T

Table Compression

2003; Buchsbaum, Fowler, Giancarlo

Adam L. Buchsbaum[1], Raffaele Giancarlo[2]
[1] Shannon Laboratory, AT&T Labs, Inc., Florham Park, NJ, USA
[2] Department of Mathematics and Computer Science, University of Palermo, Palermo, Italy

Keywords and Synonyms

Compression of multi-dimensional data; Storage, compression and transmission of tables; Compressive estimates of entropy

Problem Definition

Table compression was introduced by Buchsbaum et al. [2] as a unique application of compression, based on several distinguishing characteristics. Tables are collections of fixed-length records and can grow to be terabytes in size. They are often generated by information systems and kept in data warehouses to facilitate ongoing operations. These data warehouses will typically manage many terabytes of data online, with significant capital and operational costs. In addition, the tables must be transmitted to different parts of an organization, incurring additional costs for transmission. Typical examples are tables of transaction activity, like phone calls and credit card usage, which are stored once but then shipped repeatedly to different parts of an organization: for fraud detection, billing, operations support, etc. The goals of table compression are to be fast, online, and effective: eventual compression ratios of 100:1 or better are desirable. Reductions in required storage and network bandwidth are obvious benefits.

Tables are different than general databases [2]. Tables are written once and read many times, while databases are subject to dynamic updates. Fields in table records are fixed in length, and records tend to be homogeneous; database records often contain intermixed fixed- and variable-length fields. Finally, the goals of compression differ. Database compression stresses index preservation, the ability to retrieve an arbitrary record, under compression [6]. Tables are typically not indexed at the level of individual records; rather, they are scanned in toto by downstream applications.

Consider each record in a table to be a row in a matrix. A naive method of table compression is to compress the string derived from scanning the table in row-major order. Buchsbaum et al. [2] observe experimentally that partitioning the table into contiguous intervals of columns and compressing each interval separately in this fashion can achieve significant compression improvement. The partition is generated by a one-time, offline training procedure, and the resulting compression strategy is applied online to the table. In their application, tables are generated continuously, so offline training time can be ignored. They also observe heuristically that certain rearrangements of the columns prior to partitioning further improve compression by grouping dependent columns more closely. For example, in a table of addresses and phone numbers, the area code can often be predicted by the zip code when both are defined geographically. In information-theoretic terms, these dependencies are *contexts*, which can be used to predict parts of a table. Analogously to strings, where knowledge of context facilitates succinct codings of a symbols, the existence of contexts in tables implies, in principle, the existence of a more succinct representation of the table.

Two main avenues of research have followed, one based on the notion of combinatorial dependency [2,3] and the other on the notion of column dependency [14, 15]. The first formalizes dependencies analogously to the joint entropy of random variables, while the second does so analogously to conditional entropy [7]. These approaches to table compression have deep connections to universal similarity metrics [11], based on Kolmogorov complexity and compression, and their later uses in classification [5]. Both approaches are instances of a new emerging paradigm for data compression, referred to as *boost-*

ing [8], where data are reorganized to improve the performance of a given compressor. A software platform to facilitate the investigation of such invertible data transformations is described by Vo [16].

Notations

Let T be a table of $n = |T|$ columns and m rows. Let $T[i]$ denote the ith column of T. Given two tables T_1 and T_2, let T_1T_2 be the table formed by their juxtaposition. That is, $T = T_1T_2$ is defined so that $T[i] = T_1[i]$ for $1 \le i \le |T_1|$ and $T[i] = T_2[i - |T_1|]$ for $|T_1| < i \le |T_1| + |T_2|$. We use the shorthand $T[i, j]$ to represent the projection $T[i] \cdots T[j]$ for any $j \ge i$. Also, given a sequence P of column indices, we denote by $T[P]$ the table obtained from T by projecting the columns with indices in P.

Combinatorial Dependency and Joint Entropy of Random Variables

Fix a compressor C: e. g., gzip, based on LZ77 [17]; compress, based on LZ78 [18]; or bzip, based on Burrows–Wheeler [4]. Let $H_C(T)$ be the size of the result of compressing table T as a string in row-major order using C. Let $H_C(T_1, T_2) = H_C(T_1T_2)$.$H_C(\cdot)$ is thus a cost function defined on the ordered power set of columns. Two tables T_1 and T_2, which might be projections of columns from a common table T, are *combinatorially dependent* if $H_C(T_1, T_2) < H_C(T_1) + H_C(T_2)$ – if compressing them together is better than compressing them separately – and *combinatorially independent* otherwise. Buchsbaum et al. [3] show that combinatorial dependency is a compressive estimate of statistical dependency when formalized by the joint entropy of two random variables, i. e., the statistical relatedness of two objects is measured by the gain realized by compressing them together rather than separately. Indeed, combinatorial dependency becomes statistical dependency when H_C is replaced by the joint entropy function [7]. Analogous notions starting from Kolmogorov complexity are derived by Li et al. [11] and used for classification and clustering [5]. Figure 1 exemplifies why rearranging and partitioning columns may improve compression.

Problem 1 *Find a partition $\mathcal{P}$ of T into sets of contiguous columns that minimizes $\sum_{Y\in\mathcal{P}} H_C(Y)$ over all such partitions.*

Problem 2 *Find a partition $\mathcal{P}$ of T that minimizes $\sum_{Y\in\mathcal{P}} H_C(Y)$ over all partitions.*

The difference between Problems 1 and 2 is that the latter does not require the parts of $\mathcal{P}$ to be sets of contiguous columns.

9	0	8	2	7	3
9	0	8	3	7	5
9	0	8	5	7	6
9	0	8	2	7	5

Table Compression, Figure 1
The first three columns of the table, taken in row-major order, form a repetitive string that can be very easily compressed. Therefore, it may be advantageous to compress these columns separately. If the fifth column is swapped with the fourth, we get an even longer repetitive string that, again, can be compressed separately from the other two columns

Column Dependency and Conditional Entropy of Random Variables

Definition 1 For any table T, a *dependency relation* is a pair (P, c) in which P is a sequence of distinct column indices (possibly empty) and $c \notin P$ is another column index. If the length of P is less than or equal to k, then (P, c) is called a k-relation. P is the *predictor sequence* and c is the *predictee*.

Definition 2 Given a dependency relation (P, c), the *dependency transform* $\mathrm{dt}_P(c)$ of c is formed by permuting column $T[c]$ based on the permutation induced by a stable sort of the rows of P.

Definition 3 A collection D of dependency relations for table T is said to be a k-transform if and only if: (a) each column of T appears exactly once as a predictee in some dependency relation (P, c); (b) the dependency hypergraph $G(D)$ is acyclic; (c) each dependency relation (P, c) is a k-relation.

Let $\omega(P, c)$ be the cost of the dependency relation (P, c), and let $\delta(m)$ be an upper bound on the cost of computing $\omega(P, c)$. Intuitively, $\omega(P, c)$ gives an estimate of how well a rearrangement of column c will compress, using the rows of P as contexts for its symbols. We will provide an example after the formal definitions.

Problem 3 *Find a k-transform D of minimum cost $\omega(D) = \sum_{(P,c)\in D} \omega(P, c)$.*

Definition 1 extends to columns the notion of context that is well known for strings. Definition 3 defines a microtransformation that reorganizes the column symbols by grouping together those that have similar contexts. The context of a column symbol is given by the corresponding row in $T[P]$. The fundamental ideas here are the same as in the Burrows and Wheeler transform [4]. Finally, Problem 3 asks for an optimal strategy to reorganize the data prior to compression. The cost function ω provides an es-

timate of how well c can be compressed using the knowledge of $T[P]$.

Vo and Vo [14] connect these ideas to the conditional entropy of random variables. Let S be a sequence, $\mathcal{A}(S)$ its distinct elements, and f_a the frequency of each element a. The *zeroth-order empirical entropy* of S [13] is

$$H_0(S) = -\frac{1}{|S|} \sum_{\alpha \in \mathcal{A}(S)} f_a \lg \frac{f_a}{|S|} ,$$

and the *modified zeroth order empirical entropy* [13] is

$$H_0^*(S) = \begin{cases} 0 & \text{if } |S| = 0 , \\ (1 + \lg |S|)/|S| & \text{if } |S| \neq 0 \text{ and } H_0(S) = 0 , \\ H_0(S) & \text{otherwise} . \end{cases}$$

For a dependency relation (P, c) with nonempty P, the *modified conditional empirical entropy* of c given P is then defined as

$$H_P^*(c) = \frac{1}{m} \sum_{\rho \in \mathcal{A}(T[P])} |\rho_c| H_0^*(\rho_c) ,$$

where ρ_c is the string formed by catenating the symbols in c corresponding to positions of ρ in $T[P]$ [14]. A possible choice of $\omega(P, c)$ is given by $H_P^*(c)$. Vo and Vo also develop another notion of entropy, called *run length entropy*, to approximate more effectively the compressibility of low-entropy columns and define another cost function ω accordingly.

Key Results

Combinatorial Dependency

Problem 1 admits a polynomial-time algorithm, based on dynamic programming. Using the definition of combinatorial dependency, one can show:

Theorem 1 ([2]) *Let $E[i]$ be the cost of an optimal, contiguous partition of $T[1, i]$. $E[n]$ is thus the cost of a solution to Problem 1. Define $E[0] = 0$; then, for $1 \leq i \leq n$,*

$$E[i] = \min_{0 \leq j < i} E[j] + H_C(T_{j+1}, \ldots, T_i) . \qquad (1)$$

The actual partition with cost $E[n]$ can be maintained by standard backtracking.

The only known algorithmic solution to Problem 2 is the trivial one based on enumerating all possible feasible solutions to choose an optimal one. Some efficient heuristics based on asymmetric TSP, however, have been devised and tested experimentally [3]. Define a weighted, complete, directed graph, $G(T)$, with a vertex T_i for each column $T[i] \in T$; the *weight* of edge $\{T_i, T_j\}$ is $w(T_i, T_j) = \min(H_C(T_i, T_j), H_C(T_i) + H_C(T_j))$. One then generates a set of tours of various weights by iteratively applying standard optimizations (e. g., 3-opt, 4-opt). Each tour induces an ordering of the columns, which are then optimally partitioned using the dynamic program (1).

Buchsbaum et al. [3] also provide a general framework for studying the computational complexity of several variations of table compression problems based on notions analogous to combinatorial dependence, and they give some initial MAX-SNP-hardness results. Particularly relevant is the set of abstract problems in which one is required to find an optimal arrangement of a set of strings to be compressed, which establishes a nontrivial connection between table compression and the classical shortest common superstring problem [1]. Giancarlo et al. [10] connect table compression to the Burrows and Wheeler transform [4] by deriving the latter as a solution to an analog of Problem 2.

Column Dependency

Theorem 2 ([14,15]) *For $k \geq 2$, Problem 3 is NP-hard.*

Theorem 3 ([14,15]) *An optimum 1-transform for a table T can be found in $O(n^2 \delta(m))$ time.*

Theorem 4 ([14,15]) *A 2-transform can be computed in $O(n^2 \delta(m))$ time.*

Theorem 5 ([14]) *For any dependency relation (P, c) and some constant ϵ, $|C(\mathrm{dt}_P(c))| \leq 5 m H_p^*(c) + \epsilon$.*

Applications

Storage and transmission of alphanumeric tables.

Open Problems

All the techniques discussed use the general paradigms of context-dependent data rearrangement for compression boosting. It remains open to apply these paradigms to other domains, e. g., XML data [9,12], where high-level structures can be exploited, and to domains where pertinent structures are not known a priori.

Experimental Results

Buchsbaum et al. [2] showed that optimal partitioning alone (no column rearrangement) yielded about 55% better compression compared to gzip on telephone usage data, with small training sets. Buchsbaum et al. [3] experimentally supported the hypothesis that good TSP heuristics can effectively reorder the columns, yielding additional improvements of 5 to 20% relative to partitioning

alone. They extended the data sets used to include other tables from the telecom domain as well as biological data. Vo and Vo [14,15] showed further 10 to 35% improvement over these combinatorial dependency methods on the same data sets.

Data Sets

Some of the data sets used for experimentation are public [3].

URL to Code

The pzip package, based on combinatorial dependency, is available at http://www.research.att.com/~gsf/pzip/pzip.html. The Vcodex package, related to invertible transforms, is available at http://www.research.att.com/~gsf/download/ref/vcodex/vcodex.html. Although for the time being Vcodex does not include procedures to compress tabular data, it is a useful toolkit for their development.

Cross References

- ▶ Binary Decision Graph
- ▶ Burrows–Wheeler Transform
- ▶ Dictionary-Based Data Compression
- ▶ Succinct Data Structures for Parentheses Matching
- ▶ Tree Compression and Indexing

Recommended Reading

1. Blum, A., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. J. ACM **41**, 630–47 (1994)
2. Buchsbaum, A.L., Caldwell, D.F., Church, K.W., Fowler, G.S., Muthukrishnan, S.: Engineering the compression of massive tables: An experimental approach. In: Proc. 11th ACM-SIAM Symp. on Discrete Algorithms, 2000, pp. 175–84
3. Buchsbaum, A.L., Fowler, G.S., Giancarlo, R.: Improving table compression with combinatorial optimization. J. ACM **50**, 825–851 (2003)
4. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
5. Cilibrasi, R., Vitanyi, P.M.B.: Clustering by compression. IEEE Trans. Inf. Theory **51**, 1523–1545 (2005)
6. Cormack, G.: Data compression in a data base system. Commun. ACM **28**, 1336–1350 (1985)
7. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley Interscience, New York, USA (1990)
8. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM **52**, 688–713 (2005)
9. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring Labeled Trees for Optimal Succinctness, and beyond. In: Proc. 45th Annual IEEE Symposium on Foundations of Computer Science, 2005, pp. 198–207
10. Giancarlo, R., Sciortino, M., Restivo, A.: From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. Theor. Comput. Sci. (2007)
11. Li, M., Chen, X., Li, X., Ma, B., Vitanyi, P.M.B.: The similarity metric. IEEE Trans. Inf. Theory **50**, 3250–3264 (2004)
12. Liefke, H., Suciu, D.: XMILL: An efficient compressor for XML data. In: Proceedings of the 2000 ACM SIGMOD Int. Conf. on Management of Data, pp. 153–164. ACM, New York, USA (2000)
13. Lifshits, Y., Mozes, S., Weimann, O., Ziv-Ukelson, M.: Speeding up HMM decoding and training by exploiting sequence repetitions. Algorithmica to appear doi:10.1007/s00453-007-9128-0
14. Manzini, G.: An analysis of the Burrows–Wheeler transform. J. ACM **48**, 407–430 (2001)
15. Vo, B.D., Vo, K.-P.: Compressing table data with column dependency. Theor. Comput. Sci. **387**, 273–283 (2007)
16. Vo, B.D., Vo, K.-P.: Using column dependency to compress tables. In: DCC: Data Compression Conference, pp. 92–101. IEEE Computer Society TCC, Washington DC, USA (2004)
17. Vo., K.-P.: Compression as data transformation. In: DCC: Data Compression Conference. IEEE Computer Society TCC, pp. 403. Washington DCD, USA (2006)
18. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**, 337–343 (1977)
19. Ziv, J., Lempel, A.: Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theory **24**, 530–536 (1978)

Tail Bounds for Occupancy Problems

1995; Kamath, Motwani, Palem, Spirakis

Paul Spirakis
Computer Engineering and Informatics, Research and Academic Computer Technology Institute, Patras University, Patras, Greece

Keywords and Synonyms

Balls and bins

Problem Definition

Consider a *random allocation* of m balls to n bins where each ball is placed in a bin chosen uniformly and independently. The properties of the resulting distribution of balls among bins have been the subject of intensive study in the probability and statistics literature [3,4]. In computer science, this process arises naturally in randomized algorithms and probabilistic analysis. Of particular interest is the *occupancy problem* where the random variable under consideration is the number of empty bins.

In this entry a series of bounds are presented (reminiscent of the Chernoff bound for binomial distributions) on the tail of the distribution of the number of empty bins; the tail bounds are successively tighter, but each new bound

has a more complex closed form. Such strong bounds do not seem to have appeared in the earlier literature.

Key Results

The following notation in presenting sharp bounds on the tails of distributions. The notation $F \sim G$ will denote that $F = (1 + o(1))G$; further, $F \asymp G$ will denote that $\ln F \sim \ln G$. The proof that $f \asymp g$, is used for the purposes of later claiming that $2^f \asymp 2^g$. These asymptotic equalities will be treated like actual equalities and it will be clear that the results claimed are unaffected by this "approximation".

Consider now the probabilistic experiment of throwing m balls, independently and uniformly, into n bins.

Definition 1 Let Z be the number of empty bins when m balls are placed randomly into n bins, and define $r = m/n$. Define the function $H(m, n, z)$ as the probability that $Z = z$. The expectation of Z is given by

$$\mu = \mathbf{E}[Z] = n\left(1 - \frac{1}{n}\right)^m \sim n\,\mathrm{e}^{-r} .$$

The following three theorems provide the bounds on the tail of the distribution of the random variable Z. The proof of the first bound is based on a martingale argument.

Theorem 1 (Occupancy Bound 1) *For any $\theta > 0$,*

$$\boldsymbol{P}\left[|Z - \mu| \geq \theta\mu\right] \leq 2\exp\left(-\frac{\theta^2\mu^2(n - \frac{1}{2})}{n^2 - \mu^2}\right) .$$

Remark that for large r this bound is asymptotically equal to

$$2\exp\left(-\frac{\theta^2\,\mathrm{e}^{-2r}n}{1 - \mathrm{e}^{-2r}}\right) .$$

The reader may wish to compare this with the following heuristic estimate of the tail probability assuming that the distribution of Z is well approximated by the approximating normal distribution also far out in the tails [3,4].

$$\boldsymbol{P}\left[|Z - \mu| \geq \theta\mu\right] \leq 2\exp\left(-\frac{\theta^2\,\mathrm{e}^{-r}n}{2\,(1 - (1 + r)\,\mathrm{e}^{-r})}\right) .$$

The next two bounds are in terms of point probabilities rather than tail probabilities (as was the case in the Binomial Bound), but the unimodality of the distribution implies that the two differ by at most a small (linear) factor. These more general bounds on the point probability are essential for the application to the satisfiability problem. The next result is obtained via a generalization of the Binomial Bound to the case of dependent Bernoulli trials.

Theorem 2 (Occupancy Bound 2) *For $\theta > -1$,*

$$H(m, n, (1 + \theta)\mu) \leq \exp\left(-\left((1 + \theta)\ln[1 + \theta] - \theta\right)\mu\right) .$$

In particular, for $-1 \leq \theta < 0$,

$$H(m, n, (1 + \theta)\mu) \leq \exp\left(-\frac{\theta^2\mu}{2}\right) .$$

The last result is proved using ideas from large deviations theory [7].

Theorem 3 (Occupancy Bound 3) *For $|z - \mu| = \Omega(n)$,*

$$H(m, n, z) \asymp \exp\left(\left[-n\left(\int_0^{1-\frac{z}{n}} \ln\left[\frac{k - x}{1 - x}\right] \mathrm{d}x - r\ln k\right)\right]\right)$$

where k is defined implicitly by the equation $z = n(1-k(1-\mathrm{e}^{-r/k}))$.

Applications

Random allocations of balls to bins is a basic model that arises naturally in many areas in computer science involving choice between a number of resources, such as communication links in a network of processors, actuator devices in a wireless sensor network, processing units in a multi-processor parallel machine etc. For such situations, randomization can be used to "spread" the load evenly among the resources, an approach particularly useful in a parallel or distributed environment where resource utilization decisions have to be made locally at a large number of sites without reference to the global impact of these decisions. In the process of analyzing the performance of such algorithms, of particular interest is the occupancy problem where the random variable under consideration is the number of empty bins (i. e., machines with no jobs, routes with no load, etc.). The properties of the resulting distribution of balls among bins and the corresponding tails bounds may help in order to analyze the performance of such algorithms.

Cross References

- ► Approximation Schemes for Bin Packing
- ► Bin Packing

Recommended Reading

1. Kamath, A., Motwani, R., Spirakis, P., Palem, K.: Tail bounds for occupancy and the satisfiability threshold conjecture. J. Random Struct. Algorithms **7**(1), 59–80 (1995)

2. Janson, S.: Large Deviation Inequalities for Sums of Indicator Variables. Technical Report No. 34, Department of Mathematics, Uppsala University (1994)
3. Johnson, N.L., Kotz, S.: Urn Models and Their Applications. Wiley, New York (1977)
4. Kolchin, V.F., Sevastyanov, B.A., Chistyakov, V.P.: Random Allocations. Wiley, New York (1978)
5. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, New York (1995)
6. Shwartz, A., Weiss, A.: Large Deviations for Performance Analysis. Chapman-Hall, Boca Raton (1994)
7. Weiss, A.: Personal Communication (1993)

Technology Mapping

1987; Keutzer

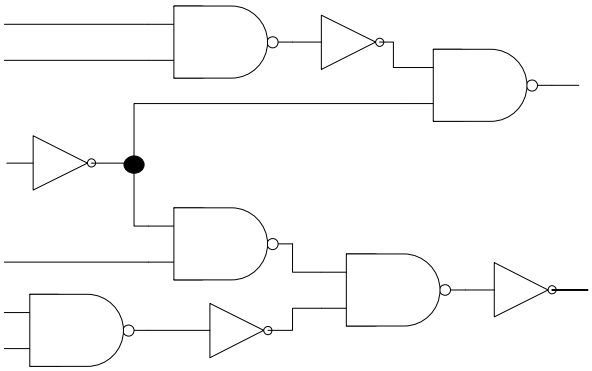
KURT KEUTZER, KAUSHIK RAVINDRAN
Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA

Keywords and Synonyms

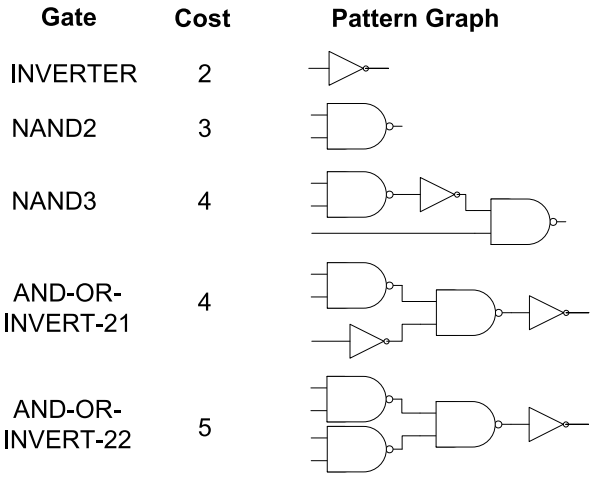
Library-based technology mapping; Technology dependent optimization

Problem Definition

Technology mapping is the problem of implementing a sequential circuit using the gates of a particular technology library. It is an integral component of any automated VLSI circuit design flow. In the prototypical chip design flow, combinational logic gates and sequential memory elements are composed to form sequential circuits. These circuits are subject to various logic optimizations to minimize area, delay, power and other performance metrics. The resulting optimized circuits still consist of primitive logic functions such as AND and OR gates. The next step is to efficiently realize these circuits in a specific VLSI technology using a library of gates available from the semiconductor vendor. Such a library would typically consist of gates of varying sizes and speeds for primitive logic functions, (AND and OR) and more complex functions (exclusive-OR, multiplexer). However, a naïve translation of generic logic elements to gates in the library will fall short of realistic performance goals. The challenge is to construct a mapping that maximally utilizes the gates in the library to implement the logic function of the circuit and achieve some performance goal—for example, minimum area with the critical path delay less than a target value. This is accomplished by *technology mapping*. For the sake of simplicity, in the following discussion it is presumed that the sequential memory elements are stripped from the digital circuit and mapped directly into memory elements of the particular technology. Then, only Boolean circuits composed of combinational logic gates remain to be mapped. Further, each remaining Boolean circuit is necessarily a directed acyclic graph (DAG).



Technology Mapping, Figure 1
Subject graph (DAG) of a Boolean circuit expressed using NAND2 and INVERTER gates



Technology Mapping, Figure 2
Library of pattern graphs (composed of NAND2 and INVERTER gates) and associated costs

The technology mapping problem can be restated in a more general graph-theoretic setting: *find a minimum cost covering of the subject graph (Boolean circuit) by choosing from the collection of pattern graphs (gates) available in a library*. The inputs to the problem are:

(a) *Subject graph*: This is a directed acyclic graph representation of a Boolean circuit expressed using a set of primitive functions (e. g., 2-input NAND gates and inverters). An example subject graph is shown in Fig. 1.

(b) *Library of pattern graphs*: This is a collection of gates available in the technology library. The pattern graphs are also DAGs expressed using the same primitive

functions used to construct the subject graph. Additionally, each gate is annotated with a number of values for different cost functions, such as area, delay, and power. An example library and associated cost model is shown in Fig. 2.

A *valid cover* is a network of pattern graphs implementing the function of the subject graph such that: (a) every vertex (i. e. gate) of the subject graph is contained in some pattern graph, and (b) each input required by a pattern graph is actually an output of some other pattern graph (i. e. the inputs of a gate must exist as outputs of other gates). Technology mapping can then be viewed as an optimization problem to find a valid cover of minimum cost of the subject graph.

Key Results

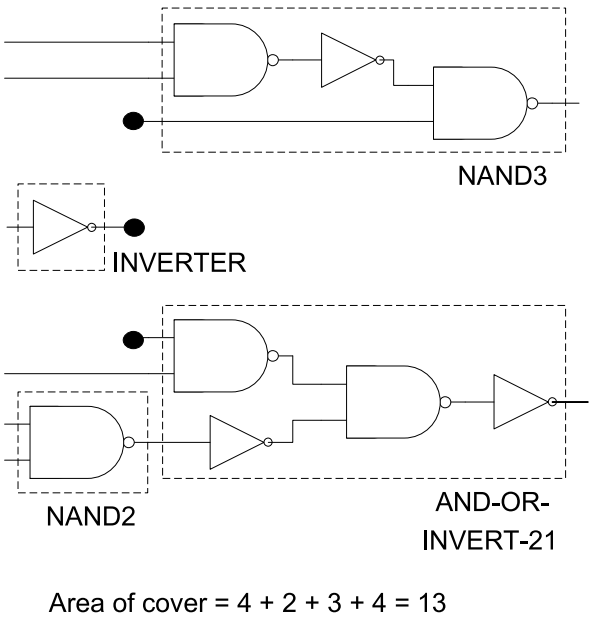
To be viable in a realistic design flow, an algorithm for minimum cost graph covering for technology mapping should ideally possess the following characteristics: (a) the algorithm should be easily adaptable to diverse libraries and cost models—if the library is expanded or replaced, the algorithm must be able to utilize the new gates effectively, (b) it should allow detailed cost models to accurately represent the performance of the gates in the library, and (c) it should be fast and robust on large subject graph instances and large libraries. One technique for solving the minimum cost graph covering problem is to formulate it as a binate-covering problem, which is a specialized integer linear program [5]. However, binate covering for a DAG is NP-Hard for any set of primitive functions and is typically unwieldy on large circuits. The DAGON algorithm suggested solving the technology mapping problem through DAG covering and advanced an alternate approach for DAG covering based on a *tree covering* approximation that produced near-optimal solutions for practical circuits and was very fast even for large circuits and large libraries [4].

DAGON was inspired by prevalent techniques for pattern matching employed in the domain of code generation for programming language compilers [1]. The fundamental concept was to partition the subject graph (DAG) into a forest of trees and solve the minimum cost covering problem independently for each tree. The approach was motivated by the existence of efficient dynamic programming algorithms for optimum tree covering [2]. The three salient components of the DAGON algorithm are: (a) subject graph partitioning, (b) pattern matching, and (c) covering.

(a) Subject graph partitioning: To apply the tree covering approximation the subject graph is first partitioned into a forest of trees. One approach is to break the graph at each vertex which has an out-degree greater than 1 (multiple fan-out point). The root of each tree is the primary output of the corresponding sub-circuit and the leaves are the primary inputs. Other heuristic partitions of the subject graph that consider duplication of vertices can also be applied to improve the quality of the final cover. Alternate subject graph partitions can also be derived starting from different decompositions of the original Boolean circuit in terms of the primitive functions.

(b) Pattern matching: The optimum covering of a tree is determined by generating the complete set of matches for each vertex in the tree (i. e. the set of pattern graphs which are candidates for covering a particular vertex) and then selecting the optimum match from among the candidates. An efficient approach for structural pattern matching is to reduce the tree matching problem to a *string matching* problem [2]. Fast string matching algorithms, such as the Aho–Corasick and the Knuth–Morris–Pratt algorithms, can then be used to find all strings (pattern graphs) which match a given vertex in the subject graph in time proportional to the length of the longest string in the set of pattern graphs. Alternatively, Boolean matching techniques can be used to find matches based on logic functions [12]. Boolean matching is slower than structural string matching, but it can compute matches independent of the actual local decompositions and under different input permutations.

(c) Covering: The final step is to generate a valid cover of the subject tree using the pattern graph matches computed at each vertex. Consider the problem of finding a valid cover of minimum area for the subject tree. Every pattern graph in the library has an associated area and the area of a valid cover is the sum of the area of the pattern graphs in the cover. The key property that makes minimum area tree covering efficient is this: *the minimum area cover of a tree rooted at some vertex v can be computed using only the minimum area covers of vertices below v*. If follows that for every pattern graph that matches at vertex v, the area of the minimum cover containing that match equals the sum of the area of the corresponding match at v and the sum of the areas of the optimal covers of the vertices which are inputs to that match. This property enables a dynamic programming algorithm to compute the minimum area cover of tree rooted at each vertex of the subject tree. The base case is the minimum area cover of a leaf (primary input) of subject tree. The area of a match at a leaf is set to 0. A recursive formulation of this dynamic programming concept is summarized in the Algorithm `minimum_area_tree_cover` shown below. As an example, the minimum area cover displayed in Fig. 3 is



Technology Mapping, Figure 3
Result of a minimum area tree covering of the subject graph in Fig. 1 using the library of pattern graphs in Fig. 2

a result of applying this algorithm to the tree partitions of the subject graph from Fig. 1 using the library from Fig. 2.

Given a vertex v in the subject tree, let M(v) denote the set of candidate matches from the library of pattern graphs for the sub-tree rooted at v.

```
Algorithm minimum_area_tree_cover (
       Vertex v ) {
   // the algorithm minimum_area_tree_cover
   //    finds an optimal cover of the tree
   //    rooted at Vertex v
   // the algorithm computes best_match(v)
   //    and areas_of_best_match(v), which
   //    denote the best pattern graph match
   //    at v and the associated areas of
   //    the optimal cover of the tree rooted
   //    at v respectively

   // check if v is a leaf of the tree
   if ( v is a leaf) {
      area_of_best_match(v) = 0;
      best_match(v) = leaf;
      return;
   }

   // compute optimal cover for each input
   //    of v
   foreach ( input of Vertex v ) {
       minimum_area_tree_cover( input );
   }
   // each tree rooted at each input of v is
   //    now annotated with its optimal cover

   // find the optimal cover of the tree
   //    rooted at Vertex v
   area_of_best_match(v) = INFINITY;
   best_match(v) = NULL;

   foreach ( Match m in the set of matches
       M(v) ) {
      // compute the area of match m at
      //    Vertex v
      // area_of_match(v,m) denotes the area
      //    of the cover when Match m is
      //    selected for v
      area_of_match(v,m) = area(m);
      foreach input pin v_i of matche m {
         area_of_match (v,m) =
            area_of_match(v,m) +
               area_of_best_match(v_i );
      }

      // update best pattern graph match
      //    and associated area of the optimal
      //    cover at Vertex v
      if ( area_of_match(v,m) <
          area_of_best_match(v) ) {
            area_of_best_match(v) =
                area_of_match(v,m);
            best_match(v) = m;
      }

   }
}
```

In this algorithm each vertex in the tree is visited exactly once. Hence, the complexity of the algorithm is proportional to the number of vertices in the subject tree times the maximum number of pattern matches at any vertex. The maximum number of matches is a function of the pattern graph library and is independent of the subject tree size. As a result, the complexity of computing the minimum cost valid cover of a tree is linear in the size of the subject tree, and the memory requirements are also linear in the size of the subject tree. The algorithm computes the optimum cover when the subject graph is a tree. In the general case of the subject graph being a DAG, empirical results have shown that the tree covering approximation yields industrial-quality results achieving aggressive area and timing requirements on large real circuit design problems [11,13].

Applications

Technology mapping is the key link between technology independent logic synthesis and technology dependent physical design of VLSI circuits. This motivates the need for efficient and robust algorithms to implement large Boolean circuits in a technology library. Early algorithms

for technology mapping were founded on rule-based local transformations [3]. DAGON was the first in advancing an algorithmic foundation in terms of graph transformations that was practicable in the inner loop of iterative procedures in the VLSI design flow [4]. From a theoretical standpoint, the graph covering formulation provided a formal description of the problem and specified optimality criteria for evaluating solutions. The algorithm was naturally adaptable to diverse libraries and cost models, and was relatively easy to implement and extend. The concept of partitioning the subject graph into trees and covering the trees optimally was effective for varied optimization objectives such as area, delay, and power. The DAGON approach has been incorporated in academic (SIS from the University of California at Berkeley [6]) and industrial (Synopsys™ Design Compiler) tool offerings for logic synthesis and optimization.

The graph covering formulation has also served as a starting point for advancements in algorithms for technology mapping over the last decade. Decisions related to logic decomposition were integrated in the graph covering algorithm, which in turn enabled technology independent logic optimizations in the technology mapping phase [9]. Similarly, heuristics were proposed to impose placement constraints and make technology mapping more aware of the physical design and layout of the final circuit [10]. To combat the problem of high power dissipation in modern submicron technologies, the graph algorithms were enhanced to minimize power under area and delay constraints [8]. Specializations of these graph algorithms for technology mapping have found successful application in design flows for Field Programmable Gate Array (FPGA) technologies [7]. We recommend the following works for a comprehensive treatment of algorithms for technology mapping and a survey of new developments and challenges in the design of modern VLSI circuits: [11,12,13].

Open Problems

The enduring problem with DAGON-related technology mappers is handling non-tree pattern graphs that arise from modeling circuit elements such as multiplexors, Exclusive-Ors, or memory-elements (e. g. flip-flops) with associated logic (e. g. scan logic). On the other hand, approaches that do not use the tree-covering formulation face challenges in easily representing diverse technology libraries and in matching the subject graph in a computationally efficient manner.

Cross References

► Sequential Exact String Matching

Recommended Reading

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. pp. 557–584. Addison Wesley, Boston (1986)
2. Aho, A., Johnson, S.: Optimal Code Generation for Expression Trees. J. ACM **23**(July), 488–501 (1976)
3. Darringer, J.A., Brand, D., Gerbi, J.V., Joyner, W.H., Trevillyan, L.H.: LSS: Logic Synthesis through Local Transformations. IBM J. Res. Dev. **25**, 272–280 (1981)
4. Keutzer, K.: DAGON: Technology Binding and Local Optimizations by DAG Matching. In: Proc. of the 24th Design Automation Conference **28**(1), pp. 341–347. Miami Beach, June 1987
5. Rudell, R.: Logic Synthesis for VLSI Design. Ph. D. thesis, University of California at Berkeley, ERL Memo 89/49, April 1989
6. Sentovich, E.M., Singh, K.J., Moon, C., Savoj, H., Brayton, R.K., Sangiovanni-Vincentelli, A.: Sequential Circuit Design using Synthesis and Optimization. In: Proc. of the IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD), pp. 328–333. Cambridge, October 1992
7. Cong, J., Ding, Y.: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs. In: Proc. of the 1992 IEEE/ACM International Conference on Computer-Aided Design (ICCAD-92) **8**(12), pp. 48–53, November 1992
8. Tiwari, V., Ashar, P., Malik, S.: Technology Mapping for Low Power in Logic Synthesis. Integr. VLSI J. **20**(3), 243–268 (1996)
9. Lehman, E., Watanabe, Y., Grodstein, J., Harkness, H.: Logic Decomposition during Technology Mapping. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **16**(8), 813–834, (1997)
10. Kutzschebauch, T., Stok, L.: Congestion Aware Layout Driven Logic Synthesis. In: Proc. of the IEEE/ACM International Conference on Computer-Aided Design, 2001, pp. 216–223
11. Devadas, S., Ghosh, A., Keutzer, K.: Logic Synthesis. McGraw Hill, New York (1994). pp. 185–200
12. De Micheli, G.: Synthesis and Optimization of Digital Circuits, 1st edn., pp. 504–533. McGraw-Hill, New York (1994)
13. Stok, L., Tiwari, V.: Technology Mapping. In: Hassoun, S., Sasou, T. (eds.) Logic Synthesis and Verification, pp. 115–139. Kluwer International Series In Engineering And Coputer Science Series. Kluwer Academic Publisher, Norwell (2002)

Teleportation of Quantum States

1993; Bennett, Brassard, Crepeau, Jozsa, Peres, Wootters

Rahul Jain
Computer Science and Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada

Keywords and Synonyms

Quantum teleportation; Teleportation

Problem Definition

An *n-qubit quantum state* is a positive semi-definite operator of unit trace in the complex Hilbert space $\mathbb{C}^{2^n}$. A *pure* quantum state is a quantum state with a unique non-zero eigenvalue. A pure state is also often represented by the unique unit eigenvector corresponding to the unique non-zero eigenvalue. In this article the standard (ket, bra) notation is followed as is often used in quantum mechanics, in which $|v\rangle$ (called as 'ket v') represents a column vector and $\langle v|$ (called as 'bra v') represents its conjugate transpose. A *classical n-bit* state is simply a probability distribution on the set $\{0,1\}^n$.

Let $\{|0\rangle, |1\rangle\}$ be the standard basis for $\mathbb{C}^2$. For simplicity of notation $|0\rangle \otimes |0\rangle$ are represented as $|0\rangle|0\rangle$ or simply $|00\rangle$. Similarly $|0\rangle\langle 0|$ represents $|0\rangle \otimes \langle 0|$. An EPR pair is a special two-qubit quantum state defined as $|\psi\rangle \triangleq \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. It is one of the four *Bell states* which form a basis for $\mathbb{C}^4$.

Suppose there are two spatially separated parties Alice and Bob and Alice wants to send an arbitrary n-qubit quantum state ρ to Bob. Since classical communication is much more reliable, and possibly cheaper, than quantum communication, it is desirable that this task be achieved by communicating just classical bits. Such a procedure is referred to as *teleportation*.

Unfortunately, it is easy to argue that this is in fact not possible if arbitrary quantum states need to be communicated faithfully. However Bennett, Brassard, Crepeau, Jozsa, Peres, Wootters [2] presented the following nice solution to it.

Key Results

Alice and Bob are said to share an EPR pair if each hold one qubit of the pair. In this article a standard notation is followed in which classical bits are called 'cbits' and shared EPR pairs are called 'ebits'. Bennett et al. showed the following:

Theorem 1 *Teleportation of an arbitrary n-qubit state can be achieved with 2n cbits and n ebits.*

These shared EPR pairs are referred to as *prior entanglement* to the protocol since they are shared at the beginning of the protocol (before Alice gets her input state) and are independent of Alice's input state. This solution is a good compromise since it is conceivable that Alice and Bob share several EPR pairs at the beginning, when they are possibly together, in which case they do not require a quantum channel. Later they can use these EPR pairs to transfer several quantum states when they are spatially separated.

Now see how Bennett el al. [2] achieve teleportation. First note that in order to show Theorem 1 it is enough to show that a single qubit, which is possibly a part of a larger state ρ can be teleported, while preserving its entanglement with the rest of the qubits of ρ, using 2 cbits and 1 ebit. Also note that the larger state ρ can now be assumed to be a pure state without loss of generality.

Theorem *Let* $|\phi\rangle_{AB} = a_0|\phi_0\rangle_{AB}|0\rangle_A + a_1|\phi_1\rangle_{AB}|1\rangle_A$, *where* a_0, a_1 *are complex numbers with* $|a_0|^2 + |a_1|^2 = 1$. *Subscripts A, B (representing Alice and Bob respectively) on qubits signify their owner.*

It is possible for Alice to send two classical bits to Bob such that at the end of the protocol the final state is $a_0|\phi_0\rangle_{AB}|0\rangle_B + a_1|\phi_1\rangle_{AB}|1\rangle_B$.

Proof For simplicity of notation, let us assume below that $|\phi_0\rangle_{AB}$ and $|\phi_1\rangle_{AB}$ do not exist. The proof is easily modified when they do exist by tagging them along. Let an EPR pair $|\psi\rangle_{AB} = \frac{1}{\sqrt{2}}(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B)$ be shared between Alice and Bob. Let us refer to the qubit under concern that needs to be teleported as the input qubit.

The combined starting state of all the qubits is

$$|\theta_0\rangle_{AB} = |\phi\rangle_{AB}|\psi\rangle_{AB} = (a_0|0\rangle_A + a_1|1\rangle_A)\left(\frac{1}{\sqrt{2}}(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B)\right)$$

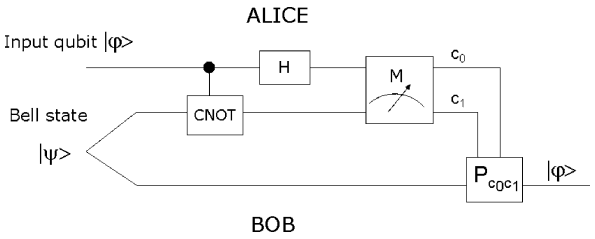
Let CNOT (*controlled-not*) gate be a two-qubit unitary operation described by the operator $|00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11|$. Alice now performs a CNOT gate on the input qubit and her part of the shared EPR pair. The resulting state is then,

$$|\theta_1\rangle_{AB} = \frac{a_0}{\sqrt{2}}|0\rangle_A(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) + \frac{a_1}{\sqrt{2}}|1\rangle_A(|1\rangle_A|0\rangle_B + |0\rangle_A|1\rangle_B)\ .$$

Let the Hadamard transform be a single qubit unitary operation with operator $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\langle 0| + \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\langle 1|$. Alice next performs a Hadamard transform on her input qubit. The resulting state then is,

$$\begin{aligned}|\theta_2\rangle_{AB} &= \frac{a_0}{2}(|0\rangle_A + |1\rangle_A)(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) \\ &\quad + \frac{a_1}{2}(|0\rangle_A - |1\rangle_A)(|1\rangle_A|0\rangle_B + |0\rangle_A|1\rangle_B) \\ &= \tfrac{1}{2}(|00\rangle_A(a_0|0\rangle_B + a_1|1\rangle_B) + |01\rangle_A(a_0|1\rangle_B \\ &\quad + a_1|0\rangle_B)) + \tfrac{1}{2}(|10\rangle_A(a_0|0\rangle_B - a_1|1\rangle_B) \\ &\quad + |11\rangle_A(a_0|1\rangle_B - a_1|0\rangle_B))\end{aligned}$$

Alice next measures the two qubits in her possession in the standard basis for $\mathbb{C}^4$ and sends the result of the measurement to Bob.



Teleportation of Quantum States, Figure 1
Teleportation protocol. H represent Hadamard transform and M represents measurement in the standard basis for $\mathbb{C}^4$

Let the four Pauli gates be the single qubit unitary operations: Identity: $P_{00} = |0\rangle\langle 0| + |1\rangle\langle 1|$, bit flip: $P_{01} = |1\rangle\langle 0| + |0\rangle\langle 1|$, phase flip: $P_{10} = |0\rangle\langle 0| - |1\rangle\langle 1|$ and bit flip together with phase flip: $P_{11} = |1\rangle\langle 0| - |0\rangle\langle 1|$. On receiving the two bits c_0c_1 from Alice, Bob performs the Pauli gate $P_{c_0c_1}$ on his qubit. It is now easily verified that the resulting state of the qubit with Bob would be $a_0|0\rangle_B + a_1|1\rangle_B$. The input qubit is successfully teleported from Alice to Bob! Please refer to Fig. 1 for the overall protocol. □

Super-Dense Coding

Super-dense coding [11] protocol is a dual to the teleportation protocol. In this Alice transmits 2 cbits of information to Bob using 1 qubit of communication and 1 shared ebit. It is discussed more elaborately in another article in the encyclopedia.

Lower Bounds on Resources

The above implementation of teleportation requires 2 cbits and 1 ebit for teleporting 1 qubit. It was argued in [2] that these resource requirements are also independently optimal. That is 2 cbits need to be communicated to teleport a qubit independent of how many ebits are used. Also 1 ebit is required to teleport one qubit independent of how much (possibly two-way) communication is used.

Remote State Preparation

Closely related to the problem of teleportation is the problem of *Remote state preparation* (RSP) introduced by Lo [10]. In teleportation Alice is just given the state to be teleported in some input register and has no other information about it. In contrast, in RSP, Alice knows a *complete description* of the input state that needs to be teleported. Also in RSP, Alice is not required to maintain any correlation of the input state with the other parts of a possibly larger state as is achieved in teleportation. The extra knowledge that Alice possesses about the input state can be used to devise protocols for probabilistically exact RSP with one cbit and one ebit per qubit asymptotically [3]. In a probabilistically exact RSP, Alice and Bob can abort the protocol with a small probability, however when they do not abort, the state produced with Bob at the end of the protocol, is exactly the state that Alice intends to send.

Teleportation as a Private Quantum Channel

The teleportation protocol that has been discussed in this article also satisfies an interesting privacy property. That is if there was a third party, say Eve, having access to the communication channel between Alice and Bob, then Eve learns nothing about the input state of Alice that she is teleporting to Bob. This is because the distribution of the classical messages of Alice is always uniform, independent of her input state. Such a channel is referred to as a *Private quantum channel* [1,6,8].

Applications

Apart from the main application of transporting quantum states over large distances using only classical channel, the teleportation protocol finds other important uses as well. A generalization of this protocol to implement unitary operations [7], is used in *Fault tolerant computation* in order to construct an infinite class of fault tolerant gates in a uniform fashion. In another application, a form of teleportation called as the error correcting teleportation, introduced by Knill [9], is used in devising quantum circuits that are resistant to very high levels of noise.

Experimental Results

Teleportation protocol has been experimentally realized in various different forms. To name a few, by Boschi et al. [4] using optical techniques, by Bouwmeester et al. [5] using photon polarization, by Nielsen et al. [12] using *Nuclear magnetic resonance* (NMR) and by Ursin et al. [13] using photons for long distance.

Cross References

► Quantum Dense Coding

Recommended Reading

1. Ambainis, A., Mosca, M., Tapp, A., de Wolf, R.: Private quantum channels. In: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, 2000, pp. 547–553
2. Bennett, C., Brassard G., Crepeau, C., Jozsa, R., Peres, A., Wootters, W.: Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. Phys. Rev. Lett. **70**, 1895–1899 (1993)

3. Bennett, C.H., Hayden, P., Leung, W., Shor, P.W., Winter, A.: Remote preparation of quantum states. IEEE Trans. Inform. Theory **51**, 56–74 (2005)
4. Boschi, D., Branca, S., Martini, F.D., Hardy, L., Popescu, S.: Experimental realization of teleporting an unknown pure quantum state via dual classical and einstein-podolski-rosen channels. Phys. Rev. Lett. **80**, 1121–1125 (1998)
5. Bouwmeester, D., Pan, J.W. , Mattle, K., Eible, M., Weinfurter, H., Zeilinger, A.: Experimental quantum teleportation. Nature **390**(6660), 575–579 (1997)
6. Boykin, P.O., Roychowdhury, V.: Optimal encryption of quantum bits. Phys. Rev. A **67**, 042317 (2003)
7. Chaung, I.L., Gottesman, D.: Quantum teleportation is a universal computational primitive. Nature **402**, 390–393 (1999)
8. Jain, R.: Resource requirements of private quantum channels and consequence for oblivious remote state preparation. Technical report (2005). arXive:quant-ph/0507075
9. Knill, E.: Quantum computing with realistically noisy devices. Nature **434**, 39–44 (2005)
10. Lo, H.-K.: Classical communication cost in distributed quantum information processing – a generalization of quantum communication complexity. Phys. Rev. A **62**, 012313 (2000)
11. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
12. Nielsen, M.A., Knill, E., Laflamme, R.: Complete quantum teleportation using nuclear magnetic resonance. Nature **396**(6706), 52–55 (1998)
13. Ursin, R., Jennewein, T., Aspelmeyer, M., Kaltenbaek, R., Lindenthal, M., Zeilinger, A.: Quantum teleportation link across the danube. Nature **430**(849), 849–849 (2004)

Text Indexing

1993; Manber, Myers

Srinivas Aluru
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA

Keywords and Synonyms

String indexing

Problem Definition

Text or string data naturally arises in many contexts including document processing, information retrieval, natural and computer language processing, and describing molecular sequences. In broad terms, the goal of text indexing is to design methodologies to store text data so as to significantly improve the speed and performance of answering queries. While text indexing has been studied for a long time, it shot into prominence during the last decade due to the ubiquity of web-based textual data and search engines to explore it, design of digital libraries for archiving human knowledge, and application of string techniques to further understanding of modern biology. Text indexing differs from the typical indexing of keys drawn from an underlying total order—text data can have varying lengths, and queries are often more complex and involve substrings, partial matches, or approximate matches.

Queries on text data are as varied as the diverse array of applications they support. Consequently, numerous methods for text indexing have been developed and this continues to be an active area of research. Text indexing methods can be classified into two categories: (i) methods that are generalizations or adaptations of indexing methods developed for an ordered set of one-dimensional keys, and (ii) methods that are specifically designed for indexing text data. The most classic query in text processing is to find all occurrences of a pattern P in a given text T (or equivalently, in a given collection of strings). Important and practically useful variants of this problem include finding all occurrences of P subject to at most k mismatches, or at most k insertions/deletions/mismatches. The focus in this entry is on these two basic problems and remarks on generalizations of one-dimensional data structures to handle text data.

Key Results

Consider the problem of finding a given pattern P in text T, both strings over alphabet Σ. The case of a collection of strings can be trivially handled by concatenating the strings using a unique end of string symbol, not in Σ, to create text T. It is worth mentioning the special case where T is structured—i. e., T consists of a sequence of words and the pattern P is a word. Consider a total order of characters in Σ. A string (or word) of length k can be viewed as a k-dimensional key and the order on Σ can be naturally extended to lexicographic order between multidimensional keys of variable length. Any one-dimensional search data structure that supports $O(\log n)$ search time can be used to index a collection of strings using lexicographic order such that a string of length k can be searched in $O(k \log n)$ time. This can be considerably improved as below [8]:

Theorem 1 *Consider a data structure on one-dimensional keys that relies on constant-time comparisons among keys (e. g., binary search trees, red-black trees etc.) and the insertion of a key identifies either its predecessor or successor. Let $O(\mathcal{F}(n))$ be the search time of the data structure storing n keys (e. g., $O(\log n)$ for red-black trees). The data structure can be converted to index n strings using $O(n)$ additional space such that the query for a string s can be performed in $O(\mathcal{F}(n))$ time if s is one of the strings indexed, and in $O(\mathcal{F}(n) + |s|)$ otherwise.*

A more practical technique that provides $O(\mathcal{F}(n) + |s|)$ search time for a string s under more restrictions on the underlying one-dimensional data structure is given in [9]. The technique is nevertheless applicable to several classic one-dimensional data structures, in particular binary search trees and its balanced variants. For a collection of strings that share long common prefixes such as IP addresses and XML path strings, a faster search method is described in [5].

When answering a sequence of queries, significant savings can be obtained by promoting frequently searched strings so that they are among the first to be encountered in a search path through the indexing data structure. Ciriani et al. [4] use self-adjusting skip lists to derive an expected bound for a sequence of queries that matches the information-theoretic lower bound.

Theorem 2 *A collection of n strings of total length N can be indexed in optimal O(N) space so that a sequence of m string queries, say $s_1,\cdots,s_m$, can be performed in $O(\sum_{j=1}^{m} |s_j| + \sum_{i=1}^{n} n_i \log(m/n_i))$ expected time, where n_i is the number of times the ith string is queried.*

Notice that the first additive term is a lower bound for reading the input, and the second additive term is a standard information-theoretic lower bound denoting the entropy of the query sequence. Ciriani et al. also extended the approach to the external memory model, and to the case of dynamic sets of strings. More recently, Ko and Aluru developed a self-adjusting tree layout for dynamic sets of strings in secondary storage that provides optimal number of disk accesses for a sequence of string or substring queries, thus providing a deterministic algorithm that matches the information-theoretic lower bound [4].

The next part of this entry deals with some of the widely used data structures specifically designed for string data, suffix trees, and suffix arrays. These are particularly suitable for querying unstructured text data, such as the genomic sequence of an organism. The following notation is used: Let $s[i]$ denote the ith character of string s, $s[i..j]$ denote the substring $s[i]s[i+1]\ldots s[j]$, and $S_i = s[i]s[i+1]\ldots s[|s|]$ denote the suffix of s starting at ith position. The suffix S_i can be uniquely described by the integer i. In case of multiple strings, the suffix of a string can be described by a tuple consisting of the string number and the starting position of the suffix within the string. Consider a collection of strings over Σ, having total length n, each extended by adding a unique termination symbol $\$ \notin \Sigma$. The suffix tree of the strings is a compacted trie of all suffixes of these extended strings. The suffix array of the strings is the lexicographic sorted order of all suffixes of these extended strings. For convenience, we list '$', the last suffix of each string, just once. The suffix tree and suffix array of strings 'apple' and 'maple' are shown in Fig. 1. Both these data structures take $O(n)$ space and can be constructed in $O(n)$ time [11, 13], both directly and from each other.

Without loss of generality, consider the problem of searching for a pattern P as a substring of a single string T. Assume the suffix tree ST of T is available. If P occurs in T starting from position i, then P is a prefix of suffix $T_i = T[i]T[i+1]\ldots T[|T|]$ in T. It follows that P matches the path from root to leaf labeled i in ST. This property results in the following simple algorithm: Start from the root of ST and follow the path matching characters in P, until P is completely matched or a mismatch occurs. If P is not fully matched, it does not occur in T. Otherwise, each leaf in the subtree below the matching position gives an occurrence of P. The positions can be enumerated by traversing the subtree in $O(occ)$ time, where occ denotes the number of occurrences of P. If only one occurrence is desired, ST can be preprocessed in $O(|T|)$ time such that each internal node contains the suffix at one of the leaves in its subtree.

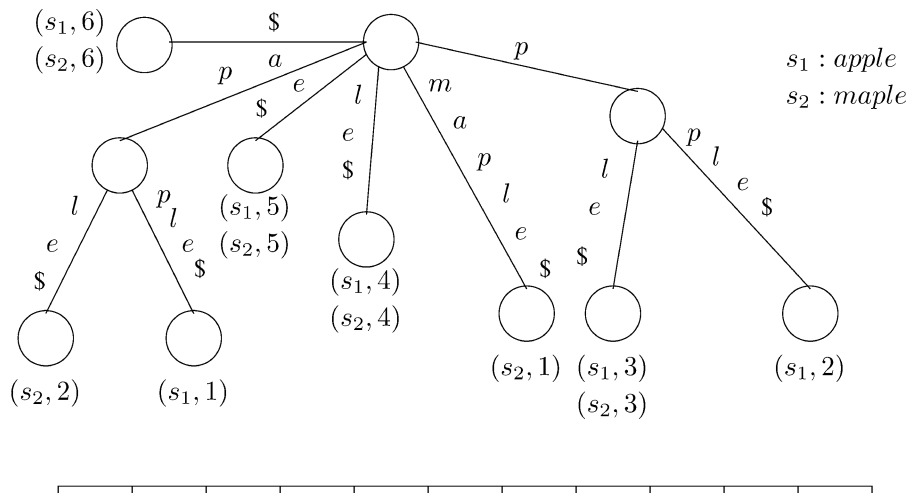
Theorem 3 *Given a suffix tree for text T and a pattern P, whether P occurs in T can be answered in $O(|P|)$ time. All occurrences of P in T can be found in $O(|P| + occ)$ time, where occ denotes the number of occurrences.*

Now consider solving the same problem using the suffix array SA of T. All suffixes prefixed by P appear in consecutive positions in SA. These can be found using binary search in SA. Naively performed, this would take $O(|P| * \log|T|)$ time. It can be improved to $O(|P| + \log|T|)$ time as follows [15]:

Let $SA[L..R]$ denote the range in the suffix array where the binary search is focused. To begin with, $L = 1$ and $R = |T|$. Let $\prec$ denote "lexicographically smaller", $\preceq$ denote "lexicographically smaller or equal", and $lcp(\alpha, \beta)$ denote the length of the longest common prefix between strings α and β. At the beginning of an iteration, $T_{SA[L]} \preceq P \preceq T_{SA[R]}$. Let $M = \lceil (L+R)/2 \rceil$. Let $l = lcp(P, T_{SA[L]})$ and $r = lcp(P, T_{SA[R]})$. Because SA is lexicographically ordered, $lcp(P, T_{SA[M]}) \geq \min(l, r)$. If $l = r$, then compare P and $T_{SA[M]}$ starting from the (l+1)th character. If $l \neq r$, consider the case when $l > r$.

Case I: $l < lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $T_{SA[M]} \prec P$ and $lcp(P, T_{SA[M]}) = lcp(P, T_{SA[L]})$. Continue search in $SA[M..R]$. No character comparisons required.

Case II: $l > lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $P \prec T_{SA[M]}$ and $lcp(P, T_{SA[M]}) = lcp(T_{SA[L]}, T_{SA[M]})$. Continue



Text Indexing, Figure 1
Suffix tree and suffix array of strings *apple* and *maple*

search in $SA[L..M]$. No character comparisons required.

Case III: $l = lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $lcp(P, T_{SA[M]}) \geq l$. Compare P and $T_{SA[M]}$ beyond lth character to determine their relative order and *lcp*.

Similarly, the case when $r > l$ can be handled such that comparisons between P and $T_{SA[M]}$, if at all needed, start from $(r+1)$th character. To start the execution of the algorithm, $lcp(P, T_{SA[1]})$ and $lcp(P, T_{SA[|T|]})$ are computed directly using at most $2|P|$ character comparisons. It remains to be described how the $lcp(T_{SA[L]}, T_{SA[M]})$ and $lcp(T_{SA[R]}, T_{SA[M]})$ values required in each iteration are computed. Let $Lcp[1 \ldots |T|-1]$ be an array such that $Lcp[i] = lcp(SA[i], SA[i+1])$. The *Lcp* array can be computed from *SA* in $O(|T|)$ time [12]. For any $1 \leq i < j \leq n$, $lcp(T_{SA[i]}, T_{SA[j]}) = \min_{k=i}^{j-1} Lcp[k]$. In order to find the *lcp* values required by the algorithm in constant time, note that the binary search can be viewed as traversing a path in the binary tree corresponding to all possible search intervals used by any execution of the binary search algorithm [15]. The root of the tree denotes the interval $[1..n]$. If $[i..j]$ $(j - i \geq 2)$ is the interval at an internal node of the tree, its left child is given by $[i..\lceil (i+j)/2 \rceil]$ and its right child is given by $[\lceil (i+j)/2 \rceil..j]$. The *lcp* value for each interval in the tree is precomputed and recorded in $O(n)$ time and space.

Theorem 4 *Given the suffix array SA of text T and a pattern P, the existence of P in T can be checked in $O(|P| + \log |T|)$ time. All occurrences of P in T can be found in O(occ) additional time, where occ denotes their number.*

Proof The algorithm makes at most $2|P|$ comparisons in determining $lcp(P, T_{SA[1]})$ and $lcp(P, T_{SA[n]})$. A comparison made in an iteration to determine $lcp(P, T_{SA[M]})$ is categorized *successful* if it contributes the *lcp*, and categorized *failed* otherwise. There is at most one failed comparison per iteration. As for successful comparisons, note that the comparisons start with $(\max(l, r) + 1)^{th}$ character of P, and each successful comparison increases the value of $\max(l, r)$ for the next iteration. Thus, each character of P is involved only once in a successful comparison. The total number of character comparisons is at most $3|P| + \log |T| = O(|P| + \log |T|)$. □

Abouelhoda et al. [1] reduce this time further to $O(|P|)$ by mimicking the suffix tree algorithm on a suffix array with some auxiliary information. The strategy is useful in other applications based on top-down traversal of suffix trees. At this stage, the distinction between suffix trees and suffix arrays is blurred as the auxiliary information stored makes the combined data structure equivalent to a suffix tree. Using clever implementation techniques, the space is reduced to approximately $6n$ bytes. A major advantage of the suffix tree and suffix array based methods is that the text T is often large and relatively static, while it is queried with several short patterns. With suffix trees and enhanced suffix arrays [1], once the text is preprocessed in $O(|T|)$ time, each pattern can be queried in $O(|P|)$ time for constant size alphabet. For large alphabets, the query can be answered in $O(|P| * \log |\Sigma|)$ time using $O(n|\Sigma|)$ space

(by storing an ordered array of $|\Sigma|$ pointers to potential children of a node), or in $O(|P| * |\Sigma|)$ time using $O(n)$ space (by storing pointers to first child and next sibling).[1] For indexing in various text-dynamic situations, see [3,7] and references therein. The problem of compressing suffix trees and arrays is covered in more detail in other entries.

While exact pattern matching has many useful applications, the need for approximate pattern matching arises in several contexts ranging from information retrieval to finding evolutionary related biomolecular sequences. The classic approximate pattern matching problem is to find substrings in the text T that have an edit distance of k or less to the pattern P, i. e., the substring can be converted to P with at most k insert/delete/substitute operations. This problem is covered in more detail in other entries. Also see [16], the references therein, and Chapter 36 of [2].

Applications

Text indexing has many practical applications—finding words or phrases in documents under preparation, searching text for information retrieval from digital libraries, searching distributed text resources such as the web, processing XML path strings, searching for longest matching prefixes among IP addresses for internet routing, to name just a few. The reader interested in further exploring text indexing is referred to the book by Crochemore and Rytter [6], and to other entries in this Encyclopedia. The last decade of explosive growth in computational biology is aided by the application of string processing techniques to DNA and protein sequence data. String indexing and aggregate queries to uncover mutual relationships between strings are at the heart of important scientific challenges such as sequencing genomes and inferring evolutionary relationships. For an in depth study of such techniques, the reader is referred to Parts I and II of [10] and Parts II and VIII of [2].

Open Problems

Text indexing is a fertile research area, making it impossible to cover many of the research results or actively pursued open problems in a short amount of space. Providing better algorithms and data structures to answer a flow of string-search queries when caches or other query models are taken into account, is an interesting research issue [4].

[1] Recently, Cole et al. (2006) showed how to further reduce the search time to $O(|P| + \log |\Sigma|)$ while still keeping the optimal $O(|T|)$ space.

Cross References

- ▶ Compressed Suffix Array
- ▶ Compressed Text Indexing
- ▶ Indexed Approximate String Matching
- ▶ Suffix Array Construction
- ▶ Suffix Tree Construction in Hierarchical Memory
- ▶ Suffix Tree Construction in RAM
- ▶ Two-Dimensional Pattern Indexing

Recommended Reading

1. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discret. Algorithms **2**, 53–86 (2004)
2. Aluru, S. (ed.): Handbook of Computational Molecular Biology. Computer and Information Science Series. Chapman and Hall/CRC Press, Boca Raton (2005)
3. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards real-time suffix tree construction. In: Proc. String Processing and Information Retrieval Symposium (SPIRE), 2005, pp. 67–78
4. Ciriani, V., Ferragina, P., Luccio, F., Muthukrishnan, S.: A data structure for a sequence of string acesses in external memory. ACM Trans. Algorithms **3** (2007)
5. Crescenzi, P., Grossi, R., Italiano, G.: Search data structures for skewed strings. In: International Workshop on Experimental and Efficient Algorithms (WEA). Lecture Notes in Computer Science, vol. 2, pp. 81–96. Springer, Berlin (2003)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company, Singapore (2002)
7. Ferragina, P., Grossi, R.: Optimal On-Line Search and Sublinear Time Update in String Matching. SIAM J. Comput. **3**, 713–736 (1998)
8. Franceschini, G., Grossi, R.: A general technique for managing strings in comparison-driven data structures. In: Annual International Colloquium on Automata, Languages and Programming (ICALP), 2004
9. Grossi, R., Italiano, G.: Efficient techniques for maintaining multidimensional keys in linked data structures. In: Annual International Colloquium on Automata, Languages and Programming (ICALP), 1999, pp. 372–381
10. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
11. Karkkainen, J., Sanders, P., Burkhardt, S.: Linear work suffix arrays construction. J. ACM **53**, 918–936 (2006)
12. Kasai, T., Lee, G., Arimura, H. et al.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. 12th Annual Symposium, Combinatorial Pattern Matching (CPM), 2001, pp. 181–192
13. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discret. Algorithms **3**, 143–156 (2005)
14. Ko, P., Aluru, S.: Optimal self-adjustring tree for dynamic string data in secondary storage. In: Proc. String Processing and Information Retrieval Symposium (SPIRE). Lect. Notes Comp. Sci. vol. 4726, pp. 184–194, Santiago, Chile (2007)
15. Manber, U., Myers, G.: Suffix arrays: a new method for on-line search. SIAM J. Comput. **22**, 935–948 (1993)

16. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**, 31–88 (2001)

Thresholds of Random k-SAT

2002; Kaporis, Kirousis, Lalas

ALEXIS KAPORIS, LEFTERIS KIROUSIS
Department of Computer Engineering and Informatics, University of Patras, Patras, Greece

Keywords and Synonyms

Phase transitions; Probabilistic analysis of a Davis–Putnam heuristic

Problem Definition

Consider n Boolean variables $V = \{x_1, \ldots, x_n\}$ and the corresponding set of $2n$ literals $L = \{x_1, \overline{x}_1 \ldots, x_n, \overline{x}_n\}$. A k-clause is a disjunction of k literals of distinct underlying variables. A random formula $\phi_{n,m}$ in k Conjunctive Normal Form (k-CNF) is the conjunction of m clauses, each selected in a uniformly random and independent way amongst the $2^k\binom{n}{k}$ possible k-clauses on n variables in V. The density r_k of a k-CNF formula $\phi_{n,m}$ is the clauses-to-variables ratio m/n.

It was conjectured that for each $k \geq 2$ there exists a critical density r_k^* such that asymptotically almost all (a.a.a.) k-CNF formulas with density $r < r_k^*$ ($r > r_k^*$) are satisfiable (unsatisfiable, respectively). So far, the conjecture has been proved only for $k = 2$ [3,11]. For $k \geq 3$, the conjecture still remains open but is supported by experimental evidence [14] as well as by theoretical, but non-rigorous, work based on Statistical Physics [15]. The value of the putative threshold r_3^* is estimated to be around 4.27. Approximate values of the putative threshold for larger values of k have also been computed.

As far as rigorous results are concerned, Friedgut [10] proved that for each $k \geq 3$ there exists a sequence $r_k^*(n)$ such that for any $\epsilon > 0$, a.a.a. k-CNF formulas $\phi_{n,\lfloor (r_k^*(n)-\epsilon)n\rfloor}$ ($\phi_{n,\lceil (r_k^*(n)+\epsilon)n\rceil}$) are satisfiable (unsatisfiable, respectively). The convergence of the sequence $r_k^*(n)$, $n = 0, 1, \ldots$ for $k \geq 3$ remains open.

Let now

$$r_k^{*-} = \underline{\lim}_{n\to\infty} r_k^*(n) = \sup\{r_k : \Pr[\phi_{n,\lfloor r_k n\rfloor} \text{ is satisfiable } \to 1]\}$$

and

$$r_k^{*+} = \overline{\lim}_{n\to\infty} r_k^*(n) = \inf\{r_k : \Pr[\phi_{n,\lceil r_k n\rceil} \text{is satisfiable} \to 0]\}\,.$$

Obviously, $r_k^{*-} \leq r_k^{*+}$. Bounding from below (from above) r_k^{*-} (r_k^{*+}, respectively) with an as large as possible (as small as possible, respectively) bound has been the subject of intense research work in the past decade.

Upper bounds to r_k^{*+} are computed by counting arguments. To be specific, the standard technique is to compute the expected number of satisfying truth assignments of a random formula with density r_k and find an as small as possible value of r_k for which this expected value approaches zero. Then, by Markov's inequality, it follows that for such a value of r_k, a random formula $\phi_{n,\lceil r_k n\rceil}$ is unsatisfiable asymptotically almost always. This argument has been refined in two directions: First, considering not all satisfying truth assignments but a subclass of them with the property that a satisfiable formula always has a satisfying truth assignment in the subclass considered. The restriction to a judiciously chosen such subclass forces the expected value of the number of satisfying truth assignments to get closer to the probability of satisfiability, and thus leads to a better (smaller) upper bound r_k. However, it is important that the subclass should be such that the expected value of the number of satisfying truth assignments can be computable by the available probabilistic techniques.

Second, make use in the computation of the expected number of satisfying truth assignments of *typical* characteristics of the random formula, i. e. characteristics shared by a.a.a. formulas. Again this often leads to an expected number of satisfying truth assignments that is closer to the probability of satisfiability (non-typical formulas may contribute to the increase of the expected number). Increasingly better upper bounds to r_3^{*+} have been computed using counting arguments as above (see the surveys [6,13]). Dubois, Boufkhad and Mandler [7] proved $r_3^{*+} < 4.506$. The latter remains the best upper bound to date.

On the other hand, for fixed and small values of k (especially for $k = 3$) lower bounds to r_k^{*-} are usually computed by algorithmic methods. To be specific, one designs an algorithm that for an as large as possible r_k it returns a satisfying truth assignment for a.a.a. formulas $\phi_{n,\lfloor r_k n\rfloor}$. Such an r_k is obviously a lower bound to r_k^{*-}. The simpler the algorithm, the easier to perform the probabilistic analysis of returning a satisfying truth assignment for a given r_k, but the smaller the r_k's for which a satisfying truth assignment is returned asymptotically almost always. In this context, backtrack-free DPLL algorithms [4,5] of increasing sophistication were rigorously analyzed (see the surveys [2,9]). At each step of such an algorithm, a literal is set to TRUE and then a *reduced* formula is obtained by (i) deleting clauses where this literal appears and by (ii) deleting the negation of this literal from the clauses it

appears. At steps at which 1-clauses exist (known as forced steps), the selection of the literal to be set to TRUE is made so as a 1-clause becomes satisfied. At the remaining steps (known as free steps), the selection of the literal to be set to TRUE is made according to a heuristic that characterizes the particular DPLL algorithm. A free step is followed by a round of consecutive forced steps. To facilitate the probabilistic analysis of DPLL algorithms, it is assumed that they never backtrack: if the algorithm ever hits a contradiction, i. e. a 0-clause is generated, it stops and reports failure, otherwise it returns a satisfying truth assignment. The previously best lower bound for the satisfiability threshold obtained by such an analysis was $3.26 < r_3^{*-}$ (Achlioptas and Sorkin [1]).

The previously analyzed such algorithms (with the exception of the `Pure Literal` algorithm [8]) at a free step take into account only the clause size where the selected literal appears. Due to this limited information exploited on selecting the literal to be set, the reduced formula in each step remains random conditional only on the current numbers of 3- and 2-clauses and the number of yet unassigned variables. This retention of "strong" randomness permits a successful probabilistic analysis of the algorithm in a not very complicated way. However, for $k = 3$ it succeeds to show satisfiability only for densities up to a number slightly larger than 3.26. In particular, in [1] it is shown that this is the optimal value that can be attained by such algorithms.

Key Results

In [12], a DPLL algorithm is described (and then probabilistically analyzed) such that each free step selects the literal to be set to TRUE taking into account its *degree* (i. e. its number of occurrences) in the current formula.

Algorithm `Greedy` [Section 4.A in 12]

The first variant of the algorithm is very simple: At each free step, a literal with the maximum number of occurrences is selected and set to TRUE. Notice that in this greedy variant, a literal is selected irrespectively of the number of occurrences of its negation. This algorithm successfully returns a satisfying truth assignment for a.a.a. formulas with density up to a number slightly larger than 3.42, establishing that $r_3^{*-} > 3.42$. Its simplicity, contrasted with the improvement over the previously obtained lower bounds, suggests the importance of analyzing heuristics that take into account degree information of the current formula.

Algorithm `CL` [Section 5.A in 12]

In the second variant, at each free step t, the degree of the negation $\overline{\tau}$ of the literal τ that is set to TRUE is also taken into account. Specifically, the literal to be set to TRUE is selected so as upon the completion of the round of forced steps that follow the free step t, the marginal expected increase of the flow from 2-clauses to 1-clauses per unit of expected decrease of the flow from 3-clauses to 2-clauses is minimized. The marginal expectation corresponding to each literal can be computed from the numbers of its positive and negative occurrences. More specifically, if m_i, $i = 2, 3$ equals the expected flow of i-clauses to $(i-1)$-clauses at each step of a round, and τ is the literal set to TRUE at the beginning of the round, then τ is chosen so as to minimize the ratio $|\frac{\triangle m_2}{\triangle m_3}|$ of the differences $\triangle m_2$ and $\triangle m_3$ between the beginning and the end of the round. This has as effect the bounding of the rate of generation of 1-clauses by the smallest possible number throughout the algorithm. For the probabilistic analysis to go through, we need to know for each i, j the number of literals with degree i whose negation has degree j. This heuristic succeeds in returning a satisfying truth assignment for a.a.a. formulas with density up to a number slightly larger than 3.52, establishing that $r_3^{*-} > 3.52$.

Applications

Some applications of SAT solvers include Sequential Circuit Verification, Artificial Intelligence, Automated deduction and Planning, VLSI, CAD, Model-checking and other type of formal verification. Recently, automatic SAT-based model checking techniques were used to effectively find attacks on security protocols.

Open Problems

The main open problem in the area is to formally show the existence of the threshold r_k^* for all (or at least some) $k \geq 3$. To rigorously compute upper and lower bounds better than the ones mentioned here still attracts some interest. Related results and problems arise in the framework of variants of the satisfiability problem and also the problem of colorability.

Cross References

- ▶ Backtracking Based k-SAT Algorithms
- ▶ Local Search Algorithms for kSAT
- ▶ Maximum Two-Satisfiability
- ▶ Tail Bounds for Occupancy Problems

Recommended Reading

1. Achioptas, D., Sorkin, G.B.: Optimal myopic algorithms for random 3-sat. In: 41st Annual Symposium on Foundations of Computer Science, pp. 590–600. IEEE Computer Society, Washington (2000)
2. Achlioptas, D.: Lower bounds for random 3-sat via differential equations. Theor. Comput. Sci. **265**(1–2), 159–185 (2001)
3. Chvátal, V., Reed, B.: Mick gets some (the odds are on his side). In: 33rd Annual Symposium on Foundations of Computer Science, pp. 620–627. IEEE Computer Society, Pittsburgh (1992)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**, 394–397 (1962)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. Assoc. Comput. Mach. **7**(4), 201–215 (1960)
6. Dubois, O.: Upper bounds on the satisfiability threshold. Theor. Comput. Sci. **265**, 187–197 (2001)
7. Dubois, O., Boufkhad, Y., Mandler, J.: Typical random 3-sat formulae and the satisfiability threshold. In: 11th ACM-SIAM symposium on Discrete algorithms, pp. 126–127. Society for Industrial and Applied Mathematics, San Francisco (2000)
8. Franco, J.: Probabilistic analysis of the pure literal heuristic for the satisfiability problem. Annal. Oper. Res. **1**, 273–289 (1984)
9. Franco, J.: Results related to threshold phenomena research in satisfiability: Lower bounds. Theor. Comput. Sci. **265**, 147–157 (2001)
10. Friedgut, E.: Sharp thresholds of graph properties, and the *k*-sat problem. J. AMS **12**, 1017–1054 (1997)
11. Goerdt, A.: A threshold for unsatisfiability. J. Comput. Syst. Sci. **33**, 469–486 (1996)
12. Kaporis, A.C., Kirousis, L.M., Lalas, E.G.: The probabilistic analysis of a greedy satisfiability algorithm. Random Struct. Algorithms **28**(4), 444–480 (2006)
13. Kirousis, L., Stamatiou, Y., Zito, M.: The unsatisfiability threshold conjecture: the techniques behind upper bound improvements. In: A. Percus, G. Istrate, C. Moore (eds.) Computational Complexity and Statistical Physics, Santa Fe Institute Studies in the Sciences of Complexity, pp. 159–178. Oxford University Press, New York (2006)
14. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distribution of sat problems. In: 10th National Conference on Artificial Intelligence, pp. 459–465. AAAI Press, Menlo Park (1992)
15. Monasson, R., Zecchina, R.: Statistical mechanics of the random *k*-sat problem. Phys. Rev. E **56**, 1357–1361 (1997)

Topology Approach in Distributed Computing

1999; Herlihy Shavit

Maurice Herlihy
Department of Computer Science, Brown University, Providence, RI, USA

Keywords and Synonyms

Wait-free renaming

Problem Definition

The application of techniques from Combinatorial and Algebraic Topology has been successful at solving a number of problems in distributed computing. In 1993, three independent teams [3,15,17], using different ways of generalizing the classical graph-theoretical model of distributed computing, were able to solve *set agreement* a long-standing open problem that had eluded the standard approaches. Later on, in 2004, journal articles by Herlihy and Shavit [15] and by Saks and Zaharoglou [17] were to win the prestigious Gödel prize. This paper describes the approach taken by the Herlihy/Shavit paper, which was the first draw the connection between Algebraic and Combinatorial Topology and Distributed Computing.

Pioneering work in this area, such as by Biran, Moran, and Zaks [2] used graph-theoretic notions to model uncertainty, and were able to express certain lower bounds in terms of graph connectivity. This approach, however, had limitations. In particular, it proved difficult to capture the effects of multiple failures or to analyze decision problems other then consensus.

Combinatorial topology generalizes the notion of a graph to the notion of a *simplicial complex*, a structure that has been well-studied in mainstream mathematics for over a century. One property of central interest to topologists is whether a simplicial complex has no "holes" below a certain dimension k, a property known as *k-connectivity*. Lower bounds previously expressed in terms of connectivity of graphs can be generalized by recasting them in terms of k-connectivity of simplicial complexes. By exploiting this insight, it was possible to solve some open problems (k-set agreement, renaming), to pose and solve some new problems ([13]), and to unify a number of disparate results and models [14].

Key Results

A *vertex* $\vec{v}$ is a point in a high-dimensional Euclidean space. Vertexes $\vec{v}_0, \ldots, \vec{v}_n$ are *affinely independent* if $\vec{v}_1 - \vec{v}_0, \ldots, \vec{v}_n - \vec{v}_0$ are linearly independent. An *n-dimensional simplex* (or *n-simplex*) $S^n = (\vec{s}_0, \ldots, \vec{s}_n)$ is the convex hull of a set of $n + 1$ affinely-independent vertexes. For example, a 0-simplex is a vertex, a 1-simplex a line segment, a 2-simplex a solid triangle, and a 3-simplex a solid tetrahedron. Where convenient, superscripts indicate dimensions of simplexes. The $\vec{s}_0, \ldots, \vec{s}_n$ are said to *span* S^n. By convention, a simplex of dimension $d < 0$ is an empty simplex.

A *simplicial complex* (or complex) is a set of simplexes closed under containment and intersection. The *dimension* of a complex is the highest dimension of any of its

simplexes. $\mathcal{L}$ is a *subcomplex* of $\mathcal{K}$ if every simplex of $\mathcal{L}$ is a simplex of $\mathcal{K}$. A map $\mu\colon \mathcal{K} \to \mathcal{L}$ carrying vertexes to vertexes is *simplicial* if it also induces a map of simplexes to simplexes.

Definition 1 A complex $\mathcal{K}$ is *k-connected* if every continuous map of the k-sphere to $\mathcal{K}$ can be extended to a continuous map of the $(k+1)$-disk. By convention, a complex is (-1)-*connected* if and only if it is nonempty, and every complex is k-*connected* for $k < -1$.

A complex is 0-connected if it is connected in the graph-theoretic sense, and a complex is k-connected if it has no holes in dimensions k or less. The definition of k-connectivity may appear difficult to use, but fortunately reasoning about connectivity can be done in a combinatorial way, using the following elementary consequence of the Mayer–Vietoris sequence.

Theorem 2 *If $\mathcal{K}$ and $\mathcal{L}$ are complexes such that $\mathcal{K}$ and $\mathcal{L}$ are k-connected, and $\mathcal{K} \cap \mathcal{L}$ is $(k-1)$-connected, then $\mathcal{K} \cup \mathcal{L}$ is k-connected.*

This theorem, plus the observation that any non-empty simplex is k-connected for all k, allows reasoning about a complex's connectivity inductively in terms of the connectivity of its components.

A set of $n+1$ sequential *processes* communicate either by sending messages to one another or by applying operations to shared objects. At any point, a process may *crash*: it stops and takes no more steps. There is a bound f on the number of processes that can fail. Models differ in their assumptions about timing. At one end of the spectrum is the *synchronous model* in which computation proceeds in a sequence of rounds. In each round, a process sends messages to the other processes, receives the messages sent to it by the other processes in that round, and changes state. (Or it applies operations to shared objects.) All processes take steps at exactly the same rate, and all messages are delivered with exactly the same message delivery time. At the other end is the *asynchronous model* in which there is no bound on the amount of time that can elapse between process steps, and there is no bound on the time it can take for a message to be delivered. Between these extremes is the *semi-synchronous model* in which process step times and message delivery times can vary, but are bounded between constant upper and lower bounds. Proving a lower bound in any of these models requires a deep understanding of the global states that can arise in the course of a protocol's execution, and of how these global states are related.

Each process starts with an *input value* taken from a set V, and then executes a deterministic *protocol* in which it repeatedly receives one or more messages, changes its local state, and sends one or more messages. After a finite number of steps, each process chooses a *decision value* and halts.

In the k-set agreement task [5], processes are required to (1) choose a decision value after a finite number of steps, (2) choose as their decision values some process's input value, and (3) collectively choose no more than k distinct decision values. When $k = 1$, this problem is usually called *consensus* [16].

Here is the connection between topological models and computation. An initial local state of process P is modeled as a vertex $\vec{v} = \langle P, v\rangle$ labeled with P's process id and initial value v. An initial global state is modeled as an n-simplex $S^n = (\langle P_0, v_0\rangle, \ldots, \langle P_n, v_n\rangle)$, where the P_i are distinct. The term $ids(S^n)$ denotes the set of process ids associated with S^n, and $vals(S^n)$ the set of values. The set of all possible initial global states forms a complex, called the *input complex*.

Any protocol has an associated *protocol complex* $\mathcal{P}$, defined as follows. Each vertex is labeled with a process id and a possible local state for that process. A set of vertexes $\langle P_0, v_0\rangle, \ldots, \langle P_d, v_d\rangle$ spans a simplex of $\mathcal{P}$ if and only if there is some protocol execution in which $P_0, \ldots, P_d$ finish the protocol with respective local states $v_0, \ldots, v_d$. Each simplex thus corresponds to an equivalence class of executions that "look the same" to the processes at its vertexes. The term $\mathcal{P}(S^m)$ to denote the subcomplex of $\mathcal{P}$ corresponding to executions in which only the processes in $ids(S^m)$ participate (the rest fail before sending any messages). If $m < n - f$, then there are no such executions, and $\mathcal{P}(S^m)$ is empty. The structure of the protocol complex $\mathcal{P}$ depends both on the protocol and on the timing and failure characteristics of the model. $\mathcal{P}$ often refers to both the protocol and its complex, relying on context to disambiguate.

A protocol *solves* k-set agreement if there is a simplicial map δ, called *decision map*, carrying vertexes of $\mathcal{P}$ to values in V such that if $\vec{p} \in \mathcal{P}(S^n)$ then $\delta(\vec{p}) \in vals(S^n)$, and δ maps the vertexes of any given simplex in $\mathcal{P}(S^n)$ to at most k distinct values.

Applications

The renaming problem is a key tool for understanding the power of various asynchronous models of computation.

Open Problems

Characterizing the full power of the topological approach to proving lower bounds remains an open problem.

Cross References

- ► Asynchronous Consensus Impossibility
- ► Renaming

Recommended Reading

Perhaps the first paper to investigate the solvability of distributed tasks was the landmark 1985 paper of Fischer, Lynch, and Paterson [6] which showed that *consensus*, then considered an abstraction of the database commitment problem, had no 1-resilient message-passing solution. Other tasks that attracted attention include *renaming* [1,12,15] and *set agreement* [3,5,12,10,15,17].

In 1988, Biran, Moran, and Zaks [2] gave a graph-theoretic characterization of decision problems that can be solved in the presence of a single failure in a message-passing system. This result was not substantially improved until 1993, when three independent research teams succeeded in applying combinatorial techniques to protocols that tolerate delays by more than one processor: Borowsky and Gafni [3], Saks and Zaharoglou [17], and Herlihy and Shavit [15].

Later, Herlihy and Rajsbaum used homology theory to derive further impossibility results for set agreement and to unify a variety of known impossibility results in terms of the theory of chain maps and chain complexes [12]. Using the same simplicial model.

Biran, Moran, and Zaks [2] gave the first decidability result for decision tasks, showing that tasks are decidable in the 1-resilient message-passing model. Gafni and Koutsoupias [7] were the first to make the important observation that the contractibility problem can be used to prove that tasks are undecidable, and suggest a strategy to reduce a specific wait-free problem for three processes to a contractibility problem. Herlihy and Rajsbaum [11] provide a more extensive collection of decidability results.

Borowsky and Gafni [3], define an iterated immediate snapshot model that has a recursive structure. Chaudhuri, Herlihy, Lynch, and Tuttle [4] give an inductive construction for the synchronous model, and while the resulting "Bermuda Triangle" is visually appealing and an elegant combination of proof techniques from the literature, there is a fair amount of machinery needed in the formal description of the construction. In this sense, the formal presentation of later constructions is substantially more succinct.

More recent work in this area includes separation results [8] and complexity lower bounds [9].

1. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM **37**(3), 524–548 (1990)
2. Biran, O., Moran, S., Zaks, S.: A combinatorial characterization of the distributed 1-solvable tasks. J. Algorithms **11**(3), 420–440 (1990)
3. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for *t*-resilient asynchronous computations. In: Proceedings of the 25th ACM Symposium on Theory of Computing, May 1993
4. Chaudhuri, S., Herlihy, M., Lynch, N.A., Tuttle, M.R.: Tight bounds for k-set agreement. J. ACM **47**(5), 912–943 (2000)
5. Chaudhuri, S.: More choices allow more faults: Set consensus problems in totally asynchronous systems. Inf. Comp. **105**(1), 132–158 (1993) A preliminary version appeared in ACM PODC 1990
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty processor. J. ACM **32**(2), 374–382 (1985)
7. Gafni, E., Koutsoupias, E.: Three-processor tasks are undecidable. SIAM J. Comput. **28**(3), 970–983 (1999)
8. Gafni, E., Rajsbaum, S., Herlihy, M.: Subconsensus tasks: Renaming is weaker than set agreement. In: Lecture Notes in Computer Science, pp. 329–338. (2006)
9. Guerraoui, R., Herlihy, M., Pochon, B.: A topological treatment of early-deciding set-agreement. In: OPODIS, pp. 20–35, (2006)
10. Herlihy, M., Rajsbaum, S.: Set consensus using arbitrary objects. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, pp. 324–333, August (1994)
11. Herlihy, M., Rajsbaum, S.: The decidability of distributed decision tasks (extended abstract). In: STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pp. 589–598. ACM Press, New York (1997)
12. Herlihy, M., Rajsbaum, S.: Algebraic spans. Math. Struct. Comput. Sci. **10**(4), 549–573 (2000)
13. Herlihy, M., Rajsbaum, S.: A classification of wait-free loop agreement tasks. Theor. Comput. Sci. **291**(1), 55–77 (2003)
14. Herlihy, M., Rajsbaum, S., Tuttle, M.R.: Unifying synchronous and asynchronous message-passing models. In: PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing, pp. 133–142. ACM Press, New York (1998)
15. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM **46**(6), 858–923 (1999)
16. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
17. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. **29**(5), 1449–1483 (2000)

Trade-Offs for Dynamic Graph Problems

2005; Demetrescu, Italiano

Camil Demetrescu, Giuseppe F. Italiano
Department of Computer & Systems Science, University of Rome, Rome, Italy

Keywords and Synonyms

Trading off update time for query time in dynamic graph problems

Problem Definition

A dynamic graph algorithm maintains a given property $\mathcal{P}$ on a graph subject to dynamic changes, such as edge insertions, edge deletions and edge weight updates. A dynamic graph algorithm should process queries on property $\mathcal{P}$ quickly, and perform update operations faster than recomputing from scratch, as carried out by the fastest static algorithm. A typical definition is given below:

Definition 1 (Dynamic graph algorithm) Given a graph and a graph property $\mathcal{P}$, a *dynamic graph algorithm* is a data structure that supports any intermixed sequence of the following operations:

`insert`(u, v):	insert edge (u, v) into the graph.
`delete`(u, v):	delete edge (u, v) from the graph.
`query`$(\ldots)$:	answer a query about property $\mathcal{P}$ of the graph.

A graph algorithm is *fully dynamic* if it can handle both edge insertions and edge deletions and *partially dynamic* if it can handle either edge insertions or edge deletions, but not both: it is *incremental* if it supports insertions only, and *decremental* if it supports deletions only. Some papers study variants of the problem where more than one edge can be deleted of inserted at the same time, or edge weights can be changed. In some cases, an update may be the insertion or deletion of a node along with all edges incident to them. Some other papers only deal with specific classes of graphs, e. g., planar graphs, directed acyclic graphs (DAGs), etc.

There is a vast literature on dynamic graph algorithms. Graph problems for which efficient dynamic solutions are known include graph connectivity, minimum cut, minimum spanning tree, transitive closure, and shortest paths (see, e. g. [3] and the references therein). Many of them update explicitly the property $\mathcal{P}$ after each update in order to answer queries in optimal time. This may be a good choice in scenarios where there are few updates and many queries. In applications where the numbers of updates and queries are comparable, a better approach would be to try to reduce the update time, possibly at the price of increasing the query time. This is typically achieved by relaxing the assumption that the property $\mathcal{P}$ should be maintained explicitly.

This entry focuses on algorithms for dynamic graph problems that maintain the graph property implicitly, and thus require non-constant query time while supporting faster updates. In particular, it considers two problems: *dynamic transitive closure* (also known as *dynamic reachability*) and *dynamic all-pairs shortest paths*, defined below.

Definition 2 (Fully dynamic transitive closure) The *fully dynamic transitive closure problem* consists of maintaining a directed graph under an intermixed sequence of the following operations:

`insert`(u, v):	insert edge (u, v) into the graph.
`delete`(u, v):	delete edge (u, v) from the graph.
`query`(x, y):	return *true* if there is a directed path from vertex x to vertex y, and *false* otherwise.

Definition 3 (Fully dynamic all-pairs shortest paths) The *fully dynamic transitive closure problem* consists of maintaining a weighted directed graph under an intermixed sequence of the following operations:

`insert`(u, v, w):	insert edge (u, v) into the graph with weight w.
`delete`(u, v):	delete edge (u, v) from the graph.
`query`(x, y):	return the distance from x to y in the graph, or $+\infty$ if there is no directed path from x to y.

Recall that the distance from a vertex x to a vertex y is the weight of a minimum-weight path from x to y, where the weight of a path is defined as the sum of edge weights in the path.

Key Results

This section presents a survey of query/update tradeoffs for dynamic transitive closure and dynamic all-pairs shortest paths.

Dynamic Transitive Closure

The first query/update tradeoff for this problem was devised by Henzinger and King [6], who proved the following result:

Theorem 1 (Henzinger and King 1995 [6]) *Given a general directed graph, there is a randomized algorithm with one-sided error for the fully dynamic transitive closure that supports a worst-case query time of $O(n/\log n)$ and an amortized update time of $O(m\sqrt{n}\log^2 n)$.*

The first subquadratic algorithm for this problem is due to Demetrescu and Italiano for the case of directed acyclic graphs [4,5]:

Theorem 2 (Demetrescu and Italiano 2000 [4,5]) *Given a directed acyclic graph with n vertices, there is a randomized algorithm with one-sided error for the fully dynamic*

Trade-Offs for Dynamic Graph Problems, Table 1
Fully dynamic transitive closure algorithms with implicit solution representation

Type of graphs	Type of algorithm	Update time	Query time	Reference
General	Monte Carlo	$O(m\sqrt{n}\log^2 n)$ *amort.*	$O(n/\log n)$	HK [6]
DAG	Monte Carlo	$O(n^{1.575})$	$O(n^{0.575})$	DI [4]
General	Monte Carlo	$O(n^{1.575})$	$O(n^{0.575})$	Sank. [13]
General	Monte Carlo	$O(n^{1.495})$	$O(n^{1.495})$	Sank. [13]
General	Deterministic	$O(m\sqrt{n})$ *amort.*	$O(\sqrt{n})$	RZ [10]
General	Deterministic	$O(m + n\log n)$ *amort.*	$O(n)$	RZ [11]

transitive closure problem that supports each query in $O(n^{\epsilon})$ time and each insertion/deletion in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$, for any $\epsilon \in [0, 1]$, where $\omega(1, \epsilon, 1)$ is the exponent of the multiplication of an $n \times n^{\epsilon}$ matrix by an $n^{\epsilon} \times n$ matrix.

Notice that the dependence of the bounds upon parameter ε leads to a full range of query/update tradeoffs. Balancing the two terms in the update bound of Theorem 2 yields that ε must satisfy the equation $\omega(1, \epsilon, 1) = 1 + 2\epsilon$. The current best bounds on $\omega(1, \epsilon, 1)$ [2,7] imply that $\epsilon < 0.575$. Thus, the smallest update time is $O(n^{1.575})$, which gives a query time of $O(n^{0.575})$:

Corollary 1 (Demetrescu and Italiano 2000 [4,5]) *Given a directed acyclic graph with n vertices, there is a randomized algorithm with one-sided error for the fully dynamic transitive closure problem that supports each query in $O(n^{0.575})$ time and each insertion/deletion in $O(n^{1.575})$ time.*

This result has been generalized to the case of general directed graphs by Sankowski [13]:

Theorem 3 (Sankowsk 2004 [13]) *Given a general directed graph with n vertices, there is a randomized algorithm with one-sided error for the fully dynamic transitive closure problem that supports each query in $O(n)$ time and each insertion/deletion in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$, for any $\epsilon \in [0, 1]$, where $\omega(1, \epsilon, 1)$ is the exponent of the multiplication of an $n \times n^{\epsilon}$ matrix by an $n^{\epsilon} \times n$ matrix.*

Corollary 2 (Sankowski 2004 [13]) *Given a general directed graph with n vertices, there is a randomized algorithm with one-sided error for the fully dynamic transitive closure problem that supports each query in $O(n^{0.575})$ time and each insertion/deletion in $O(n^{1.575})$ time.*

Sankowski has also shown how to achieve an even faster update time of $O(n^{1.495})$ at the expense of a much higher $O(n^{1.495})$ query time:

Theorem 4 (Sankowski 2004 [13]) *Given a general directed graph with n vertices, there is a randomized algorithm with one-sided error for the fully dynamic transitive closure problem that supports each query and each insertion/deletion in $O(n^{1.495})$ time.*

Roditty and Zwick presented algorithms designed to achieve better bounds in the case of sparse graphs:

Theorem 5 (Roditty and Zwick 2002 [10]) *Given a general directed graph with n vertices and m edges, there is a deterministic algorithm for the fully dynamic transitive closure problem that supports each insertion/deletion in $O(m\sqrt{n})$ amortized time and each query in $O(\sqrt{n})$ worst-case time.*

Theorem 6 (Roditty and Zwick 2004 [11]) *Given a general directed graph with n vertices and m edges, there is a deterministic algorithm for the fully dynamic transitive closure problem that supports each insertion/deletion in $O(m + n\log n)$ amortized time and each query in $O(n)$ worst-case time.*

Observe that the results of Theorem 5 and Theorem 6 are subquadratic for $m = o(n^{1.5})$ and $m = o(n^2)$, respectively. Moreover, they are not based on fast matrix multiplication, which is theoretically efficient but impractical.

Dynamic Shortest Paths

The first effective tradeoff algorithm for dynamic shortest paths is due to Roditty and Zwick in the special case of sparse graphs with unit edge weights [12]:

Theorem 7 (Roditty and Zwick 2004 [12]) *Given a general directed graph with n vertices, m edges, and unit edge weights, there is a randomized algorithm with one-sided error for the fully dynamic all-pairs shortest paths problem that supports each distance query in $O(t + \frac{n\log n}{k})$ worst-case time and each insertion/deletion in $O(\frac{mn^2\log n}{t^2} + km + \frac{mn\log n}{k})$ amortized time.*

By choosing $k = (n\log n)^{1/2}$ and $(n\log n)^{1/2} \le t \le n^{3/4}(\log n)^{1/4}$ in Theorem 7, it is possible to obtain an amortized update time of $O(\frac{mn^2\log n}{t^2})$ and a worst-case query

time of $O(t)$. The fastest update time of $O(m\sqrt{n \log n})$ is obtained by choosing $t = n^{3/4}(\log n)^{1/4}$.

Later, Sankowski devised the first subquadratic algorithm for dense graphs based on fast matrix multiplication [14]:

Theorem 8 (Sankowski 2005 [14]) *Given a general directed graph with n vertices and unit edge weights, there is a randomized algorithm with one-sided error for the fully dynamic all-pairs shortest paths problem that supports each distance query in $O(n^{1.288})$ time and each insertion/deletion in $O(n^{1.932})$ time.*

Applications

The transitive closure problem studied in this entry is particularly relevant to the field of databases for supporting transitivity queries on dynamic graphs of relations [16]. The problem also arises in many other areas such as compilers, interactive verification systems, garbage collection, and industrial robotics.

Application scenarios of dynamic shortest paths include network optimization [1], document formatting [8], routing in communication systems, robotics, incremental compilation, traffic information systems [15], and dataflow analysis. A comprehensive review of real-world applications of dynamic shortest path problems appears in [9].

Open Problems

It is a fundamental open problem whether the fully dynamic all pairs shortest paths problem of Definition 3 can be solved in subquadratic time per operation in the case of graphs with real-valued edge weights.

Cross References

- ► All Pairs Shortest Paths in Sparse Graphs
- ► All Pairs Shortest Paths via Matrix Multiplication
- ► Decremental All-Pairs Shortest Paths
- ► Fully Dynamic All Pairs Shortest Paths
- ► Fully Dynamic Transitive Closure
- ► Single-Source Fully Dynamic Reachability

Recommended Reading

1. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows: Theory, Algorithms and Applications. Prentice Hall, Englewood Cliffs (1993)
2. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. Symb. Comput. **9**, 251–280 (1990)
3. Demetrescu, C., Finocchi, I., Italiano, G.: Dynamic Graphs. In: Mehta, D., Sahni, S. (eds.) Handbook on Data Structures and Applications (CRC Press Series, in Computer and Information Science), chap. 36. CRC Press, Boca Raton (2005)
4. Demetrescu, C., Italiano, G.: Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In: Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS'00), Redondo Beach (2000), pp. 381–389
5. Demetrescu, C., Italiano, G.: Trade-offs for fully dynamic reachability on dags: Breaking through the $O(n^2)$ barrier. J. ACM **52**, 147–156 (2005)
6. Henzinger, M., King, V.: Fully dynamic biconnectivity and transitive closure. In: Proc. 36th IEEE Symposium on Foundations of Computer Science (FOCS'95), Milwaukee (1995), pp. 664–672
7. Huang, X., Pan, V.: Fast rectangular matrix multiplication and applications. J. Complex. **14**, 257–299 (1998)
8. Knuth, D., Plass, M.: Breaking paragraphs into lines. Software-practice Exp. **11**, 1119–1184 (1981)
9. Ramalingam, G.: Bounded incremental computation. In: Lecture Notes in Computer Science, vol. 1089. Springer, New York (1996)
10. Roditty, L., Zwick, U.: Improved dynamic reachability algorithms for directed graphs. In: Proceedings of 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Vancouver (2002), pp. 679–688
11. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC), Chicago (2004), pp. 184–191
12. Roditty, L., Zwick, U.: On Dynamic Shortest Paths Problems. In: Proceedings of the 12th Annual European Symposium on Algorithms (ESA), Bergen (2004), pp. 580–591
13. Sankowski, P.: Dynamic transitive closure via dynamic matrix inverse. In: FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04), pp. 509–517. IEEE Computer Society, Washington, DC (2004)
14. Sankowski, P.: Subquadratic algorithm for dynamic shortest distances. In: 11th Annual International Conference on Computing and Combinatorics (COCOON'05), Kunming (2005), pp. 461–470
15. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. In: Proc. 3rd Workshop on Algorithm Engineering (WAE'99), London (1999), pp. 110–123
16. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proc. 9-th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville (1990). pp. 230–242

Traveling Sales Person with Few Inner Points

2004; Deĭneko, Hoffmann, Okamoto, Woeginger

Yoshio Okamoto
Department of Information and Computer Sciences, Toyohashi University of Technology, Toyohashi, Japan

Keywords and Synonyms

Traveling salesman problem; Traveling salesperson problem; Minimum-cost Hamiltonian circuit problem;

Minimum-weight Hamiltonian circuit problem; Minimum-cost Hamiltonian cycle problem; Minimum-weight Hamiltonian cycle problem

Problem Definition

In the *traveling salesman problem* (TSP) n cities $1, 2, \ldots, n$ together with all the pairwise distances $d(i,j)$ between cities i and j are given. The goal is to find the shortest tour that visits every city exactly once and in the end returns to its starting city. The TSP is one of the most famous problems in combinatorial optimization, and it is well-known to be NP-hard. For more information on the TSP, the reader is referred to the book by Lawler, Lenstra, Rinnooy Kan, and Shmoys [14].

A special case of the TSP is the so-called *Euclidean TSP*, where the cities are points in the Euclidean plane, and the distances are simply the Euclidean distances. A special case of the Euclidean TSP is the *convex Euclidean TSP*, where the cities are further restricted so that they lie in convex position. The Euclidean TSP is still NP-hard [4,17], but the convex Euclidean TSP is quite easy to solve: Running along the boundary of the convex hull yields a shortest tour. Motivated by these two facts, the following natural question is posed: What is the influence of the number of inner points on the complexity of the problem? Here, an *inner point* of a finite point set P is a point from P which lies in the interior of the convex hull of P. Intuition says that "Fewer inner points make the problem easier to solve."

The result below answers this question and supports the intuition above by providing simple exact algorithms.

Key Results

Theorem 1 *The special case of the Euclidean TSP with few inner points can be solved in the following time and space complexity. Here, n denotes the total number of cities and k denotes the number of cities in the interior of the convex hull. 1. In time* $O(k!kn)$ *and space* $O(k)$. *2. In time* $O(2^k k^2 n)$ *and space* $O(2^k kn)$ *[1].*

Here, assume that the convex hull of a given point set is already determined, which can be done in time $O(n \log n)$ and space $O(n)$. Further, note that the above space bounds do not count the space needed to store the input but they just count the space in working memory (as usual in theoretical computer science).

Theorem 1 implies that, from the viewpoint of parameterized complexity [2,3,16], these algorithms are fixed-parameter algorithms, when the number k of inner points is taken as a parameter, and hence the problem is fixed-parameter tractable (FPT). (A *fixed-parameter algorithm* has running time $O(f(k)\text{poly}(n))$, where n is the input size, k is a parameter and $f: \mathbb{N} \to \mathbb{N}$ is an arbitrary computable function. For example, an algorithm with running time $O(5^k n)$ is a fixed-parameter algorithm whereas one with $O(n^k)$ is not.) Observe that the second algorithm gives a polynomial-time exact solution to the problem when $k = O(\log n)$.

The method can be extended to some generalized versions of the TSP. For example, Deĭneko et al. [1] stated that the prize-collecting TSP and the partial TSP can be solved in a similar manner.

Applications

The theorem is motivated more from a theoretical side rather than an application side. No real-world application has been assumed.

As for the theoretical application, the viewpoint (introduced in the problem definition section) has been applied to other geometric problems. Some of them are listed below.

The Minimum Weight Triangulation Problem: Given n points in the Euclidean plane, the problem asks to find a triangulation of the points which has minimum total length. The problem is now known to be NP-hard [15].

Hoffmann and Okamoto [10] proved that the problem is fixed-parameter tractable with respect to the number k of inner points. The time complexity they gave is $O(6^k n^5 \log n)$. This is subsequently improved by Grantson, Borgelt, and Levcopoulos [6] to $O(4^k kn^4)$ and by Spillner [18] to $O(2^k kn^3)$. Yet other fixed-parameter algorithms have also been proposed by Grantson, Borgelt, and Levcopoulos [7,8]. The currently best time complexity was given by Knauer and Spillner [13] and it is $O(2^{c\sqrt{k}\log k} k^{3/2} n^3)$ where $c = (2+\sqrt{2})/(\sqrt{3}-\sqrt{2}) < 11$.

The Minimum Convex Partition Problem: Given n points in the Euclidean plane, the problem asks to find a partition of the convex hull of the points into the minimum number of convex regions having some of the points as vertices.

Grantson and Levcopoulos [9] gave an algorithm running in $O(k^{6k-5}2^{16k}n)$ time. Later, Spillner [19] improved the time complexity to $O(2^k k^3 n^3)$.

The Minimum Weight Convex Partition Problem: Given n points in the Euclidean plane, the problem asks to find a convex partition of the points with minimum total length.

Grantson [5] gave an algorithm running in $O(k^{6k-5}2^{16k}n)$ time. Later, Spillner [19] improved the time complexity to $O(2^k k^3 n^3)$.

The Crossing Free Spanning Tree Problem: Given an n-vertex geometric graph (i. e., a graph drawn on the Euclidean plane where every edge is a straight line segment connecting two distinct points), the problem asks to determine whether it has a spanning tree without any crossing of the edges. Jansen and Woeginger [11] proved this problem is NP-hard.

Knauer and Spillner [12] gave algorithms running in $O(175^k k^2 n^3)$ time and $O(2^{33\sqrt{k}\log k}k^2 n^3)$ time.

The method proposed by Knauer and Spillner [12] can be adopted to the TSP as well. According to their result, the currently best time complexity for the TSP is $2^{O(\sqrt{k}\log k)}\text{poly}(n)$.

Open Problems

Currently, no lower bound result for the time complexity seems to be known. For example, is it possible to prove under a reasonable complexity-theoretic assumption the impossibility for the existence of an algorithm running in $2^{O(\sqrt{k})}\text{poly}(n)$ for the TSP?

Cross References

On the traveling salesman problem:
- ▶ Euclidean Traveling Salesperson Problem
- ▶ Hamilton Cycles in Random Intersection Graphs
- ▶ Implementation Challenge for TSP Heuristics
- ▶ Metric TSP

On fixed-parameter algorithms:
- ▶ Closest Substring
- ▶ Parameterized SAT
- ▶ Vertex Cover Kernelization
- ▶ Vertex Cover Search Trees

On others:
- ▶ Minimum Weight Triangulation

Recommended Reading

1. Deĭneko, V.G., Hoffmann, M., Okamoto, Y., Woeginger, G.J.: The traveling salesman problem with few inner points. Oper. Res. Lett. **31**, 106–110 (2006)
2. Downey, R.G., Fellows, M.R.: Parameterized Complexity. In: Monographs in Computer Science. Springer, New York (1999)
3. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science An EATCS Series. Springer, Berlin (2006)
4. Garey, M.R., Graham, R.L., Johnson, D.S.: Some NP-complete geometric problems. In: Proceedings of 8th Annual ACM Symposium on Theory of Computing (STOC '76), pp. 10–22. Association for Computing Machinery, New York (1976)
5. Grantson, M.: Fixed-parameter algorithms and other results for optimal partitions. Lecentiate Thesis, Department of Computer Science, Lund University (2004)
6. Grantson, M., Borgelt, C., Levcopoulos, C.: A fixed parameter algorithm for minimum weight triangulation: Analysis and experiments. Tech. Rep. 154, Department of Computer Science, Lund University (2005)
7. Grantson, M., Borgelt, C., Levcopoulos, C.: Minimum weight triangulation by cutting out triangles. In: Deng, X., Du, D.-Z. (eds.) Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC). Lecture Notes in Computer Science, vol. 3827, pp. 984–994. Springer, New York (2005)
8. Grantson, M., Borgelt, C., Levcopoulos, C.: Fixed parameter algorithms for the minimum weight triangulation problem. Tech. Rep. 158, Department of Computer Science, Lund University (2006)
9. Grantson, M., Levcopoulos, C.: A fixed parameter algorithm for the minimum number convex partition problem. In: Akiyama, J., Kano, M., Tan, X. (eds.) Proceedings of Japanese Conference on Discrete and Computational Geometry (JCDCG 2004). Lecture Notes in Computer Science, vol. 3742, pp. 83–94. Springer, New York (2005)
10. Hoffmann, M., Okamoto, Y.: The minimum weight triangulation problem with few inner points. Comput. Geom. Theory Appl. **34**, 149–158 (2006)
11. Jansen, K., Woeginger, G.J.: The complexity of detecting crossingfree configurations in the plane. BIT **33**, 580–595 (1993)
12. Knauer, C., Spillner, A.: Fixed-parameter algorithms for finding crossing-free spanning trees in geometric graphs. Tech. Rep. 06–07, Department of Computer Science, Friedrich-Schiller-Universität Jena (2006)
13. Knauer, C., Spillner, A.: A fixed-parameter algorithm for the minimum weight triangulation problem based on small graph separators. In: Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG). Lecture Notes in Computer Science, vol. 4271, pp. 49–57. Springer, New York (2006)
14. Lawler, E., Lenstra, J., Rinnooy Kan, A., Shmoys, D. (eds.): The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, Chichester (1985)
15. Mulzer, W., Rote, G.: Minimum Weight Triangulation is NP-hard. In: Proceedings of the 22nd Annual ACM Symposium on Computational Geometry (SoCG), Association for Computing Machinery, New York 2006, pp. 1–10
16. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and Its Applications, vol. 31. Oxford University Press, Oxford (2006)
17. Papadimitriou, C.H.: The Euclidean travelling salesman problem is NP-complete. Theor. Comput. Sci. **4**, 237–244 (1977)
18. Spillner, A.: A faster algorithm for the minimum weight triangulation problem with few inner points. In: Broersma, H., Johnson, H., Szeider, S. (eds.) Proceedings of the 1st ACiD Workshop. Texts in Algorithmics, vol. 4, pp. 135–146. King's College, London (2005)
19. Spillner, A.: Optimal convex partitions of point sets with few inner points. In: Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG), 2005, pp. 34–37

Traveling Salesperson Problem

► Traveling Sales Person with Few Inner Points

Tree Agreement

► Maximum Agreement Subtree (of 2 Binary Trees)

Tree Alignment

► Maximum Agreement Subtree (of 3 or More Trees)

Tree Compression and Indexing

2005; Ferragina, Luccio, Manzini, Muthukrishnan

PAOLO FERRAGINA[1], S. SRINIVASA RAO[2]
[1] Department of Computer Science, University of Pisa, Pisa, Italy
[2] Computational Logic and Algorithms Group, IT University of Copenhagen, Copenhagen, Denmark

Keywords and Synonyms

XML compression and indexing

Problem Definition

Trees are a fundamental structure in computing. They are used in almost every aspect of modeling and representation for explicit computation like searching for keys, maintaining directories, and representations of parsing or execution traces—to name just a few. One of the latest uses of trees is XML, the de facto format for data storage, integration, and exchange over the Internet (see http://www.w3.org/XML/). Explicit storage of trees, with one pointer per child as well as other auxiliary information (e. g. label), is often taken as given but can account for the dominant storage cost. Just to have an idea, a simple tree encoding needs at least 16 bytes per tree node: one pointer to the auxiliary information (e. g. node label) plus three node pointers to the parent, the first child, and the next sibling. This large space occupancy may even prevent the processing of medium size trees, e. g. XML documents. This entry surveys the best known storage solutions for unlabeled and labeled trees that are space efficient and support fast navigational and search operations over the tree structure. In the literature, they are referred to as *succinct/compressed tree indexing* solutions.

Notation and Basic Facts

The information-theoretic storage cost for any item of a universe U can be derived via a simple *counting argument*: at least $\log |U|$ bits are needed to distinguish any two items of U.[1] Now, let $\mathcal{T}$ be a rooted tree of arbitrary degree and shape, and consider the following three main classes of trees:

Ordinal Trees. $\mathcal{T}$ is unlabeled and its children are left-to-right *ordered*. The number of ordinal trees on t nodes is $C_t = \binom{c2t}{t}/(t+1)$ which induces a lower bound of $2t - \Theta(\log t)$ bits.

Cardinal k-ary Trees $\mathcal{T}$ is labeled on its *edges* with symbols drawn from the alphabet $\Sigma = \{1, \ldots, k\}$. Any node has degree at most k because the edges outgoing from each node have *distinct* labels. Typical examples of cardinal trees are the binary tree ($k = 2$), the (uncompacted) tree and the *Patricia tree*. The number of k-ary cardinal trees on t nodes is $C_t^k = \binom{kt+1}{t}/(kt+1)$ which induces a lower bound of $t(\log k + \log e)$ bits, when k is a slowly-growing function of t.

(Multi-)Labeled Trees. $\mathcal{T}$ is an ordinal tree, labeled on its *nodes* with symbols drawn from the alphabet Σ. In the case of multi-labeled trees, every node has *at least* one symbol as its label. The *same* symbols may repeat among sibling nodes, so that the degree of each node is *unbounded*, and the same labeled-subpath may occur many times in $\mathcal{T}$, anchored anywhere. The information-theoretic lower bound on the storage complexity of this class of trees on t nodes comes easily from the decoupling of the tree structure and the storage of tree labels. For labeled trees it is $\log C_t + t \log |\Sigma| = t(\log |\Sigma| + 2) - \Theta(\log t)$ bits.

The following query operations should be supported over $\mathcal{T}$:

Basic Navigational Queries. They ask for the parent of node u, the ith child of u, the degree of u. These operations may be restricted to some label $c \in \Sigma$, if $\mathcal{T}$ is labeled.

Sophisticated Navigational Queries. They ask for the jth level-ancestor of u, the depth of u, the subtree size of u, the lowest common ancestor of a pair of nodes, the ith node according to some node ordering over $\mathcal{T}$, possibly restricted to some label $c \in \Sigma$ (if $\mathcal{T}$ is labeled). For even more operations see [2,11].

[1] Throughout the entry, all logarithms are taken to the base 2, and it is assumed $0 \log 0 = 0$.

Subpath Query. Given a labeled subpath Π, it asks for the (number *occ* of) nodes of $\mathcal{T}$ that immediately descend from Π. Every subpath occurrence may be anchored anywhere in the tree (i. e. not necessarily in its root).

The elementary solution to the tree indexing problem consists of encoding the tree $\mathcal{T}$ via a mixture of pointers and arrays, thus taking a total of $\Theta(t \log t)$ bits. This supports basic navigational queries in constant time, but it is not space efficient and requires the whole visit of the tree to implement the subpath query or the more sophisticated navigational queries. Here the goal is to design tree storage schemes that are either *succinct*, namely "close to the information-theoretic lower bound" mentioned before, or *compressed* in that they achieve "entropy-bounded storage." Furthermore, these storage schemes do *not* require the whole visit of the tree for most navigational operations. Thus, succinct/compressed tree indexing solutions are distinct from simply compressing the input, and then uncompressing it later on at query time.

In this entry, it is assumed that $t \geq |\Sigma|$ and the Random Access Machine (RAM) with word size $\Theta(\lg t)$ is taken as the model of computation. This way, one can perform various arithmetic and bit-wise boolean operations on single words in constant time.

Key Results

The notion of *succinct* data structures was introduced by Jacobson [10] in a seminal work over 18 years ago. He presented a storage scheme for ordinal trees using $2t + o(t)$ bits and supporting basic navigational queries in $O(\log \log t)$ time (i. e. parent, first child and next sibling of a node). Later, Munro and Raman [13] closed the issue for ordinal trees on basic navigational queries and the subtree-size query by achieving constant query-time and $2t + o(t)$ bits of storage. Their storage scheme is called *Balanced Parenthesis* (BP).[2] Subsequently, Benoit et al. [3] proposed a storage scheme called *Depth-First Unary Degree Sequence* (shortly, DFUDS) that still uses $2t + o(t)$ bits but performs more navigational queries like ith child, child rank, and node degree in constant time. Geary et al. [8] gave another representation still taking optimal space that extends DFUDS's operations to the level-ancestor query.

Although these three representations achieve the optimal space occupancy, none of them supports every existing operation in constant time: e. g. BP does not support ith child and child rank, DFUDS and Geary et al.'s representation do not support LCA. Recently, Jansson et al. [11] extended the DFUDS storage scheme in two directions: (1) they showed how to implement in constant time all navigational queries above;[3] (2) they showed how to compress the new tree storage scheme up to $H^*(\mathcal{T})$, which denotes the entropy of the distribution of node degrees in $\mathcal{T}$.

Theorem 1 ([Jansson et al. 2007]) *For any rooted tree $\mathcal{T}$ with t nodes, there exists a tree indexing scheme that uses $tH^*(\mathcal{T}) + O(t(\log \log t)^2/\log t)$ bits and supports all navigational queries in constant time.*

This improves the standard tree pointer-based representation, since it needs no more than $H^*(\mathcal{T})$ bits per node and does not compromise the performance of sophisticated navigational queries. Since it is $H^*(\mathcal{T}) \leq 2$, this solution is also never worse than BP or DFUDS, but its improvement may be significant! This result can be extended to achieve the kth order entropy of the DFUDS sequence, by adopting any compressed-storage scheme for strings (see e. g. [7] and references therein).

Benoit et al. [3] extended the use of DFUDS to cardinal trees, and proposed a tree indexing scheme whose space occupancy is close to the information-theoretic lower bound and supports various navigational queries in constant time. Raman et al. [15] improved the space by using a different approach (based on storing the tree as a set of edges) thus proving the following:

Theorem 2 ([Raman et al. 2002]) *For any k-ary cardinal tree $\mathcal{T}$ with t nodes, there exists a tree indexing scheme that uses $\log C_t^k + o(t) + O(\log \log k)$ bits and supports in constant time the following operations: finding the parent, the degree, the ordinal position among its siblings, the child with label c, the ith child of a node.*

The subtree size operation cannot be supported efficiently using this representation, so [3] should be resorted to in case this operation is a primary concern.

Despite this flurry of activity, the fundamental problem of indexing *labeled* trees succinctly has remained mostly unsolved. In fact, the succinct encoding for ordered trees mentioned above might be replicated $|\Sigma|$ times (one per possible symbol of Σ), and then the divide-and-conquer approach of [8] might be applied to reduce the final space occupancy. However, the final space bound

[2] Some papers [Chiang et al., ACM-SIAM SODA '01; Sadakane, ISAAC '01; Munro et al., J.ALG '01; Munro and Rao, ICALP '04] have extended BP to support in constant time other sophisticated navigational queries like LCA, node degree, rank/select on leaves and number of leaves in a subtree, level-ancestor and level-successor.

[3] The BP representation and the one of Geary et al. [8] have been recently extended to support further operations—like depth/height of a node, next node in the same level, rank/select over various node orders—still in constant time and $2t + o(t)$ bits (see [9] and references therein).

would be $2t + t\log|\Sigma| + O(t|\Sigma|(\log\log\log t)/(\log\log t))$ bits, which is nonetheless far from the information-theoretic storage bound even for moderately large Σ. On the other hand, if subpath queries are of primary concern (e. g. XML), one can use the approach of [12] which consists of a variant of the suffix-tree data structure properly designed to index all $\mathcal{T}$'s labeled paths. Subpath queries can be supported in $O(|\Pi|\log|\Sigma| + occ)$ time, but the required space would be still $\Theta(t\log t)$ bits (with large hidden constants due to the use of suffix trees). Recently, some papers [1,2,5] addressed this problem in its whole generality by either dealing simultaneously with subpath and basic navigational queries [5], or by considering *multi*-labeled trees and a larger set of navigational operations [1,2].

The tree-indexing scheme of [5] is based on a *transform* of the labeled tree $\mathcal{T}$, denoted $\mathsf{xbw}[\mathcal{T}]$, which linearizes it into *two* coordinated arrays $\langle S_{\mathsf{last}}, S_\alpha\rangle$: the former capturing the tree structure and the latter keeping a permutation of the labels of $\mathcal{T}$. $\mathsf{xbw}[\mathcal{T}]$ has the optimal (up to lower-order terms) size of $2t + t\log|\Sigma|$ bits and can be built and inverted in optimal linear time. In designing the XBW-Transform, the authors were inspired by the elegant Burrows–Wheeler transform for strings [4]. The power of $\mathsf{xbw}[\mathcal{T}]$ relies on the fact that it allows one to transform compression and indexing problems on labeled trees into easier problems over strings. Namely, the following two string-search primitives are key tools for indexing $\mathsf{xbw}[\mathcal{T}]$: $\mathsf{rank}_c(S, i)$ returns the number of occurrences of the symbol c in the string prefix $S[1, i]$, and $\mathsf{select}_c(S, j)$ returns the position of the jth occurrence of the symbol c in string S. The literature offers many time/space efficient solutions for these primitives that could be used as a *black-box* for the compressed indexing of $\mathsf{xbw}[\mathcal{T}]$ (see e. g. [2,14] and references therein).

Theorem 3 ([Ferragina et al. 2005]) *Consider a tree $\mathcal{T}$ consisting of t nodes labeled with symbols drawn from alphabet Σ. There exists a compressed tree-indexing scheme that achieves the following performance:*

- *If $|\Sigma| = O(\text{polylog(t)})$, the index takes at most $tH_0(S_\alpha) + 2t + o(t)$ bits, supports basic navigational queries in constant time and (counting) subpath queries in $O(|\Pi|)$ time.*
- *For any alphabet Σ, the index takes less than $tH_k(S_\alpha) + 2t + o(t\log|\Sigma|))$ bits, but label-based navigational queries and (counting) subpath queries are slowed down by a factor $o((\log\log|\Sigma|)3)$.*

Here $H_k(s)$ is the kth order empirical entropy of string s, with $H_k(s) \le H_{k-1}(s)$ for any $k > 0$.

Since $H_k(S_\alpha) \le H_0(S_\alpha) \le \log|\Sigma|$, the indexing of $\mathsf{xbw}[\mathcal{T}]$ takes at most as much space as its plain representation, up to lower order terms, but with the additional feature of being able to navigate and search $\mathcal{T}$ efficiently. This is indeed a sort of *pointerless representation* of the labeled tree $\mathcal{T}$ with additional search functionalities (see [5] for details).

If sophisticated navigational queries over labeled trees are a primary concern, and subpath queries are not necessary, then the approach of Barbay et al. [1,2] should be followed. They proposed the novel concept of *succinct index*, which is different from the concept of *succinct/compressed encoding* implemented by all the above solutions. A succinct index does not *touch* the data to be indexed, it just accesses the data via basic operations offered by the underlying abstract data type (ADT), and requires asymptotically less space than the information-theoretic lower bound on the storage of the data itself. The authors reduce the problem of indexing labeled trees to the one of indexing ordinal trees and strings; and the problem of indexing multi-labeled trees to the one of indexing ordinal trees and binary relations. Then, they provide succinct indexes for strings and binary relations. In order to present their result, the following definitions are needed. Let m be the total number of symbols in $\mathcal{T}$, t_c be the number of nodes labeled c in $\mathcal{T}$, and let ρ_c be the maximum number of labels c in any rooted path of $\mathcal{T}$ (called the *recursivity* of c). Define ρ as the average recursivity of $\mathcal{T}$, namely $\rho = (1/m)\sum_{c\in\Sigma}(t_c\rho_c)$.

Theorem 4 ([Barbay et al. 2007]) *Consider a tree $\mathcal{T}$ consisting of t nodes (multi-)labeled with possibly many symbols drawn from alphabet Σ. Let m be the total number of symbols in $\mathcal{T}$, and assume that the underlying ADT for $\mathcal{T}$ offers basic navigational queries in constant time and retrieves the ith label of a node in time f. There is a succinct index for $\mathcal{T}$ using $m(\log\rho + o(\log(|\Sigma|\rho)))$ bits that supports for a given node u the following operations (where $L = \log\log|\Sigma|\ \log\log\log|\Sigma|$):*

- *Every c-descendant or c-child of u can be retrieved in $O(L(f + \log\log|\Sigma|))$ time.*
- *The set A of c-ancestors of u can be retrieved in $O(L(f + \log\log|\Sigma|) + |A|(\log\log\rho_c + \log\log\log|\Sigma|(f + \log\log|\Sigma|)))$ time.*

Applications

As trees are ubiquitous in many applications, this section concentrates just on two examples that, in their simplicity, highlight the flexibility and power of succinct/compressed tree indexes.

The first example regards suffix trees, which are a crucial algorithmic block of many string processing applications—ranging from bioinformatics to data mining, from data compression to search engines. Standard implementations of suffix trees take at least 80 bits per node. The compressed suffix tree of a string $S[1, s]$ consists of three components: the tree topology, the string depths stored into the internal suffix-tree nodes, and the suffix pointers stored in the suffix-tree leaves (also called *suffix array* of S). The succinct tree representation of [11] can be used to encode the suffix-tree topology and the string depths taking $4s + o(s)$ bits (assuming w.l.o.g. that $|\Sigma| = 2$). The suffix array can be compressed up to the kth order entropy of S via any solution surveyed in [14]. The overall result is never worse than 80 bits per node, but can be significantly better for highly compressible strings.

The second example refers to the XML format which is often modeled as a labeled tree. The succinct/compressed indexes in [1,2,5] are theoretical in flavor but turn out to be relevant for practical XML processing systems. As an example, [6] has published some initial encouraging experimental results that highlight the impact of the XBW-Transform on real XML datasets. The authors show that a proper adaptation of the XBW-Transform allows one to compress XML data up to state-of-the-art XML-conscious compressors, and to provide access to its content, navigate up and down the XML tree structure, and search for simple path expressions and substrings in a few milliseconds over MBs of XML data, by uncompressing only a tiny fraction of them at each operation. Previous solutions took several seconds per operation!

Open Problems

For a complete set of open problems and further directions of research, the interested reader is referred to the recommended readings. Here two main problems, which naturally derive from the discussion above, are commented.

Motivated by XML applications, one may like to extend the subpath search operation to the *efficient* search for all leaves of $\mathcal{T}$ whose labels *contain* a substring β and that *descend from* a given subpath Π. The term "efficient" here means in time proportional to $|\Pi|$ and to the number of retrieved occurrences, but independent as much as possible of $\mathcal{T}$'s size in the worst case. Currently, this search operation is possible only for the leaves which are immediate descendants of Π, and even for this setting, the solution proposed in [6] is not optimal.

There are two main encodings for trees which lead to the results above: ordinal tree representation (BP, DFUDS or the representation of Geary et al. [8]) and XBW. The former is at the base of solutions for sophisticated navigational operations, and the latter is at the base of solutions for sophisticated subpath searches. Is it possible to devise *one unique* transform for the labeled tree $\mathcal{T}$ which combines the best of the two worlds and is still compressible?

Experimental Results

See http://cs.fit.edu/~mmahoney/compression/text.html and at the paper [6] for numerous experiments on XML datasets.

Data Sets

See http://cs.fit.edu/~mmahoney/compression/text.html and the references in [6].

URL to Code

Paper [6] contains a list of software tools for compression and indexing of XML data.

Cross References

- ▶ Compressed Text Indexing
- ▶ Rank and Select Operations on Binary Strings
- ▶ Succinct Encoding of Permutations: Applications to Text Indexing
- ▶ Table Compression
- ▶ Text Indexing

Recommended Reading

1. Barbay, J., Golynski, A., Munro, J.I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. In: Proc. 17th Combinatorial Pattern Matching (CPM). LNCS n. 4009 Springer, Barcelona (2006), pp. 24–35
2. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), New Orleans, USA, (2007), pp. 680–689
3. Benoit, D., Demaine, E., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica **43**, 275–292 (2005)
4. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Report 124, Digital Equipment Corporation (1994)
5. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 184–193. Cambridge, USA (2005)
6. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and searching XML data via two zips. In: Proc. 15th World Wide Web Conference (WWW), pp. 751–760. Edingburg, UK(2006)

7. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. **372**, (1):115–121 (2007)
8. Geary, R., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. In: Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1–10. New Orleans, USA (2004)
9. He, M., Munro, J.I., Rao, S.S.: Succinct ordinal trees based on tree covering. In: Proc. 34th International Colloquium on Algorithms, Language and Programming (ICALP). LNCS n. 4596, pp. 509–520. Springer, Wroclaw, Poland (2007)
10. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 549–554. Triangle Park, USA (1989)
11. Jansson, J., Sadakane, K., Sung, W.: Ultra-succinct representation of ordered trees. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 575–584. New Orleans, USA(2007)
12. Kosaraju, S.R.: Efficient tree pattern matching. In: Proc. 20th IEEE Foundations of Computer Science (FOCS), pp. 178–183. Triangle Park, USA (1989)
13. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
14. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Comput. Surv. **39**(1) (2007)
15. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 233–242. San Francisco, USA (2002)

Treewidth of Graphs

1987; Arnborg, Corneil, Proskurowski

Hans L. Bodlaender
Institute of Information and Computing Sciences
Algorithms and Complexity Group, Center for Algorithmic Systems, University of Utrecht,
Utrecht, The Netherlands

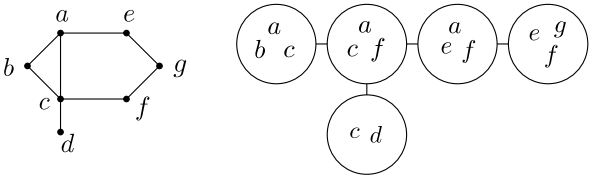
Keywords and Synonyms

Partial k-tree; Dimension; k-decomposable graphs

Problem Definition

The treewidth of graphs is defined in terms of tree decompositions. A *tree decomposition* of a graph $G = (V, E)$ is a pair $(\{X_i | i \in I\}, T = (I, F))$ with $\{X_i | i \in I\}$ a collection of subsets of V, called *bags*, and T a tree, such that

- $\bigcup_{i \in I} X_i = V$.
- For all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$.
- For all $v \in V$, the set $\{i \in I | v \in X_i\}$ induces a connected subtree of T.



Treewidth of Graphs, Figure 1
A graph and a tree decomposition of width 2

The *width* of a tree decomposition is $\max_{i \in I} |X_i| - 1$, and the treewidth of a graph G is the minimum width of a tree decomposition of G.

An alternative definition is in terms of chordal graphs. A graph $G = (V, E)$ is *chordal*, if and only if each cycle of length at least 4 has a chord, i. e., an edge between two vertices that are not successive on the cycle. A graph G has treewidth at most k, if and only if G is a subgraph of a chordal graph H that has maximum clique size at most k.

A third alternative definition is in terms of orderings of the vertices. Let π be a permutation (called *elimination scheme* in this context) of the vertices of $G = (V, E)$. Repeat the following step for $i = 1, \ldots, |V|$: take vertex $\pi(i)$, turn the set of its neighbors into a clique, and then remove v. The *width* of π is the maximum over all vertices of its degree when it was eliminated. The treewidth of G equals the minimum width over all elimination schemes.

In the treewidth problem, the given input is an undirected graph $G = (V, E)$, assumed to be given in its adjacency list representation, and a positive integer $k < |V|$. The problem is to decide if G has treewidth at most k, and if so, to give a tree decomposition of G of width at most k.

Key Results

Theorem 1 (Arnborg et al. [1]) *The problem, given a graph G and an integer k, is to decide if the treewidth of G of at most k is nondeterministic polynomial-time (NP) complete.*

For many applications of treewidth and tree decompositions, the case where k is assumed to be a fixed constant is very relevant. Arnborg et al. [1] gave in 1987 an algorithm that solves this problem in $O(n^{k+2})$ time. A number of faster algorithms for the problem with k fixed have been found; see, e. g., [6] for an overview.

Theorem 2 (Bodlaender [4]) *For each fixed k, there is an algorithm, that given a graph $G = (V, E)$ and an integer k, decides if the treewidth of G is at most k, and if so, that finds a tree decomposition of width at most k in $O(n)$ time.*

This result of Theorem 2 is of theoretical importance only: in a practical setting, the algorithm appears to be much too slow owing to the large constant factor, hidden in the O-notation. For treewidth 1, the problem is equivalent to recognizing trees. Efficient algorithms based on a small set of reduction rules exist for treewidth 2 and 3 [2].

Two often-used heuristics for treewidth are the *minimum fill-in* and *minimum degree* heuristic. In the *minimum degree* heuristic, a vertex v of minimum degree is chosen. The graph G', obtained by making the neighborhood of v a clique and then removing v and its incident edges, is built. Recursively, a chordal supergraph H' of G' is made with the heuristic. Then, a chordal supergraph H of G is obtained, by adding v and its incident edges from G to H'. The *minimum fill-in* heuristic works similarly, but now a vertex is selected such that the number of edges that is added to make the neighborhood of v a clique is as small as possible.

Theorem 3 (Fomin et al. [9]) *There is an algorithm that, given a graph $G = (V, E)$, determines the treewidth of G and finds a tree decomposition of G of minimum width that uses $O(1.8899^n)$ time.*

Bouchitté and Todinca [8] showed that the treewidth can be computed in polynomial time for graphs that have a polynomial number of minimal separators. This implies polynomial-time algorithms for several classes of graphs, e. g., permutation graphs, weakly triangulated graphs.

Applications

One of the main applications of treewidth and tree decomposition is that many problems that are intractable (e. g., NP-hard) on arbitrary graphs become polynomial time or linear time solvable when restricted to graphs of bounded treewidth. The problems where this technique can be applied include many of the classic graph and network problems, like Hamiltonian circuit, Steiner tree, vertex cover, independent set, and graph coloring, but it can also be applied to many other problems. It is also used in the algorithm by Lauritzen and Spiegelhalter [11] to solve the inference problem on probabilistic ("Bayesian", or "belief") networks. Such algorithms typically have the following form. First, a tree decomposition of bounded width is found, and then a dynamic programming algorithm is run that uses this tree decomposition. Often, the running time of this dynamic programming algorithm is exponential in the width of the tree decomposition that is used, and thus one wants to have a tree decomposition whose width is as small as possible.

There are also general characterizations of classes of problems that are solvable in linear time on graphs of bounded treewidth. Most notable is the class of problems that can be formulated in *monadic second order logic* and extensions of these.

Treewidth has been used in the context of several applications or theoretical studies, including graph minor theory, data bases, constraint satisfaction, frequency assignment, compiler optimization, and electrical networks.

Open Problems

There are polynomial-time approximation algorithms for treewidth that guarantee a width of $O(k\sqrt{\log k})$ for graphs of treewidth k, but it is an open question whether there is a polynomial-time approximation algorithm for treewidth with a constant quality ratio. Another long-standing open problem is whether there is a polynomial-time algorithm to compute the treewidth of planar graphs.

Also open is to find an algorithm for the case where the bound on the treewidth k is fixed and whose running time as a function on n is polynomial, and as a function on k improves significantly on the algorithm of Theorem 2.

The base of the exponent of the running time of the algorithm of Theorem 3 can possibly be improved.

Experimental Results

Many algorithms (upper-bound heuristics, lower-bound heuristics, exact algorithms, and preprocessing methods) for treewidth have been proposed and experimentally evaluated. An overview of many of such results is given in [7]. A variant of the algorithm by Arnborg et al. [1] was implemented by Shoikhet and Geiger [15]. Röhrig [14] has experimentally evaluated the linear-time algorithm of Bodlaender [4], and established that it is not practical, even for small values of k. The *minimum degree* and *minimum fill-in* heuristics are frequently used [10].

Data Sets

A collection of test graphs and results for many of the algorithms on these graphs can be found in the TreewidthLIB collection [16].

Cross References

► Branchwidth of Graphs

Recommended Reading

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebr. Discret. Methods **8**, 277–284 (1987)

2. Arnborg, S., Proskurowski, A.: Characterization and recognition of partial 3-trees. SIAM J. Algebr. Discret. Methods **7**, 305–314 (1986)
3. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybernetica **11**, 1–23 (1993)
4. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**, 1305–1317 (1996)
5. Bodlaender, H.L.: A partial *k*-arboretum of graphs with bounded treewidth. Theor. Comp. Sci. **209**, 1–45 (1998)
6. Bodlaender, H.L.: Discovering treewidth. In: P. Vojtáš, M. Bieliková, B. Charron-Bost (eds.) Proceedings 31st Conference on Current Trends in Theory and Practive of Computer Science, SOFSEM 2005. Lecture Notes in Computer Science, vol. 3381, pp. 1–16. Springer, Berlin (2005)
7. Bodlaender, H.L.: Treewidth: Characterizations, applications, and computations. In: Fomin, F.V. (ed.) Proceedings 32nd International Workshop on Graph-Theoretic Concepts in Computer Science WG'06. Lecture Notes in Computer Science, vol. 4271, pp. 1–14. Springer, Berlin (2006)
8. Bouchitté, V., Todinca, I.: Listing all potential maximal cliques of a graph. Theor. Comput. Sci. **276**, 17–32 (2002)
9. Fomin, F.V., Kratsch, D., Todinca, I., Villanger, I.: Exact (exponential) algorithms for treewidth and minimum fill-in (2006). To appear in SIAM Journal of Computing, Preliminary version appeared in ICALP 2004
10. Koster, A.M.C.A., Bodlaender, H.L., van Hoesel, S.P.M.: Treewidth: Computational experiments. In: Broersma, H., Faigle, U., Hurink, J., Pickl, S. (eds.) Electronic Notes in Discrete Mathematics, vol. 8, pp. 54–57. Elsevier, Amsterdam (2001)
11. Lauritzen, S.J., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems. J. Royal Stat. Soc. Ser. B (Methodological) **50**, 157–224 (1988)
12. Reed, B.A.: Tree width and tangles, a new measure of connectivity and some applications, LMS Lecture Note Series, vol. 241, pp. 87–162. Cambridge University Press, Cambridge (1997)
13. Reed, B.A.: Algorithmic aspects of tree width, pp. 85–107. CMS Books Math. Ouvrages Math. SMC, 11. Springer, New York (2003)
14. Röhrig, H.: Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (1998)
15. Shoikhet, K., Geiger, D.: A practical algorithm for finding optimal triangulations. In: Proc. National Conference on Artificial Intelligence (AAAI '97), pp. 185–190. Morgan Kaufmann, San Fransisco (1997)
16. Bodlaender, H.L.: Treewidthlib. http://www.cs.uu.nl/people/hansb/treewidthlib (2004)

Triangle Finding

► Quantum Algorithm for Finding Triangles

Trip Planner

► Shortest Paths Approaches for Timetable Information

Truthful

► Nash Equilibria and Dominant Strategies in Routing

Truthful Auctions

► Truthful Mechanisms for One-Parameter Agents

Truthful Mechanisms for One-Parameter Agents

2001; Archer, Tardos

Moshe Babaioff
Microsoft Research, Silicon Valley,
Mountain View, CA, USA

Keywords and Synonyms

Incentive compatible mechanisms; Dominant strategy mechanisms; Single-parameter agents; Truthful auctions

Problem Definition

This problem is concerned with designing truthful (dominant strategy) mechanisms for problems where each agent's private information is expressed by a single positive real number. The goal of the mechanisms is to allocate loads placed on the agents, and an agent's private information is the cost she incurs per unit load. Archer and Tardos [4] give an exact characterization for the algorithms that can be used to design truthful mechanisms for such load-balancing problems using appropriate side payments. The characterization shows that the allocated load must be monotonic in the cost (decreasing when the cost on an agent increases, fixing the costs of the others). Thus, truthful mechanisms are characterized by a condition on the allocation rule and on payments that ensures voluntary participation can be calculated using the given characterization.

The characterization is used to design polynomial-time truthful mechanisms for several problems in combinatorial optimization to which the celebrated Vickrey-Clarke-Groves (VCG) mechanism does not apply. For scheduling related parallel machines to minimize makespan ($Q\|C_{\max}$), Archer and Tardos [4] presented a 3-approximation mechanism based on randomized rounding of the optimal fractional solution. This mech-

anism is truthful only in expectation (a weaker notion of truthfulness in which truthful bidding maximizes the agent's *expected* utility). Archer [3] improved it to a randomized 2-approximation truthful mechanism. Andelman et al. [2] provided a deterministic truthful mechanism that is 5-approximation. Kovács improved it to 3-approximation in [9], and to 2.8-approximation in [10] (Kovács also gave other results for two special cases). Andelman et al. [2] also presented a deterministic fully polynomial time approximation scheme (FPTAS) for scheduling on a fixed number of machines, as well as a suitable payment scheme that yields a deterministic truthful mechanism. Archer and Tardos [4] presented results for goals other than minimizing the makespan. They presented a truthful mechanism for $Q\|\sum C_j$ (scheduling related machines to minimize the sum of completion times), and showed that for $Q\|\sum w_j C_j$ (minimizing the weighted sum of completion times) $2/\sqrt{3}$ is the best approximation ratio achievable by a truthful mechanism.

This family of problems belongs to the field of algorithmic mechanism design, initiated in the seminal paper of Nisan and Ronen [12]. Nisan and Ronen considered makespan minimization for scheduling on *unrelated* machines and proved upper and lower bounds (note that for unrelated machines agents have more than one parameter). Mu'alem and Schapira [11] presented improved lower bounds. The problem of scheduling on related machines to minimize the makespan has been considered in other papers. Auletta et al. [5] and Ambrosio and Auletta [1] presented truthful mechanisms for several nondeterministic polynomial-time hard restrictions of this problem. Nisan and Ronen [12] also introduced a model in which the mechanism is allowed to observe the machines' actual processing time and compute the payments afterwards (in such a model the machines essentially cannot claim to be faster than they are). Auletta et al. [6] presented additional results for this model. In particular, they showed that it is possible to overcome the lower bound of $2/\sqrt{3}$ for $Q\|\sum w_j C_j$ (minimizing the weighted sum of completion times) and provided a polynomial-time $(1+\epsilon)$-approximation truthful mechanism (with verification) when the number of machines (m) is constant.

The Mechanism Design Framework

Let I be the set of agents. Each agent $i \in I$ has some private *value* (type) consisting of a single parameter $t_i \in \Re$ that describes the agent, and which only i knows. Everything else is public knowledge. Each agent will report a *bid* b_i to the mechanism. Let t denote the vector of true values, and b the vector of bids.

There is some set of *outcomes* O, and given the bids b the mechanism's output algorithm computes an outcome $o(b) \in O$. For any types t, the mechanism aims to choose an outcome $o \in O$ that minimizes some function $g(o, t)$. Yet, given the bids b the mechanism can only choose the outcome as a function of the bids ($o = o(b)$) and has no knowledge of the true types t. To overcome the problem that the mechanism knows only the bids b, the mechanism is designed to be truthful (using payments), that is, in such a mechanism it is a dominant strategy for the agents to reveal their true types ($b = t$). For such mechanisms minimizing $g(o, t)$ is done by assuming that the bids are the true types (and this is justified by the fact that truth-telling is a dominant strategy).

In the framework discussed here we assume that outcome $o(b)$ will assign some amount of *load* or work $w_i(o(b))$ to each agent i, and given $o(b)$ and t_i, agent i incurs some monetary *cost*, $cost_i(t_i, o(b)) = t_i w_i(o(b))$. Thus, agent i's private data t_i measure her cost per unit work.

Each agent i attempts to maximize her *utility* (profit), $u_i(t_i, b) = P_i(b) - cost_i(t_i, o(b))$, where $P_i(b)$ is the *payment* to agent i.

Let b_{-i} denote the vector of bids, not including agent i, and let $b = (b_{-i}, b_i)$. Truth-telling is a *dominant strategy* for agent i if bidding t_i always maximizes her utility, regardless of what the other agents bid. That is, $u_i(t_i, (b_{-i}, t_i)) \geq u_i(t_i, (b_{-i}, b_i))$ for all b_{-i} and b_i.

A mechanism M consists of the pair $M = (o(\cdot), P(\cdot))$, where $o(\cdot)$ is the *output function* and $P(\cdot)$ is the *payment scheme*, i. e., the vector of payment functions $P_i(\cdot)$. An output function *admits a truthful payment scheme* if there exist payments $P(\cdot)$ such that for the mechanism $M = (o(\cdot), P(\cdot))$, truth-telling is a dominant strategy for each agent. A mechanism that admits a truthful payment scheme is *truthful*.

Mechanism M satisfies the *voluntary participation* condition if agents who bid truthfully never incur a net loss, i. e., $u_i(t_i, (b_{-i}, t_i)) \geq 0$ for all agents i, true values t_i, and other agents' bids b_{-i}.

Definition 1 With the other agents' bids b_{-i} fixed, the *work curve* for agent i is $w_i(b_{-i}, b_i)$, considered as a single-variable function of b_i. The output function o is *decreasing* if each of the associated work curves is decreasing (i. e., $w_i(b_{-i}, b_i)$ is a decreasing function of b_i, for all i and b_{-i}).

Scheduling on Related Machines

There are n jobs and m machines. The jobs represent amounts of work $p_1 \geq p_2 \geq \cdots \geq p_n$, and let p denote the

set of jobs. Machine i runs at some speed s_i, so it must spend p_j/s_i units of time processing each job j assigned to it. The input to an algorithm is b, the (reported) speed of the machines, and the output is $o(b)$, an assignment of jobs to machines. The load on machine i for outcome $o(b)$ is $w_i(b) = \sum p_j$, where the sum runs over jobs j assigned to i. Each machine incurs a cost proportional to the time it spends processing its jobs. The cost of machine i is $cost_i(t_i, o(b)) = t_i w_i(o(b))$, where $t_i = 1/s_i$ and $w_i(b)$ is the total load assigned to i when the speeds are b. Let C_j denote the completion time of job j. One can consider the following goals for scheduling related parallel machines:

- Minimizing the makespan ($Q\|C_{\max}$), the mechanism's goal is to minimize the completion time of the last job on the last machine, i. e., $g(o, t) = C_{\max} = \max_i t_i \cdot w_i(b)$.
- Minimize the sum of completion times ($Q\|\sum C_j$), i. e. $g(o, t) = Q\|\sum C_j = \sum_j C_j$.
- Minimize the weighted sum of completion times ($Q\|\sum w_j C_j$), i. e., $g(o, t) = Q\|\sum w_j C_j = \sum_j w_j C_j$, where w_j is the weight of job j.

An algorithm is a *c-approximation algorithm* with respect to g, if for every instance (p, t) it outputs an outcome of cost at most $c \cdot g(o(t), t)$. A *c-approximation mechanism* is one whose output algorithm is a c-approximation. Note that if the mechanism is truthful the approximation is with respect to the true speeds. A polynomial-time approximation scheme (PTAS) is a family of algorithms such that for every $\epsilon > 0$ there exists a $(1 + \epsilon)$-approximation algorithm. If the running time is also polynomial in $1/\epsilon$, the family of algorithms is a FPTAS.

Key Results

The following two theorems hold for the mechanism design framework as defined in Sect. Problem Definition.

Theorem 1 ([4]) *The output function o(b) admits a truthful payment scheme if and only if it is decreasing. In this case, the mechanism is truthful if and only if the payments $P_i(b_{-i}, b_i)$ are of the form*

$$h_i(b_{-i}) + b_i w_i(b_{-i}, b_i) - \int_0^{b_i} w_i(b_{-i}, u)\,\mathrm{d}u\,,$$

where the h_i are arbitrary functions.

Theorem 2 ([4]) *A decreasing output function admits a truthful payment scheme satisfying voluntary participation if and only if $\int_0^\infty w_i(b_{-i}, u)du < \infty$ for all i, b_{-i}. In this case, the payments can be defined by*

$$P_i(b_{-i}, b_i) = b_i w_i(b_{-i}, b_i) + \int_{b_i}^{\infty} w_i(b_{-i}, u)\,\mathrm{d}u\,.$$

Theorem 3 ([4]) *There is a truthful mechanism (not polynomial time) that outputs an optimal solution for $Q\|C_{max}$ and satisfies voluntary participation.*

Theorem 4 *For the problem of minimizing the makespan ($Q\|C_{max}$):*

- *There is a polynomial-time randomized algorithm that deterministically yields a 2-approximation, and admits a truthful payment scheme that creates a mechanism that is truthful in expectation and satisfies voluntary participation [3] .*
- *There is a polynomial-time deterministic 2.8-approximation algorithm that admits a truthful payment scheme that creates a mechanism that satisfies voluntary participation [10].*
- *There is a deterministic FPTAS for scheduling on a fixed number of machines that admits a truthful payment scheme that creates a mechanism that satisfies voluntary participation [2].*

Theorem 5 ([4]) *There is a truthful polynomial-time mechanism that outputs an optimal solution for $Q\|\sum C_j$ and satisfies voluntary participation.*

Theorem 6 ([4]) *No truthful mechanism for $Q\|\sum w_j C_j$ can achieve an approximation ratio better than $2/\sqrt{3}$, even on instances with just two jobs and two machines.*

Applications

Archer and Tardos [4] applied the characterization of truthful mechanisms to problems other than scheduling. They presented results for the uncapacitated facility location problem as well as the maximum-flow problem.

Kis and Kapolnai [8] considered the problem of scheduling of groups of identical jobs on related machines with sequence-independent setup times ($Q|u_j, p_{jk} = p_j\|C_{\max}$). They provided a truthful, polynomial-time, randomized mechanism for the batch-scheduling problem with a deterministic approximation guarantee of 4 to the minimal makespan, based on the characterization of truthful mechanisms presented above.

Open Problems

Considering scheduling on related machines to minimize the makespan, Hochbaum and Shmoys [7] presented a PTAS for this problem, but it is not monotonic. Is there

a truthful PTAS for this problem when the number of machines is not fixed? It is still an open problem whether such a mechanism exists or not. Finding such a mechanism would be an interesting result. Proving a lower bound that shows that such a mechanism does not exist would be even more interesting as it will show that there is a "cost of truthfulness" for this computational problem. A gap between the best approximation algorithm and the best monotonic algorithm (which creates a truthful mechanism), if it exists for this problem, would be a major step in improving our understanding of the combined effect of computational and incentive constraints.

Cross References

- ► Algorithmic Mechanism Design
- ► Competitive Auction
- ► Generalized Vickrey Auction
- ► Incentive Compatible Selection

Recommended Reading

1. Ambrosio, P., Auletta, V.: Deterministic monotone algorithms for scheduling on related machines. In: 2nd Ws. on Approx. and Online Alg. (WAOA), 2004, pp. 267–280
2. Andelman, N., Azar, Y., Sorani, M.: Truthful approximation mechanisms for scheduling selfish related machines. In: 22nd Ann. Symp. on Theor. Aspects of Comp. Sci. (STACS), 2005, pp. 69–82
3. Archer, A.: Mechanisms for Discrete Optimization with Rational Agents. Ph. D. thesis, Cornell University (2004)
4. Archer, A., Tardos, É.: Truthful mechanisms for one-parameter agents. In: 42nd Annual Symposium on Foundations of Computer Science (FOCS), 2001, pp. 482–491
5. Auletta, V., De Prisco, R., Penna, P., Persiano, G.: Deterministic truthful approximation mechanisms for scheduling related machines. In: 21st Ann. Symp. on Theor. Aspects of Comp. Sci. (STACS), 2004, pp. 608–619
6. Auletta, V., De Prisco, R., Penna, P., Persiano, G., Ventre, C.: New constructions of mechanisms with verification. In: 33rd International Colloquium on Automata, Languages and Programming (ICALP) (1), 2006, pp. 596–607
7. Hochbaum, D., Shmoys, D.: A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. SIAM J. Comput. **17**(3), 539–551 (1988)
8. Kis, T., Kapolnai, R.: Approximations and auctions for scheduling batches on related machines. Operat. Res. Let. **35**(1), 61–68 (2006)
9. Kovács, A.: Fast monotone 3-approximation algorithm for scheduling related machines. In: 13th Annual European Symposium (ESA), 2005, pp. 616–627
10. Kovács, A.: Fast Algorithms for Two Scheduling Problems. Ph. D. thesis, Universität des Saarlandes (2007)
11. Mu'alem, A., Schapira, M.: Setting lower bounds on truthfulness. In: SODA, 2007
12. Nisan, N., Ronen, A.: Algorithmic mechanism design. Game. Econ. Behav. **35**, 166–196 (2001)

Truthful Multicast

2004; Wang, Li, Wang

Weizhao Wang[1], Xiang-Yang Li[2], Yu Wang[3]

[1] Google Inc., Irvine, CA, USA

[2] Department of Computer Science, Illinois Institute of Tech., Chicago, IL, USA

[3] Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC, USA

Keywords and Synonyms

Truthful multicast routing; Strategyproof multicast mechanism

Problem Definition

Several mechanisms [1,3,5,9], which essentially all belong to the VCG mechanism family, have been proposed in the literature to prevent the selfish behavior of unicast routing in a wireless network. In these mechanisms, the least cost path, which maximizes the social efficiency, is used for routing. Wang, Li, and Wang [8] studied the *truthful* multicast routing protocol for a selfish wireless network, in which selfish wireless terminals will follow their own interests. The multicast routing protocol is composed of two components: (1) the tree structure that connects the sources and receivers, and (2) the payment to the relay nodes in this tree. Multicast poses a unique challenge in designing strategyproof mechanisms due to the reason that (1) a VCG mechanism uses an output that maximizes the *social efficiency*; (2) it is NP-hard to find the tree structure with the minimum cost, which in turn maximizes the social efficiency. A range of multicast structures, such as the least cost path tree (LCPT), the pruning minimum spanning tree (PMST), virtual minimum spanning tree (VMST), and Steiner tree, were proposed to replace the optimal multicast tree. In [8], Wang et al. showed how payment schemes can be designed for existing multicast tree structures so that rational selfish wireless terminals will follow the protocols for their own interests.

Consider a communication network $G = (V, E, c)$, where $V = \{v_1, \cdots, v_n\}$ is the set of communication terminals, $E = \{e_1, e_2, \cdots, e_m\}$ is the set of links, and c is the cost vector of all agents. Here agents are terminals in a node weighted network and are links in a link weighted network. Given a set of sources and receivers $Q = \{q_0, q_1, q_2, \cdots, q_{r-1}\} \subset V$, the multicast problem is to find a tree $T \subset G$ spanning all terminals Q. For simplic-

ity, assume that $s = q_0$ is the sender of a multicast session if it exists. All terminals or links are required to declare a cost of relaying the message. Let d be the declared costs of all nodes, i. e., agent i declared a cost d_i. On the basis of the declared cost profile d, a multicast tree needs to be constructed and the payment $p_k(d)$ for each agent k needs to be decided. The utility of an agent is its payment received, minus its cost if it is selected in the multicast tree. Instead of reinventing the wheels, Wang et al. still used the previously proposed structures for multicast as the output of their mechanism. Given a multicast tree, they studied the design of strategyproof payment schemes based on this tree.

Notations

Given a network H, $\omega(H)$ denotes the total cost of all agents in this network. If the cost of any agent i (link e_i or node v_i) is changed to c'_i, the new network is denoted as $G' = (V, E, c|^i c'_i)$, or simply $c|^i c'_i$. If one agent i is removed from the network, it is denoted as $c|^i \infty$. For the simplicity of notation, the cost vector c is used to denote the network $G = (V, E, c)$ if no confusion is caused. For a given source s and a given destination q_i, $\mathsf{LCP}(s, q_i, c)$ represents the shortest path between s and q_i when the cost of the network is represented by vector c. $|\mathsf{LCP}(s, q_i, d)|$ denotes the total cost of the least cost path $\mathsf{LCP}(s, q_i, d)$. The notation of several multicast trees is summarized as follows.

1. Link Weighted Multicast Tree
   - **LCPT**: The union of all least cost paths from the source to receivers is called the *least cost path tree*, denoted by $LCPT(d)$.
   - **PMST**: First construct the minimum spanning tree $MST(G)$ on the graph G. Take the tree $MST(G)$ rooted at sender s, prune all subtrees that do not contain a receiver. The final structure is called the Pruning Minimum Spanning Tree (PMST).
   - **LST**: The Link Weighted Steiner Tree (LST) can be constructed by the algorithm proposed by Takahashi and Matsuyama [6].
2. Node Weighted Multicast Tree
   - **VMST**: First construct a virtual graph using all receivers plus the sources as the vertices and the cost of LCP as the link weight. Then compute the minimum spanning tree on the virtual graph, which is called virtual minimum spanning tree (VMST). Finally, choose all terminals on the VMST as the relay terminals.
   - **NST**: The node weighted Steiner tree (NST) can be constructed by the algorithm proposed by [4].

Key Results

If the LCPT tree is used as the multicast tree, Wang et al. proved the following theorem.

Theorem 1 *The VCG mechanism combined with LCPT is not truthful.*

Because of the failure of the VCG mechanism, they designed their non-VGC mechanism for the LCPT-based multicast routing as follows.

1: For each receiver $q_i \neq s$, computes the least cost path from the source s to q_i, and compute a payment $p^i_k(d)$ to every link e_k on the $\mathsf{LCP}(s, q_i, d)$ using the scheme for unicast

$$p^i_k(d) = d_k + |\mathsf{LCP}(s, q_i, d|^k \infty)| - |\mathsf{LCP}(s, q_i, d)|.$$

2: The final payment to link $e_k \in LCPT$ is then

$$p_k(d) = \max_{q_i \in Q} p^i_k(d). \quad (1)$$

The payment to each link not on LCPT is simply 0.

Truthful Multicast, Algorithm 1
Non-VCG mechanism for LCPT

Theorem 2 *Payment (defined in Eq. (1)) based on LCPT is truthful and it is minimum among all truthful payments based on LCPT.*

More generally, Wang et al. [8] proved the following theorem.

Theorem 3 *The VCG mechanism combined with either one of the LCPT, PMST, LST, VMST, NST is not truthful.*

1: Apply VCG mechanism on the MST. The payment for edge $e_k \in PMST(d)$ is

$$p_k(d) = \omega(MST(d|^k \infty)) - \omega(MST(d)) + d_k. \quad (2)$$

2: For every edge $e_k \notin PMST(d)$, its payment is 0.

Truthful Multicast, Algorithm 1
Non-VCG mechanism for PMST

Because of this negative result, they designed their non-VCG mechanisms for all multicast structures they studied: LCPT, PMST, LST, VMST, NST. For example, Algorithm 2 is the algorithm for PMST. For other algorithms, please refer to [8].

Regarding all their non-VGC mechanisms, they proved the following theorem.

Theorem 4 *The non-VCG mechanisms designed for the multicast structures LCPT, PMST, LST, VMST, NST are not only truthful, but also achieve the minimum payment among all truthful mechanisms.*

Applications

In wireless ad hoc networks, it is commonly assumed that, each terminal contributes its local resources to forward the data for other terminals to serve the common good, and benefits from resources contributed by other terminals to route its packets in return. On the basis of such a fundamental design philosophy, wireless ad hoc networks provide appealing features such as enhanced system robustness, high service availability and scalability. However, the critical observation that individual users who own these wireless devices are generally selfish and non-cooperative may severely undermine the expected performances of the wireless networks. Therefore, providing incentives to wireless terminals is a must to encourage contribution and thus maintains the robustness and availability of wireless networking systems. On the other hand, to support a communication among a group of users, multicast is more efficient than unicast or broadcast, as it can transmit packets to destinations using fewer network resources, thus increasing the social efficiency. Thus, most results of the work of Wang et al. can apply to multicast routing in wireless networks in which nodes are selfish. It not only guarantees that multicast routing behaves normally but also achieves good social efficiency for both the receivers and relay terminals.

Open Problems

There are several unsolved challenges left as future work in [8]. Some of these challenges are listed below.

- How to design algorithms that can compute these payments in asymptotically optimum time complexities is presently unknown.
- Wang et al. [8] only studied the tree-based structures for multicast. Practically, mesh-based structures may be more needed for wireless networks to improve the fault tolerance of the multicast. It is unknown whether a strategyproof multicast mechanism can be designed for some mesh-based structures used for multicast.
- All of the tree construction and payment calculations in [8] are performed in a centralized way, it would be interesting to design some distributed algorithms for them.
- In the work by Wang et al. [8] it was assumed that the receivers will always relay the data packets for other receivers for free, the source node of the multicast will pay the relay nodes to compensate their cost, and the source node will not charge the receivers for getting the data. As a possible future work, the budget balance of the source node needs to be considered if the receivers have to pay the source node for getting the data.
- Fairness of payment sharing needs to be considered in a case where the receivers share the total payments to all relay nodes on the multicast structure. Notice that this is different from the cost-sharing studied in [2], in which they assumed a fixed multicast tree, and the link cost is publicly known; in that work they showed how to share the total link cost among receivers.
- Another important task is to study how to implement the protocols proposed in [8] in a distributed manner. Notice that, in [3,9], distributed methods have been developed for a truthful unicast using some cryptography primitives.

Cross References

► Algorithmic Mechanism Design

Recommended Reading

1. Anderegg, L., Eidenbenz, S.: Ad hoc-VCG: a truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In: Proceedings of the 9th annual international conference on Mobile computing and networking. pp. 245–259 ACM Press, New York (2003)
2. Feigenbaum, J., Papadimitriou, C., Shenker, S.: Sharing the cost of multicast transmissions. J. Comput. Syst. Sci. **63**(1), 21–41 (2001)
3. Feigenbaum, J., Papadimitriou, C., Sami, R., Shenker, S.: A BGP-based mechanism for lowest-cost routing. In: Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing, pp. 173–182. Monterey, 21–24 July 2002
4. Klein, P., Ravi, R.: A nearly best-possible approximation algorithm for node-weighted Steiner trees. J. Algorithms **19**(1), 104–115 (1995)
5. Nisan, N., Ronen, A.: Algorithmic mechanism design. In: Proc. 31st Annual Symposium on Theory of Computing (STOC99), Atlanta, 1–4 May 1999, pp. 129–140 (1999)
6. Takahashi, H., Matsuyama, A.: An approximate solution for the Steiner problem in graphs. Math. Jap. **24**(6), 573–577 (1980)
7. Wang, W., Li, X.-Y.: Low-Cost routing in selfish and rational wireless ad hoc networks. IEEE Trans. Mobile Comput. **5**(5), 596–607 (2006)
8. Wang, W., Li, X.-Y., Wang, Y.: Truthful multicast in selfish wireless networks. In: Proceedings of the 10th ACM Annual international Conference on Mobile Computing and Networking, Philadelphia, 26 September – 1 October 2004
9. Zhong, S., Li, L., Liu, Y., Yang, Y.R.: On designing incentive-compatible routing and forwarding protocols in wireless ad-hoc networks –an integrated approach using game theoretical and cryptographic techniques. In: Proceedings of the 11th ACM Annual international Conference on Mobile Computing and Networking, Cologne, 28 August – 2 September 2005

Truthful Multicast Routing

► Truthful Multicast

t-Spanners

► Geometric Spanners

TSP-Based Curve Reconstruction

2001; Althaus, Mehlhorn

Edgar Ramos
School of Mathematics, National University of Colombia, Medellín, Colombia

Problem Definition

An instance of the curve reconstruction problem is a finite set of *sample* points V in the plane, which are assumed to be taken from an unknown planar curve γ. The task is to construct a geometric graph G on V such that two points in V are connected by an edge in G if and only if the points are adjacent on γ. The curve γ may consist of one or more connected components, and each of them may be closed or open (with endpoints), and may be smooth everywhere (tangent defined at every point) or not.

Many heuristic approaches have been proposed to solve this problem. This work continues a line of reconstruction algorithms with *guaranteed* performance, i. e. algorithms which probably solve the reconstruction problem under certain assumptions of γ and V. Previous proposed solutions with guaranteed performances were mostly *local*: a subgraph of the complete geometric graph defined by the points is considered (in most cases the Delaunay edges), and then *filtered* using a local criteria into a subgraph that will constitute the reconstruction. Thus, most of these algorithms fail to enforce that the solution have the global property of being a path/tour or collection of paths/tours and so usually require a dense sampling to work properly and have difficulty handling nonsmooth curves. See [6,7,8] for surveys of these algorithms.

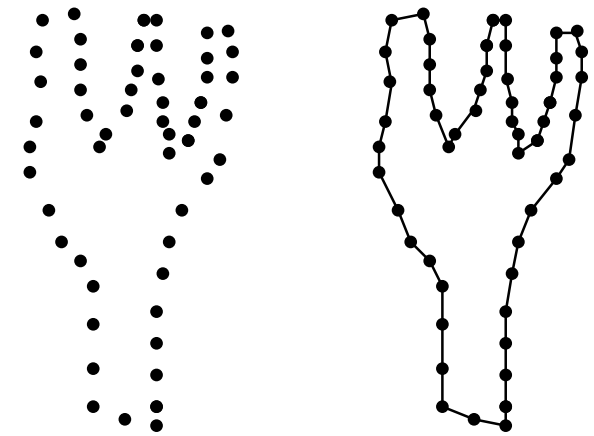
This work concentrates on a solution approach based on the *traveling salesman problem (TSP)*. Recall that a *traveling salesman path (tour)* for a set V of points is a path (cycle) passing through all points in V. An optimal traveling salesman path (tour) is a traveling salesman path (tour) of shortest length. The first question is under which conditions for γ and V a traveling salesman path (tour) is a correct reconstruction. Since the construction of an optimal traveling salesman path (tour) is an NP-hard problem, a second question is whether for the specific instances under consideration, an efficient algorithm is possible.

A previous work of Giesen [9] gave a first weak answer to the first question: For every benign semiregular closed curve γ, there exists an $\epsilon > 0$ with the following property: If V is a finite sample set from γ so that for every $x \in \gamma$ there is a $p \in V$ with $\|pv\| \leq \epsilon$, then the optimal traveling salesman tour is a polygonal reconstruction of γ. For a curve $\gamma : [0,1] \to R^2$, its left and right tangents at $\gamma(t_0)$, are defined as the limits of the ratio $|\gamma(t_2) - \gamma(t_1)| / |t_2 - t_1|$ as (t_1, t_2) converges to (t_0, t_0) from the right $(t_0 < t_1 < t_2)$ and from the right $(t_1 < t_2 < t_0)$ respectively. A curve is semiregular if both tangents exist at every points and regular if the tangents exist and coincide at every point. The *turning* angle of γ at p is the angle between the left and right tangents at a points p. A semiregular curve is *benign* if the turning angle is less than π.

To investigate the TSP-based solution of the reconstruction problem, this work considers its integer linear programming *(ILP)* formulation and the corresponding linear programming *(LP)* relaxation. The motivation is that a successful method for solving the TSP is to use a branch-and-cut algorithm based on the LP-relaxation. See Chapter 7 in [5]. For a path with endpoints a and b, the formulation is based on variables $x_{u,v} \in \{0,1\}$ for each pair u, v in V (indicating whether the edge uv is in the path $(x_{uv} = 1)$ or not $(x_{uv} = 0)$ and consists of the following objective function and constraints ($x_{uu} = 0$ for all $u \in V$):

$$
\begin{aligned}
\text{minimize} \quad & \sum_{u,v \in V} \|uv\| \cdot x_{uv} & \\
\text{subject to} \quad & \sum_{v \in V} x_{uv} = 2 & \text{for all } u \in V \setminus \{a,b\} \\
& \sum_{v \in V} x_{uv} = 1 & \text{for } u \in \{a,b\} \\
& \sum_{u,v \in V'} x_{uv} \leq |V'| - 1 & \text{for } V' \subseteq V, V' \neq \emptyset \\
& x_{uv} \in \{0,1\} & \text{for all } u, v \in V.
\end{aligned}
$$

Here $\|uv\|$ denotes the Euclidean distance between u and v and so the objective function is the total length of the selected edges. This is called the *subtour-ILP for the TSP with specified endpoints*. The equality constraints are called the *degree constraints*, the inequality ones are called *subtour elimination constraints* and the last ones are called the *integrality constraints*. If the degree and integrality constraints hold, the corresponding graph could include disconnected cycles (subtours), hence the need for the subtour elimination constraints. The relaxed LP is obtained by replacing



TSP-Based Curve Reconstruction, Figure 1
Sample data and its reconstruction

the integrality constraints by the constraints $0 \le x_{uv} \le 1$ and is called the *subtour-LP for the TSP with specified endpoints*. There is a polynomial time algorithm that given a candidate solution returns a violated constraint if it exists: the degree constraints are trivial to check and the subtour elimination constraints are checked using a min cut algorithm (if a, b are joined by an edge and all edge capacities are made equal to one, then a violated subtour constraint corresponds to a cut smaller than two). This means that the subtour-LP for the TSP with specified endpoints can potentially be solved in polynomial time in the bit size of the input description, using the ellipsoid method [10].

Key Results

The main results of this paper are that, given a sample set V with $a, b \in V$ from a benign semiregular open curve γ with endpoints a, b and satisfying certain *sampling condition* [it], then

- the optimal traveling salesman path on V with endpoints a, b is a polygonal reconstruction of γ from V,
- the subtour-LP for traveling salesman paths has an optimal integral solution which is unique.

This means that, under the sampling conditions, the subtour-LP solution provides a TSP solution and also suggests a reconstruction algorithm: solve the subtour-LP and, if the solution is integral, output it. If the input satisfies the sampling condition, then the solution will be integral and the result is indeed a polygonal reconstruction. Two algorithms are proposed to solve the subtour-LP. First, using the simplex method and the cutting plane framework: it starts with an LP consisting of only the degree constraints and in each iteration solves the current LP and checks whether that solution satisfies all the subtour elimination constraints (using a min cut algorithm) and, if not, adds a violated constraint to the current LP. This algorithm has a potentially exponential running time. Second, using a similar approach but with the ellipsoid method. This can be implemented so that the running time is polynomial in the bit size of the input points. This requires justification for using approximate point coordinates and distances.

The main tool in deriving these results is the connection between the subtour-LP and the so-called *Held–Karp bound*. The line of argument is as follows:

- Let $c(u, v) = \|uv\|$ and $\mu : V \to R$ be a *potential function*. The corresponding *modified distance function* c_μ is defined by $c_\mu(u, v) = c(u, v) - \mu(u) - \mu(v)$.
- For any traveling salesman path T with endpoints a, b,

$$c_\mu(T) = c(T) - 2\sum_{v \in V} \mu(v) + \mu(a) + \mu(b),$$

  and so an optimal traveling salesman path with endpoints a, b for c_μ is also optimal for c.
- Let C_μ be the cost of a minimum spanning tree MST_μ under c_μ, then since a traveling salesman path is a spanning tree, the optimal traveling salesman T_0 satisfies $C_\mu \le c_\mu(T_0) = c(T_0) - 2\sum_{v \in V} \mu(v) + \mu(a) + \mu(b)$, and so

$$\max_\mu \left(C_\mu + 2\sum_{v \in V} \mu(v) - \mu(a) - \mu(b) \right) \le c(T_0)\,.$$

  The term on the left is the so called Held–Karp bound.
- Now, if for a particular μ, MST_μ is a path with endpoints a, b, then MST_μ is in fact an optimal traveling salesman path with endpoints a, b, and the Held–Karp bound matches $c(T_0)$.
- The Held–Karp bound is equal to the optimal objective value of the subtour-LP. This follows by relaxation of the degree constraints in a Lagrangian fashion (see [5]) and gives an effective way to compute the Held-Karp bound: solve the subtour-LP.
- Finally, a potential function μ is constructed for γ so that, for an appropriately dense sample set V, MST_μ is unique and is a polygonal reconstruction with endpoints a, b. This then implies that solving the subtour-LP will produce a correct polygonal reconstruction.

Note that the potential function μ enters the picture only as an analysis tool. It is not needed by the algorithm. The authors extend this work to the case of open curves without specified endpoints and of closed curves using variations of the ILP formulation and a more restricted sampling condition. They also extend it to the case of a collection of closed curves. The latter requires preprocessing

that partitions points into groups that are expected to form individual curves. Then each subgroup is processed with the subtour-LP approach and then the quality of the result assessed and then that partition may be updated.

Finite Precision

The above results are obtained assuming exact representation of point samples and the distances between them, so claiming a polynomial time algorithm is not immediate as the running time of the ellipsoid method is polynomial in the bit size of the input. The authors extend the results to the case in which points and the distances between them are known only approximately and from this they can conclude the polynomial running time.

Relation to Local Feature Size

The defined potential function μ is related to the so called *local feature size* function f used in the theory of smooth curve reconstruction, where $f(p)$ is defined as the distance from p to the medial axis of the curve γ. In this paper, $\mu(p)$ is defined as $d(p)/3$ where $d(p)$ is the size of the largest neighborhood of p so that γ in that neighborhood does not *deviate significantly* from a flat segment of curve. This paper shows $f(p) < 3d(p)$. In fact, $\mu(p)$ amounts to a generalization of the local feature size to nonsmooth curves (for a corner point p, $\mu(p)$ is proportional to the size of the largest neighborhood of p such that γ inside does not deviate significantly from a corner point with two nearly flat legs incident to it, and for points near the corner, μ is defined as an appropriate interpolation of the two definitions), and is in fact similar to definitions proposed elsewhere.

Applications

The curve reconstruction problem appears in applied areas such as cartography. For example, to determine level sets, features, object contours, etc. from samples. Admittedly, these applications usually may require the ability to handle very sparse sampling and noise. The 3D version of the problem is very important in areas such as industrial manufacturing, medical imaging, and computer animation. The 2D problem is often seen as a simpler (toy) problem to test algorithmic approaches.

Open Problems

A TSP-based solution when the curve γ is a collection of curves, not all closed, is not given in this paper. A solution similar to that for closed curves (partitioning and then application of subtour-LP for each) seems feasible for general collections, but some technicalities need to be solved. More interesting is the study of corresponding reconstruction approaches for surfaces in 3D.

Experimental Results

The companion paper [2] presents results of experiments comparing the TSP-based approach to several (local) Delaunay filtering algorithms. The TSP implementation uses the simplex method and the cutting plane framework (with a potentially exponential running time algorithm). The experiments show that the TSP-based approach has a better performance, allowing for much sparser samples than the others. This is to be expected given the global nature of the TSP-based solution. On the other hand, the speed of the TSP-based solution is reported to be competitive when compared to the speed of the others, despite its potentially bad worst-case behavior.

Data Sets

None reported. Experiments in [2] were performed with a simple reproducible curve based on a sinusoidal with varying number of periods and samples.

URL to Code

The code of the TSP-based solution as well as the other solutions considered in the companion paper [2] are available from: http://www.mpi-inf.mpg.de/~althaus/LEP:Curve-Reconstruction/curve.html

Cross References

- ► Engineering Geometric Algorithms
- ► Euclidean Traveling Salesperson Problem
- ► Minimum Weight Triangulation
- ► Planar Geometric Spanners
- ► Robust Geometric Computation

Recommended Reading

1. Althaus, E., Mehlhorn, K.: Traveling salesman-based curve reconstruction in polynomial time. SIAM J. Comput. **31**, 27–66 (2001)
2. Althaus, E., Mehlhorn, K., Näher, S., Schirra, S.: Experiments on curve reconstruction. In: ALENEX, 2000, pp. 103–114
3. Amenta, N., Bern, M.: Surface reconstruction by Voronoi filtering. Discret. Comput. Geom. **22**, 481–504 (1999)
4. Amenta, N., Bern, M., Eppstein, D.: The crust and the β-skeleton: Combinatorial curve reconstruction. Graph. Model. Image Process. **60**, 125–135 (1998)
5. Cook, W., Cunningham, W., Pulleyblank, W., Schrijver, A.: Combinatorial Optimization. Wiley, New York (1998)

6. Dey, T.K.: Curve and surface reconstruction. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn. CRC, Boca Raton (2004)
7. Dey, T.K.: Curve and Surface Reconstruction: Algorithms with Mathematical Analysis. Cambridge University Press, New York (2006)
8. Edlesbrunner, H.: Shape reconstruction with the Delaunay complex. In: LATIN'98, Theoretical Informatics. Lecture Notes in Computer Science, vol. 1380, pp. 119–132. Springer, Berlin (1998)
9. Giesen, J.: Curve reconstruction, the TSP, and Menger's theorem on length. Discret. Comput. Geom. **24**, 577–603 (2000)
10. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1986)

Two-Dimensional Compressed Matching

► Multidimensional Compressed Pattern Matching

Two-Dimensional Pattern Indexing

2005; Na, Giancarlo, Park

Joong Chae Na[1], Paolo Ferragina[2], Raffaele Giancarlo[3], Kunsoo Park[4]
[1] Department of Computer Science and Engineering, Sejong University, Seoul, Korea
[2] Department of Computer Science, University of Pisa, Pisa, Italy
[3] Department of Mathematics and Applications, University of Palermo, Palermo, Italy
[4] School of Computer Science and Engineering, Seoul National University, Seoul, Korea

Keywords and Synonyms

Two-Dimensional indexing for pattern matching; Two-Dimensional index data structures; Index data structures for matrices or images; Indexing for matrices or images

Problem Definition

This entry is concerned with designing and building indexes of a two-dimensional matrix, which is basically the generalization of indexes of a string, the *suffix tree* [12] and the *suffix array* [11], to a two-dimensional matrix. This problem was first introduced by Gonnet [7]. Informally, a two-dimensional analog of the suffix tree is a tree data structure storing all submatrices of an $n \times m$ matrix, $n \geq m$. The *submatrix tree* [2] is an incarnation of such indexes. Unfortunately, building such indexes requires $\Omega(nm^2)$ time [2]. Therefore, much of the attention paid has been restricted to square matrices and submatrices, the important special case in which much better results are available.

For square matrices, the *Lsuffix tree* and its array form, storing all *square submatrices* of an $n \times n$ matrix, have been proposed [3,9,10]. Moreover, the general framework for these index families is also introduced [4,5]. Motivated by LZ1-type image compression [14], the on-line case, i.e, the matrix is given one row or column at a time, has been also considered. These data structures can be built in time close to n^2. Building these data structures is a nontrivial extension of the algorithms for the standard suffix tree and suffix array. Generally, a tree data structure and its array form of this type for square matrices are referred to as the *two-dimensional suffix tree* and the *two-dimensional suffix array*, which are the main concerns of this entry.

Notations

Let A be an $n \times n$ matrix with entries defined over a finite alphabet Σ. $A[i..k, j..l]$ denotes the submatrix of A with corners (i,j), (k,j), (i,l), and (k,l). When $i = k$ or $j = l$, one of the repeated indexes is omitted. For $1 \leq i, j \leq n$, the *suffix* $A(i,j)$ of A is the largest square submatrix of A that starts at position (i,j) in A. That is, $A(i,j) = A[i..i+k, j..j+k]$ where $k = n - \max(i,j)$. Let $\$_i$ be a special symbol not in Σ such that $\$_i$ is lexicographically smaller than any other character in Σ. Assume that $\$_i$ is lexicographically smaller than $\$_j$ for $i < j$. For notational convenience, assume that the last entries of the ith row and column are $\$_i$. It makes all suffixes distinct. See Fig. 1a and b for an example.

Let $L\Sigma = \bigcup_{i=1}^{\infty} \Sigma^{2i-1}$. The strings of $L\Sigma$ are referred to as *Lcharacters*, and each of them is considered as an atomic item. $L\Sigma$ is called the *alphabet of Lcharacters*. Two Lcharacters are equal if and only if they are equal as strings over Σ. Moreover, given two Lcharacters La and Lb of equal length, La is *lexicographically smaller than or equal to* Lb if and only if the string corresponding to La is lexicographically smaller than or equal to that corresponding to Lb. A *chunk* is the concatenation of Lcharacters with the following restriction: an Lcharacter in Σ^{2i-1} can precede only one in $\Sigma^{2(i+1)-1}$ and succeed only one in $\Sigma^{2(i-1)-1}$. An *Lstring* is a chunk such that the first Lcharacter is in Σ.

For dealing with matrices as strings, a linear representation of square matrices is needed. Given $A[1..n, 1..n]$, divide A into n L-shaped characters. Let $a(i)$ be the concatenation of row $A[i, 1..i-1]$ and column $A[1..i, i]$. Then

$a(i)$ can be regarded as an Lcharacter. The linearized string of matrix A, called the Lstring of matrix A, is the concatenation of Lcharacters $a(1), \ldots, a(n)$. See Fig. 1c for an example. Slightly different linearizations have been used [9,10,13], but they are essentially the same in the aspect of two-dimensional functionality.

Two-Dimensional Suffix Trees

The suffix tree of matrix A is a compacted trie over the alphabet $L\Sigma$ that represents Lstrings corresponding to all suffixes of A. Formally, the *two-dimensional suffix tree* of matrix A is a rooted tree that satisfies the following conditions (see Fig. 1d for an example):

1. Each edge is labeled with a chunk.
2. There is no internal node of outdegree one.
3. Chunks assigned to sibling edges start with different Lcharacters, which are of the same length as strings in Σ^*.
4. The concatenation of the chunks labeling the edges on the path from the root to a leaf gives the Lstring of exactly one suffix of A, say $A(i, j)$. It is said that this leaf is associated with $A(i, j)$.
5. There is exactly one leaf associated with each suffix.

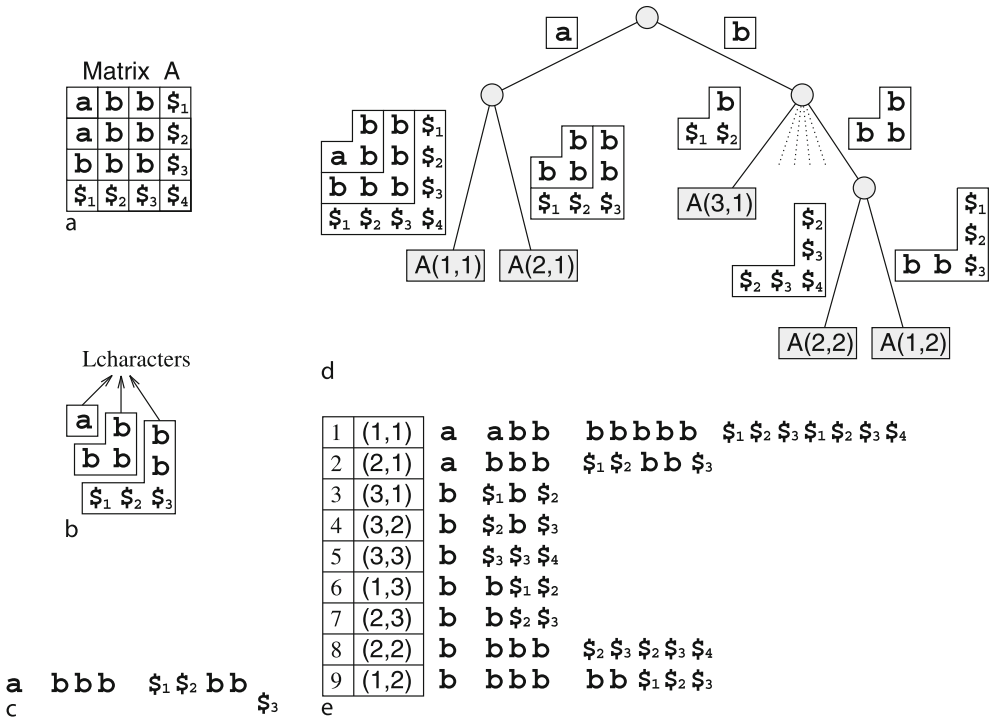
Conditions 4 and 5 mean that there is a one-to-one correspondence between the leaves of the tree and the suffixes of A (which are all distinct because $\$_i$ is unique).

Problem 1 (Construction of 2D suffix tree)
INPUT: *An $n \times n$ matrix A.*
OUTPUT: *A two-dimensional suffix tree storing all square submatrices of A.*

On-Line Suffix Trees

Assume that A is read *on-line* in row major order (column major order can be considered similarly). Let $A_t = A[1..t, 1..n]$ and $row_t = A[t, 1..n]$. At time $t-1$, nothing but A_{t-1} is known about A. At time t, row_t is read and so A_t is known. After time t, the on-line suffix tree of A is storing all suffixes of A_t. Note that Condition 4 may not be satisfied during the on-line construction of the suffix tree. A leaf may be associated with more than one suffix, because the suffixes of A_t are not all distinct.

Problem 2 (On-line construction of 2D suffix tree)
INPUT: *A sequence of rows of $n \times n$ matrix A, row_1, $row_2, \ldots, row_n$.*



Two-Dimensional Pattern Indexing, Figure 1
a A matrix A, **b** the suffix $A(2, 1)$ and Lcharacters composing $A(2, 1)$, **c** the Lstring of $A(2, 1)$, **d** the suffix tree of A, and **e** the suffix array of A (omitting the suffixes started with $\$_i$)

OUTPUT: *A two-dimensional suffix tree storing all square submatrices of A_t after reading row$_t$.*

Two-Dimensional Suffix Arrays

The *two-dimensional suffix array* of matrix A is basically a sorted list of all Lstrings corresponding to suffixes of A. Formally, the kth element of the array has the start position (i, j) if and only if the Lstring of $A(i, j)$ is the kth smallest one among the Lstrings of all suffixes of A. See Fig. 1e for an example. The two-dimensional suffix array is also coupled with additional information tables, called *Llcp* and *Rlcp*, to enhance its performance like the standard suffix array. The two-dimensional suffix array can be constructed from the two-dimensional suffix tree in linear time.

Problem 3 (Construction of 2D suffix array)
INPUT: *An $n \times n$ matrix A.*
OUTPUT: *The two-dimensional suffix array storing all square submatrices of A.*

Submatrix Trees

The *submatrix tree* is a tree data structure storing all submatrices. This entry just gives a result on submatrix trees. See [2] for details.

Problem 4 (Construction of a submatrix tree)
INPUT: *An $n \times m$ matrix B, $n \geq m$.*
OUTPUT: *The submatrix tree and its array form storing all submatrices of B.*

Key Results

Theorem 1 (Kim and Park 1999 [10], Cole and Hariharan 2000 [1]) *Given an $n \times n$ matrix A over an integer alphabet, one can construct the two-dimensional suffix tree in $O(n^2)$ time.*

Kim and Park's result is a deterministic algorithm, Cole and Hariharan's result is a randomized one. For an arbitrary alphabet, one needs first to sort it and then to apply the theorem above.

Theorem 2 (Na et al. 2005 [13]) *Given an $n \times n$ matrix A, one can construct on-line the two-dimensional suffix tree of A in $O(n^2 \log n)$ time.*

Theorem 3 (Kim et al. 2003 [9]) *Given an $n \times n$ matrix A, one can construct the two-dimensional suffix array of A in $O(n^2 \log n)$ time without constructing the two-dimensional suffix tree.*

Theorem 4 (Giancarlo 1993 [2]) *Given an $n \times m$ matrix B, one can construct the submatrix tree of B in $O(nm^2 \log(nm))$ time.*

Applications

Two-dimensional indexes can be used for many pattern-matching problems of two-dimensional applications such as low-level image processing, image compression, visual data bases, and so on [3,6]. Given an $n \times n$ text matrix and an $m \times m$ pattern matrix over an alphabet Σ, the *two-dimensional pattern retrieval problem*, which is a basic pattern matching problem, is to find all occurrences of the pattern in the text. The two-dimensional suffix tree and array of the text can be queried in $O(m^2 \log |\Sigma| + occ)$ time and $O(m^2 + \log n + occ)$ time, respectively, where occ is the number of occurrences of the pattern in the text. This problem can be easily extended to a set of texts. These queries have the same procedure and performance as those of indexes for strings. On-line construction of the two-dimensional suffix tree can be applied to LZ-1-type image compression [6].

Open Problems

The main open problems on two-dimensional indexes are to construct indexes in optimal time. The linear-time construction algorithm for two-dimensional suffix trees is already known [10]. The on-line construction algorithm due to [13] is optimal for unbounded alphabets, but not for integer or constant alphabets. Another open problem is to construct two-dimensional suffix arrays directly in linear time.

Experimental Results

An experiment that compares construction algorithms of two-dimensional suffix trees and suffix arrays was presented in [8]. Giancarlo's algorithm [2] and Kim et al.'s algorithm [8] were implemented for two-dimensional suffix trees and suffix arrays, respectively. Random matrices of sizes $200 \times 200 \sim 800 \times 800$ and alphabets of sizes 2, 4, 16 were used for input data. According to experimental results, the construction of two-dimensional suffix arrays is ten-times faster and five-times more space-efficient than that of two-dimensional suffix trees.

Cross References

- ▶ Multidimensional String Matching
- ▶ Suffix Array Construction
- ▶ Suffix Tree Construction in RAM

Recommended Reading

1. Cole, R. Hariharan, R.: Faster suffix tree construction with missing suffix links. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing, 2000, pp. 407–415
2. Giancarlo, R.: An index data structure for matrices, with applications to fast two-dimensional pattern matching. In: Proceed-

ings of Workshop on Algorithm and Data Structures, vol. 709, pp. 337–348. Springer Lect. Notes Comp. Sci. Montréal, Canada (1993)
3. Giancarlo, R.: A generalization of the suffix tree to square matrices, with application. SIAM J. Comput. **24**, 520–562 (1995)
4. Giancarlo, R., Grossi, R.: On the construction of classes of suffix trees for square matrices: Algorithms and applications. Inf. Comput. **130**, 151–182 (1996)
5. Giancarlo, R., Grossi, R.: Suffix tree data structures for matrices. In: Apostolico, A., Galil, Z. (eds.) Pattern Matching Algorithms, ch. 11„ pp. 293–340. Oxford University Press, Oxford (1997)
6. Giancarlo, R., Guaiana, D.: On-line construction of two-dimensional suffix trees. J. Complex. **15**, 72–127 (1999)
7. Gonnet, G.H.: Efficient searching of text and pictures. Tech. Report OED-88-02, University of Waterloo (1988)
8. Kim, D.K., Kim, Y.A., Park, K.: Constructing suffix arrays for multi-dimensional matrices. In: Proceedings of the 9th Symposium on Combinatorial Pattern Matching, 1998, pp. 249–260
9. Kim, D.K., Kim, Y.A., Park, K.: Generalizations of suffix arrays to multi-dimensional matrices. Theor. Comput. Sci. **302**, 401–416 (2003)
10. Kim, D.K., Park, K.: Linear-time construction of two-dimensional suffix trees. In: Proceedings of the 26th International Colloquium on Automata, Languages, and Programming, 1999, pp. 463–372
11. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**, 935–948 (1993)
12. McCreight, E.M.: A space-economical suffix tree construction algorithms. J. ACM **23**, 262–272 (1976)
13. Na, J.C., Giancarlo, R., Park, K.: $O(n^2 \log n)$ time on-line construction of two-dimensional suffix trees. In: Proceedings of the 11th International Computing and Combinatorics Conference, 2005, pp. 273–282
14. Storer, J.A.: Lossless image compression using generalized LZ1-type methods. In: Proceedings of Data Compression Conference, 1996, pp. 290–299

Two-Dimensional Pattern Matching with Scaling

► Two-Dimensional Scaled Pattern Matching

Two-Dimensional Scaled Pattern Matching

2006; Amir, Chencinski

Amihood Amir[1]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel

[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

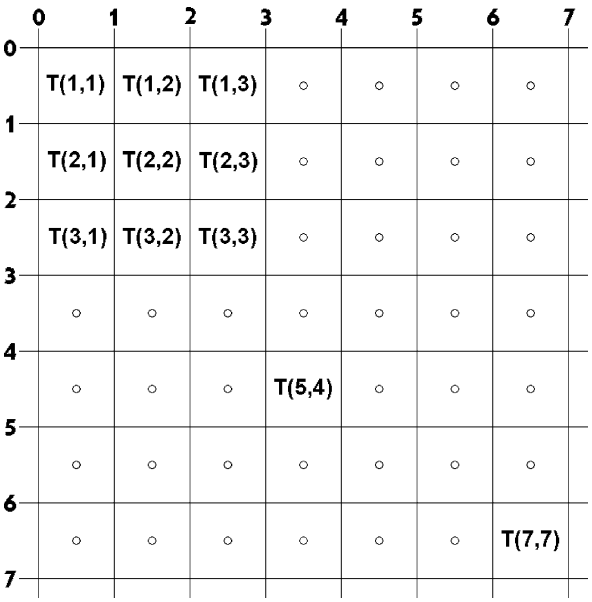
Keywords and Synonyms

Pattern matching in scaled images; 2d scaled matching; Two dimensional pattern matching with scaling; Multidimensional scaled search

Problem Definition

Definition 1 Let T be a two-dimensional $n \times n$ array over some alphabet Σ.

1. The *unit pixels array* for T (T^{1X}) consists of n^2 unit squares, called pixels in the real plane $\Re^2$. The corners of the pixel $T[i, j]$ are $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$, and (i, j). Hence the pixels of T form a regular $n \times n$ array that covers the area between $(0, 0)$, $(n, 0)$, $(0, n)$, and (n, n). Point $(0, 0)$ is the *origin* of the unit pixel array. The *center* of each pixel is the geometric center point of its square location. Each pixel $T[i, j]$ is identified with the value from Σ that the original array T had in that position. Say that the pixel has a color or a character from Σ. See Fig. 1 for an example of the grid and pixel centers of a 7×7 array.
2. Let $r \in \Re$, $r \geq 1$. The *r-ary pixels array* for T (T^{rX}) consists of n^2 *r-squares*, each of dimension $r \times r$ whose *origin* is $(0, 0)$ and covers the area between $(0, 0)$, $(nr, 0)$, $(0, nr)$, and (nr, nr). The corners of the pixel $T[i, j]$ are $((i-1)r, (j-1)r)$, $(ir, (j-1)r)$, $((i-1)r, jr)$, and (ir, jr). The *center* of each pixel is the geometric center point of its square location.



Two-Dimensional Scaled Pattern Matching, Figure 1
The grid and pixel centers of a unit pixel array for a 7 × 7 array

Two-Dimensional Scaled Pattern Matching, Figure 2
An original image, scaled by 1.3 and scaled by 2, using the geometric model definition of scaling

Notation: Let $r \in \Re$. $[r]$ denotes the *rounding* of r, i. e.

$$[r] = \begin{cases} \lfloor r \rfloor & \text{if } r - \lfloor r \rfloor < .5; \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

Definition 2 Let T be an $n \times n$ text array, P be an $m \times m$ pattern array over alphabet Σ, and let $r \in \Re$, $1 \leq r \leq \frac{n}{m}$. Say that there is an *occurrence of P scaled to r* at text location (i, j) if the following conditions hold:

Let T^{1X} be the unit pixels array of T and P^{rX} be the r-ary pixel arrays of P. Translate P^{rX} onto T^{1X} in a manner that the origin of P^{rX} coincides with location $(i-1, j-1)$ of T^{1X}. Every center of a pixel in T^{1X} which is within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ has the same color as the r-square of P^{rX} in which it falls.

The colors of the centers of the pixels in T^{1X} which are within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ define a $[mr] \times [mr]$ array over Σ. This array is denoted by $P^{s(r)}$ and called *P scaled to r*.

The above definition is the one provided in the *geometric model*, pioneered by Landau and Vishkin [15], and Fredriksson and Ukkonen [14]. Prior to the advent of the geometric model, the only discrete definition of scaling was to natural scales, as defined by Amir, Landau and Vishkin [10]:

Definition 3 Let $P[m \times m]$ be a two-dimensional matrix over alphabet Σ (not necessarily bounded). Then P scaled by s (P^s) is the $sm \times sm$ matrix where every symbol $P[i, j]$ of P is replaced by a $s \times s$ matrix whose elements all equal the symbol in $P[i, j]$. More precisely,

$$P^s[i, j] = P[\lceil \frac{i}{s} \rceil, \lceil \frac{j}{s} \rceil] .$$

Say that pattern $P[m \times m]$ *occurs* (or an occurrence of P starts) at location (k, l) of the text T if for any $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, m\}$, $T[k+i-1, l+j-1] = P[i, j]$.

The *two dimensional pattern matching problem with natural scales* is defined as follows.

INPUT: Pattern matrix $P[i, j]$ $i = 1, \ldots m; j = 1, \ldots, m$ and Text matrix $T[i, j]$ $i = 1, \ldots, n; j = 1, \ldots, n$ where $n > m$.

OUTPUT: all locations in T where an occurrence of P scaled by s (an s-occurrence) starts, for any $s = 1, \ldots, \lfloor \frac{n}{m} \rfloor$.

The natural scales definition cannot answer normal everyday occurrences such as an image scaled to, say, 1.3. This led to the geometric model. The geometric model is a discrete adaptation, without smoothing, of scaling as used in computer graphics. The definition is pleasing in a "real-world" sense. Figure 2 shows "lenna" scaled to non-discrete scales by the geometric model definition. The results look natural.

It is possible, of course, to consider a *one dimensional* version of scaling, or scaling in *strings*. Both above definitions apply for one dimensional scaling where the text and pattern are taken to be matrices having a single row. The interest in one dimensional scaling lies because of two reasons: (1) There is a faster algorithm for one dimensional scaling in the geometric model than the restriction of the two dimensional scaling algorithm to one dimension. (2) Historically, before the geometric model was defined, there was an attempt [3] to define real scaling on strings as follows.

Definition 4 Denote the string $aa \cdots a$, where a is repeated r times, by a^r. The *one dimensional floor real scaled matching problem* is the following.

INPUT: A pattern $P = a_1^{r_1} a_2^{r_2} \ldots a_j^{r_j}$, of length m, and a text T of length n.
OUTPUT: All locations in the text where the substring $a_1^{c_1} a_2^{\lfloor r_2 k \rfloor} \ldots a_{j-1}^{\lfloor r_{j-1} k \rfloor} a_j^{c_j}$ appears, where $c_1 \geq \lfloor r_1 k \rfloor$ and $c_j \geq \lfloor r_j k \rfloor$.

This definition indeed handles real scaling but has a significant weakness in that a string of length m scaled to r may be significantly shorter than mr. For this reason the definition could not be generalized to two dimensions. The geometric model does not suffer from these deficiencies.

Key Results

The first results in scaled natural matching dealt with fixed finite-sized alphabets.

Theorem 1 (Amir, Landau, and Vishkin 1992 [10]) *There exists an $O(|T| \log |\Sigma|)$ worst-case time solution to the two dimensional pattern matching problem with natural scales, for fixed finite alphabet Σ.*

The main idea behind the algorithm is analyzing the text with the aid of *power columns*. Those are the text columns appearing $m - 1$ columns apart, where P is an $m \times m$ pattern. This dependence on the pattern size make the power columns useless where a dictionary of different sized patterns is involved. A significantly simpler algorithm with an additional advantage of being alphabet-independent was presented in [6].

Theorem 2 (Amir and Calinescu 2000 [6]) *There exists an $O(|T|)$ worst-case time solution to the two dimensional pattern matching problem with natural scales.*

The alphabet independent time complexity of this algorithm was achieved by developing a scaling-invariant "signature" of the pattern. This idea was further developed to scaled dictionary matching.

Theorem 3 (Amir and Calinescu 2000 [6]) *Given a static dictionary of square pattern matrices. It is possible in $O(|D| \log k)$ preprocessing, where $|D|$ is the total dictionary size and k is the number of patterns in the dictionary, and $O(|T| \log k)$ text scanning time, for input text T, to find all occurrences of dictionary patterns in the text in all natural scales.*

This is identical to the time at [8], the best non-scaled matching algorithm for a static dictionary of square patterns. It is somewhat surprising that scaling does not add to the complexity of single matching nor dictionary matching.

The first algorithm to solve the scaled matching problem for real scales, was a one dimensional real scaling algorithm using Definition 4.

Theorem 4 (Amir, Butman, and Lewenstein 1998 [3]) *There exists an $O(|T|)$ worst-case time solution to the one dimensional floor real scaled matching problem.*

The first algorithm to solve the two dimensional scaled matching problem for real scales in the geometric model is the following.

Theorem 5 (Amir, Butman, Lewenstein, and Porat 2003 [4]) *Given an $n \times n$ text and $m \times m$ pattern. It is possible to find all pattern occurrences in all real scales in time $O(nm^3 + n^2 m \log m)$ and space $O(nm^3 + n^2)$.*

The above result was improved.

Theorem 6 (Amir and Chencinski 2006 [7]) *Given an $n \times n$ text and $m \times m$ pattern. It is possible to find all pattern occurrences in all real scales in time $O(n^2 m)$ and space $O(n^2)$.*

This algorithm achieves its time by exploiting geometric characteristics of nested scales occurrences and a sophisticated use of dueling [1,16].

The assumption in both above algorithms is that the scaled occurrence of the pattern starts at the top left corner of some pixel.

It turns out that one can achieve faster times in the *one dimensional* real scaled matching problem, even in the geometric model.

Theorem 7 (Amir, Butman, Lewenstein, Porat, and Tsur 2004 [5]) *Given a text string T of length n and a pattern string P of length m, there exists an $O(n \log m + m\sqrt{nm \log m})$ worst-case time solution to the one dimensional pattern matching problem with real scales in the geometric model.*

Applications

The problem of finding approximate occurrences of a template in an image is a central one in digital libraries and web searching. The current algorithms to solve this problem use methods of computer vision and computational geometry. They model the image in another space and seek a solution there. A deterministic worst-case algorithm in pixel-level images does not yet exist. Yet, such an algorithm could be useful, especially in raw data that has not been modeled, e. g. movies. The work described here advances another step toward this goal from the scaling point of view.

Open Problems

Finding all scaled occurrences without fixing the scaled pattern start at the top left corner of the text pixel would be important from a practical point of view. The final goal is an integration of scaling with rotation [2,11,12,13] and local errors (edit distance) [9].

Cross References

- ► Multidimensional Compressed Pattern Matching
- ► Multidimensional String Matching

Recommended Reading

1. Amir, A., Benson, G., Farach, M.: An alphabet independent approach to two dimensional pattern matching. SIAM J. Comput. **23**(2), 313–323 (1994)
2. Amir, A., Butman, A., Crochemore, M., Landau, G.M., Schaps, M.: Two-dimensional pattern matching with rotations. Theor. Comput. Sci. **314**(1–2), 173–187 (2004)
3. Amir, A., Butman, A., Lewenstein, M.: Real scaled matching. Inf. Proc. Lett. **70**(4), 185–190 (1999)
4. Amir, A., Butman, A., Lewenstein, M., Porat, E.: Real two dimensional scaled matching. In: Proc. 8th Workshop on Algorithms and Data Structures (WADS '03), pp. 353–364 (2003)
5. Amir, A., Butman, A., Lewenstein, M., Porat, E., Tsur, D.: Efficient one dimensional real scaled matching. In: Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE '04), pp. 1–9 (2004)
6. Amir, A., Calinescu, G.: Alphabet independent and dictionary scaled matching. J. Algorithms **36**, 34–62 (2000)
7. Amir, A., Chencinski, E.: Faster two dimensional scaled matching. In: Proc. 17th Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 4009, pp. 200–210. Springer, Berlin (2006)
8. Amir, A., Farach, M.: Two dimensional dictionary matching. Inf. Proc. Lett. **44**, 233–239 (1992)
9. Amir, A., Landau, G.: Fast parallel and serial multidimensional approximate array matching. Theor. Comput. Sci. **81**, 97–115 (1991)
10. Amir, A., Landau, G.M., Vishkin, U.: Efficient pattern matching with scaling. J. Algorithms **13**(1), 2–32 (1992)
11. Amir, A., Tsur, D., Kapah, O.: Faster two dimensional pattern matching with rotations. In: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM '04), pp. 409–419 (2004)
12. Fredriksson, K., Mäkinen, V., Navarro, G.: Rotation and lighting invariant template matching. In: Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04). LNCS, pp. 39–48 (2004)
13. Fredriksson, K., Navarro, G., Ukkonen, E.: Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In: Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002). LNCS, vol. 2373, pp. 235–248 (2002)
14. Fredriksson, K., Ukkonen, E.: A rotation invariant filter for two-dimensional string matching. In: Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 1448, pp. 118–125. Springer, Berlin (1998)
15. Landau, G.M., Vishkin, U.: Pattern matching in a digitized image. Algorithmica **12**(3/4), 375–408 (1994)
16. Vishkin, U.: Optimal parallel pattern matching in strings. Proc. 12th ICALP, pp. 91–113 (1985)

Two-Interval Pattern Problems

2004; Vialette
2007; Cheng, Yang, Yuan

Stéphane Vialette
IGM-LabInfo, University of Paris-East, Descartes, France

Keywords and Synonyms

2-intervals; RNA structures

Problem Definition

The problem is concerned with finding large constrained patterns in sets of 2-intervals. Given a single-stranded RNA molecule, a sequence of contiguous bases of the molecule can be represented as an interval on a single line, and a possible pairing between two disjoint sequences can be represented as a 2-interval, which is merely the union of two disjoint intervals. Derived from arc-annotated sequences, 2-interval representation considers thus only the bonds between the bases and the pattern of the bonds, such as hairpin structures, knots and pseudoknots. A maximum cardinality disjoint subset of a candidate set of 2-intervals restricted to certain prespecified geometrical constraints can provide a useful valid approximation for RNA secondary structure determination.

The geometric properties of 2-intervals provide a possible guide for understanding the computational complexity of finding structured patterns in RNA sequences. Using a model to represent nonsequential information allows us to vary restrictions on the complexity of the pattern structure. Indeed, two disjoint 2-intervals, i. e., two 2-intervals that do not intersect in any point, can be in precedence order ($<$), be allowed to nest ($\sqsubset$) or be allowed to cross ($\between$). Furthermore, the set of 2-intervals and the pattern can have different restrictions, e. g., all intervals have the same length or all the intervals are disjoint. These different combinations of restrictions alter the computational complexity of the problems, and need to be examined separately. This examination produces efficient algorithms for more restrictive structured patterns, and hardness results for those that are less restrictive.

Notations

Let $I = [a, b]$ be an interval on the line. Write $\mathsf{start}(I) = a$ and $\mathsf{end}(I) = b$. A *2-interval* is the union of two dis-

joint intervals defined over a single line and is denoted by $D = (I, J)$; I is completely to the left of J. Write $\mathsf{left}(D) = I$ and $\mathsf{right}(D) = J$. Two 2-intervals $D_1 = (I_1, J_1)$ and $D_2 = (I_2, J_2)$ are said to be *disjoint* (or *nonintersecting*) if both 2-intervals share no common point, i. e., $(I_1 \cup J_1) \cap (I_2 \cup J_2) = \emptyset$. For such disjoint pairs of 2-intervals, three natural binary relations, denoted $<$, $\sqsubset$ and $\between$, are of special interest:

- $D_1 < D_2$ (D_1 *precedes* D_2), if $I_1 < J_1 < I_2 < J_2$,
- $D_1 \sqsubset D_2$ (D_1 is *nested in* D_2), if $I_2 < I_1 < J_1 < J_2$, and
- $D_1 \between D_2$ (D_1 *crosses* D_2), if $I_1 < I_2 < J_1 < J_2$.

A pair of 2-intervals D_1 and D_2 is said to be *R-comparable* for some $R \in \{<, \sqsubset, \between\}$, if either $D_1 R D_2$ or $D_2 R D_1$. Note that any two disjoint 2-intervals are R-comparable for some $R \in \{<, \sqsubset, \between\}$. A set of disjoint 2-intervals $\mathcal{D}$ is said to be $\mathcal{R}$-*comparable* for some $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, $\mathcal{R} \neq \emptyset$, if any pair of distinct 2-intervals in $\mathcal{D}$ is R-comparable for some $R \in \mathcal{R}$. The nonempty subset $\mathcal{R}$ is called a *model* for $\mathcal{D}$.

The 2-interval-pattern problem asks one to find in a set of 2-intervals a largest subset of pairwise compatible 2-intervals. In the present context, compatibility denotes the fact that any two 2-intervals in the solution are (1) nonintersecting and (2) satisfy some prespecified geometrical constraints. The 2-interval-pattern problem is formally defined as follows:

Problem 1 (2-interval-pattern)
INPUT: *A set of 2-intervals $\mathcal{D}$ and a model $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$.*
SOLUTION: *A $\mathcal{R}$-comparable subset $\mathcal{D}' \subseteq \mathcal{D}$.*
MEASURE: *The size of the solution, i. e., $|\mathcal{D}'|$.*

According to the above definition, any solution for the 2-interval-pattern problem for some model $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$ corresponds to an RNA structure constrained by $\mathcal{R}$. For example, a solution for the 2-interval-pattern problem for the $\mathcal{R} = \{<, \sqsubset\}$ model corresponds to a pseudoknot-free structure (a *pseudoknot* in an RNA sequence $S = s_1, s_2, \ldots, s_n$ is composed of two interleaving nucleotide pairings (s_i, s_j) and $(s_{i'}, s_{j'})$ such that $i < i' < j < j'$).

Some additional definitions are needed for further algorithmic analysis. Let $\mathcal{D}$ be a set of 2-intervals. The *width* (respectively *height*, *depth*) is the size of a maximum cardinality $\{<\}$-comparable (respectively $\{\sqsubset\}$-comparable, $\{\between\}$-comparable) subset $\mathcal{D}' \subseteq \mathcal{D}$. The *interleaving distance* of a 2-interval $D_i \in \mathcal{D}$ is defined to be the distance between the two intervals of D_i, i. e., $\mathsf{start}(\mathsf{right}(D_i)) - \mathsf{end}(\mathsf{left}(D_i))$. The *total interleaving distance* of the set of 2-intervals $\mathcal{D}$, written $\mathcal{L}(\mathcal{D})$, is the sum of all interleaving distances, i. e., $\mathcal{L}(\mathcal{D}) = \sum_{D_i \in \mathcal{D}} \mathsf{start}(\mathsf{right}(D_i)) - \mathsf{end}(\mathsf{left}(D_i))$. The *interesting coordinates* of $\mathcal{D}$ are defined to be the set $X(\mathcal{D}) = \bigcup_{D_i \in \mathcal{D}} \{\mathsf{end}(\mathsf{left}(D_i)), \mathsf{start}(\mathsf{right}(D_i))\}$. The *density* of $\mathcal{D}$, written $d(\mathcal{D})$, is the maximum number of 2-intervals in $\mathcal{D}$ over a single point. Formally, $d(\mathcal{D}) = \max_{x \in X(\mathcal{D})} \{D \in \mathcal{D} : \mathsf{end}(\mathsf{left}(D) \leq x < \mathsf{start}(\mathsf{right}(D))\}$.

Constraints

The structure of the set of all (simple) intervals involved in a set of 2-intervals $\mathcal{D}$ turns out to be of particular importance for algorithmic analysis of the 2-interval-pattern problem. The *interval ground set* of $\mathcal{D}$, denoted $\mathcal{I}(\mathcal{D})$, is the set of all intervals involved in $\mathcal{D}$, i. e., $\mathcal{I}(\mathcal{D}) = \{\mathsf{left}(D_i) : D_i \in \mathcal{D}\} \cup \{\mathsf{right} < (D_i) : D_i \in \mathcal{D}\}$. In [7,20], four types of interval ground sets were introduced.

1. *Unlimited*: no restriction on the structure.
2. *Balanced*: each 2-interval $D_i \in \mathcal{D}$ is composed of two intervals having the same length, i. e., $|\mathsf{left}(D_i)| = |\mathsf{right}(D_i)|$.
3. *Unit*: the interval ground set $\mathcal{I}(\mathcal{D})$ is solely composed of unit length intervals.
4. *Disjoint*: no two distinct intervals in the interval ground set $\mathcal{I}(\mathcal{D})$ intersect.

Observe that a unit 2-interval set is balanced, while the converse is not necessarily true. Furthermore, for most applications, one may assume that a disjoint 2-interval set is unit. Observe that in this latter case, a set of 2-intervals reduces to a graph $G = (V, E)$ equipped with a numbering of its vertices from 1 to $|V|$, and hence the 2-interval-pattern problem for disjoint interval ground sets reduces to finding a constrained maximum matching in a linear graph. Considering additional restrictions such as:

- Bounding the width, the height or the depth of either the input set of 2-intervals or the solution subset
- Bounding the interleaving distances

is also of interest for practical applications.

Key Results

The different combinations of the models and interval ground sets alter the computational complexity of the 2-interval-pattern problem. The main results are summarized in Tables 1 (time complexity and hardness) and 2 (approximation for hard instances).

Theorem 1 *The 2-interval-pattern problem is approximable (APX) hard for models $\mathcal{R} = \{<, \sqsubset, \between\}$ and $\mathcal{R} = \{\sqsubset, \between\}$, and is nondeterministic polynomial-time (NP) complete – in its natural decision version – for model $\mathcal{R} = \{<, \between\}$, even when restricted to unit interval ground sets.*

Two-Interval Pattern Problems, Table 1
Complexity of the 2-interval-pattern problem for all combinations of models and interval ground sets. For the polynomial-time cases, $n = |\mathcal{D}|$, $\mathcal{L} = \mathcal{L}(\mathcal{D})$ and $d = d(\mathcal{D})$

Model $\mathcal{R}$	Interval ground set $\mathcal{I}(\mathcal{D})$	
	Unlimited, Balanced, Unit	Disjoint
$\{<,\sqsubset,\between\}$	APX-hard [1]	$O(n\sqrt{n})$ [15]
$\{<,\between\}$	NP-complete [3]	**unknown**
$\{\sqsubset,\between\}$	APX-hard [19]	$O(n \log n + \mathcal{L})$ [8]
$\{<,\sqsubset\}$	$O(n \log n + nd)$ [8]	
$\{<\}$	$O(n \log n)$ [19]	
$\{\sqsubset\}$	$O(n \log n)$ [3]	
$\{\between\}$	$O(n \log n + \mathcal{L})$ [8]	

Notice here that the 2-interval-pattern problem for model $\mathcal{R} = \{<,\between\}$ is not APX-hard. Two hard cases of the 2-interval-pattern turn out to be polynomial-time-solvable when restricted to disjoint-interval ground sets.

Theorem 2 *The 2-interval-pattern problem for a disjoint-interval ground set is solvable in*

- $O(n\sqrt{n})$ *time for model* $\mathcal{R} = \{<,\sqsubset,\between\}$ *(trivial reduction to the standard maximum matching problem)*
- $O(n \log n + \mathcal{L})$ *time for model* $\mathcal{R} = \{\sqsubset,\between\}$

The complexity of the 2-interval-pattern problem for model $\mathcal{R} = \{<,\between\}$ and a disjoint-interval ground set is still unknown. Three cases of the 2-interval-pattern problem are polynomial-time-solvable, regardless of the structure of the interval ground sets.

Theorem 3 *The 2-interval-pattern problem is solvable in*

- $O(n \log n + nd)$ *time for model* $\mathcal{R} = \{<,\sqsubset\}$
- $O(n \log n)$ *time for models* $\mathcal{R} = \{<\}$ *and* $\mathcal{R} = \{\sqsubset\}$
- $O(n \log n + \mathcal{L})$ *time for model* $\mathcal{R} = \{\between\}$

One may now turn to approximating hard instances of the 2-interval-pattern problem. Surprisingly enough, no significant differences (in terms of approximation guarantees) have yet been found for the 2-interval-pattern problem between the model $\mathcal{R} = \{<,\sqsubset,\between\}$ and the model $\mathcal{R} = \{\sqsubset,\between\}$ (the approximation algorithms are, however, different).

Theorem 4 *The 2-interval-pattern problem for model* $\mathcal{R} = \{<,\sqsubset,\between\}$ *or model* $\mathcal{R} = \{\sqsubset,\between\}$ *is approximable within ratio*

- *4 for unlimited-interval ground sets, and*
- $2 + \epsilon$ *for unit-interval ground sets.*

The 2-interval-pattern problem for model $\mathcal{R} = \{<,\between\}$ *is approximable within ratio* $1 + 1/\epsilon$, $\epsilon \geq 2$ *for all models.*

A practical 3-approximation algorithm for model $\mathcal{R} = \{<,\sqsubset,\between\}$ (resp. $\mathcal{R} = \{\sqsubset,\between\}$) and unit interval ground set that runs in $\mathcal{O}(n \lg n)$ (resp. $\mathcal{O}(n^2 \lg n)$) time has been proposed in [1] (resp. [7]). For model $\mathcal{R} = \{<,\between\}$, a more practical 2-approximation algorithm that runs in $\mathcal{O}(n^3 \lg n)$ time has been proposed in [10]. Notice that Theorem 4 holds true for the weighted version of the 2-interval-pattern problem [7] except for models $\mathcal{R} = \{<,\sqsubset,\between\}$ and $\mathcal{R} = \{\sqsubset,\between\}$ and unit interval ground set where the best approximation ratio is $2.5 + \epsilon$ [5].

Applications

Sets of 2-intervals can be used for modeling stems in RNA structures [20,21], determining DNA sequence similarities [13] or scheduling jobs that are given as groups of non-intersecting segments in the real line [1,9]. In all these applications, one is concerned with finding a maximum cardinality subset of nonintersecting 2-intervals. Some other classical combinatorial problems are also of interest [5]. Also, considering sets of t-intervals (each element is the union of at most t disjoint intervals) and their corresponding intersection graph has proved to be useful.

It is computationally challenging to predict RNA structures including pseudoknots [14]. Practical approaches to cope with intractability are either to restrict the class of pseudoknots under consideration [18] or to use heuristics [6,17,19]. The general problem of establishing a general representation of structured patterns, i. e., *macroscopic describers* of RNA structures, was considered in [20]. Sets of 2-intervals provide such a natural geometric description.

Constructing a relevant 2-interval set from a RNA sequence is relatively easy: stable stems are selected, usually according to a simplified thermodynamic model without accounting for loop energy [2,16,19,20,21]. Predicting a reliable RNA structure next reduces to finding a maximum subset of nonconflicting 2-intervals, i. e., a subset of disjoint 2-intervals. Considering in addition a model $\mathcal{R} \subseteq \{<,\sqsubset,\between\}$ allows us to vary restrictions on the complexity of the pattern structure. In [21], the treewidth of the intersection graph of the set of 2-intervals is considered for speeding up the computation.

For sets of 2-intervals involved in practical applications, restrictions on the interval ground set are needed. Unit interval ground sets were considered in [7]. Of particular importance in the context of molecular biology (RNA structures and DNA sequence similarities) are balanced interval ground sets, where each 2-interval is composed of two equally length intervals.

Two-Interval Pattern Problems, Table 2
Performance ratios for hard instances of the 2-interval-pattern problem. LP stands for *Linear Programming* and N/A stands for *Not Applicable*

Model $\mathcal{R}$	Interval ground set $\mathcal{I}(\mathcal{D})$			
	Unlimited	Balanced	Unit	Disjoint
$\{<, \sqsubset, \between\}$	4 LP [1]	4 $\mathcal{O}(n \lg n)$ [7]	$2+\epsilon$ $\mathcal{O}(n^2 + n^{\mathcal{O}(\log 1/\epsilon)})$ [13]	N/A
$\{\sqsubset, \between\}$	4 LP [7]	4 $\mathcal{O}(n^2 \lg n)$ [7]	$2+\epsilon$ $\mathcal{O}(n^2 + n^{\mathcal{O}(\log 1/\epsilon)})$ [13]	N/A
$\{<, \between\}$	$1 + 1/\epsilon$ $\quad \mathcal{O}(n^{2\epsilon+3})$, $\epsilon \geq 2$ [14]			

Open Problems

A number of problems related to the 2-interval-pattern problem remain open. First, improving the approximation ratios for the various flavors of the 2-interval-pattern problem is of particular importance. For example, the existence of a fast approximation algorithm with good performance guarantee for the 2-interval-pattern problem for model $\mathcal{R} = \{<, \sqsubset, \between\}$ remains an apparently challenging open problem. A related open research area is concerned with balanced-interval ground sets. In particular, no evidence has shown yet that the 2-interval-pattern problem becomes easier to approximate for balanced-interval ground sets. This question is of special importance in the context of RNA structures where most 2-intervals are balanced.

A number of important question are still open for model $\mathcal{R} = \{<, \between\}$. First, it is still unknown whether the 2-interval-pattern problem for disjoint-interval ground sets and model $\mathcal{R} = \{<, \between\}$ is polynomial-time-solvable. Observe that this problem trivially reduces to the following graph problem: Given a graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, find a maximum cardinality matching $\mathcal{M} \subseteq E$ such that for any two distinct edges $\{i, j\}$ and $\{k, l\}$ of $\mathcal{M}$, $i < j$, $k < l$ and $i < k$, either $j < k$ or $j < l$. Another open question concerns the approximation of the 2-interval-pattern problem for balanced interval ground set. Is this special case better approximable than the general case?

A last direction of research is concerned with the parameterized complexity of the 2-interval-pattern problem. For example, it is not known whether the 2-interval-pattern problem for models $\mathcal{R} = \{<, \sqsubset, \between\}$, $\mathcal{R} = \{\sqsubset, \between\}$ or $\mathcal{R} = \{<, \between\}$ is fixed-parameter-tractable when parameterized by the size of the solution. Also, investigating the parameterized complexity for parameters such as the maximum number of pairwise crossing intervals in the input set or the treewidth of the corresponding intersection 2-interval graph, which are expected to be relatively small for most practical applications, is of particular interest.

Cross References

- ▶ RNA Secondary Structure Prediction Including Pseudoknots
- ▶ RNA Secondary Structure Prediction by Minimum Free Energy

Recommended Reading

1. Bar-Yehuda, R., Halldorsson, M., Naor, J., Shachnai, H., Shapira, I.: Scheduling split intervals. In: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 732–741
2. Billoud, B., Kontic, M., Viari, A.: Palingol a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database. Nucleic. Acids. Res. **24**, 1395–1403 (1996)
3. Blin, G., Fertin, G., Vialette, S.: Extracting 2-intervals subsets from 2-interval sets. Theor. Comput. Sci. **385**(1–3), 241–263 (2007)
4. Blin, G., Fertin, G., Vialette, S.: New results for the 2-interval pattern problem. In: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM). Lecture Notes in Computer Science, vol. 3109. Springer, Berlin (2004)
5. Butman, A., Hermelin, D., Lewenstein, M., Rawitz, D.: Optimization problems in multiple-interval graphs. In: Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM-SIAM, 2007, pp. 268–277
6. Chen, J.-H., Le, S.-Y., Maize, J.: Prediction of common secondary structures of RNAs: a genetic algorithm approach. Nucleic. Acids. Res. **28**, 991–999 (2000)
7. Crochemore, M., Hermelin, D., Landau, G., Rawitz, D., Vialette, S.: Approximating the 2-interval pattern problem, Theoretical Computer Science (special issue for Alberto Apostolico) (2008)
8. Erdong, C., Linji, Y., Hao, Y.: Improved algorithms for 2-interval pattern problem. J. Combin. Optim. **13**(3), 263–275 (2007)
9. Halldorsson, M., Karlsson, R.: Strip graphs: Recognition and scheduling. In: Proc. 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG). Lecture Notes in Computer Science, vol. 4271, pp. 137–146. Springer, Berlin (2006)
10. Jiang, M.: A 2-approximation for the preceding-and-crossing structured 2-interval pattern problem, J. Combin. Optim. **13**, 217–221 (2007)
11. Jiang, M.: Improved approximation algorithms for predicting RNA secondary structures with arbitrary pseudoknots. In: Proc. 3rd International Conference on Algorithmic Aspects in Infor-

mation and Management (AAIM), Portland, OR, USA, Lecture Notes in Computer Science, vol. 4508, pp. 399–410. Springer (2007)
12. Jiang, M.: A PTAS for the weighted 2-interval pattern problem over the preceding-and-crossing model. In: Y.X. A.W.M. Dress, B. Zhu (eds.) Proc. 1st Annual International Conference on Combinatorial Optimization and Applications (COCOA), Xi'an, China, Lecture Notes in Computer Science, vol. 4616, pp. 378–387. Springer (2007)
13. Joseph, D., Meidanis, J., Tiwari, P.: Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In: Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT). Lecture Notes in Computer Science, pp. 326–337. Springer, Berlin (1992)
14. Lyngsø, R., Pedersen, C.: RNA pseudoknot prediction in energy-based models. J. Comput. Biol. **7**, 409–427 (2000)
15. Micali, S., Vazirani, V.. An $O(sqrt|V||E|)$ algorithm for finding maximum matching in general graphs. In: Proc. 21st Annual Symposium on Foundation of Computer Science (FOCS), IEEE, 1980, pp. 17–27
16. Nussinov, R., Pieczenik, G., Griggs, J., Kleitman, D.: Algorithms for loop matchings. SIAM J. Appl. Math. **35**, 68–82 (1978)
17. Ren, J., Rastegart, B., Condon, A., Hoos, H.: HotKnots: Heuristic prediction of rna secondary structure including pseudoknots. RNA **11**, 1194–1504 (2005)
18. Rivas, E., Eddy, S.: A dynamic programming algorithm for RNA structure prediction including pseudoknots. J. Mol. Biol. **285**, 2053–2068 (1999)
19. Ruan, J., Stormo, G., Zhang, W.: An iterated loop matching approach to the prediction of RNA secondary structures with pseudoknots. Bioinformatics **20**, 58–66 (2004)
20. Vialette, S.: On the computational complexity of 2-interval pattern matching. Theor. Comput. Sci. **312**, 223–249 (2004)
21. Zhao, J., Malmberg, R., Cai, L.: Rapid ab initio rna folding including pseudoknots via graph tree decomposition. In: Proc. Workshop on Algorithms in Bioinformatics. Lecture Notes in Computer Science, vol. 4175, pp. 262–273. Springer, Berlin (2006)

Two-Level Boolean Minimization

1956; McCluskey

Robert Dick
Department Electrical Engineering and Computer Systems, Northwestern University, Evanston, IL, USA

Keywords and Synonyms

Logic minimization; Quine–McCluskey algorithm; Tabular method

Problem Definition

Summary

Find a minimal sum-of-products expression for a Boolean function.

Two-Level Boolean Minimization, Table 1
Equivalent representations with different implementation complexities

Expression	Meaning in english	Boolean logic identity
$\overline{a} \wedge \overline{b} \vee \overline{a} \wedge b$	not a and not b or not a and b	Distributivity
$\overline{a} \wedge (\overline{b} \vee b)$	not a and either not b or b	Complements
$\overline{a} \wedge True$	not a and *True*	Boundedness
$\overline{a}$	not a	

Extended Definition

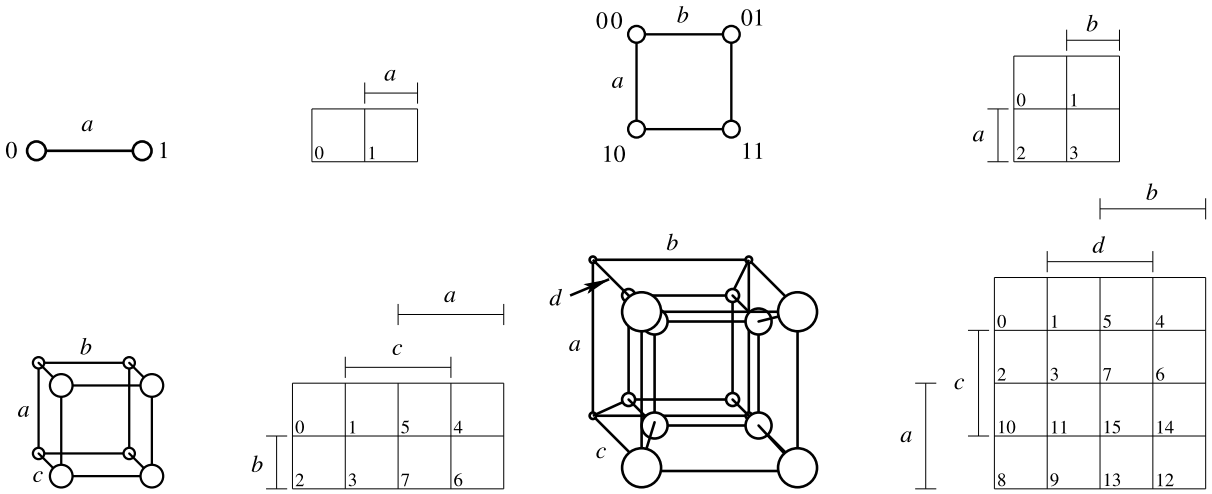
Consider a Boolean algebra with two elements: *False* or *True*. A Boolean function $f(y_1, y_2, \cdots, y_n)$ of n Boolean input variables specifies an output value for each combination of input variable values. It is possible to represent the same function with a number of different expressions. For example, the first and last expressions in Table 1 correspond to this function. Assuming access to complemented input variables, straight-forward implementations of these expressions would require two *and* gates and an *or* gate for $\overline{a} \wedge \overline{b} \vee \overline{a} \wedge b$ and only a wire for $\overline{a}$. Although the implementation efficiency depends on target technology, in general terser expressions enable greater efficiency. Boolean minimization is the task of deriving the tersest expression for a function. Elegant and optimal algorithms exist for solving the variant of this problem in which the expression is limited to two levels, i. e., a layer of *and* gates followed by a single *or* gate or a layer of *or* gates followed by a single *and* gate.

Key Results

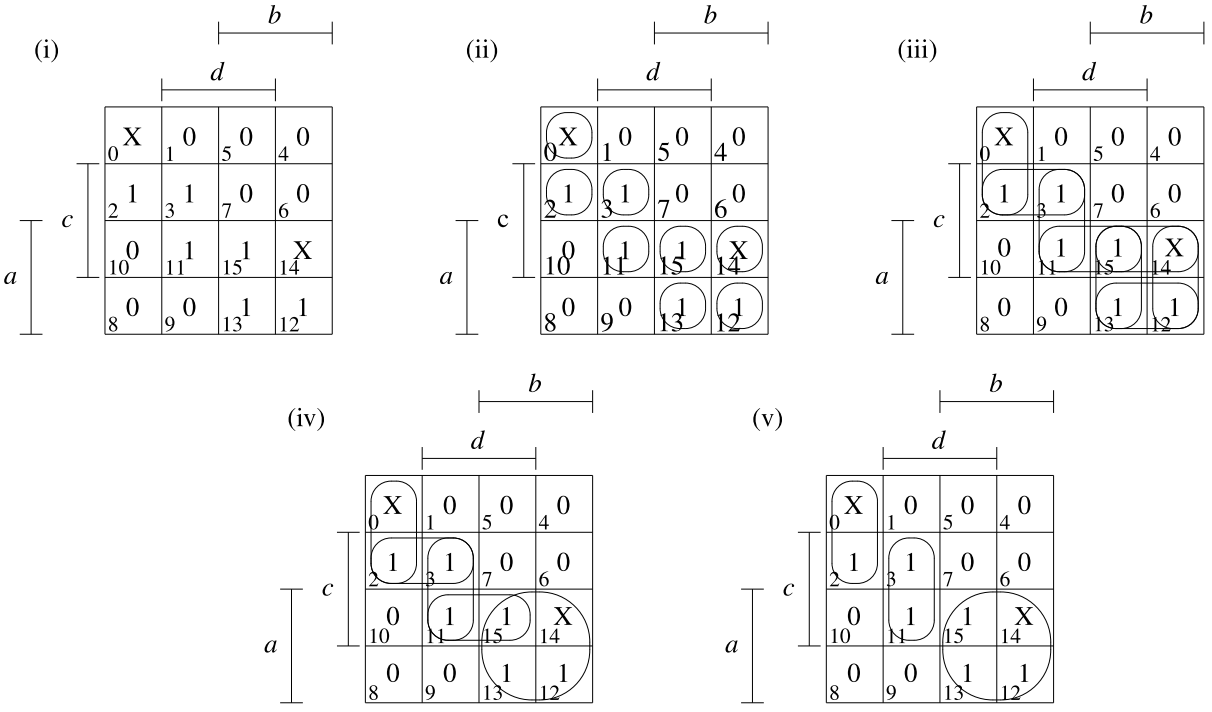
This survey will start by introducing the Karnaugh Map visualization technique, which will be used to assist in the subsequent explanation of the Quine–McCluskey algorithm for two-level Boolean minimization. This algorithm is optimal for its constrained problem variant. It is one of the fundamental algorithms in the field of computer-aided design and forms the basis or inspiration for many solutions to more general variants of the Boolean minimization problem.

Karnaugh Maps

Karnaugh Maps [4] provide a method of visualizing adjacency in Boolean space. A Karnaugh Map is a projection of an n-dimensional hypercube onto two-dimensional surface such that adjacent points in the hypercube remain adjacent in the two-dimensional projection. Figure 1 illustrates Karnaugh Maps of 1, 2, 3, and 4 variables: a, b, c, and d.



Two-Level Boolean Minimization, Figure 1
Boolean function spaces from one to four dimensions and their corresponding Karnaugh Maps



Two-Level Boolean Minimization, Figure 2
(i) Karnaugh Map of function $f(a, b, c, d)$, (ii) elementary implicants, (iii) second-order implicants, (iv) prime implicants, and (v) a minimal cover

A *literal* is a single appearance of a complemented or uncomplemented input variable in a Boolean expression. A product term or *implicant* is the Boolean product, or *and*, of one or more literals. Every implicant corresponds to a repeated balanced bisection of Boolean space, or of the corresponding Karnaugh Map, i. e., an implicant is a rectangle in a Karnaugh Map with width m and height n where $m = 2^j$ and $n = 2^k$ for arbitrary non-negative integers j and k, e.g, the ovals in Fig. 2(ii–v). An *elementary implicant* is an implicant in which, for each variable of the cor-

responding function, the variable or its complement appears, e. g., the circles in Fig. 2(ii). Implicant A *covers* implicant B if every elementary implicant in B is also in A.

Prime implicants are implicants that are not covered by any other implicants, e. g., the ovals and circle in Fig. 2(iv). It is unnecessary to consider anything but prime implicants when seeking a minimal function representation because, if a non-prime implicants could be used to cover some set of elementary implicants, there is guaranteed to exist a prime implicant that covers those elementary implicants and contains fewer literals. One can draw the largest implicants covering each elementary implicant and covering no positions for which the function is *False*, thereby using Karnaugh Maps to identify prime implicants. One can then manually seek a compact subset of prime implicants covering all elementary implicants in the function.

This Karnaugh Map based approach is effective for functions with few inputs, i. e., those with low dimensionality. However, representing and manipulating Karnaugh Maps for functions of many variables is challenging. Moreover, the Karnaugh Map method provides no clear set of rules to follow when selecting a minimal subset of prime implicants to implement a function.

The Quine–McCluskey Algorithm

The Quine–McCluskey algorithm provides a formal, optimal way of solving the two-level Boolean minimization problem. W. V. Quine laid the essential theoretical groundwork for optimal two-level logic minimization [7,8]. However, E. J. McCluskey first proposed a precise algorithm to fully automate the process [6].

The Quine–McCluskey method has two phases: (1) produce all prime implicants and (2) select a minimal subset of prime implicants covering the function. In the first phase, the elementary implicants of a function are iteratively combined to produce implicants with fewer literals. Eventually, all prime implicants are thus produced. In the second phase, a minimal subset of prime implicants covering the on-set elementary implicants is selected using unate covering.

The Quine–McCluskey method may be illustrated using an example. Consider the function indicated by the Karnaugh Map in Fig. 2(i) and the truth table in Table 2. For each combination of Boolean input variable values, the function $f(a, b, c, d)$ is required to output a 0 (*False*), a 1 (*True*), or has no requirement. The lack of a requirement is indicated with an X, or don't-care symbol.

Expanding implicants as much as possible will ultimately produce the prime implicants. To do this, combine on-set and don't-care elementary implicants using the reduction theorem ($\overline{a}b \vee ab = b$) shown in Table 1. The elementary implicants are circled in Fig. 2(ii) and listed in the second column of Table 3. In this table, 0s indicate complemented variables and 1s indicate uncomplemented variables, e. g., 0010 corresponds to $\overline{a}\,\overline{b}c\overline{d}$. It is necessary to determine all possible combinations of implicants. It is impossible to combine non-adjacent implicants, i. e., those that differ in more than one variable. Therefore, it is not necessary to consider combining any pair of implicants with a number of uncomplemented variables differing by any value other than 1. This fact can be exploited by organizing the implicants based on the number of ones they contain, as indicated by the first column in Table 3. All possible combinations of implicants in adjacent subsets are considered. For example, consider combining 0010 with 0011, which results in 001X or $\overline{a}\,\overline{b}c$, and also consider combining 0010 with 1100, which is impossible due to differences in more than one variable. Whenever an implicant is successfully merged, it is marked. These marked implicants are clearly not prime implicants because the implicants they produced cover them and contain fewer literals. Note that marked implicants should still be used for subsequent combinations. The merged implicants in

Two-Level Boolean Minimization, Table 2
Truth table of function $f(a, b, c, d)$

Elementary implicant (a, b, c, d)	Function value (a, b, c, d)	Elementary implicant	Function value
0000	X	1000	0
0001	0	1001	0
0010	1	1010	0
0011	1	1011	1
0100	0	1100	1
0101	0	1101	1
0110	0	1110	X
0111	0	1111	1

Two-Level Boolean Minimization, Table 3
Identifying prime implicants

Number of ones	Elementary implicant (a, b, c, d)	Second-order implicant	Third-order implicant
0	0000 ✓	00X0	
1	0010 ✓	001X	
2	0011 ✓ 1100 ✓	X011 110X ✓ 11X0 ✓	11XX
3	1011 ✓ 1101 ✓ 1110 ✓	1X11 11X1 ✓ 111X ✓	
4	1111 ✓		

Two-Level Boolean Minimization, Table 4
Solving unate covering problem to select minimal cover

Requirements (elementary implicants)	Resources (prime implicants)				
	00X0	001X	X011	1X11	11XX
0010	✓	✓			
0011		✓	✓		
1011			✓	✓	
1100					✓
1101					✓
1111				✓	✓

the third column of Table 3 correspond to those depicted in Fig. 2(iii).

After all combinations of elementary implicants have been considered, and successful combinations listed in the third column, this process is repeated on the second-order merged implicants in the third column, producing the implicants in the fourth column. Implicants that contain don't-care marks in different locations may not be combined. This process is repeated until a column yielding no combinations is arrived at. The unmarked implicants in Table 3 are the prime implicants, which correspond to the implicants depicted in Fig. 2(iv).

After a function's prime implicants have been identified, it is necessary to select a minimal subset that covers the function. The problem can be formulated as unate covering. As shown in Table 4, label each column of a table with a prime implicant; these are resources that may be used to fulfill the requirements of the function. Label each row with an elementary implicant from the on-set; these rows correspond to requirements. Do not add rows for don't-cares. Don't-cares impose no requirements, although they were useful in simplifying prime implicants. Mark each row–column intersection for which the elementary implicant corresponding to the row is covered by the prime implicant corresponding to the column. If a column is selected, all the rows for which the column contains marks are *covered*, i. e., those requirements are satisfied. The goal is to cover all rows with a minimal-cost subset of columns. McCluskey defined minimal cost as having a minimal number of prime implicants, with ties broken by selecting the prime implicants containing the fewest literals. The most appropriate cost function depends on the implementation technology. One can also use a similar formulation with other cost functions, e. g., minimize the total number of literals by labeling each column with a cost corresponding to the number of literals in the corresponding prime implicant.

One can use a number of heuristics to accelerate solution of the unate covering problem, e. g., neglect rows that have a superset of the marks of any other row, for they will be implicitly covered and neglect columns that have a subset of the marks of any other column if their costs are as high, for the other column is at least as useful. One can easily select columns as long as there exists a row with only one mark because the marked column is required for a valid solution. However, there exist problem instances in which each row contains multiple marks. In the worst case, the best existing algorithms are required to make tentative decisions, determine the consequences, then backtrack and evaluate alternative decisions.

The unate covering problem appears in many applications. It is $\mathcal{NP}$-complete [5], even for the instances arising during two-level minimization [9]. Its use in the Quine–McCluskey method predates its categorization as an $\mathcal{NP}$-complete problem by 16 years. A detailed treatment of this problem would go well beyond the scope of this entry. However, Gimpel [3] as well as Coudert and Madre [2] provide good starting points for further reading.

Some families of logic functions have optimal two-level representations that grow in size exponentially in the number of inputs, but have more compact multi-level implementations. These families are frequently encountered in arithmetic, e. g., a function indicating whether the number of on inputs is odd. Efficient implementation of such functions requires manual design or multilevel minimization [1].

Applications

Digital computers are composed of precisely two things: (1) implementations of Boolean logic functions and (2) memory elements. The Quine–McCluskey method is used to permit efficient implementation of Boolean logic functions in a wide range of digital logic devices, including computers. The Quine–McCluskey method served as a starting point or inspiration for most currently-used logic minimization algorithms. Its direct use is contradicted when functions are not amenable to efficient two-level implementation, e. g., many arithmetic functions.

Cross References

► Local Approximation of Covering and Packing Problems

Recommended Reading

1. Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A.L.: Multilevel logic synthesis. Proc. IEEE **78**(2), 264–300 (1990)
2. Coudert, O., Madre, J.C.: New ideas for solving covering problems. In: Proc. Design Automation Conf., 1995, pp. 641–646

3. Gimpel, J.F.: A reduction technique for prime implicant tables. IEEE Trans. Electron. Comput. **14**(4), 535–541 (1965)
4. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Trans. AIEE, Commun. Electron. **72**, 593–599 (1953)
5. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
6. McCluskey, E.J.: Minimization of Boolean functions. Bell Syst. Tech. J. **35**(6), 1417–1444 (1956)
7. Quine, W.V.: The problem of simplyfying truth functions. Am. Math. Mon. **59**(8), 521–531 (1952)
8. Quine, W.V.: A way to simplify truth functions. Am. Math. Mon. **62**(9), 627–631 (1955)
9. Umans, C., Villa, T., Sangiovanni-Vincentelli, A.L.: Complexity of two-level logic minimization. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **25**(7), 1230–1246 (2006)

Two-Person Game

► Complexity of Bimatrix Nash Equilibria

Two-Player Game

► Complexity of Bimatrix Nash Equilibria

Two-Player Nash

► Complexity of Bimatrix Nash Equilibria