Gray, F.: Pulse code communications. U.S. Patent (2632058) (1953)
26. Jansens, D.: Improved Methods for Generating Quasi-Gray Codes. Master's thesis, School of Computer Science, Carleton University (April 2010)
27. Knuth, D.E.: Sorting and searching: The art of computer programming III. Addison-Wesley, Reading (1973)

28. Knuth, D.E.: The Art of Computer Programming. Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming), vol. 4. Addison-Wesley Professional (2005)
29. Lewenstein, M., Munro, J.I., Nicholson, P.K., Raman, V.: Explicit data structures in the bitprobe model (submitted manuscript, 2013)
30. Miltersen, P.B.: The bit probe complexity measure revisited. In: Enjalbert, P., Wagner, K.W., Finkel, A. (eds.) STACS 1993. LNCS, vol. 665, pp. 662–671. Springer, Heidelberg (1993), http://portal.acm.org/citation.cfm?id=646509.694667
31. Minsky, M.L., Papert, S.: Perceptrons: An Introduction to Computational Geometry. The MIT Press (1969)
32. Mortensen, C.W., Pagh, R., Patrascu, M.: On dynamic range reporting in one dimension. In: Gabow, H.N., Fagin, R. (eds.) STOC, pp. 104–111. ACM (2005)
33. Nisan, N., Wigderson, A.: Hardness vs randomness. Journal of Computer and System Sciences 49(2), 149–167 (1994)
34. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
35. Pagh, A., Pagh, R., Rao, S.S.: An optimal bloom filter replacement. In: SODA, pp. 823–829. SIAM (2005)
36. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM Journal on Computing 31(2), 353–363 (2001)
37. Pagh, R.: On the cell probe complexity of membership and perfect hashing. In: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing, STOC 2001, pp. 425–432. ACM, New York (2001)
38. Pătraşcu, M.: Succincter. In: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science, pp. 305–313. IEEE Computer Society (2008)
39. Pătraşcu, M., Tarniţă, C.E.: On dynamic bit-probe complexity. Theoretical Computer Science 380(1), 127–142 (2007)
40. Radhakrishnan, J., Raman, V., Rao, S.S.: Explicit deterministic constructions for membership in the bitprobe model. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 290–299. Springer, Heidelberg (2001)
41. Radhakrishnan, J., Shah, S., Shannigrahi, S.: Data structures for storing small sets in the bitprobe model. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 159–170. Springer, Heidelberg (2010)
42. Rahman, M.Z.: Data Structuring Problems in the Bit Probe Model. Master's thesis, University of Waterloo (2007)
43. Rahman, M.Z., Munro, J.I.: Integer representation and counting in the bit probe model. Algorithmica 56(1), 105–127 (2010)
44. Rao, S.S.: Succinct Data Structures. Ph.D. thesis, Institute of Mathematical Sciences (2001)
45. Savage, C.: A survey of combinatorial Gray codes. SIAM Review 39, 605–629 (1996)
46. Ta-Shma, A.: Storing information with extractors. Information Processing Letters 83(5), 267–274 (2002)
47. Tao, T., Croot III., E., Helfgott, H.: Deterministic methods to find primes. Mathematics of Computation 81(278), 1233–1246 (2012)
48. Viola, E.: Bit-probe lower bounds for succinct data structures. SIAM Journal on Computing 41(6), 1593–1604 (2012)
49. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
50. Yao, A.C.: Should tables be sorted? J. ACM 28(3), 615–628 (1981)
51. Yao, A.C., Yao, F.F.: Dictionary look-up with one error. Journal of Algorithms 25(1), 194–202 (1997)

Succinct Representations of Ordinal Trees

Rajeev Raman[1] and S. Srinivasa Rao[2,⋆]

[1] Department of Computer Science, University of Leicester, UK
r.raman@mcs.le.ac.uk
[2] School of Computer Science and Engineering, Seoul National University, S. Korea
ssrao@cse.snu.ac.kr

Abstract. We survey *succinct* representations of ordinal, or rooted, ordered trees. Succinct representations use space that is close to the appropriate information-theoretic minimum, but support operations on the tree rapidly, usually in $O(1)$ time.

1 Introduction

An increasing number of applications such as information retrieval, XML processing, data mining etc. require large amounts of tree-structured data to be represented in main memory. Unfortunately, the memory consumption of classical ways of representing such data is often prohibitively large: for example, the standard in-memory representation of XML documents in a number of common programming languages such as C++, Java or Python requires memory an order of magnitude more than the size of the XML document on disk [40]. A similar situation arises when attempting to build a *suffix tree* data structure to index a collection of documents [31]. The large memory usage of standard tree representations severely affects the scalability of such applications.

This problem has led to the intensive study of *succinct*, or highly space-efficient, tree representations. This survey focusses on succinct *ordinal* trees, i.e. rooted, ordered trees. The standard way to represent an ordinal tree is to store pointers from each node to its first child and its next sibling; this represents the structure of the tree using $2n$ pointers, or $2n\lceil \lg n \rceil$ bits[1]. However, this is significantly more space than necessary. As there are $C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1}$ ordinal trees on n nodes, there is an information-theoretic worst-case lower bound of $\lg C_{n-1} = 2n - O(\lg n)$ bits on any ordinal tree representation, and there is a trivial encoding of an n-node ordinal tree as an integer of $\lceil \lg C_{n-1} \rceil$ bits: the tree is encoded by its position in any systematic enumeration of all n-node ordinal trees. It has been known for several years (see e.g. [30] and references therein)

⋆ Work partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant number 2012-0008241).

[1] We use lg to denote the logarithm base 2.

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 319–332, 2013.

that there are many—less brute-force—ways of encoding the structure of an n-node ordinal tree in $2n + O(1)$ bits. Thus, a succinct tree representation can potentially offer a $\Theta(\lg n)$ factor space savings in the worst case.

At first sight it appears that it would be difficult to perform operations quickly on such a compact representation: in particular, converting a compact representation to a standard one to perform operations defeats the purpose of considering the compact representation. However, standard representations also have limitations: for instance, in the basic first-child next-sibling representation described above, navigating to the parent of a node takes $O(n)$ time, unless a parent pointer is added, increasing the space usage to $3n\lceil \lg n \rceil$ bits from $2n\lceil \lg n \rceil$ bits. Determining the size of the subtree rooted at a node in $O(1)$ time would require storing an additional $\Theta(n \lg n)$ bits, and adding more complex functionality such as level-ancestor [6] or lowest common ancestor [1] would add a further $\Theta(n \lg n)$ bits each. In short, while standard representations can support a number of navigational and other queries in 'linear' space ($\Theta(n)$ words or $\Theta(n \lg n)$ bits), as the set of operations increases, so does the constant within the $\Theta(n)$. To mitigate this space blow-up, real-world software libraries can sometimes attempt to minimize the constant factor in the space usage by omitting some pointers, with disastrous results [14, p144]. On the other hand, as we will see, a number of succinct representations have been developed that use only $2n + o(n)$ bits of space, and support a wide variety of navigational and other operations in $O(1)$ time on the word RAM model. A key advantage of succinct representations is that as their functionality increases, usually only the constant in the $o(n)$ increases. Thus, adding new functionality has very little additional cost.

Furthermore, as a number of implementations and empirical evaluations have demonstrated (see Section 5), the practical performance of succinct ordinal tree representations is excellent, both in terms of memory usage and speed. They have also only been integrated successfully into a range of applications [13,3,45]. Furthermore, they are currently available in well-documented open-source libraries such as SDSL [22] and succinct [37].

The paper is organised as follows. After some preliminaries, we discuss the main succinct representations of ordinal trees. We then cover specialized topics including dynamization and the redundancy of tree representations, and conclude with implementations, empirical evaluations and practical issues.

2 Preliminaries

2.1 Terminology

Succinct ordinal trees can be *dynamic*, i.e. allow changes to the tree, or *static*. Our focus will be more on static trees, though we address dynamization issues in Section4.1. For static trees, we assume that the input tree is pre-processed to obtain a succinct representation: we do not normally focus on the time and space requirements for the pre-processing (which is, however, an important issue in practice). The space used by the succinct representation of an n-node tree can be expressed as $I(n) + R(n)$ bits where $I(n) = \lceil \lg C_{n-1} \rceil = 2n - O(\lg n)$ is the

information-theoretic lower bound and $R(n) \geq 0$ is called the *redundancy*. The redundancy is a term of great practical significance, and studying the trade-off between redundancy and query time is of fundamental importance.

Succinct tree representations are of two kinds: *systematic* (also known as *succinct indices*) and *non-systematic*. In systematic tree representations, we separate the storage for the encoding of the tree (which usually takes $2n + O(1)$ bits) from the *succinct index*, an auxiliary data structure created during preprocessing to help answer queries rapidly. A query is answered by reading $O(1)$ words (each of $O(\lg n)$ consecutive bits) from the query and the succinct index. In a systematic representation, the redundancy is essentially the size of the succinct index. Separating the 'structure of the tree' from auxiliary data structures not only makes it easier to show lower bounds on the redundancy needed to achieve a particular query time [19,24], it also has several algorithmic advantages [4]. Non-systematic representations do not have this conceptual separation, but it is known that the redundancy needed to achieve a particular query time is lower for non-systematic representations than for systematic representations.

2.2 Features of Succinct Tree Representations

Succinct tree representations require some care in their use. Firstly, the numbering of nodes is based upon the position of the representation of the nodes in the encoding of the tree. This is particularly problematic in the dynamic case, since a single update may change the numbers of every node in the tree. Furthermore, certain operations can be implemented efficiently in one tree encoding, but are hard or impossible to implement in another. Since different encodings number nodes differently, it may not be possible to come up with a succinct representation that supports the union of the operations of two encodings. Finally, this way of numbering nodes sometimes means that the 'natural' numbering of nodes is a sequence of non-consecutive integers from the range $\{1, \ldots, 2n + O(1)\}$.

2.3 FIDs

Given a subset S from a universe U, we define a *fully indexable dictionary* (FID, from now on) on S to be any data structure that supports the following operations on S in constant time, for any $x \in U$, and $1 \leq i \leq |U|$:

- $rank(x)$: return the number of elements in S that are less than x,
- $select(i, S)$: return the i-th smallest element in S, and
- $select(i, \bar{S})$: return the i-th smallest element in $U \setminus S$.

Lemma 1. *[41] Given a subset of size n from the universe $[m]$, there is a FID that uses $\lg \binom{m}{n} + O(m \lg \lg m / \lg m)$ bits.*

Given a bitvector B, we define its FID to be the FID for the set S where B is the characteristic vector of set S.

3 Ordinal Tree Representations

In this section we describe the main succinct ordinal tree representations.

3.1 Operations on Ordinal Trees

We first introduce a set of useful operations that are supported by various ordinal tree representations. Given an ordinal tree, we consider the following operations:

- `parent`(x): returns the parent of node x;
- `child`(x, i): returns the i-th child of node x, for $i \geq 1$;
- `child_rank`(x): returns the number of left siblings of node x plus 1;
- `depth`(x): returns the depth of node x;
- `level_ancestor`(x, i): returns the ancestor of node x that is i levels above x, for $i \geq 0$ (if x is at depth d, it returns the ancestor of x at depth $d - i$);
- `desc`(x): returns the number of descendants of node x;
- `degree`(x): returns the degree of node x;
- `height`(x): returns the height of the subtree rooted at node x;
- `LCA`(x, y): returns the lowest common ancestor of the nodes x and y;
- `left_leaf`(x) (`right_leaf`(x)): returns the leftmost (rightmost) leaf of the subtree rooted at node x;
- `leaf_rank`(x): returns the number of leaves up to node x in preorder;
- `leaf_select`(i): returns the i-th leaf among all the leaves from left to right;
- `leaf_size`(x): returns the number of leaves in the subtree rooted at node x;
- $\texttt{rank}_{\text{PRE/POST/DFUDS/LEVEL}}(x)$: returns the position of node x in the preorder, postorder, DFUDS order or level order traversal of the tree;
- $\texttt{select}_{\text{PRE/POST/DFUDS/LEVEL}}(j)$: returns the j-th node in the preorder, postorder or DFUDS order or level order traversal of the tree;
- `level_left`(i) (`level_right`(i)): returns the first (last) node visited in a preorder traversal among all the nodes whose depths are i;
- `level_succ`(x) (`level_pred`(x)): the *level successor* (*predecessor)* of node x, i.e. the node visited immediately after (before) node x in a preorder traversal among all the nodes that are at the same level as node x.

3.2 Jacobson's Representation

Jacobson [26] proposed the first succinct tree representation for ordinal trees. His representation is based on the *level order unary degree sequence* (LOUDS) of a tree, which visits the nodes in a level-order traversal of the tree and encodes their degrees in unary. (In a level-order traversal of a tree, the root is visited first, followed by all the children of the root, from left to right, followed by all the nodes at each successive level.) See Figure 1 for an example. This representation uses $2n+O(1)$ bits to encode an ordinal tree on n nodes, and an additional $o(n)$ bits to support `rank` and `select` operations on the encoding. Jacobson considered these operations in the bit probe model, and showed how to support them in $O(\lg n)$

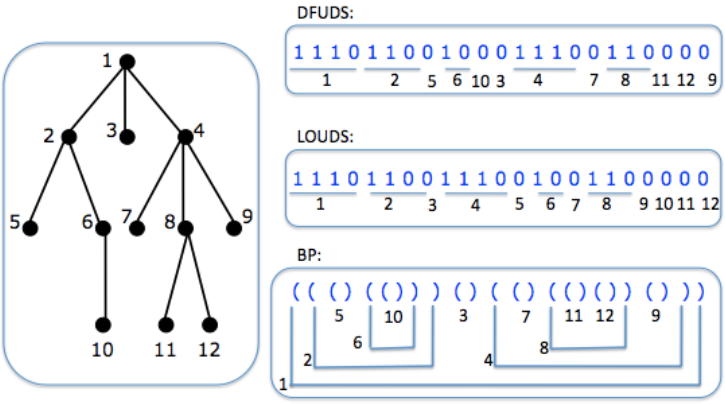


Fig. 1. An example of the DFUDS, LOUDS and BP sequences of a given ordinal tree

probes. Clark and Munro [10] further showed how to support the `rank` and `select` operations in $O(1)$ time under the word RAM model with $\Theta(\lg n)$ word size. Thus the LOUDSrepresentation supports `parent`, `child` and `child_rank` operations in $O(1)$ time. In addition to these operations, it is straightforward to support $\texttt{rank}_{\text{LEVEL}}$, $\texttt{select}_{\text{LEVEL}}$, `level_succ` and `level_pred` operations in constant time.

3.3 Balanced Parenthesis Representation

Munro and Raman [33] proposed another type of succinct representation of trees based on the isomorphism between *balanced parenthesis sequences* (BP) and ordinal trees. The BP sequence of a given tree is obtained by performing a depth-first traversal, and writing an opening parenthesis each time a node is visited, and a closing parenthesis immediately after all its descendants are visited. This gives a $2n$-bit encoding of an n-node tree as a sequence of balanced parentheses, and a node can be identified e.g. by the position of the opening parenthesis of the pair corresponding to it. See Figure 1 for an example. Extending the ideas from Jacobson [26], Munro and Raman showed how to support a few basic operations in constant time on a balanced parenthesis sequence of length n using an auxiliary structure of size $o(n)$ bits. By showing how to translate the operations on the tree to the basic operation on the parenthesis sequence representing it, Munro and Raman [33] presented a succinct representation of an ordinal tree on n nodes in $2n + o(n)$ bits based on BP, which supports `parent`, `desc`, `depth`, $\texttt{rank}_{\text{PRE/POST}}$ and $\texttt{select}_{\text{PRE/POST}}$ in constant time, and $\texttt{child}(x, i)$ in $O(i)$ time. Munro et al. [34] provided constant-time support for `leaf_rank`, `leaf_select`, `left_leaf`, `right_leaf` and `leaf_size` on the BP representation using $o(n)$ additional bits, which were used to design space-efficient suffix trees. Chiang et al. [8] showed how to support `degree` in constant time. The support for `level_ancestor`, `level_succ` and `level_pred` in constant time was further

provided by Munro and Rao [36], which has applications in the succinct representations of functions. Lu and Yeh [32] showed how to support `child`, `child_rank`, `height` and `LCA` operations in constant time.

Using a different approach to support the operations, Sadakane and Navarro [43] augmented the BP sequence with $o(n)$-bit auxiliary structure to obtain a "fully functional" representation as described in Section 3.7.

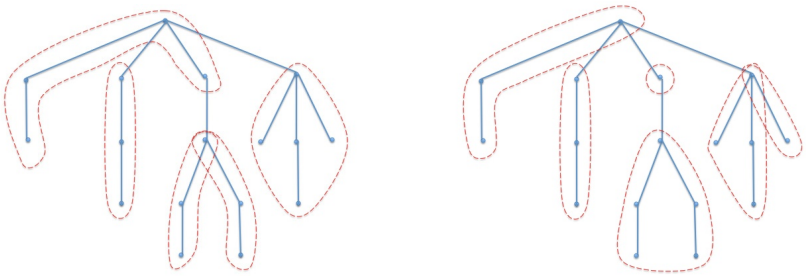
3.4 Depth-First Unary Degree Sequence Representation

Benoit et al. [5] observed that by visiting the nodes in depth-first order (i.e., preorder) and encoding their degrees in unary (instead of visiting them in level order to produce the LOUDS encoding), one can support other useful operations such as `desc`. The resulting encoding of the tree is called the *depth first unary degree sequence* (DFUDS). More specifically, the DFUDS sequence represents a node of degree d by d opening parentheses followed by a closing parenthesis. All the nodes are listed in preorder (an extra opening parenthesis is added to the beginning of the sequence). See Figure 1 for an example. The DFUDS number of a node is defined to be the rank of the opening parenthesis in its parent's description that corresponds to this node. Benoit et al. [5] presented a succinct tree representation based on DFUDS that occupies $2n + o(n)$ bits and supports `child`, `parent`, `degree` and `desc` in constant time. In their representation, each node is referred to by the position of the first parenthesis in the representation of the node. Jansson et al. [27] extended this representation using $o(n)$ additional bits to provide constant-time support for `child_rank`, `depth`, `level_ancestor`, `LCA`, `left_leaf`, `right_leaf`, `leaf_rank`, `leaf_select`, `leaf_size`, $\texttt{rank}_{\text{PRE}}$ and $\texttt{select}_{\text{PRE}}$. Barbay et al. [4] further showed how to support $\texttt{rank}_{\text{DFUDS}}$ and $\texttt{select}_{\text{DFUDS}}$. The operations $\texttt{rank}_{\text{PRE}}$, $\texttt{select}_{\text{PRE}}$, $\texttt{rank}_{\text{DFUDS}}$ and $\texttt{select}_{\text{DFUDS}}$ support the constant-time conversions between the preorder number and DFUDS number of the same node, which is used to support various queries on labeled trees by Barbay et al. [4].

3.5 Representations Based on Tree Covering

Another approach to represent ordinal trees is based on a *tree covering* algorithm (TC). This approach, first proposed by Geary et al. [20], is based on an algorithm to cover an ordinal tree with a set of *mini-trees*, each of which is further covered by a set of *micro-trees*.

Geary et al. [20] proposed an algorithm to cover a given ordinal tree on n nodes into $O(n/M)$ mini-trees, each of size $O(M)$, for a given parameter M. This decomposition guarantees that any two mini-trees computed by this algorithm are either disjoint, or only joined at their common root, and in addition, every mini-tree, except possibly the one containing the root, has size $\Theta(M)$. See Figure 2(a) for an example. The tree structure representing how these $O(n/M)$ mini-trees are connected is then stored using $O(n \lg n/M)$ bits. Similarly, each mini-tree is further decomposed into $O(M/M')$ micro-trees, each of size $O(M')$,



(a) (b)

Fig. 2. An example of covering an ordinal tree using the algorithm of (a) Geary et al. [20], and (b) Farzan and Munro [15], with parameter $M = 3$.

for a parameter M', using the same algorithm, and the tree structures of all the mini-trees (representing their micro-tree decomposition) is stored using a total of $O(n(\lg M)/M')$ bits. Each of the micro-trees is represented as a pointer to a table of size at most $2^{2M'}$. This table stores the representations of all possible micro-trees of size M'.

By choosing $M = \lceil \lg^4 n \rceil$ and $M' = \lceil (\lg n)/24 \rceil$, Geary et al. [20] obtain a representation that takes $2n + o(n)$ bits. They further show how to support various operations (namely, `child`, `child_rank`, `depth`, `level_ancestor`, `desc`, `degree`, $\texttt{rank}_{\text{PRE/POST}}$ and $\texttt{select}_{\text{PRE/POST}}$) on the ordinal tree in constant time by storing various auxiliary structures using $o(n)$ bits. He et al. [25] extended the representation by supporting several additional operations, namely `LCA`, `height`, `left_leaf`, `right_leaf`, `leaf_rank`, `leaf_select`, `leaf_size`, $\texttt{rank}_{\text{DFUDS}}$, $\texttt{select}_{\text{DFUDS}}$, `level_left`, `level_right`, `level_succ` and `level_pred`.

Farzan and Munro [15] modified the tree covering algorithm so that each mini-tree has at most one node, other than the root of the mini tree, that is connected to the root of another mini-tree. See Figure 2(b) for an example. This simplifies the representation of the tree, as well as the auxiliary structures needed to support various operations on the tree. They call this approach as the *uniform approach*, and justify the name by demonstrating that this approach can be applied to obtain succinct representations for various other families of trees, including cardinal trees.

3.6 "Universal" Representation

The succinct tree representations described so far are systematic encodings: they encode the *structure* of the tree as a bit-string of length $2n + o(n)$ bits, together with an index of $o(n)$ bits, where the index depends upon the choice of structure bit-string and the operations to be supported. Operations are supported in $O(1)$ time by reading $O(1)$ words from the structure bit-string and/or the index.

Since the index depends on the choice of structure bit-string, which also determines the node numbering, this approach leads to certain difficulties [17]. For example, certain operations can be implemented efficiently using one encoding, but are hard or impossible to implement in another. As noted in the introduction, it is not possible in general to create a representation that supports the union of the sets of operations of two representations without losing optimality. Even if we represent the given tree as two separate copies, each using the respective structure bit-strings and index data structures (thus losing optimality) we would still face the problem that a series of linked operations using both representations would require us to be able to map nodes of one numbering to the other, which is not always easy to achieve.

Farzan et al. [17] proposed an optimal-space succinct encoding for ordinal trees that can return $b = O(w)$ consecutive bits from the structure bit-strings of other (BP, DFUDS or TC) encodings, where w is the word-size, in $O(1)$ time. Since we can emulate access to the structure bit-strings of other encodings, by adding the appropriate index of $o(n)$ bits, one can directly support any operations supported by those encodings, with only a constant factor slowdown and negligible additional space cost. This representation is called the *universal representation* (UNIVERSAL).

3.7 "Fully-Functional" Representation

Sadakane and Navarro [43] proposed an ordinal tree representation that is based on storing the structure bit-string of BP, and constructing auxiliary structures to support the operations. But it differs from the BP representation by significantly simplifying the auxiliary structures, and also improving the lower-order term in space. In particular, they base navigation on the key primitive of *excess search*. The *excess* of a position in a parentheses sequence is the difference between the numbers of open and closed parentheses from the start of the sequence to the given position (the depth of a node equals the excess of its open parenthesis in the BP sequence). An excess search operation such as `fwd_excess`(i, d) returns the closest position $j \geq i$ whose where the excess equals the excess at i plus d. They propose a simple and flexible data structure, called the *range min-max tree* for supporting excess search, and then reduce a wide range of operations on the ordinal trees to simple combinations excess search and other primitive operations. The resulting representation, which they call as a *fully-functional* (FF) representation, (i) is conceptually simpler than most of the earlier representations, (ii) has smaller redundancy than the earlier ones, (iii) can be easily implemented, and (iv) can be efficiently dynamized.

Table 1 shows the comparison between the functionalities of various succinct tree representations that we described in this section.

Table 1. Navigational operations supported in $O(1)$ time on various succinct ordinal tree representations. All these representations use $2n + o(n)$ bits.

Operations	LOUDS	BP	DFUDS	TC/ UNIVERSAL	FF
`parent`, `child`, `child_rank`	✓	✓	✓	✓	✓
`depth`, `level_ancestor`		✓	✓	✓	✓
`degree`	✓	✓	✓	✓	✓
`height`		✓		✓	✓
`LCA`		✓	✓	✓	✓
`desc`, `leaf_size`		✓	✓	✓	✓
`left_leaf`, `right_leaf`		✓	✓	✓	✓
`leaf_rank`, `leaf_select`		✓	✓	✓	✓
$\texttt{rank}_{\text{PRE}}$, $\texttt{select}_{\text{PRE}}$		✓	✓	✓	✓
$\texttt{rank}_{\text{POST}}$, $\texttt{select}_{\text{POST}}$		✓		✓	✓
$\texttt{rank}_{\text{DFUDS}}$, $\texttt{select}_{\text{DFUDS}}$			✓	✓	
$\texttt{rank}_{\text{LEVEL}}$, $\texttt{select}_{\text{LEVEL}}$	✓				
`level_left`, `level_right`				✓	✓
`level_succ`, `level_pred`	✓	✓		✓	✓

4 Additional Topics

4.1 Dynamization

Following upon dynamization of binary trees [35,42], dynamization of succinct ordinal trees was considered by Sadakane and Navarro [43] and Farzan and Munro [16]. In the former, the update operations are taken to be edits on the BP sequence representing the tree, e.g. adding or deleting a parenthesis pair, while Farzan and Munro consider the slightly less general operations of adding a leaf or a new node breaking an existing edge. In both cases, navigation and other operations should continue to be supported once the update is complete.

A fundamental difference is in the way the operations are specified in the two approaches. Sadakane and Navarro follow the static API of [33], and perform many navigation operations via excess search. A key feature is that operations on a node are specified by the position of that node in the BP bit-string. As noted by Joannou and Raman [28], when using this approach, any navigation operation has several steps that are known require $\Omega(\lg n / \lg \lg n)$ time (unless updates are allowed to take more than poly-log time). However, the resulting dynamization does support the full range of navigational operations supported by the FF representation, together with the updates, in $O(\lg n / \lg \lg n)$ time.

An alternative is the *finger* model [16,42,35], where updates are done using a finger, which is effectively a "pointer" to a node in the tree. A finger may be moved using some navigational operations, and crucially, updates can only happen in the vicinity of a finger. In the finger model, the non-constant lower bounds above do not apply and both updates and navigation operations can in fact be performed in $O(1)$ time [16] (the time for updates is amortized).

However, the set of operations supported by Farzan and Munro is smaller than that of Sadakane and Navarro: in addition to the basic navigation operations, only `child`, `child_rank` and `desc` are supported.

4.2 Compressibility

When considering the information-theoretic lower bound of $2n - O(\lg n)$ bits, it must be noted that this is a *worst-case* bound. The same information-theoretic lower bound applies if the tree is a random tree or highly regular (e.g. complete k-ary tree). On the other hand, trees we find in practice are usually compressible. Although there has been work on compressing labelled trees [18], the work on representing compressible ordinal trees is far less mature. Jansson et al. [27] provided a definition of tree compressibility based on degree sequences (and gave a combinatorial justification thereof). They showed that it is possible to compress a given ordinal tree in the minimum possible space (according to their measure of compressibility), plus lower-order terms, and still support operations in $O(1)$ time. Bille et al. [7] considered ordinal trees that are compressed by sharing subtrees, which is a form of grammar-based compression. They showed that by generalizing excess search to such grammar-compressed BP strings, it is possible to support many of the operations of the representation of [43] in $O(\lg n)$ time. The space used is $O(m \lg n)$ bits, where m is the size of the grammar that generates the given tree.

4.3 Redundancy

The redundancy of a succinct representation is a quantity of both practical and fundamental importance, and papers have increasingly focussed on obtaining the best possible redundancy while still obtaining the best running times for operations (typically $O(1)$ time). As noted in the introduction, the redundancy is viewed slightly differently for systematic and non-systematic representations. For systematic representations, it is known that for the BP encoding, an index of size $\Theta(n \lg \lg n / \lg n)$ bits is needed to perform a full set of navigational operations [24]. However, using a non-systematic encoding, it is possible to obtain a redundancy of $O(n/(\lg n)^c)$ for any constant $c > 0$ [39,43].

We close on a lighter note. We note that even without the redundancy caused by requiring $O(1)$-time operations, the basic representations such as BP, DFUDS etc. are at least $2n - O(1)$ bits long, while the lower bound is $2n - O(\lg n)$ bits. We consider how to eliminate this $O(\lg n)$-bit additive overhead. As noted in the introduction, a trivial encoding—encoding a given tree by its position in an enumeration—achieves the lower bound, albeit at the cost of making operations quite difficult. However, Golin et al. [23] note that binary trees can be encoded as follows: write down the size of the left subtree of the root (which is a number from $\{0, \ldots, n-1\}$), and then recurse on the left and right subtrees; finally encode this sequence as a mixed-radix number. This approach gives an optimal-space (zero redundancy) encoding of binary trees such that operations can be performed in

polynomial time; via the standard equivalence between binary and ordinal trees, we also obtain an optimal succinct ordinal tree representation where navigation can be performed in polynomial time.

5 Implementations and Experimental Evaluation

Succinct representations of trees have been applied in a number of practical contexts. In the context of dictionary indexing, Clark [11] implemented succinct binary trees, and an implementation of a compact trie, the *Bonsai* tree, was used for text prediction and compression [12].

Ordinal tree implementations were apparently first studied in detail by Geary et al. [21] who implemented a simplified version of the BP representation of Munro and Raman (although [33] mentions a previous BP implementation by Chupa [9], details appear not to be in the public domain). Delpratt et al. [14] considered engineering the LOUDS representation and suggested a useful practical trick called *double numbering*. This addresses the issue that while the 'natural' numbering of notes in the succinct representation of an n-node tree is often as non-consecutive integers from $\{1, \ldots, 2n + O(1)\}$, for a variety of applications, a 'compact' numbering from $\{1, \ldots, n\}$ is desirable, particularly in order to associate information with the nodes. In theory, this can often be achieved in $O(1)$ time by means of `select`, to convert a user-provided node number (in a compact numbering) to the representation's natural numbering, followed by the appropriate tree operation, finally followed by a `rank` operation to convert the answer back to a node number in a compact numbering. However, this is a significant overhead in practice, and Delpratt et al. note that it is often better to number a node as a pair $\langle x, y \rangle$, where x and y number the node according to the compact and natural numberings, as updating the *pair* during operations is often trivial.

Subsequently, Arroyuelo et al. [2] implemented a simplified $O(\lg n)$-time excess search algorithm based on [43] for navigating in the BP representation; the resulting implementation appears to have similar speed performance, greater functionality and better space usage to that of Geary et al. [21]. Joannou and Raman [28] observe that the reason that the simple $O(\lg n)$-time excess search of Arroyuelo et al. performs comparably to the $O(1)$-time implementation of Geary et al. for navigation is that navigation in ordinal trees (i.e. a sequence of steps moving from a node to an adjacent one) induces a kind of locality of access in the BP sequence; consequently, the $O(\lg n)$-time worst-case cost may not be paid frequently. They advocate the use of *splay* trees [44] as a data structure for excess search to exploit this locality, and show good empirical performance. However, they do not perform a theoretical worst-case analysis. They also present the first empirical study of dynamic ordinal trees. Grossi and Ottaviano [38] present a highly engineered version of excess search.

In general, the experimental results show that succinct ordinal tree representations are competitive with naive representations, even when both the succinct and naive representations are too large to fit in cache, and small enough to fit in main memory. Succinct representations are usually much faster in case the

succinct representation fits into faster memory (cache/main memory) while the naive representation does not. Admittedly, an $O(1)$-time operation on a succinct ordinal tree can be more complex (in terms of the number of instructions performed) than the same operation in a classical ordinal tree representation, where an operation may be as simple as just following a pointer. However, when storing relatively large data, following a pointer can incur a cache miss or a TLB miss (which is even worse [29]). On the other hand, the best practical implementations of a succinct data structure will spend many instructions sequentially accessing memory locations, which is usually very fast in practice. Furthermore, since more information is packed into the faster levels of the memory hierarchy, succinct data structures show better locality as well.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. In: Aho, A.V., Borodin, A., Constable, R.L., Floyd, R.W., Harrison, M.A., Karp, R.M., Strong, H.R. (eds.) STOC, pp. 253–265. ACM (1973)
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Blelloch, G.E., Halperin, D. (eds.) ALENEX, pp. 84–97. SIAM (2010)
3. Arroyuelo, D., Claude, F., Maneth, S., Mäkinen, V., Navarro, G., Nguyen, K., Sirén, J., Välimäki, N.: Fast in-memory XPath search using compressed indexes. In: Li, F., Moro, M.M., Ghandeharizadeh, S., Haritsa, J.R., Weikum, G., Carey, M.J., Casati, F., Chang, E.Y., Manolescu, I., Mehrotra, S., Dayal, U., Tsotras, V.J. (eds.) ICDE, pp. 417–428. IEEE (2010)
4. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
5. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
6. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. 48(2), 214–230 (1994)
7. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: SODA, pp. 373–389. SIAM (2011)
8. Chiang, Y.T., Lin, C.C., Lu, H.I.: Orderly spanning trees with applications. SIAM J. Comput. 34(4), 924–945 (2005)
9. Chupa, K.: Efficient Representation of Binary Search Trees. Master's thesis, University of Waterloo (1997)
10. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage (extended abstract). In: ACM-SIAM SODA, pp. 383–391 (1996)
11. Clark, D.R.: Compact PAT trees. Ph.D. thesis, University of Waterloo, Waterloo, Ontario., Canada (1998)
12. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Softw. Pract. Exper. 23(3), 277–291 (1993)
13. Delpratt, O., Raman, R., Rahman, N.: Engineering succinct DOM. In: EDBT. ACM Intl. Conference Proceeding Series, vol. 261, pp. 49–60. ACM (2008)
14. Delpratt, O., Rahman, N., Raman, R.: Engineering the LOUDS succinct tree representation. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 134–145. Springer, Heidelberg (2006)

15. Farzan, A., Munro, J.I.: A uniform approach towards succinct representation of trees. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 173–184. Springer, Heidelberg (2008)
16. Farzan, A., Munro, J.I.: Succinct representation of dynamic trees. Theor. Comput. Sci. 412(24), 2668–2678 (2011)
17. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 451–462. Springer, Heidelberg (2009)
18. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM 57(1) (2009)
19. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379(3), 405–417 (2007)
20. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
21. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368(3), 231–246 (2006)
22. Gog, S.: Succinct data structure library, SDSL (2013), `https://github.com/simongog/sdsl`
23. Golin, M.J., Iacono, J., Krizanc, D., Raman, R., Rao, S.S., Shende, S.: Encoding 2-d range maximum queries. CoRR abs/1109.2885v2 (2012)
24. Golynski, A., Grossi, R., Gupta, A., Raman, R., Rao, S.S.: On the size of succinct indices. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 371–382. Springer, Heidelberg (2007)
25. He, M., Munro, J.I., Rao, S.S.: Succinct ordinal trees based on tree covering. ACM Transactions on Algorithms 8(4), 42 (2012)
26. Jacobson, G.J.: Space-efficient static trees and graphs. In: IEEE Symposium on Foundations of Computer Science, pp. 549–554 (1989)
27. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct representation of ordered trees with applications. Journal of Computer and System Sciences 78(2), 619–631 (2012)
28. Joannou, S., Raman, R.: Dynamizing succinct tree representations. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 224–235. Springer, Heidelberg (2012)
29. Jurkiewicz, T., Mehlhorn, K.: The cost of address translation. In: Sanders, P., Zeh, N. (eds.) ALENEX, pp. 148–162. SIAM (2010)
30. Katajainen, J., Mäkinen, E.: Tree compression and optimization with applications. Int. J. Found. Comput. Sci. 1(4), 425–448 (1990)
31. Kurtz, S.: Reducing the space requirement of suffix trees. Softw., Pract. Exper. 29(13), 1149–1171 (1999)
32. Lu, H., Yeh, C.: Balanced parentheses strike back. ACM Trans. Algorithms 4(3), 1–13 (2008)
33. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
34. Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. J. Algorithms 39(2), 205–222 (2001)
35. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: Kosaraju, S.R. (ed.) SODA, pp. 529–536. ACM/SIAM (2001)
36. Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1006–1015. Springer, Heidelberg (2004)

37. Ottaviano, G.: The succinct library (2013), https://github.com/ot/succinct
38. Grossi, R., Ottaviano, G.: Design of practical succinct data structures for large data collections. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 5–17. Springer, Heidelberg (2013)
39. Patrascu, M.: Succincter. In: FOCS, pp. 305–313. IEEE Computer Society (2008)
40. Poyias, A.: XXML: Handling extra-large XML documents. siXML technology whitepaper (February 2013), https://lra.le.ac.uk/bitstream/2381/27744/1/SiXDOMWhitepaper.pdf
41. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 43 (2007); Preliminary version in SODA 2002
42. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 357–368. Springer, Heidelberg (2003)
43. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Charikar, M. (ed.) SODA, pp. 134–149. SIAM (2010)
44. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
45. Tabei, Y.: Succinct multibit tree: Compact representation of multibit trees by using succinct data structures in chemical fingerprint searches. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 201–213. Springer, Heidelberg (2012)

Array Range Queries

Matthew Skala

University of Manitoba, Winnipeg, Canada
mskala@cs.umanitoba.ca

Abstract. Array range queries are of current interest in the field of data structures. Given an array of numbers or arbitrary elements, the general array range query problem is to build a data structure that can efficiently answer queries of a given type stated in terms of an interval of the indices. The specific query type might be for the minimum element in the range, the most frequently occurring element, or any of many other possibilities. In addition to being interesting in themselves, array range queries have connections to computational geometry, compressed and succinct data structures, and other areas of computer science. We survey recent and relevant past work on this class of problems.

Keywords: array, range query, document retrieval, range search, selection, range frequency, top-k.

1 Introduction

Given an array of numbers or arbitrary elements, the general array range query problem is to build a data structure that can efficiently answer queries of a given type stated in terms of an interval of the indices. For instance, we might ask for the minimum element value that occurs in the range. It is possible to define a nearly endless variety of such problems by making different choices for the type of query to answer, the amount of space allowed, the model of computation, and so on. Much work has been done on these problems, especially in the last few years. Besides the steady procession of improvements in time and space bounds, work on array ranges has led to new techniques and insights in data structure design applicable to many other problems.

In this paper we survey current results on array range queries. In this introduction we first define the scope of the survey and discuss classification of range queries. Then we introduce some commonly-used techniques for these problems, and in subsequent sections we describe current work in the field organized by our classification.

We are primarily interested in range queries on arrays as such; that is, sequences of data indexed by consecutive integers. Geometric problems are typically approached using different techniques and are not the main focus here. The 1999 survey of Agarwal and Erickson gives detailed coverage of geometric range query work up to that date [1], and we mention some recent geometric work relevant to array range queries. Similarly, although many of the data structures used

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 333–350, 2013.

for array range queries are related to those used for string searching problems, our focus here is not on string search. Array ranges can be naturally generalized to queries on paths in trees, and we discuss some of that work without attempting to cover it all.

Array range queries tend to be distinguished by small input and output sizes. A query typically consists of just the starting and ending indices of the range; in some problems there may also be something else in the query, such as a color or a threshold, but seldom more than a constant number of such things. The answer to a query is usually also of small size, such as a single element or just an answer of "yes" or "no." It is unusual for the output size to be as big as the entire contents of the range.

Unless otherwise specified, we assume a RAM model of computation in which the machine can perform operations on words of length $O(\log n)$ in unit time; data structures of $O(n)$ such words; and a static analysis, in which we are allowed to see the entire array in advance and possibly process it in larger working space to build the data structure, after which the element values cannot change. In dynamic versions of array range query problems, the question arises of what is an update. Changing the value of the element at one index, without changing the number of elements nor the values at any other indices, may be the most natural form of dynamic update for an array. However, because of the way they use geometric data structures internally, many dynamic array range data structures also support insertions and deletions, shifting all later elements up or down by one index position, at no additional cost over changing the value of a single element. Unless otherwise specified, we assume that a dynamic update can be an insertion or deletion.

1.1 Classification of Array Range Queries

The unending desire for novel results has led researchers to consider a bewildering assortment of different kinds of array range query problems, and it may not be possible to fit them all into a consistent classification scheme. However, we can discern a few general categories that may be useful in organizing the discussion. Very often, a single paper may describe several different problems, possibly connected by a common data structure that cuts across our classification.

First, an array range query usually considers one of three things about array element values to determine how they can satisfy the query. We discuss each of these in its own section.

- The array elements may be values that have an order, typically real numbers, with the ordering or magnitude of the numbers relevant to the query result. In this case the element values are often called *weights*. These queries are discussed in Section 2.
- The interesting thing about array elements may be only whether each value does or does not occur in the range at least once, without regard for an ordering of the values nor how many times each value occurs. Elements considered this way are often called *colors*. These queries are discussed in Section 3.

– The number of times each value occurs in the range, not only whether it is zero or nonzero, may be important to the query result. Considering element *frequencies* in this way can be said to combine weights (because some values can be ranked above others) and colors (because the values as such are not ordered). These queries are discussed in Section 4.

When considering the ordered element value (weight), a query may meaningfully be stated in terms of the actual value, or its rank among the values that occur in the query range. Similarly, a query based on color could be concerned with an actual color value, or "the first (or k-th) color to occur in the range." And a frequency query could be in terms of a given value of the frequency (either a raw number or a ratio to the query range length) or as a rank among frequencies, such as the mode. In all three categories, then, we can describe range query problems based on raw value and on rank, and that provides a second dimension for our classifiation.

Having defined the property of interest, and whether to consider it as a raw value or as a rank, we can describe different range query problems depending on what information is given as input to the query.

– The values of interest may be completely determined by the problem, so that a query consists of only a range of indices. Such problems can often be seen as special cases of more general problems, of separate interest because they may be easier than the general versions.
– A single value or rank of interest may be specified in the query.
– A threshold may be part of the query, with all values or ranks above or below the threshold being of interest. In some cases, upper and lower thresholds may give rise to significantly different problems. A closely-related category allows an interval (two thresholds) to be given in the query and then matches values within the interval, giving the effect of the intersection of an upper and a lower threshold query, possibly with better bounds.
– A few problems have been described in which the query includes an exact value or a threshold that can be chosen with generality, but the choice is made during preprocessing, and must be the same for all queries.

Finally, there may be several different forms in which a query can return its result, and different kinds of results from what would otherwise be the same problem may require different data structures and have different bounds.

– The query may return a single element weight, color, or frequency matching the criteria, or a rank on one of those properties. These six possibilities correspond to the first two dimensions of our categorization.
– The query may return the index of an element at which the match occurs. Many queries usually thought of as returning values (the previous category) implicitly do it by returning indices where those values occur; but asking for the index is a slightly stronger question, and may be more difficult.
– The query may return a list of all matching values or indices.
– The query may return only a decision result: "yes" or "no" to whether a matching result occurs in the range at all.

1.2 Techniques for Array Range Query

Many range query data structures begin their preprocessing by doing *rank space reduction* [38,22]. Rank space reduction starts with an array $A[1..n]$ of arbitrary elements and creates an array $B[1..n]$ of integers in the range 1 to n, where each element of $B[i]$ represents the rank of $A[i]$ among the set of values that occur in A. A similar array B' is created to translate the ranks back to element values from A. If original values are ever needed, then the ranks from B can be looked up in B'. This reduction means that the rest of the data structure can deal exclusively with integers the size of n; and in particular, it can use the usual RAM techniques to do predecessor queries in $O(\log \log n)$ time.

If a query consists solely of an interval of indices into an array of size n, only $O(n^2)$ distinct queries are possible. Then we could store the answers to all of them in a table that size, and answer queries in constant time just by looking up the answers in the table. Quadratic space is rarely satisfactory, but if we split the input into blocks of size $\Omega(\sqrt{n})$, there are $O(n)$ intervals that start and end on block boundaries and we could afford to store a constant-sized answer for each of them in linear space. Many array range data structures use such a table as the first step in a recursion that will eventually answer arbitrary queries.

It is also common practice to reduce to some other, already-solved array range query problem. Indeed, many of the array range queries we discuss here originally arise as reductions used to solve other problems. The *rank* and *select* queries of the succinct data structures literature [61,60,72,46,79] are often invoked as building blocks for more complicated array range problems. Given a vector of bits, the rank query asks for the number of 1 bits that occur between the start of the vector and a given index; the select query is the reverse, asking for the index of the i-th 1 bit. Rank and select within a range of indices (instead of only counting from the start) are easy to answer by doing, and subtracting out, one additional rank query at the start of the range.

Wavelet trees [48] are also popular building blocks; they generalize succinct bit vectors with rank and select to larger alphabets while providing a few other queries useful in solving array range problems. Navarro provides a general survey [74]. A wavelet tree stores an array (usually called a string in this context) of elements from some alphabet, in a binary structure that divides the alphabet in half at each level. Each node contains a succinct bit vector naming the subtrees responsible for each remaining array elements at that level. The overall space requirement for the basic data structure is $O(n \log \sigma)$ bits where σ is the size of the alphabet, and it can support rank and select on any letter in $O(\log \sigma)$ time; but it can also be used as part of the solution to many other kinds of queries.

2 Weight Queries

When the values stored in the array have meanings directly relevant to the query, they are usually called *weights*. Weights might not be real numbers, but typically have some structure such as a total order, a group, or a semigroup.

2.1 Range Minimum Query

Range Minimum Query (RMQ) on weights equipped with an order is one of the earliest-studied array range query problems. *Range maximum* is equivalent. This problem has the important property that the minimum of a union of two sets must be the minimum of one of the sets; so the query range can be decomposed into a constant number of smaller ranges and the answer found by solving those as subproblems. Highlights of the work on this problem, including the variation for two-dimensional arrays, are shown in Table 1.

Table 1. Selected results on array range minimum query

space (*bits*)	query time	year	ref.	note
$O(n \log n)$	$O(1)$	1984	[51]	LCA
$O(n \log n)$	$O(\log n)$	1984	[38]	2-D
$O(n \log n)$	$O(1)$	1989,1993	[9,10]	
$4n + o(n)$	$O(1)$	2002	[84]	$O(n \log n)$ bits prep. space
array $+ 2n + o(n)$	$O(1)$	2007	[35]	$2n - o(n)$ space lower bound
$O(n \log n)$	$O(1)$	2007	[4]	2-D, $O(n \log^{(k)} n)$ prep. time
$O(n \log n)$	$O(1)$	2009	[26]	cache-oblivious, $O(n/B)$ prep. time
$2n + o(n)$	$O(1)$	2010	[34]	
array $+ O(n)$	$O(1)$	2012	[14]	2-D
$O(n \log n)$	$O(1)$	2013	[29]	simplified

The one-dimensional array RMQ problem can be reduced to *Lowest Common Ancestor* (LCA), and constant-time linear-space solutions for that problem go back at least as far as the work of Harel and Tarjan in 1984 [51]. It is generally taken for granted (whether explicitly stated or not) that data structures for this problem must return the *index* of a minimum element, not only the *weight*. Berkman and Vishkin's work [9,10] is usually cited as the first significant data structure for RMQ. Their main concern was with doing the preprocessing in a parallel model, and with applying the data structure to other problems beyond range queries; but they achieved constant time RMQ with a linear space data structure by reducing from RMQ to LCA and then, by means of *Cartesian trees*, back to RMQ with the constraint that successive elements differ by at most one. Bender and Farach-Colton [8] give a simplified presentation of this data structure. Preprocessing time can be linear with a careful implementation, and subsequent results have also achieved linear preprocessing time.

Sadakane gives a succinct data structure for RMQ, again as part of a solution to LCA [84]. It uses $4n+o(n)$ bits, but it requires asymptotically more than that ($O(n \log n)$ bits) temporarily during preprocessing. Fischer and Huen improve the data structure space bound to $2n + o(n)$ bits with access to the original array [35], and never require more than that even during preprocessing. They also give a lower bound of $2n-o(n)$ bits, and claim an advantage of not using bit vector rank and select (which may make implementation easier). This is usually cited as the current best result for the one-dimensional array RMQ problem in

the basic RAM model. Fischer [34] later improves it to avoid the requirement for access to the original array. In this volume, Durocher [29] gives a new linear-space data structure with constant time query, using simple, practical techniques.

For RMQ on two-dimensional arrays, Gabow, Bentley, and Tarjan [38] give a solution with $O(n \log n)$ space and preprocessing time, and $O(\log n)$ query. Note that we describe the array as $\sqrt{n}$ by $\sqrt{n}$ for a total of n elements overall to make these results more easily comparable to the one-dimensional case. Amir, Fischer, and Lewenstein [4] give a linear-space, constant query time data structure for two-dimensional array RMQ with $O(n \log^{(k)} n)$ preprocessing time, where $\log^{(k)}$ means logarithm iterated any constant number of times.

Demaine, Landau, and Weimann [26] extend RMQ in several directions at once, as well as reviewing some extensions of the problem in more detail. They give an algorithm for one-dimensional arrays with the usual linear space and constant query time as well as optimal $O(n/B)$ preprocessing time in the cache-oblivious model (B is the block size); they solve *Bottleneck Edge Query* (BEQ), which is a generalization to paths in graphs, in linear space, $O(k)$ query time, and $O(n \log^{(k)} n)$ preprocessing time, as well as giving some results on a dynamic version; and they show a combinatorial lower bound that suggests Cartesian trees cannot usefully be applied to the two-dimensional array RMQ problem. Other techniques than Cartesian trees may still be applicable. Brodal, Davoodi, and Rao [14] give a data structure for two-dimensional RMQ in $O(n/c)$ bits (not words; c can be chosen) plus access to the original array, with linear preprocessing and $O(c)$ query time, and they show a matching lower bound.

2.2 Counting and Reporting

Purely counting elements in an array range is trivial, so the term *range counting* usually refers to counting elements subject to some restriction, such as elements whose weight is in a specified interval. Similarly, *range reporting* on arrays usually refers to reporting elements whose weights are within an interval. These definitions make the array range queries equivalent to geometric range queries on a two-dimensional grid, and the best results use geometric range query techniques. Conversely, solutions to geometric versions of these problems often start with rank-space reduction to integer coordinates, so the equivalence is close.

We summarize interesting results on counting, reporting, and emptiness in Table 2. The 1988 data structure of Chazelle [22] is a precursor to the wavelet tree, without the optimization of using succinct bit vectors in the nodes; it achieves $O(\log n)$ time for counting and $O(\log n + k \log^{\epsilon} n)$ for reporting, where k is the number of results, in linear space. By using a wavelet tree, Mäkinen and Navarro [70] achieve $O(\log n)$ time for counting and $O(k \log n)$ for reporting. Bose et al. [11] improve both these query times by a factor of $\log \log n$. Other significant results also involve $\log^{\epsilon} n$ trade-offs: for instance, $O(\log n + k)$ reporting with $O(n \log^{\epsilon} n)$ space [3]. Nekrich [75,76] gives a good survey of previous work up to 2009, as well as a $O(\log n + k \log^{\epsilon} n)$ dynamic reporting data structure (linear space, $O(\log^{3+\epsilon} n)$ updates) and a $O(\log n / \log \log n + k \log^{\epsilon} n)$ static linear-space reporting data structure.

Table 2. Selected results on range counting and reporting

problem	space	query time	year	ref.
counting	$O(n)$	$O(\log n)$	1988	[22]
	$n + o(n)$	$O(\log n)$	2007	[70]
	$n + o(n)$	$O(\log n / \log\log n)$	2009	[11]
	$n + o(n)$	$O((\log n / \log\log n)^2)^*$	2011	[52]
	$O(n)$	$O(\log \sigma)^\dagger$	2011	[55]
	$n + o(n)$	$O(\log \sigma \log n)^\dagger$	2012	[78]
	compressed	$O(\log \sigma / \log\log n)^\dagger$	2012	[56]
reporting	$O(n)$	$O(\log n + k \log^\epsilon n)$	1988	[22]
	$O(n \log^\epsilon n)$	$O(\log n + k)$	2000	[3]
	$n + o(n)$	$O(k \log n)$	2007	[70]
	$n + o(n)$	$O(k \log n / \log\log n)$	2009	[11]
	$O(n)$	$O(\log n + k \log^\epsilon n)^*$	2007,2009	[75,76]
	$O(n)$	$O((1 + k) \log \sigma)^\dagger$	2011	[55]
	$O(n \log\log n)$	$O(\log \sigma + k \log\log \sigma)^\dagger$	2011	[55]
	$n + o(n)$	$O((\log n + k) \log \sigma \log n)^\dagger$	2012	[78]
	compressed	$O((1 + k) \log \sigma / \log\log n)^\dagger$	2012	[56]

*dynamic †on trees

Range counting results are often presented as barely-discussed corrollaries in papers on range reporting, but He and Munro [52] give a succinct dynamic data structure specifically for counting, with $O((\log n / \log\log n)^2)$ time for queries and updates in the worst case of large alphabet size; they state the bounds in a more complicated form taking into account the possibility of small alphabets, and discuss applications to geometric range counting. He, Munro, and Zhou [55] also propose extending counting and reporting queries to paths in trees; they achieve $O((1 + k) \log \sigma)$ reporting time (to report k elements, σ distinct weights in the tree) with linear space, and $O(\log \sigma + k \log\log \sigma)$ time with $O(n \log\log n)$ space. Patil, Shah, and Thankachan [78] give succinct data structures for tree path counting and reporting at a $\log n$ time penalty. Then He, Munro, and Zhou improve their earlier linear-space results to compressed space [56], also improving the query times by $\log\log n$.

2.3 Other Weight Queries

There is a near-trivial folklore solution to *range sum*, or equivalently *range mean*, in a static array: just precompute and store the sums for all ranges beginning at the start of the array, and then subtract the sums for the endpoints to answer an arbitrary range query in constant time. This approach generalizes to arbitrary dimension by applying the principle of inclusion and exclusion on ranges with one corner fixed at the origin, although the constant in the query time is exponential in the number of dimensions. Fredman [37] introduced the dynamic one-dimensional version of the problem (with updates consisting of changing a single element's value) and gave a $O(\log n)$ time, linear-space solution.

A typical modern approach might use any standard balanced tree augmented with subtree sums to allow query, update, insertion, and deletion in $O(\log n)$ time. Matching $\Omega(\log n)$ bounds have been shown in various models of computation [37,85,50].

There are matching upper and lower bounds of $\Theta(n+m\alpha(m,n))$ time to do m array range sum queries *off-line*, where α is the inverse Ackermann function [23]. Brodnik et al. [18] give a constant-time solution using $O\big((n+U^{O(\log n)})\log U\big)$ bits of memory in the *RAMBO* model of computation, where bits can be shared among memory words, summing integers modulo U. The high-dimensional dynamic array case was extensively studied in the database community around the turn of the century [57,44,24,82]. Almost all of these results still apply when "sum" is generalized to arbitrary group operations; some of the related work can be further generalized to arbitrary semigroups, forming a connection to the results on range minimum because minimum is a semigroup operation.

Range minimum was generalized in one direction to semigroups, but a different generalization is to the *range selection* problem of returning a chosen order statistic. The special case of *range median* was the first to be studied: Krizanc, Morin, and Smid [67,68] gave multiple results for it trading off time and space, with $O(n^\epsilon)$ time for linear space on array ranges. They also considered tree paths. Gagie, Puglisi, and Turpin [43] review existing results on range median and then supersede most of them by showing how to use wavelet trees to answer selection, for general order statistics chosen at query time, in $O(\log \sigma)$ time with a linear-space data structure. Subsequent results by Brodal and Jørgensen [17]; Gfeller and Sanders [45]; and then all four of those authors together [16] include queries in $O(\log n/\log\log n)$ time with linear space, a dynamic version with $O((\log n/\log\log n)^2)$ query time in $O(n\log n/\log\log n)$ space, and various results on other models including a cell-probe lower bound of $O(\log n/\log\log n)$ for dynamic range median if updates are $O(\log^{O(1)} n)$. Jørgensen and Larsen [63] give additional lower bounds, notably including $\Omega(n^{1+\Omega(1)})$ space for constant-time queries; and a linear-space static data structure with query time $O((1+\log k)/\log\log n)$ where k is the order statistic desired, optimal by their lower bounds except for small values of k. In compressed space, He, Munro, and Nicholson [53] give a dynamic data structure with query time, in a recently-posted correction to the conference paper [54], $O((\log n/\log\log n)(\log\sigma/\log\log\sigma))$. The earlier-mentioned tree path counting results of He, Munro, and Zhou [55,56] and of Patil, Shah, and Thankachan [78] can also be applied to tree path selection with nearly the same bounds; in the case of the He, Munro, and Zhou results the denominator in the query time changes from $\log\log n$ to $\log\log\sigma$.

Instead of selecting a single order statistic, we could ask for the first k order statistics: a *top-k* problem. Range reporting answers top-k queries if the results are returned in sorted order, preferably with output-sensitive time, and many current range reporting data structures work that way. However, Brodal et al. [15] describe a linear-space data structure for top-k reporting with $O(k)$ output time, and that is better than the lower bound for range reporting. As we discussed, "range reporting" in current use implies that queries include thresholds

on element weights, in effect an extra dimension of the geometric problem, and that is not part of the definition of top-k queries. Nekrich and Navarro [77] give a linear-space data structure for sorted reporting *with* a threshold in $O(k \log^{\epsilon} n)$ time, as well as some other time-space trade-offs.

3 Color Queries

If we consider array elements primarily with respect to distinctness, possibly imposing an order on the distinct values, we can define array range queries based on these *colors*. Color range query problems often arise in the document retrieval literature, and are often presented along the way to solving other problems rather than as independent results.

Gagie et al. [42] have a useful framework for understanding recent work on colored range queries. As they explain, the *colored range listing* problem is fundamental. Given a range, colored range listing returns the index of the first element of each distinct color in the range. Looking at the array elements is easy; the difficulty lies in removing the duplicates to give output-sensitive time. Some of the first output-sensitive results are due to Janardan and Lopez [62], whose linear-space static data structure reports k results in $O(\log n + k)$ time. Gupta, Janardan, and Smid [49] suggest transforming each element $A[i]$ in the original array A into a point (i, j) on the plane using the largest $j < i$ such that $A[j] = A[i]$, and give dynamic results. Muthukrishnan [73] describes an equivalent transformation in terms of an array C. In either form, appropriate range queries on the transformed input give the answers to colored range listing. Muthukrishnan's data structure uses $O(n)$ space and $O(k)$ query time. The subsequent improvements to range minimum queries allow tighter space bounds with or without compromises on the query time, and much subsequent work on colored range listing comes down to applying better RMQ results in this setting. Gagie, Puglisi, and Turpin [43] describe how to use wavelet trees for range selection, the generalization of range minimum, and apply it to colored range listing. Using range selection instead of range minimum eliminates the need to store the suffix array in the document retrieval application they consider. The recent publication of Gagie et al. on compressed document retrieval [41] includes a good survey of colored range problems within their framework.

Merely counting the distinct colors instead of returning them is the *range color counting* problem. Bozanis et al. [13] give a linear-space $O(\log n)$-time data structure for it. Lai, Poon, and Shi [69] apply a result of Gupta et al. [49] to solve the dynamic version with a $\log^2$ penalty: either $O(n \log n)$ space and $O(\log n)$ query, or linear space and $O(\log^2 n)$ query. Gagie and Kärkkäinen [40] reduce it to compressed space with query time logarithmic in the length of the query range, slightly better than $O(\log n)$. They also give some dynamic results, expanded in the journal version by Gagie et al. [41].

If we ask for colors to be listed in order, combining the distinctness of colors with the ordering of weights, we have the *top-k color reporting* problem. Karpinski and Nekrich [66] give a linear-space data structure for this problem

with optimal $O(k)$ query time. This top-k problem defined by an ordering of the element values is different from a top-k problem defined by frequency rank within the range, as considered in the next section. Authors working on such problems use similar or identical terminology for the two.

4 Frequency Queries

When queries concern element frequencies, most combinations of the other parameters in our classification yield interesting and distinct problems: we can ask about raw frequencies or their ranks, provide exact values or thresholds, and return several different kinds of results. High frequencies and low frequencies often necessitate different techniques and sometimes even have different bounds.

4.1 Queries Related to Raw Frequency

The well-known element uniqueness problem, of determining whether any element in an array occurs more than once, has a lower bound of $\Omega(n \log n)$ in the algebraic decision tree model [7]. Determining whether the frequency exceeds k requires $\Theta(n \log(n/k))$ (matching upper and lower bounds) [71,27]. These bounds also apply to the time for *preprocessing plus one query* on the array range versions of the problems. The array range query of whether any element has frequency at least k in the range, k chosen during preprocessing, is trivial to solve with a linear-space data structure and constant-time queries: just store, for each range starting point, the end of the shortest range for which the answer is "yes." Greve et al. [47] describe that data structure. But for $k > 1$, even if chosen during preprocessing, testing the existence of an element with frequency exactly k in the range is more difficult. Greve et al. [47] show matching upper and lower time bounds of $\Theta(\log n / \log \log n)$ with k chosen at query time, assuming a linear-space data structure, the cell probe model for the lower bound, and word RAM for the upper bound.

A closely related but not identical class of problems considers a threshold on the *proportion* of array elements in a range that contain a given value. These problems represent a stepping stone to the frequency-rank problems in the next subsection. Results for these problems are summarized in Table 3.

Given β fixed at preprocessing time, the *range β-majority* query problem is to return an element that occurs (or all such elements) in at least β proportion of the elements of the query range. A geometric data structure of Karpinski and Nekrich [65] can be applied to solve this in $O(n/\beta)$ space and $O((1/\beta)(\log \log n)^2)$ query time. Durocher et al. [30,31] give a solution in $O(n \log(1+1/\beta))$ space and $O(1/\beta)$ query time, and Gagie et al. [39] in $O(n)$ space and $O((1/\beta) \log \log n)$ query time; in recent work, Belazzougui, Gagie, and Navarro improve the query time to optimal $O(1/\beta)$ [6]. Elmasry et al. [32] study a dynamic version of this problem, mostly in a geometric setting with β fixed and updates consisting of point addition and removal; for arrays, their data structure gives $O(n)$ space and $O((1/\beta) \log n / \log \log n)$ query time, with $O((1/\beta) \log^3 n / \log \log n)$ amortized insertion and $O((1/\beta) \log n)$ amortized deletion.

Table 3. Selected results on array range majority and minority

problem	space	query time	year	ref.
	$O(n/\beta)$	$O(1/\beta)$	2008	[65]
	$O(n \log(1 + 1/\beta))$	$O(1/\beta)$	2011,2013	[30,31]
β-majority	$O(n)$	$O((1/\beta) \log \log n)$	2011	[39]
	$O(n)$	$O((1/\beta) \log n / \log \log n)^*$	2011	[32]
	$O(n)$	$O(1/\beta)$	2012	[6]
	$O(n(H + 1))$	$O(1/\alpha)$	2011	[39]
	$O(n \log n)$	$O(1/\alpha)$	2012	[21]
α-majority	$n \log \log \sigma + O(n)$	$O(1/\alpha)$	2012	[6]
	$O(n)$	$O((1/\alpha) \log \log(1/\alpha))$	2012	[6]
	$n + o(n)$	$O((1/\alpha) \log \log \sigma)$	2012	[6]
α-minority	$O(n)$	$O(1/\alpha)$	2012	[21]

*dynamic

When the proportion threshold is α chosen at query time, it is easy to solve range majority at a factor of $\log n$ space penalty by building copies of a slightly modified β-majority data structure for each power of two value of β and then choosing the closest one at query time. Chan et al. describe that technique [21]. Gagie et al. [39] give a compressed range α-majority data structure ($O(n(H+1))$ words where H is entropy, bounded above by $\log \sigma$ where σ is the number of distinct elements) with $O(1/\alpha)$ query time. Belazzougui, Gagie, and Navarro [6] give multiple new results for this problem with query time varying depending on space, as summarized in our table.

Another possibility is to make the frequency threshold depend on the element value. De Berg and Haverkort [25] describe *significant-presence* queries. An element value has a significant presence in a query range if at least a specified fraction of the elements with that value in the entire array occur within the query range: the range covers a fraction of the color class rather than the color class necessarily covering a fraction of the range. The significant-presence range query is to find all the values which have significant presence in the range. The main results of de Berg and Haverkort concern approximate versions of this problem in an arbitrary-dimensional geometric setting, but they solve the exact problem on one-dimensional arrays with the threshold fixed during preprocessing using linear space and $O(\log n + k)$ query time.

Querying for a threshold on frequency in the opposite direction, that is, considering elements that occur at least once in the query range but less than some number of times, requires significantly different techniques. Chan et al. [21] solve range α-minority (α may be chosen at query time) using $O(n)$ space and $O(1/\alpha)$ query time.

4.2 Queries Related to Frequency Rank

Range *mode* (most frequent element) is arguably a more natural and useful query than range majority; indeed, previous results on range majority are often inspired

by or connected with work on the more difficult question of range mode. Range mode and more general frequency rank problems are of great current interest because of applications in document retrieval: in an array of document identifiers corresponding to a suffix array, high-frequency elements in a range identify the documents that contain a given substring many times.

Krizanc et al. [67,68] introduce the range mode problem for arrays, immediately also generalizing it to paths in trees. They give data structures for arrays and trees with $O(n^{2-2\epsilon})$ space and $O(n^{\epsilon} \log n)$ query time, where ϵ can be chosen. Setting it to $1/2$ gives linear space and $O(\sqrt{n} \log n)$ time. They also give a $O(n^2 \log\log n / \log n)$-space, constant-time data structure, slightly beating the obvious quadratic-sized table of all possible answers. In the same work they consider range median; and the connection between median and mode has persisted in later work by others, with results on both problems often appearing simultaneously.

Petersen [80] improves the $O(n^{\epsilon} \log n)$-time result to $O(n^{\epsilon})$ with the same $O(n^{2-2\epsilon})$ space; however, the hidden constant in the space requirement goes to infinity as ϵ approaches $1/2$, so linear space is no longer achievable. He improves the space bound for constant time to $O(n^2/\log n)$; then, with Grabowski [81], to $O(n^2 \log\log n / \log^2 n)$.

Bose et al. [12] consider approximate range mode, where the goal is to return an element with frequency at least β (chosen at preprocessing) times the frequency of the true mode. Their general data structure requires $O(n/(1-\beta))$ space and $O(\log\log_{1/\beta} n)$ time, but for $\beta \in \{1/2, 1/3, 1/4\}$ they give data structures with constant time and $O(n \log n)$, $O(n \log\log n)$, or $O(n)$ space, respectively. They also give results on an approximate version of the range median.

Chan et al. [19,20] have the best current results on linear-space array range mode, including $O(\sqrt{n/\log n})$ time with linear space; actually $O(\sqrt{n/w})$ on a w-sized word RAM, thus $O(\sqrt{n/B})$ in external memory. They give an argument, based on a reduction from boolean matrix multiplication, suggesting that the $\sqrt{n}$ factor in the time may be difficult or impossible to remove with a linear-space data structure; and some results on the dynamic one-dimensional version where updates consist of changing single element values (not insertion or deletion), and some higher-dimensional geometric range mode queries.

Top-k array range frequency is closely related to *top-k document retrieval*, the problem of returning the documents with top k most occurrences of a search substring, or the top k best values of some relevance function related to counting substring occurrences. Top-k document retrieval reduces to top-k array range frequency on the suffix array; however, because the queries are related to substrings, they correspond to internal nodes of the suffix tree, and there are only a linear number of possible query ranges. True top-k array range frequency would allow a quadratic number of distinct query ranges. This distinction allows the data structures for top-k document retrieval to achieve better results than would be possible for top-k array range frequency; and the techniques used, although related, are not directly applicable to the general array range problems. Hon, Shah, and Vitter [59], and Hon, Shah, and Thankachan [58] have some interesting

recent work on the document retrieval problem. On the array version, there is very little work on the exact problem; recall that merely finding the mode is a difficult problem, and the top-k version must be at least as hard. Some of the work of Janardin and Lopez [62] is applicable to special cases of top-k frequency. Gagie et al. [41], as well as giving results on document retrieval, solve an ϵ-approximate version, in $O((n/\epsilon)\log n)$ space (worst case; they state a more precise bound in terms of entropy) and $O(k \log \sigma \log(1/\epsilon))$ query time. Chan et al. [21] use a one-sided version (one end of the query must be index 0, making the problem much easier) with $O(n)$ space and $O(k)$ query time as part of their range majority data structure.

Top-k array range frequency naturally suggests a selection version, of finding the k-th most frequent element in an array range. That is an open problem in the case of general k. We have described the special case of range mode already. Chan et al. [21] introduce the *range least frequent element* problem, giving a linear-time data structure with $O(\sqrt{n})$ query time and an argument from boolean matrix multiplication suggesting that, as with mode, a significantly better query time may not be possible.

5 Conclusion

We have surveyed work in array range queries, an active field of current data structure research. We have also described a classification scheme for array range query problems. Not all imaginable categories in our classification are covered by existing work; the missing ones may suggest open problems of interest.

Acknowledgement. We gratefully acknowledge the suggestions made by an anonymous reviewer.

References

1. Agarwal, P.K., Erickson, J.: Geometric range searching and its relatives. In: Chazelle, B., Goodman, J.E., Pollack, R. (eds.) Advances in Discrete and Computational Geometry. Contemporary Mathematics, vol. 223, pp. 1–56. American Mathematical Society, Providence (1999)
2. Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.): ICALP 2009, Part I. LNCS, vol. 5555. Springer, Heidelberg (2009)
3. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, November 12-14, pp. 198–207 (2000)
4. Amir, A., Fischer, J., Lewenstein, M.: Two-dimensional range minimum queries. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 286–294. Springer, Heidelberg (2007)
5. Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.): ISAAC 2011. LNCS, vol. 7074. Springer, Heidelberg (2011)

6. Belazzougui, D., Gagie, T., Navarro, G.: Better space bounds for parameterized range majority and minority. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 121–132. Springer, Heidelberg (2013)
7. Ben-Or, M.: Lower bounds for algebraic computation trees (preliminary report). In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, Boston, Massachusetts, April 25-27, pp. 80–86 (1983)
8. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
9. Berkman, O., Vishkin, U.: Recursive *-tree parallel data-structure. In: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1989, Research Triangle Park, NC, October 30-November 1, pp. 196–202. IEEE Computer Society Press, Los Alamitos (1989)
10. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. 22(2), 221–242 (1993)
11. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
12. Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 377–388. Springer, Heidelberg (2005)
13. Bozanis, P., Kitsios, N., Makris, C., Tsakalidis, A.: New upper bounds for generalized intersection searching problems. In: Fülöp, Z., Gécseg, F. (eds.) ICALP 1995. LNCS, vol. 944, pp. 464–474. Springer, Heidelberg (1995)
14. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica 63(4), 815–830 (2012)
15. Brodal, G.S., Fagerberg, R., Greve, M., López-Ortiz, A.: Online sorted range reporting. In: [28], pp. 173–182
16. Brodal, G.S., Gfeller, B., Jørgensen, A.G., Sanders, P.: Towards optimal range medians. Theoret. Comput. Sci. 412(24), 2588–2601 (2011)
17. Brodal, G.S., Jørgensen, A.G.: Data structures for range median queries. In: [28], pp. 822–831
18. Brodnik, A., Karlsson, J., Munro, J.I., Nilsson, A.: An O(1) solution to the prefix sum problem on a specialized memory architecture. In: Navarro, G., Bertossi, L.E., Kohayakawa, Y. (eds.) TCS 2006. IFIP, vol. 209, pp. 103–114. Springer, Boston (2006)
19. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. In: Dürr, C., Wilke, T. (eds.) 29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, Paris, France, February 29-March 3. LIPIcs, vol. 14, pp. 290–301. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
20. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. Theory Comput. Sys., 1–23 (2013)
21. Chan, T.M., Durocher, S., Skala, M., Wilkinson, B.T.: Linear-space data structures for range minority query in arrays. In: [36], pp. 295–306
22. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. 17(3), 427–462 (1988)

23. Chazelle, B., Rosenberg, B.: The complexity of computing partial sums off-line. Internat. J. Comput. Geom. Appl. 1(1), 33–45 (1991)
24. Chun, S.J., Chung, C.W., Lee, J.H., Lee, S.L.: Dynamic update cube for range-sum queries. In: Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T. (eds.) Proceedings of the Twenty-seventh International Conference on Very Large Data Bases, Roma, Italy, September 11-14, pp. 521–530. Morgan Kaufmann Publishers (2001)
25. de Berg, M., Haverkort, H.J.: Significant-presence range queries in categorical data. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 462–473. Springer, Heidelberg (2003)
26. Demaine, E.D., Landau, G.M., Weimann, O.: On Cartesian trees and range minimum queries. In: [2], pp. 341–353
27. Dobkin, D., Munro, J.I.: Determining the mode. Theoret. Comput. Sci. 12(3), 255–263 (1980)
28. Dong, Y., Du, D.-Z., Ibarra, O. (eds.): ISAAC 2009. LNCS, vol. 5878. Springer, Heidelberg (2009)
29. Durocher, S.: A simple linear-space data structure for constant-time range minimum query. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Munro Festschrift 2013. LNCS, vol. 8066, pp. 48–60. Springer, Heidelberg (2013)
30. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 244–255. Springer, Heidelberg (2011)
31. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. Inform. and Comput. 222, 169–179 (2013)
32. Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. In: [5], pp. 150–159
33. Eppstein, D. (ed.): Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Mathematics (SODA 2002), January 6-8. ACM Press, New York (2002)
34. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
35. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
36. Fomin, F.V., Kaski, P. (eds.): SWAT 2012. LNCS, vol. 7357. Springer, Heidelberg (2012)
37. Fredman, M.L.: The complexity of maintaining an array and computing its partial sums. J. ACM 29(1), 250–260 (1982)
38. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, April 30-May 2, pp. 135–143. ACM Press, Washington, DC (1984)
39. Gagie, T., He, M., Munro, J.I., Nicholson, P.K.: Finding frequent elements in compressed 2D arrays and strings. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 295–300. Springer, Heidelberg (2011)
40. Gagie, T., Kärkkäinen, J.: Counting colours in compressed strings. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 197–207. Springer, Heidelberg (2011)

41. Gagie, T., Kärkkäinen, J., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. Theoret. Comput. Sci. 483, 36–50 (2013)
42. Gagie, T., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
43. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
44. Geffner, S., Agrawal, D., Abbadi, A.E., Smith, T.R.: Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In: Kitsuregawa, M., Papazoglou, M.P., Pu, C. (eds.) Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, pp. 328–335. IEEE Computer Society (1999)
45. Gfeller, B., Sanders, P.: Towards optimal range medians. In: [2], pp. 475–486
46. Golynski, A.: Optimal lower bounds for rank and select indexes. Theoret. Comput. Sci. 387(3), 348–359 (2007)
47. Greve, M., Jørgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds and approximations for range mode. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part I. LNCS, vol. 6198, pp. 605–616. Springer, Heidelberg (2010)
48. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), pp. 841–850. ACM/SIAM (2003)
49. Gupta, P., Janardan, R., Smid, M.: Further results on generalized intersection searching problems: Counting, reporting, and dynamization. J. Algorithms 19(2), 282–317 (1995)
50. Hampapuram, H., Fredman, M.L.: Optimal biweighted binary trees and the complexity of maintaining partial sums. SIAM J. Comput. 28(1), 1–9 (1998)
51. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
52. He, M., Munro, J.I.: Space efficient data structures for dynamic orthogonal range counting. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 500–511. Springer, Heidelberg (2011)
53. He, M., Munro, J.I., Nicholson, P.K.: Dynamic range selection in linear space. In: [5], pp. 160–169
54. He, M., Munro, J.I., Nicholson, P.K.: Dynamic range selection in linear space. CoRR abs/1106.5076v3 (2013), http://arxiv.org/abs/1106.5076v3
55. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: [5], pp. 140–149
56. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 575–586. Springer, Heidelberg (2012)
57. Ho, C.T., Agrawal, R., Megiddo, N., Srikant, R.: Range queries in OLAP data cubes. In: Peckman, J.M. (ed.) Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997, Tucson, Arizona, USA, May 13-15. SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 26(2), pp. 73–88. ACM Press (1997)
58. Hon, W.K., Shah, R., Thankachan, S.V.: Towards an optimal space-and-query-time index for top-k document retrieval. In: [64], pp. 173–184

59. Hon, W.K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, Atlanta, Georgia, USA, October 25-27, pp. 713–722. IEEE Computer Society (2009)
60. Jacobson, G.: Space-efficient static trees and graphs. In: 30th Annual Symp. on Foundations of Computer Science, vol. 30, pp. 549–554 (1989)
61. Jacobson, G.: Succinct Static Data Structures. PhD thesis, Carnegie-Mellon, Technical Report CMU-CS-89-112 (January 1989)
62. Janardan, R., Lopez, M.: Generalized intersection searching problems. Internat. J. Comput. Geom. Appl. 3(1), 39–69 (1993)
63. Jørgensen, A.G., Larsen, K.G.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: [83], pp. 805–813
64. Kärkkäinen, J., Stoye, J. (eds.): CPM 2012. LNCS, vol. 7354. Springer, Heidelberg (2012)
65. Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proceedings of the 20th Annual Canadian Conference on Computational Geometry, Montréal, Canada, August 13-15 (2008)
66. Karpinski, M., Nekrich, Y.: Top-k color queries for document retrieval. In: [83], pp. 401–411
67. Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, pp. 517–526. Springer, Heidelberg (2003)
68. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nord. J. Comput. 12(1), 1–17 (2005)
69. Lai, Y., Poon, C., Shi, B.: Approximate colored range and point enclosure queries. J. Discrete Algorithms 6(3), 420–432 (2008)
70. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoret. Comput. Sci. 387(3), 332–347 (2007)
71. Munro, J.I., Spira, P.M.: Sorting and searching in multisets. SIAM J. Comput. 5(1), 1–8 (1976)
72. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
73. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: [33], pp. 657–666
74. Navarro, G.: Wavelet trees for all. In: [64], pp. 2–26
75. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 15–26. Springer, Heidelberg (2007)
76. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Comput. Geom. 42(4), 342–351 (2009)
77. Nekrich, Y., Navarro, G.: Sorted range reporting. In: [36], pp. 271–282
78. Patil, M., Shah, R., Thankachan, S.V.: Succinct representations of weighted trees supporting path queries. J. Discrete Algorithms 17, 103–108 (2012)
79. Patrascu, M.: Succincter. In: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, Philadelphia, Pennsylvania, USA, October 25-23, pp. 305–313. IEEE (2008)
80. Petersen, H.: Improved bounds for range mode and range median queries. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 418–423. Springer, Heidelberg (2008)

81. Petersen, H., Grabowski, S.: Range mode and range median queries in constant time and sub-quadratic space. Inf. Process. Lett. 109(4), 225–228 (2009)
82. Poon, C.K.: Dynamic orthogonal range queries in OLAP. Theoret. Comput. Sci. 296(3), 487–510 (2003)
83. Randall, D. (ed.): Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25. SIAM (2011)
84. Sadakane, K.: Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In: [35], pp. 225–232
85. Yao, A.C.C.: On the complexity of maintaining partial sums. SIAM J. Comput. 14(2), 277–288 (1985)

Indexes for Document Retrieval with Relevance*

Wing-Kai Hon[1], Manish Patil[2], Rahul Shah[2],
Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Louisiana State University, USA
{mpatil,rahul,thanks}@csc.lsu.edu
[3] The University of Kansas, USA
jsv@ku.edu

Abstract. Document retrieval is a special type of pattern matching that is closely related to information retrieval and web searching. In this problem, the data consist of a collection of text documents, and given a query pattern P, we are required to report all the documents (not all the occurrences) in which this pattern occurs. In addition, the notion of *relevance* is commonly applied to rank all the documents that satisfy the query, and only those documents with the highest relevance are returned. Such a concept of relevance has been central in the effectiveness and usability of present day search engines like Google, Bing, Yahoo, or Ask. When relevance is considered, the query has an additional input parameter k, and the task is to report only the k documents with the highest relevance to P, instead of finding all the documents that contains P. For example, one such relevance function could be the frequency of the query pattern in the document. In the information retrieval literature, this task is best achieved by using inverted indexes. However, if the query consists of an arbitrary string—which can be a partial word, multiword phrase, or more generally any sequence of characters—we cannot take advantages of the word boundaries and we need a different approach.

This leads to one of the active research topics in string matching and text indexing community in recent years, and various aspects of the problem have been studied, such as space-time tradeoffs, practical solutions, multipattern queries, and I/O-efficiency. In this article, we review some of the initial frameworks for designing such indexes and also summarize the developments in this area.

1 Introduction

Query processing forms a central aspect of databases which in turn is supported by data structures that are commonly referred to as *indexes*. In databases, the notion of queries is semantically well-defined; hence a tuple (or a record) either

* This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah and J. S. Vitter) and CCF–1218904 (R. Shah).

A. Brodnik et al. (Eds.): Munro Festschrift, LNCS 8066, pp. 351–362, 2013.

qualifies or does not qualify for the query, and a database operation will return exactly all those tuples that satisfy the query conditions. In contrast, information retrieval takes a somewhat fuzzy approach on query processing. The data are often unstructured and the notions of precision, recall, and relevance add their flavors to which tuples are returned. Often the criteria for a tuple to satisfy the query is not just a binary decision. The notion of *relevance-ranking* is central to information retrieval where the output is ranked by relevance score—which is an indicator of how strongly the tuple (or a web document, in case of search engines) matches the query. In recent times, various extensions to the standard relational database model have been proposed to cope with an increasing need to integrate databases and information retrieval. Top-k query processing is one such line of research, which adds the notion of relevance to database query processing.

Formally, a top-k query comes with a parameter k. Amongst all tuples that satisfy the query, they are ranked by their relevance scores, and only the k most relevant tuples are reported. In document retrieval and duplicate elimination (as a part of the projection operation) in databases, we get multiple occurrences of the same tuple (or key) satisfying the query and relevance depends on the contribution of each such tuple. In this case, only one tuple (out of the multiple occurrences) is to be reported with composite score. A simple example of such a score function is the frequency —which is number of times a particular attribute occurs in the query result. In terms of web-search this is known as *term-frequency*, which is the number of times the query term occurs in a given document. There can be even more complex statistical scoring functions, for instance when one considers OLAP queries (with slice-and-dice type ranges).

In terms of document retrieval, we are given a set $\mathcal{D}=\{d_1, d_2, d_3, ..., d_D\}$ of D string documents of total length n. We build an index on this collection. Then pattern P (of length p) comes as an query, and we are required to output the list of all *ndoc* documents in which the pattern P appears (not all *occ* occurrences). This is called the *document listing* problem and was introduced by Matias et al. [27]. Muthukrishnan [28] gave the first optimal $O(p + ndoc)$ query time solution in linear space, i.e., $O(n)$ words. Since then, this has been an active research area [37,40,15] with focus on making the index space-efficient. In *top-k document retrieval*, there is a relevance score involved in addition to the uniqueness condition. Let $S(P, d_i)$ be the set of occurrences of pattern P in the document d_i. The relevance score of P with respect to d_i is a function $w(P, d_i)$ that depends only on the set $S(P, d_i)$. Now, as a query result, we are required to report only the top-k highest scoring documents. The formal definition is given below:

Problem 1 (Top-k document retrieval problem). *Let $w(P, d)$ be the score function capturing the relevance of a pattern P with respect to a document d. Given a document collection $\mathcal{D}= \{d_1, d_2, ..., d_D\}$ of D documents, build an index answering the following query: given input P and k, find k documents d with the highest $w(P, d)$ values in sorted (or unsorted) order.*

This problem was introduced in [17], where they proposed an $O(n \log D)$ words index with query time $O(p + k + \log D \log\log D)$ (works only for

document-frequency as the score function). The recent flurry of activities [5,10,14,19,23,25,31,36,18,24,21,26,39,41] came with Hon et al.'s work [22]. In this survey article, we review the various aspects of top-k document retrieval as listed below:

- We begin by describing the linear space and optimal time (internal memory) framework based on the work of Hon et al. [22] and of Navarro and Nekrich [30].
- In Section 3, we focus on the I/O model [3] solution for top-k document retrieval by Shah et al. [38] that occupies almost-linear $O(n \log^* n)$ space and can answer queries in $O(p/B + \log_B n + k/B)$ I/Os.
- In Section 4 we briefly explain the first succinct index that was proposed by Hon et al. [22] occupying roughly twice the size of text with $O(p+k \log^{O(1)} n)$ query time and also review the later developments in this line of work.
- We also briefly discuss variants of document retrieval problem in Section 5 such as multipattern queries, queries with forbidden pattern, parameterized top-k queries.
- Finally, we conclude in Section 6 by listing some of the interesting open problems in this research area.

2 Linear Space Framework

This section briefly explains the linear space framework for top-k document retrieval based on the work of Hon, Shah and Vitter [22] and Navarro and Nekrich [30]. The generalized suffix tree (GST) of a document collection $\mathcal{D}= \{d_1, d_2, d_3, \ldots, d_D\}$ is the combined compact trie (a.k.a. Patricia trie) of all the non-empty suffixes of all the documents. We use n to denote the total length of all the documents, which is also the number of the leaves in GST. For each node u in GST, consider the path from the root node to u. Let $\mathit{depth}(u)$ be the number of nodes on the path, and $\mathit{prefix}(u)$ be the string obtained by concatenating all the edge labels of the path. For a pattern P that appears in at least one document, the *locus* of P, denoted as u_P, is the node closest to the root satisfying that P is a prefix of $\mathit{prefix}(u_P)$. By numbering all the nodes in GST in the pre-order traversal manner, the part of GST relevant to P (i.e., the subtree rooted at u_P) can be represented as a range.

Nodes are marked with document-ids. A leaf node ℓ is marked with a document $d \in \mathcal{D}$ if the suffix represented by ℓ belongs to d. An internal node u is marked with d if it is the lowest common ancestor of two leaves marked with d. Notice that a node can be marked with multiple documents. For each node u and each of its marked documents d, define a *link* to be a quadruple $(\mathit{origin}, \mathit{target}, \mathit{doc}, \mathit{score})$, where $\mathit{origin} = u$, target is the lowest proper ancestor[1] of u marked with d, $\mathit{doc} = d$, and $\mathit{score} = w\big(\mathit{prefix}(u), d\big)$. Two crucial properties of the links identified in [22] are listed below.

[1] Define a dummy node as the parent of the root node, marked with all the documents.

- For each document d that contains a pattern P, there is a unique link whose *origin* is in the subtree of u_P and whose *target* is a proper ancestor of u_P. The *score* of the link is exactly the score of d with respect to P.
- The total number of links is bounded by $O(n)$.

We say that a link is *stabbed* by node u if it is originated in the subtree of u and targets a proper ancestor of u. Therefore, top-k document retrieval can be viewed as the problem of indexing the $O(n)$ links described above to efficiently report the k highest scored links stabbed by any given node u_P. By mapping each link $L_i = (o_i, t_i, doc, score_i)$ to a 3d point $(x_i, y_i, z_i) = (o_i, depth(t_i), score_i)$, the above problem can be reduced to the following range searching query: report k points with the highest z coordinate among those points with $x_i \in [u_P, u'_P]$ and $y_i < depth(u_P)$, which is a 4-constraint query. Here u'_P represents the pre-order rank of the right most leaf in the subtree of u_P.

While general 4-sided orthogonal range searching is proved hard [6], the main idea is to make use of the special property that the reduce subproblem can only have p distinct values, hence it can be decomposed into p 3-constrained queries (which can be solved optimally). Thus a linear space index with near-optimal $O(p + k \log k)$ is achieved by Hon et al. [22]. This query time is improved to optimal $O(p + k)$ by Navarro and Nekrich [30].

Theorem 1. *There exists a linear space index of $O(n)$-word space for answering top-k document retrieval queries in optimal $O(p + k)$ time.*

Nekrich [30] showed that the index space can be reduced to $O(n(\log \sigma + \log D + \log \log n))$ bits, if the requirement is to retrieve only the top-k documents without their associated scores. With *term-frequency* as the score they achieved the index that is further compressed occupying $O(n(\log \sigma + \log D))$ bits. Hon et al. [18] proposed an alternative approach to directly compress the index to achieve an $n(1 + o(1))(\log \sigma + 2 \log D)$ bits index with $O(p + k \log \log n + poly \log \log n)$ query time.

3 External-Memory Framework

With the advent of enterprise search, deep desktop search, and email search technologies, the indexes that reside on disks (external memory) are more and more important. Unfortunately, the (linear space) approach described in the previous section cannot lead to an optimal external memory solution as it inevitably adds an extra $O(p)$ additive factor in query time. Therefore, we need to explore some other properties that can potentially simplify the problem. In this section, we briefly describe the I/O-efficient framework by

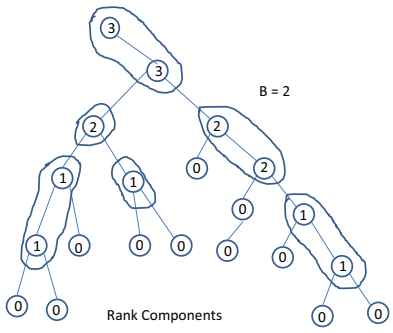


Fig. 1. Rank Components

Shah et al. [38]. They showed how to decompose the 4-constrained query (as described in the previous section) into at most $\log(n/B)$ (instead of p) 3-constrained queries, by exploring the fact that, out of four constraints in the given query, two of them always correspond to a tree range. Here B denotes the disk block size.

Here we solve a threshold variant of the problem (i.e., among all those links stabbed by u_P, retrieve those with weight at least a given threshold τ). Note that, both threshold and top-k variants are equivalent due to the existence of a linear-space structure to compute threshold τ given (u_P, k) in $O(1)$ time such that the number of number of outputs reported by threshold variant of the problem is between k and $k + O(k + \log n)$. It is known that no linear-space external memory structure can answer the (even the simpler) 1d top-k range reporting query in $O(\log^{O(1)} n + k/B)$ I/Os if the output order must be ensured [2]. We thus turn our attention to solving the unordered variant of the top-k document retrieval problem.

We start with some definitions: Let $size(u)$ denote the number of leaves in the subtree of u. We define the *rank* of u,

$$rank(u) = \left\lfloor \log\lceil \frac{size(u)}{B} \rceil \right\rfloor$$

Note that $rank(\cdot) \in [0, \lfloor \log\lceil \frac{n}{B} \rceil \rfloor]$ and nodes with the same rank will form a contiguous subtree, and we call each subtree a *component* (see Figure 1). The *rank* of a component is defined as the rank of nodes within it.

We classify the links into the following three types based on the *rank* of its target with respect to the *rank* of query node u_P: *low-ranked links*: links with $rank(target) < rank(u_P)$, *high-ranked links*: links with $rank(target) > rank(u_P)$, *equi-ranked links*: links with $rank(target) = rank(u_P)$. The links within each of these categories can be processed separately as follows:

1. None of the low-ranked links can be an output as their target will not be an ancestor of u_P, hence can be ignored while querying.
2. For a high-ranked link L_i, if $o_i \in [u_P, u'_P]$, then the condition that t_i is an ancestor of u_P will be implicitly satisfied. Thus, we are left with only 3-constraints, which can be modeled as a 3-sided query [2,4].
3. We group together all the links whose target node t_i belongs to component C to form a set S_C. Further we replace the origin o_i in each of the links by its lowest ancestor s_i within C (Figure 2). Then, an equi-ranked link $L_i \in C$ is an output iff $t_i < u_P \leq s_i$ and $score_i \geq \tau$, which can be modeled as a 3d dominance query [1].

Putting everything together, the top-k document retrieval problem can be reduced to $O(\log(n/B))$ 3-constraint queries. Thus, by maintaining appropriate structures for handling such queries, we can obtain a linear-space index with $O(\log^2(n/B) + k/B)$ I/Os, which is optimal for $k \geq B \log^2(n/B)$. For optimally handling the case when k is small, bootstrapping techniques are introduced (for details we refer to [38]). We summarize the main result in the following Theorem.

Theorem 2. *There exists external memory index of almost-linear* $O(n \log^* n)$ *words space for answering top-*k *document retrieval queries in optimal* $O(p/B + \log_B n + k/B)$ *I/Os.*

If the score function is monotonic, the top-k document retrieval problem can be reduced to the *top-k categorical range maxima query* (Top-CRMQ) problem. Given an integer array $A[1...n]$ and associated category (color) array $C[1...n]$, where each $A[i]$ has an associated color $C[i]$, we apply range top-k query (a, b, k) to find the top-k (distinct) colors in the range $[a, b]$. The notion of top-k associates a score with each color c occurring in the query range, where the score of a color c in the range $[a, b]$ is $max\{A[i]|i \in [a, b]$ *and* $C[i] = c\}$. We can now model the top-k document retrieval problem into Top-CRMQ: arrange all links in the ascending order of origin, then construct arrays A and C such that $A[i]$ represents the score of the ith link and $C[i]$ represents the document to which it belongs. Now, top-k document retrieval is equivalent to Top-CRMQ on A with $[a, b]$ as the input range, where $[a, b]$ represents the maximal range of all links with origin within the subtree of u_P. Thus by integrating with the recent solution for the Top-CRMQ problem by Nekrich et al. [35], an external memory top-k document retrieval index with space $O(n\alpha(B))$-words and query I/O bound $O(p/B + k/B + \log_B n + \alpha(B))$ can be obtained, where $\alpha(\cdot)$ is the inverse Ackermann function.

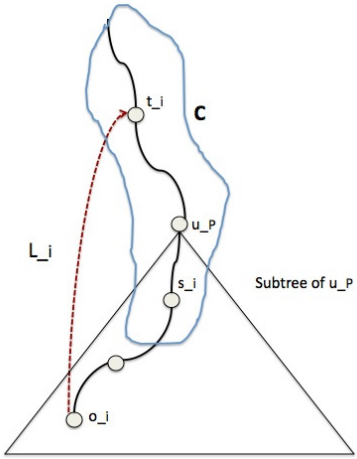


Fig. 2. Pseudo Origin

4 Succinct Frameworks

In the succinct framework, the goal is achieve the index space proportional to the size of text (i.e., $n \log \sigma$ bits). We use the *score* function to be term-frequency. We begin this section by briefly explaining the marking scheme introduced by Hon et al. [22] and then review the later developments in this line of work.

Marked Nodes in GST: Certain nodes in the *GST* can be identified as marked nodes with respect to a parameter g called the *grouping factor* as follows. The procedure starts by combining every g consecutive leaves (from left to right) together as a group, and marking the lowest common ancestor (LCA) of the first and last leaves in each group. Further, we mark the LCA of all pairs of marked nodes. Additionally, we ensure that the root is always marked. At the end of this procedure, the number of marked nodes in *GST* will be $O(n/g)$. Hon et al. [22] showed that, given any node u with u^* being its highest marked

descendent (if exists), number of leaves in $GST(u \backslash u^*)$ (i.e., the number of leaves in the subtree of u, but not in the subtree of u^*) is at most $2g$.

We begin by the describing the data structure for a top-k document retrieval problem for a fixed k. First, We implement the marking scheme in GST as described above with $g = k \log^{2+\epsilon} n$, where $\epsilon > 0$ is any constant. The top-k documents corresponding to each of the $O(n/g)$ marked nodes (as the locus) are maintained explicitly in $O(k \log n)$ bits, for a total of $O((n/g)k \log n) = o(n/\log n)$ bits. In order to answer a top-k query, we first find the locus node u_P, and then its highest marked descendent node u_P^*. If a document d is in the top-k list with respect to node u_P, then either it is in the top-k list with respect to u_P^* as well or there is at least one leaf in the $GST(u_P \backslash u_P^*)$ with the corresponding suffix in document d. By using this observation, we can obtain a set of $O(g+k)$ possible candidate documents. By computing the term frequencies of each document in the candidate set, we can identify the documents in the final output. Note that instead of a GST, we maintain its compressed variant. An additional $|CSA|$ bits structure is used for computing term-frequency in $O(\log^{2+\epsilon} n)$ time, where CSA represents the compressed suffix array [11,16] of the concatenated text of all documents, and $|CSA|$ represents its size in bits. Thus the query time can be bounded by $O(p + k \log^{4+2\epsilon} n)$. In order to handle top-k queries for any general k, we maintain the above described data structure for $k = 1, 2, 4, 8, ...$, with overall space requirement roughly equal to twice that of the input text.

Theorem 3. *There exists a succinct data structure of space roughly twice the size of text (in compressed form) with query time $O(\log^{4+\epsilon} n)$ per reported document.*

A series of work has been done to improve the above succinct index. The per-document retrieval time is improved to $O(\log k \log^{2+\epsilon} n)$ by Belazzougui and Navarro [5], whereas the fastest succinct index is by Hon et al. [21], where the query time is $O(\log k \log^{1+\epsilon} n)$. Note that the space occupancy of all these succinct indexes is roughly twice the size of text. An interesting open question to design a space optimal index (i.e., $|CSA| + o(n)$ bits) has been positively answered by Tsur [39], where the per-document report time is $O(\log k \log^{2+\epsilon} n)$. Very recently, Navarro and Thankachan [33] improved the query time of Tsur's index to $O(\log^2 k \log^{1+\epsilon} n)$, and is currently the fastest space optimal index.

Instead of using an additional CSA for document frequency computation of the candidate document, an alternative approach is to use a data structure called the document array $E[1...n]$, where $E[i]$ denotes the document to which the suffix corresponding to ith leftmost leaf in GST belongs to. The resulting index space is $|CSA| + n \log D(1 + o(1))$ bits. The first result of the kind is due to Gagie et al. [14] with per-document report time is $O(\log^{2+\epsilon} n)$, which was improved to $O(\log k \log^{1+\epsilon} n)$ by Belazzougui and Navarro [5], and to $O((\log \sigma \log \log n)^{1+\epsilon} n)$ by Hon et al. [18]. Here σ represents the alphabet size. Culpepper et al. [10] have proposed another document array-based index. Even though their query algorithm is only a heuristic (no worst-case bound), it is one of the simplest and most efficient indexes in practice. Another trade-off is by Gagie et al. [14], where

the index space is $|CSA| + O(\frac{n \log D}{\log \log D})$ bits and query time is $O(\log^{3+\epsilon} n)$. This result is also improved by Belazzougui and Navarro [5], where they achieved by a per-document report time of $O(\log k \log^{2+\epsilon} n)$ with an index space of $|CSA| + O(n \log \log \log D)$ bits.

5 Variants of Document Retrieval

In this section, we briefly describe some of the variants of document retrieval problem along with the known results.

5.1 Two-Pattern Document Listing

In this case, the query consists of two patterns P_1 and P_2 (of length p_1 and p_2 respectively), and the task is to report all those *ndoc* documents containing both P_1 and P_2. The first solution was given by [28], which requires $\tilde{O}(n^{3/2})$ space and answers a query in $O(p_1+p_2+\sqrt{n}+ndoc)$ time[2]. Clearly, this solution is not practical due to its huge space requirement. Cohen and Porat [8] showed that this problem can be reduced to set-intersection. Based on their elegant framework for the the set-intersection problem, they proposed an $O(n \log n)$-word space index with $O(p_1 + p_2 + \sqrt{n \times ndoc} \log^{2.5} n)$ query time. Later Hon et al. [19] improved the space as well as the query time of Cohen and Porat's index to $O(n)$-word and $O(p_1+p_2+\sqrt{n \times ndoc} \log^{1.5} n)$ respectively. In addition, Hon et al. [19] extended their solution to handle multipattern queries (i.e., query input consists of two or more patterns) and also to top-k queries. Using Geometric-BWT techniques [7], Fischer et al. [12] showed that in pointer machine model, any index for two-pattern document listing with query time $O(p_1 + p_2 + \log^{O(1)} n + ndoc)$ must require $\Omega(n(\log n/ \log \log n)^3)$ bits space.

5.2 Forbidden/Excluded Pattern Queries

A variant of a two-pattern document listing is pattern matching with forbidden (excluded) pattern. Given two patterns P_1 and P_2, the goal is to list all *ndoc* documents containing P_1 *but not* P_2. Fischer et al. [12] introduced the problem and proposed an index of size $O(n^{1.5})$ bits with query time $O(p_1 + p_2 + \sqrt{n} + ndoc)$. Recently, Hon et al. [20] gave a space-efficient solution for this problem, occupying linear space of $O(n)$ words. However, the query time is increased to $O(p_1 + p_2 + \sqrt{n \times ndoc} \log^{2.5} n)$.

5.3 Parameterized Top-k Queries

In this case, the query consists of two parameters x and y ($x \leq y$) in addition to P and k and the task is to retrieve the top-k documents with highest $w(P, \cdot)$ among

[2] The notation $\tilde{O}$ ignores poly-logarithmic factors. Precisely, $\tilde{O}(f(n)) \equiv O(f(n) \log^{O(1)} n)$.

only those documents d with $Par(P, d) \in [x, y]$, where $Par(\cdot, \cdot)$ is a predefined function. Navarro and Nekrich [30] showed that such queries can be answered in $O(p + (k + \log n) \log^{\epsilon} n)$ time by maintaining a linear-space index. For the case when $w(\cdot, \cdot)$ is page rank, $Par(\cdot, \cdot)$ is term-frequency and y is unbounded, Karpinski and Nekrich [25] gave an optimal query time data structure with $O(n \log D)$-word space.

6 Conclusions and Open Problems

In this article, we briefly reviewed some of the theoretical frameworks for designing top-k document retrieval indexes in different settings. However, we have not covered the details of practical solutions [24,36,26,32,34] as well as some of the other related topics (we recommend the recent article by Navarro [29] for an exhaustive survey). Even though many efficient solutions are already available for the central problem, there are still many interesting variations and open questions one could ask. We conclude with some of them as listed below:

1. The current I/O-optimal index requires $O(n \log^* n)$-word space. It is interesting to see if we can bring down this space to linear (i.e., $O(n)$ words) without sacrificing the optimality in the I/O bound. Designing these indexes in the Cache-Oblivious model is another future research direction.
2. The optimal space-compressed index (by Navarro and Thankachan [33]) takes $O(\log^2 k \log^{1+\epsilon} n)$ query time. The fastest compressed space index (by Hon et al. [21]) takes twice the size of text. An interesting problem is to design a space-optimal index, while keeping the query time the same (or better) as that of the fastest compressed index known.
3. Top-kth document retrieval: instead of reporting all top-k documents, report the kth highest-scored document corresponding to the query.
4. Top-k version of forbidden pattern query: the query consists of P_1, P_2, and k, and the task is to report the top-k documents based on $w(P_1, \cdot)$ among all those documents d which does not contain the forbidden pattern P_2.
5. Another space-time trade-off for parametrized top-k query. For example, design an optimal query time index using $O(n \log^{\epsilon} n)$ words of space.
6. Currently the gap between the upper and lower bound for two-pattern query problem is huge. It is interesting to see if this gap can be reduced. Can we obtain similar (or better) lower bounds for the forbidden pattern query problem. We strongly believe that the lower bounds for this problems are different from the currently known upper bounds [12,20] by at most $poly \log n$ factors only.
7. Even though many succinct indexes have been proposed for top-k queries for frequency or page-rank based score functions, it is still unknown if such a succinct index can be designed if the score function is term-proximity (i.e., $w(P, d)$ is the difference between the positions of the closest occurrences of P in document d). Designing such an index even for special cases (say, long patterns or allow approximate score, etc), or deriving lower bounds are interesting research directions.

8. Approximate pattern matching (i.e., allowing bounded errors and don't cares) is another active research area [9]. Adding this aspect to document retrieval leads to many new problems. The following is one such problem: report all those documents in which the edit (or hamming) distance between one of its substrings and P is at most π, where $\pi \geq 1$ is an input parameter.
9. Indexing a highly repetitive document collection (which is highly compressible using LZ-based compression techniques) is an active line of research. In the recent work by Gagie et al. [13], an efficient document retrieval index suitable for a repetitive collection is proposed. An open problem is to extend these results for handling top-k queries.

References

1. Afshani, P.: On dominance reporting in 3D. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 41–51. Springer, Heidelberg (2008)
2. Afshani, P., Brodal, G.S., Zeh, N.: Ordered and unordered top-k range reporting in large data sets. In: SODA, pp. 390–400 (2011)
3. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
4. Arge, L., Samoladas, V., Vitter, J.S.: On two-dimensional indexability and optimal range search indexing. In: Proc. 18th Symposium on Principles of Database Systems (PODS), pp. 346–357 (1999)
5. Belazzougui, D., Navarro, G.: Improved compressed indexes for full-text document retrieval. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 386–397. Springer, Heidelberg (2011)
6. Chazelle, B.: Lower bounds for orthogonal range searching: I. the reporting case. J. ACM 37(2), 200–212 (1990)
7. Chien, Y.-F., Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: Geometric burrows-wheeler transform: Compressed text indexing via sparse suffixes and range searching. Algorithmica (2013)
8. Cohen, H., Porat, E.: Fast set intersection and two-patterns matching. Theor. Comput. Sci. 411(40-42), 3795–3800 (2010)
9. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100 (2004)
10. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
11. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
12. Fischer, J., Gagie, T., Kopelowitz, T., Lewenstein, M., Mäkinen, V., Salmela, L., Välimäki, N.: Forbidden patterns. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 327–337. Springer, Heidelberg (2012)
13. Gagie, T., Karhu, K., Navarro, G., Puglisi, S.J., Sirén, J.: Document listing on repetitive collections. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 107–119. Springer, Heidelberg (2013)
14. Gagie, T., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)

15. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. 426, 25–41 (2012)
16. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
17. Hon, W.-K., Patil, M., Shah, R., Wu, S.-B.: Efficient index for retrieving top-k most frequent documents. J. Discrete Algorithms 8(4), 402–417 (2010)
18. Hon, W.-K., Shah, R., Thankachan, S.V.: Towards an optimal space-and-query-time index for top-k document retrieval. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 173–184. Springer, Heidelberg (2012)
19. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: String retrieval for multi-pattern queries. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 55–66. Springer, Heidelberg (2010)
20. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: Document listing for queries with excluded pattern. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 185–195. Springer, Heidelberg (2012)
21. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed top-k document retrieval. In: DCC (2013)
22. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: FOCS 2009, pp. 713–722 (2009)
23. Hon, W.-K., Shah, R., Vitter, J.S.: Compression, indexing, and retrieval for massive string data. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 260–274. Springer, Heidelberg (2010)
24. Culpepper, M.P.J.S., Scholer, F.: Efficient in-memory top-k document retrieval. In: SIGIR (2012)
25. Karpinski, M., Nekrich, Y.: Top-k color queries for document retrieval. In: SODA, pp. 401–411 (2011)
26. Konow, R., Navarro, G.: Faster Compact Top-k Document Retrieval. In: DCC (2013)
27. Matias, Y., Muthukrishnan, S.M., Şahinalp, S.C., Ziv, J.: Augmenting suffix trees, with applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
28. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
29. Navarro, G.: Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. CoRR, abs/1304.6023 (2013)
30. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: SODA, pp. 1066–1077 (2012)
31. Navarro, G., Puglisi, S.J.: Dual-sorted inverted lists. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 309–321. Springer, Heidelberg (2010)
32. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical compressed document retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)
33. Navarro, G., Thankachan, S.V.: Faster top-k document retrieval in optimal space (submitted)
34. Navarro, G., Valenzuela, D.: Space-efficient top-k document retrieval. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 307–319. Springer, Heidelberg (2012)
35. Nekrich, Y., Patil, M., Shah, R., Thankachan, S.V., Vitter, J.S.: Top-k categorical range maxima queries (submitted)
36. Patil, M., Thankachan, S.V., Shah, R., Hon, W.-K., Vitter, J.S., Chandrasekaran, S.: Inverted indexes for phrases and strings. In: SIGIR, pp. 555–564 (2011)

37. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discrete Algorithms 5(1), 12–22 (2007)
38. Shah, R., Sheng, C., Thankachan, S.V., Vitter, J.S.: On optimal top-k string retrieval. CoRR, abs/1207.2632 (2012)
39. Tsur, D.: Top-k document retrieval in optimal space. Inf. Process. Lett. 113(12), 440–443 (2013)
40. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
41. Vitter, J.S.: Compressed data structures with relevance. In: CIKM, pp. 4–5 (2012)

Author Index