S

Schedulers for Optimistic Rate Based Flow Control

2005; Fatourou, Mavronicolas, Spirakis

Panagiota Fatourou
Department of Computer Science,
University of Ioannina, Ioannina, Greece

Keywords and Synonyms

Rate allocation; Rate adjustment; Bandwidth allocation

Problem Definition

The problem concerns the design of efficient rate-based flow control algorithms for virtual-circuit communication networks where a connection is established by allocating a fixed path, called *session*, between the source and the destination. Rate-based flow-control algorithms repeatedly adjust the transmission rates of different sessions in an end-to-end manner with primary objectives to optimize the network utilization and achieve some kind of fairness in sharing bandwidth between different sessions.

A widely-accepted fairness criterion for flow-control is *max-min fairness* which requires that the rate of a session can be increased only if this increase does not cause a decrease to any other session with smaller or equal rate. Once max-min fairness has been achieved, no session rate can be increased any further without violating the above condition or exceeding the bandwidth *capacity* of some link. Call *max-min* rates the session rates when max-min fairness has been reached.

Rate-based flow control algorithms perform rate adjustments through a sequence of *operations* in a way that the capacities of network links are never exceeded. Some of these algorithms, called *conservative* [3,6,10,11,12], employ operations that gradually increase session rates until they converge to the max-min rates without ever performing any rate decreases. On the other hand, *optimistic* algorithms, introduced more recently by Afek, Mansour, and Ostfeld [1], allow for decreases, so that a session's rate may be intermediately be larger than its final max-min rate.

Optimistic algorithms [1,7] employ a specific rate adjustment operation, called *update operation* (introduced in [1]). The goal of an `update` operation is to achieve fairness among a set of neighboring sessions and optimize the network utilization in a local basis. More specifically, an `update` operation calculates an increase for the rate of a particular session (the *updated* session) for each link the session traverses. The calculated increase on a particular link is the maximum possible that respects the max-min fairness condition between the sessions traversing the link; that is, this increase should not cause a decrease to the rate of any other session traversing the link with smaller rate than the rate of the updated session after the increase. Once the maximum increase on each link has been calculated the minimum among them is applied to the session's rate (let e be the link for which the minimum increase has been calculated). This causes the decrease of the rates of those sessions traversing e which had larger rates than the increased rate of the updated session to the new rate. Moreover, the `update` operation guarantees that all the capacity of link e is allocated to the sessions traversing it (so the bandwidth of this link is fully utilized).

One important performance parameter of a rate-based flow control algorithm is its *locality* which is characterized by the amount of knowledge the algorithm requires to decide which session's rate to update next. *Oblivious* algorithms do not assume any knowledge of the network topology or the current session rates. *Partially oblivious* algorithms have access to session rates but they are unaware of the network topology, while *non-oblivious* algorithms require full knowledge of both the network topology and the session rates. Another crucial performance parameter of rate-based flow control algorithms is the *convergence complexity* measured as the maximum number of rate-

adjustment operations performed in any execution until max-min fairness is achieved.

Key Results

Fatourou, Mavronicolas and Spirakis [7] have studied the convergence complexity of optimistic rate-based flow control algorithms under varying degrees of locality. More specifically, they have proved lower and upper bounds on the convergence complexity of oblivious, partially-oblivious and non-oblivious algorithms. These bounds are expressed in terms of n the number of sessions laid out on the network.

Theorem 1 (Lower Bound for Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *Any optimistic, oblivious, rate-based flow control algorithm requires $\Omega(n^2)$ `update` operations to compute the max-min rates.*

Fatourou, Mavronicolas and Spirakis [7] have presented algorithm `RoundRobin`, which applies `update` operations to sessions in a round robin order. Obviously, `RoundRobin` is oblivious. It has been proved [7] that the convergence complexity of `RoundRobin` is $O(n^2)$. This shows that the lower bound for oblivious algorithms is tight.

Theorem 2 (Upper Bound for Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *`RoundRobin` computes the max-min rates after performing $O(n^2)$ `update` operations.*

`RoundRobin` belongs to a class of oblivious algorithms, called `Epoch` [7]. Each algorithm of this class repeatedly chooses some permutation of all session indices and applies `update` operations on the sessions in the order determined by this permutation. This is performed n times. Clearly, `Epoch` is a class of oblivious algorithms. It has been proved [7] that each of the algorithms in this class has convergence complexity $O(n^2)$.

Another oblivious algorithm, called `Arbitrary`, has been presented in [1]. The algorithm works in a very simple way by choosing the next session to be updated in an arbitrary way, but it requires an exponential number of `update` operations to compute the max-min rates.

Fatourou, Mavronicolas and Spirakis [7] have proved that partially-oblivious algorithms do not achieve better convergence complexity than oblivious algorithms despite the knowledge they employ.

Theorem 3 (Lower Bound for Partially Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *Any optimistic, partially oblivious, rate-based flow control algorithm requires $\Omega(n^2)$ `update` operations to compute the max-min rates.*

Afek, Mansour and Ostfeld [1] have presented a partially oblivious algorithm, called `GlobalMin`. The algorithm chooses as the session to update next the one with the minimum rate among all sessions. The convergence complexity of `GlobalMin` is $O(n^2)$ [1]. This shows that the lower bound for partially-oblivious algorithms is tight.

Theorem 4 (Upper Bound for Partially Oblivious algorithms, Afek, Mansour and Ostfeld [1]) *`GlobalMin` computes the max-min rates after performing $O(n^2)$ `update` operations.*

Another partially-oblivious algorithm, called `LocalMin`, is also presented in [1]. The algorithm chooses to schedule next a session which has a minimum rate among all the sessions that share a link with it. `LocalMin` has time complexity $O(n^2)$.

Fatourou, Mavronicolas and Spirakis [7] have presented a non-oblivious algorithm, called `Linear`, that exhibits linear convergence complexity. `Linear` follows the classical idea [3,12] of selecting as the next updated session one of the sessions that traverse the most congested link in the network. To discover such a session, `Linear` requires knowledge of the network topology and the session rates.

Theorem 5 (Upper Bound for Non-Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *`Linear` computes the max-min rates after performing $O(n)$ `update` operations.*

The convergence complexity of `Linear` is optimal, since n rate adjustments must be performed in any execution of an optimistic rate-based flow control algorithm (assuming that the initial session rates are zero). However, this comes at a remarkable cost in locality which makes `Linear` impractical.

Applications

Flow control is the dominant technique used in most communication networks for preventing data traffic congestion when the externally injected transmission load is larger than what can be handled even with optimal routing. Flow control is also used to ensure high network utilization and fairness among the different connections. Examples of networking technologies where flow control techniques have been extensively employed to achieve these goals are TCP streams [5] and ATM networks [4]. An overview of flow control in practice is provided in [3].

The idea of controlling the rate of a traffic source originates back to the data networking protocols of the ANSI Frame Relay Standard. Rate-based flow control is considered attractive due to its simplicity and its low hardware requirements. It has been chosen by the ATM Forum on Traffic Management as the best suited technique for the goals of ABR service [4].

A substantial amount of research work has been devoted in past to conservative flow control algorithms [3,6,10,11,12]. The optimistic framework has been introduced much later by Afek et al. [1] as a more suitable approach for real dynamic networks where decreases of session rates may be necessary (e. g., for accommodating the arrival of new sessions). The algorithms presented in [7] improve upon the original algorithms proposed in [1] in terms of either convergence complexity, or locality, or both. Moreover, they identify that certain classical scheduling techniques, such as round-robin [11], or adjusting the rates of sessions traversing one of the most congested links [3,12] can be efficient under the optimistic framework. The first general lower bounds on the convergence complexity of rate-based flow control algorithms are also presented in [7].

The performance of optimistic algorithms has been theoretically analyzed in terms of an abstraction, namely the `update` operation, which has been designed to address most of the intricacies encountered by rate-based flow control algorithms. However, the `update` operation masks low-level implementation details, while it may incur non-trivial, local computations on the switches of the network. Fatourou, Mavronicolas and Spirakis [9] have studied the impact on the efficiency of optimistic algorithms of local computations required at network switches in order to implement the `update` operation, and proposed a distributed scheme that implements a broad class of such algorithms. On a different avenue, Afek, Mansour and Ostfeld [2] have proposed a simple flow control scheme, called `Phantom`, which employs the idea of considering an imaginary session on each link [10,12], and they have discussed how `Phantom` can be applied to ATM networks and networks of TCP routers.

A broad class of modern distributed applications (e. g., remote video, multimedia conferencing, data visualization, virtual reality, etc.) exhibit highly differing bandwidth requirements and need some kind of quality of service guarantees. To efficiently support a wide diversity of applications sharing available bandwidth, a lot of research work has been devoted on incorporating priority schemes on current networking technologies. Priorities offer a basis for modeling the diverse resource requirements of modern distributed applications, and they have been used to accommodate the needs of network management policies, traffic levels, or pricing. The first efforts for embedding priority issues into max-min fair, rate-based flow control were performed in [10,12]. An extension of the classical theory of max-min fair, rate-based flow control to accommodate priorities of different sessions has been presented in [8]. (A number of other papers addressing similar generalizations of max-min fairness to account for priorities and utility have been presented after the original publication of [8].)

Many modern applications are not based solely on point-to-point communication but they rather require multipoint-to-multipoint transmissions. A max-min fair rate-based flow control algorithm for multicast networks is presented in [14]. Max-min fair allocation of bandwidth in wireless adhoc networks is studied in [15].

Open Problems

The research work on optimistic, rate-based flow control algorithms leaves open several interesting questions. The convergence complexity of the proposed optimistic algorithms has been analyzed only for a static set of sessions laid out on the network. It would be interesting to evaluate these algorithms under a dynamic network setting, and possibly extend the techniques they employ to efficiently accommodate arriving and departing sessions.

Although max-min fairness has emerged as the most frequently praised fairness criterion for flow control algorithms, achieving it might be expensive in highly dynamic situations. Afek et al. [1] have proposed a modified version of the `update` operation, called *approximate* `update`, which applies an increase to some session only if it is larger than some quantity $\delta > 0$. An *approximate* optimistic algorithm uses the approximate `update` operation and terminates if no session rate can be increased by more than δ. Obviously such an algorithm does not necessarily reach max-min fairness. It has been proved [1] that for some network topologies every approximate optimistic algorithm may converge to session rates that are away from their max-min counterparts by an exponential factor. The consideration of other versions of `update` operation or different termination conditions might lead to better max-min fairness approximations and deserves more study; different choices may also significantly impact the convergence complexity of approximate optimistic algorithms. It would be also interesting to derive trade-off results between the convergence complexity of such algorithms and the distance of the terminating rates they achieve to the max-min rates.

Fairness formulations that naturally approximate the max-min condition have been proposed by Kleinberg et al. [13] as suitable fairness criteria for certain routing and load balancing applications. Studying these formulations under the rate-based flow control setting is an interesting open problem.

Cross References

► Multicommodity Flow, Well-linked Terminals and Routing Problems

Recommended Reading

1. Afek, Y., Mansour, Y., Ostfeld, Z.: Convergence complexity of optimistic rate based flow control algorithms. J. Algorithms **30**(1), 106–143 (1999)
2. Afek, Y., Mansour, Y., Ostfeld, Z.: Phantom: a simple and effective flow control scheme. Comput. Netw. **32**(3), 277–305 (2000)
3. Bertsekas, D.P., Gallager, R.G.: Data Networks, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
4. Bonomi, F., Fendick, K.: The Rate-Based Flow Control for Available Bit Rate ATM Service. IEEE/ACM Trans. Netw. **9**(2), 25–39 (1995)
5. Brakmo, L.S., Peterson, L.: TCP Vegas: End-to-end Congestion Avoidance on a Global Internet. IEEE J. Sel. Areas Commun. **13**(8), 1465–1480 (1995)
6. Charny, A.: An algorithm for rate-allocation in a packet-switching network with feedback. Technical Report MIT/LCS/TR-601, Massachusetts Institute of Technology, April 1994
7. Fatourou, P., Mavronicolas, M., Spirakis, P.: Efficiency of oblivious versus non-oblivious schedulers for optimistic, rate-based flow control. SIAM J. Comput. **34**(5), 1216–1252 (2005)
8. Fatourou, P., Mavronicolas, M., Spirakis, P.: Max-min fair flow control sensitive to priorities. J. Interconnect. Netw. **6**(2), 85–114 (2005) (also in Proceedings of the 2nd International Conference on Principles of Distributed Computing, pp. 45–59 (1998)
9. Fatourou, P., Mavronicolas, M., Spirakis, P.: The global efficiency of distributed, rate-based flow control algorithms. In: Proceedings of the 5th Colloqium on Structural Information and Communication Complexity, pp. 244–258, June 1998
10. Gafni, E., Bertsekas, D.: Dynamic control of session input rates in communication networks. IEEE Trans. Autom. Control **29**(11), 1009–1016 (1984)
11. Hahne, E.: Round Robin Scheduling for Max-min Fairness in Data Networks. IEEE J. Sel. Areas Commun. **9**(7), 1024–1039 (1991)
12. Jaffe, J.: Bottleneck Flow Control. IEEE Trans. Commun. **29**(7), 954–962 (1981)
13. Kleinberg, J., Rabani, Y., Tardos, É.: Fairness in routing and load balancing. In: Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, pp. 568–578, October 1999
14. Sarkar, S., Tassiulas, L.: Fair distributed congestion control in multirate multicast networks. IEEE/ACM Trans. Netw. **13**(1), 121–133 (2005)
15. Tassiulas, L., Sarkar, S.: Maxmin fair scheduling in wireless adhoc networks. IEEE J. Sel. Areas Commun. **23**(1), 163–173 (2005)

Scheduling with Equipartition

2000; Edmonds

JEFF EDMONDS
York University, Toronto, ON, Canada

Keywords and Synonyms

Round Robin and Equi-partition are the same algorithm. Average Response time and Flow are basically the same measure.

Problem Definition

The task is to schedule a set of n on-line jobs on p processors. The jobs are $J = \{J_1, \ldots, J_n\}$ where *job* J_i has a *release/arrival time* r_i and a sequence of phases $\langle J_i^1, J_i^2, \ldots, J_i^{q_i}\rangle$. Each phase is represented by $\langle w_i^q, \Gamma_i^q\rangle$, where w_i^q denotes the amount of *work* and Γ_i^q is the *speedup function* specifying the rate $\Gamma_i^q(\beta)$ at which this work is executed when given β processors.

A phase of a job is said to be *fully parallelizable* ⌞ if its speedup function is $\Gamma(\beta) = \beta$. It is said to be *sequential* ⌞ if its speedup function is $\Gamma(\beta) = 1$.[1] A speedup function Γ is *nondecreasing* iff $\Gamma(\beta_1) \le \Gamma(\beta_2)$ whenever $\beta_1 \le \beta_2$;[2] is *sublinear* ⌞ iff $\Gamma(\beta_1)/\beta_1 \ge \Gamma(\beta_2)/\beta_2$;[3] and is *strictly-sublinear* by α iff $\Gamma(\beta_2)/\Gamma(\beta_1) \le (\beta_2/\beta_1)^{1-\alpha}$.

An *s-speed scheduling algorithm* $S_s(J)$ allocates $s \times p$ processors each point in time to the jobs J in a way such that all the work completes.[4] More formally, it constructs a function $S(i,t)$ from $\{1,\ldots,n\} \times [0,\infty)$ to $[0, sp]$ giving the number of processors allocated to job J_i at time t. (A job is allowed to be allocated a non-integral number of processors.) Requiring that for all t, $\sum_{i=1}^{n} S(i,t) \le sp$ ensures that at most sp processors are allocated at any given time. Requiring that for all i, there exist $r_i = c_i^0 < c_i^1 < \cdots < c_i^{q_i}$ such that for all $1 \le q \le q_i$,

[1] Note that an odd feature of this definition is that a sequential job completes work at a rate of 1 even when absolutely no processors are allocated to it. This assumption makes things easier for the adversary and harder for any non-clairvoyant algorithm. Hence, it only makes these results stronger.

[2] A job phase with a nondecreasing speedup function executes no slower if it is allocated more processors.

[3] A measure of how efficient a job utilizes its processors is $\Gamma(\beta)/\beta$, which is the work completed by the job per time unit per processor. A sublinear speedup function is one whose efficiency does not increase with more processors. This is a reasonable assumption if in practice β_1 processors can simulate the execution of β_2 processors in a factor of at most β_2/β_1 more time.

[4] $S^s(J)$ is defined to be the scheduler with p processors of speed s. S_s and S^s are equivalent on fully parallelizable jobs and S^s is s times faster than S_s on sequential jobs.

$\int_{c_i^{q-1}}^{c_i^q} \Gamma_i^q(S(i,t))\,dt = w_i^q$ ensures that before a phase of a job begins, the job must have been released and all of the previous phases of the job must have completed. The *completion* time of a job J_i, denoted c_i, is the completion time of the last phase of the job.

The goal of a scheduling algorithm is to minimize the *average response time*, $\frac{1}{n}\sum_{i\in J}(c_i - r_i)$, of the jobs or equivalently its *flow time* $S_s(J) = \sum_{i\in J}(c_i - r_i)$. An alternative formalization is to integrate over time the number of jobs n_t alive at time t, $S_s(J) = \sum_{i\in J}\int_0^\infty (J_i \text{ is alive a time } t)\delta t = \int_0^\infty n_t \delta t$.

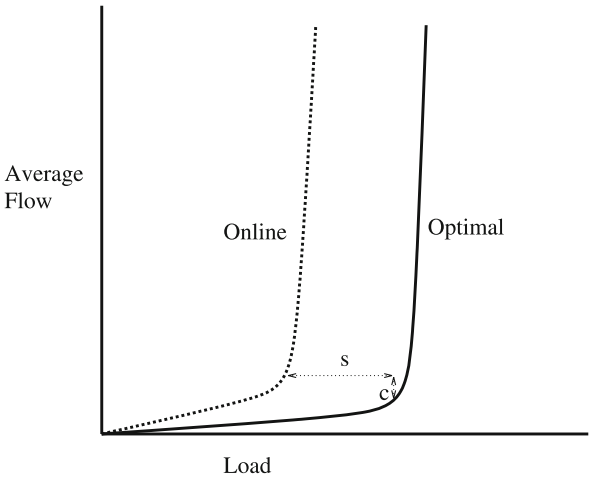
A scheduling algorithm is said to be *on-line* if it lacks knowledge of which jobs will arrive in the future. It is said to be *non-clairvoyant* if it also lacks all knowledge about the jobs that are currently in the system, except for knowing when a job arrives and knowing when it completes.

The two examples of *non-clairvoyant* schedulers that are often used in practice are *Equi-partition* (also called *Round Robin*) and *Balance*. $EQUI_s$ is defined to be the scheduler that allocates an equal number of processors to each job that is currently alive. That is, for all i and t, if job J_i is alive at time t, then $\mathcal{EQUI}(i,t) = sp/n_t$, where n_t is the number of jobs that are alive at time t. The schedule BAL_s is defined in [8] to be the schedule that allocates all of its processors to the job that has been allocated processors for the shortest length of time. (Though no one implements Balance directly, Unix uses a multi-level feedback (MLF) queue algorithm which in a way approximates Balance).

The most obvious worst-case measure of the goodness of an online non-clairvoyant scheduling algorithm S is its *competitive ratio*. This compares the perform of the algorithm to that of the optimal scheduler. However, in many cases, the limited algorithm is unable to compete against an all knowing all powerful optimal scheduler. To compensate the algorithm S_s, it is given extra speed s. An online scheduling algorithm S is said to be *s-speed c-competitive* if: $\max_J S_s(J)/Opt(J) \le c$. For example, being s-speed 2-competitive means that the cost $S_s(J)$ of scheduler S with $s \times p$ processors on any instance J is at most twice the optimal cost for the same jobs when only given p processors.

Key Results

If all jobs arrive at time zero (batch), then the flow time of *EQUI* is 2-competitive on fully parallelizable jobs [10] and $(2+\sqrt{3})$-competitive on jobs with nondecreasing sublinear speedup functions [3]. (The time until the last job completes (makespan) on fully parallelizable jobs is the same for *EQUI* and *OPT*, but can be a factor of $\Theta(\log n/\log\log n)$ worse for *EQUI* if the jobs can also have sequential phases [11].) Table 1 summarizes all the results.



Scheduling with Equipartition, Figure 1
To understand the motivation for this *resource augmentation analysis* [8], note that it is common for the quality of service of a system to have a *threshold property* with respect to the load that it is given. In this example, it seems that the online scheduling algorithm *S* performs reasonably well in comparison to the optimal scheduling algorithm. Despite this, one can see that the competitive ratio of *S* is huge by looking at the vertical gap between the curves when the load is near capacity. To explain why these curves are close, one must also measure the horizontal gap between curves. *S* performs at most *c* times worse than optimal, when either the load is decreased or equivalently the speed is increased by factor of *s*

When the jobs have arbitrary arrival times and are fully parallelizable, the optimal schedule simply allocates all the processors to the jobs with least remaining work. This, however, requires the scheduler to know the amount of work per job. Without this knowledge, *EQUI* and *BAL* are unable to compete with the optimal and hence can do a factor of $\Omega(n/\log n)$ and $\Omega(n)$ respectively worse and no non-clairvoyant schedulers has a better competitive ratio than $\Omega(n^{1/3})$ [9,10]. Randomness improves the competitive ratio of *BAL* to $\Theta(\log n \log\log n)$ [7]. Having more (or faster) processors also helps. BAL_s achieves a $s = 1+\epsilon$ speed competitive ratio of $\frac{s}{s-1} = 1 + \frac{1}{\epsilon}$ [8].

If some of the jobs are fully parallelizable and some are sequential jobs, it is hard to believe that any non-clairvoyant scheduler, even with sp processors, can perform well. Not knowing which jobs are which, it waists too many processors on the sequential jobs. Being starved, the fully parallelizable jobs fall further and further behind and then other fully parallelizable jobs arrive which fall behind as well. For example, even the randomized version of *BAL*

Scheduling with Equipartition, Table 1
Each row represents a specific scheduler and a class $\mathcal{J}$ of job sets. Here $EQUI_s$ denotes the Equi-partition scheduler with s times as many processors and $EQUI^s$ the one with processors that are s times as fast. The graphs give examples of speedup functions from the class of those considered. The columns are for different extra resources ratios s. Each entry gives the corresponding ratio between the given scheduler and the optimal

	$s = 1$	$s = 1 + \epsilon$	$s = 2 + \epsilon$	$s = 4 + 2\epsilon$	$s = \mathcal{O}(\log p)$
Batch ⌞, ◺, or ◿	$[2.71, 3.74]$				
Det. Non-clair ◿	$\Omega(n^{\frac{1}{3}})$	–			
Rand. Non-clair ◿	$\widetilde{\Theta}(\log n)$	–			
Rand. Non-clair ⌞ or ◿	$\Omega(n^{\frac{1}{2}})$	$\Omega(\frac{1}{\epsilon})$			
BAL_s ◿	$\Omega(n)$	$1 + \frac{1}{\epsilon}$	$\frac{2}{s}$		
BAL_s ◺	$\Omega(s^{-1/\alpha} n)$				
$EQUI_s$ ⌞, ◺, or ◿	$\Omega(\frac{n}{\log n})$	$\Omega(n^{1-\epsilon})$	$[1 + \frac{1}{\epsilon}, 2 + \frac{4}{\epsilon}]$	≥ 1	
$EQUI^s$ ⌞, ◺, or ◿	$\Omega(\frac{n}{\log n})$	$\Omega(n^{1-\epsilon})$	$[\frac{2}{3}(1 + \frac{1}{\epsilon}), 2 + \frac{4}{\epsilon}]$	$[\frac{2}{s}, \frac{16}{s}]$	
$EQUI$ ⌞ or ◺	$[1.48^{1/\alpha}, 2^{1/\alpha}]$				
$EQUI'_s$ Few Preempts			$\Omega(n^{1-\epsilon})$	$\Theta(1)$	
$HEQUI_s$ ◺ or ⌟			$\Omega(n^{1-\epsilon})$	$\Theta(1)$	
$HEQUI'_s$ ⌙ or ◸	$\Omega(n)$				$\Theta(1)$

can have an arbitrarily bad competitive ratio, even when given arbitrarily fast processors.

EQUI, however, does amazingly well. $EQUI_s$ achieves a $s = 2 + \epsilon$ speed competitive ratio of $2 + \frac{4}{\epsilon}$ [1]. This was later improved to $1 + \mathcal{O}(\sqrt{s}/(s-2))$, which is better for large s [1]. The intuition is that $EQUI_s$ is able to automatically "self adjust" the number of processors wasted on the sequential jobs. As it falls behind, it has more uncompleted jobs in the system and hence allocates fewer processors to each job and hence each job utilizes the processors that it is given more efficiently. The extra processors are enough to compensate for the fact that some processors are still wasted on sequential jobs. For example, suppose the job set is such that *OPT* has ℓ_t sequential jobs and at most one fully parallelizable job alive at any point in time t. (The proof starts by proving that this is the worst case.) It may take a while for the system under $EQUI_s$ to reach a "steady state", but when it does, m_t, which denotes the number of fully parallelizable jobs it has alive at time t, converges to $\frac{\ell_t}{s-1}$. At this time, $EQUI_s$ has $\ell_t + m_t$ jobs alive and *OPT* has $\ell_t + 1$. Hence, the competitive ratio is $EQUI_s(J)/OPT(J) = (\ell_t + \frac{\ell_t}{s-1}))/(\ell_t + 1) \leq \frac{s}{s-1}$, which is $1 + \frac{1}{e}$ for $s = 1 + \epsilon$. This intuition makes it appear that speed $s = 1 + \epsilon$ is sufficient. However, unless the speed is at least 2 then the competitive ratio can be bad during the time until it reaches this steady state, [8].

More surprisingly if all the jobs are *strictly sublinear*, i. e., are not fully parallel, then *EQUI* performs competitively with no extra processors [1]. More specifically, it is shown that if all the speedup functions are no more fully parallelizable than $\Gamma(\beta) = \beta^{1-\alpha}$ than the competitive ratio is at most $2^{\frac{1}{\alpha}}$. For intuition, suppose the adversary allocates $\frac{p}{n}$ processors to each of n jobs and *EQUI* falls behind enough so that it has $2^{\frac{1}{\alpha}} n$ uncompleted jobs. Then it allocates $p/(2^{\frac{1}{\alpha}} n)$ processors to each, completing work at an overall rate of $(2^{\frac{1}{\alpha}} n)\Gamma(p/(2^{\frac{1}{\alpha}} n)) = 2 \cdot n\Gamma(p/n)$. This is a factor of 2 more than that by the adversary. Hence, as in the previous result, *EQUI* has twice the speed and so performs competitively.

The results for $EQUI_s$ can be extended further. There is a competitive $s = (8 + \epsilon)$-speed non-clairvoyant scheduler that only preempts when the number of jobs in the system goes up or down by a factor of two (in some sense $\log n$ times). There is $s = (4 + \epsilon)$-speed one that includes both sublinear ◿ and superlinear ⌟ jobs. Finally, there is a $s = \mathcal{O}(\log p)$ speed one that includes both nondecreasing ⌙ and gradual ◸ jobs.

The proof of these results for $EQUI_s$ require techniques that are completely new. For example, the previous results prove that their algorithm is competitive by proving that at every point in time, the number of jobs alive under their algorithm is within a constant fraction of that under the optimal schedule. This, however, is simply not true with this less restricted model. There are job sets such that for a period of time the ratio between the numbers of alive jobs under the two schedules is unbounded. Instead,

a potential function is used to prove that this can only happen for a relatively short period of time.

The proof first transforms each possible input into a canonical input that as described above only has parallelizable or sequential phases. Having the number of fully parallelizable jobs alive under $EQUI_s$ at time t be much bigger than the number of sequential jobs alive at this same time is bad for $EQUI_s$ because it then has many more jobs alive then OPT and hence is currently incurring much higher costs. On the other hand, this same situation is also good for $EQUI_s$ because it means that it is allocating a larger fraction of its processors to the fully parallelizable jobs and hence is catching up to OPT. Both of these aspects of the current situation is carefully measured in a potential function $\Phi(t)$. It is proven that at each point in time, the occurred cost to $EQUI_s$ plus the gain $(\mathrm{d}\Phi(t))/(\mathrm{d}t)$ in this potential function is at most c times the costs occurred by OPT. Assuming that the potential function begins and ends at zero, the result follows.

More formally, the potential function is $\Phi(t) = F(t) + Q(t)$ where $Q(t)$ is total sequential work finished by $EQUI_s$ by time t minus the total sequential work finished by the adversary by time t. To define $F(t)$ requires some preliminary definitions. For $u \geq t$, define $m_u(t)$ ($\ell_u(t)$) to be number of fully parallelizable (sequential) phases executing under $EQUI_s$ at time u, for which $EQUI_s$ at time u has still not processed as much work as the adversary processed at time t. Let $n_u(t) = m_u(t) + \ell_u(t)$. Then $F(t) = \int_t^\infty f_u(m_u(t), \ell_u(t))\,\mathrm{d}u$, where $f_u(m, \ell) = \frac{s}{s-2}\frac{(m-\ell)(m+\ell)}{n_u}$. As the definition of the potential function suggests, the analysis is quite complicated.

Applications

In addition to being interesting results on their own, they have been powerful tools for the theoretical analysis of other on-line algorithms. For example, in [2,4] TCP was reduced to this problem and in [5], the online broadcast scheduling problem was reduced to this problem.

Open Problems

An open question is whether there is an algorithm that is competitive when given processors of speed $s = 1 + \epsilon$ (as opposed to $s = 2 + \epsilon$). There is a candidate algorithm that is part way between $EQUI_s$ and BAL_s.

Cross References

- ▶ Flow Time Minimization
- ▶ List Scheduling
- ▶ Load Balancing
- ▶ Minimum Flow Time
- ▶ Minimum Weighted Completion Time
- ▶ Online List Update
- ▶ Schedulers for Optimistic Rate Based Flow Control
- ▶ Shortest Elapsed Time First Scheduling

Recommended Reading

1. Edmonds, J.: Scheduling in the dark. Improved results: manuscript 2001. In: Theor. Comput. Sci. **235**, 109–141 (2000). In: 31st Ann. ACM Symp. on Theory of Computing, 1999
2. Edmonds, J.: On the Competitiveness of AIMD-TCP within a General Network. In: LATIN, Latin American Theoretical Informatics, vol. 2976, pp. 577–588 (2004). Submitted to Journal Theoretical Computer Science and/or Lecture Notes in Computer Science
3. Edmonds, J., Chinn, D., Brecht, T., Deng, X.: Non-clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics. In: 29th Ann. ACM Symp. on Theory of Computing, 1997, pp. 120–129. Submitted to SIAM J. Comput.
4. Edmonds, J., Datta, S., Dymond, P.: TCP is Competitive Against a Limited Adversary. In: SPAA, ACM Symp. of Parallelism in Algorithms and Achitectures, 2003, pp. 174–183
5. Edmonds, J., Pruhs, K.: Multicast pull scheduling: when fairness is fine. Algorithmica **36**, 315–330 (2003)
6. Edmonds, J., Pruhs, K.: A maiden analysis of longest wait first. In: Proc. 15th Symp. on Discrete Algorithms (SODA)
7. Kalyanasundaram, B., Pruhs, K.: Minimizing flow time nonclairvoyantly. In: Proceedings of the 38th Symposium on Foundations of Computer Science, October 1997
8. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. In: Proceedings of the 36th Symposium on Foundations of Computer Science, October 1995, pp. 214–221
9. Matsumoto: Competitive Analysis of the Round Robin Algorithm. in: 3rd International Symposium on Algorithms and Computation, 1992, pp. 71–77
10. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. Theor. Comput. Sci. **130** (Special Issue on Dynamic and On-Line Algorithms), 17–47 (1994). Preliminary Version in: Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 422–431
11. Robert, J., Schabanel, N.: Non-Clairvoyant Batch Sets Scheduling: Fairness is Fair enough. Personal Correspondence (2007)

Scheduling with Unknown Job Sizes

- ▶ Multi-level Feedback Queues
- ▶ Shortest Elapsed Time First Scheduling

Searching

- ▶ Deterministic Searching on the Line

Selfish Unsplittable Flows: Algorithms for Pure Equilibria

2005; Fotakis, Kontogiannis, Spirakis

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

Keywords and Synonyms

Atomic network congestion games; Cost of anarchy

Problem Definition

Consider having a set of resources E in a system. For each $e \in E$, let $d_e(\cdot)$ be the delay per user that requests its service, as a function of the total usage of this resource by all the users. Each such function is considered to be non−decreasing in the total usage of the corresponding resource. Each resource may be represented by a pair of points: an entry point to the resource and an exit point from it. So, each resource is represented by an arc from its entry point to its exit point and the model associates with this arc the cost (e. g., the delay as a function of the load of this resource) that each user has to pay if she is served by this resource. The entry/exit points of the resources need not be unique; they may coincide in order to express the possibility of offering joint service to users, that consists of a sequence of resources. Here, denote by V the set of all entry/exit points of the resources in the system. Any nonempty collection of resources corresponding to a directed path in $G \equiv (V, E)$ comprises an *action* in the system.

Let $N \equiv [n]$ be a set of users, each willing to adopt some action in the system. $\forall i \in N$, let w_i denote user i's *demand* (e. g., the flow rate from a source node to a destination node), while $\Pi_i \subseteq 2^E \setminus \emptyset$ is the collection of actions, any of which would satisfy user i (e. g., alternative routes from a source to a destination node, if G represents a communication network). The collection Π_i is called the *action set* of user i and each of its elements contains at least one resource. Any vector $\boldsymbol{r} = (r_1, \ldots, r_n) \in \Pi \equiv \times_{i=1}^{n} \Pi_i$ is a *pure strategies profile*, or a *configuration* of the users. Any vector of real functions $\boldsymbol{p} = (p_1, p_2, \ldots, p_n)$ s.t. $\forall i \in [n]$, $p_i : \Pi_i \to [0, 1]$ is a probability distribution over the set of allowable actions for user i (i. e., $\sum_{r_i \in \Pi_i} p_i(r_i) = 1$), and is called a *mixed strategies profile* for the n users.

A congestion model typically deals with users of identical demands, and thus, user cost function depending on the *number* of users adopting each action ([1,4,6]). In this work the more general case is considered, where a *weighted congestion model* is the tuple $((w_i)_{i\in N}, (\Pi_i)_{i\in N}, (d_e)_{e\in E})$. That is, the users are allowed to have different demands for service from the whole system, and thus affect the resource delay functions in a different way, depending on their own weights. A *weighted congestion game* associated with this model, is a game in strategic form with the set of users N and user demands $(w_i)_{i\in N}$, the action sets $(\Pi_i)_{i\in N}$ and cost functions $(\lambda^i_{r_i})_{i\in N, r_i\in\Pi_i}$ defined as follows: For any configuration $\boldsymbol{r} \in \Pi$ and $\forall e \in E$, let $\Lambda_e(\boldsymbol{r}) = \{i \in N : e \in r_i\}$ be the set of users exploiting resource e according to $\boldsymbol{r}$ (called the *view* of resource e wrt configuration $\boldsymbol{r}$). The *cost* $\lambda^i(\boldsymbol{r})$ *of user i for adopting strategy* $r_i \in \Pi_i$ in a given configuration $\boldsymbol{r}$ is equal to the cumulative *delay* $\lambda_{r_i}(\boldsymbol{r})$ along this path:

$$\lambda^i(\boldsymbol{r}) = \lambda_{r_i}(\boldsymbol{r}) = \sum_{e\in r_i} d_e(\theta_e(\boldsymbol{r})) \tag{1}$$

where, $\forall e \in E, \theta_e(\boldsymbol{r}) \equiv \sum_{i\in\Lambda_e(\boldsymbol{r})} w_i$ is the load on resource e wrt the configuration $\boldsymbol{r}$.

On the other hand, for a mixed strategies profile $\boldsymbol{p}$, the *expected cost of user i for adopting strategy* $r_i \in \Pi_i$ is

$$\lambda^i_{r_i}(\boldsymbol{p}) = \sum_{\boldsymbol{r^{-i}}\in\Pi^{-i}} P(\boldsymbol{p^{-i}}, \boldsymbol{r^{-i}}) \cdot \sum_{e\in r_i} d_e\left(\theta_e(\boldsymbol{r^{-i}} \oplus r_i)\right) \tag{2}$$

where, $\boldsymbol{r^{-i}}$ is a configuration of all the users except for user i, $\boldsymbol{p^{-i}}$ is the mixed strategies profile of all users except for i, $\boldsymbol{r^{-i}} \oplus r_i$ is the new configuration with user i choosing strategy r_i, and $P(\boldsymbol{p^{-i}}, \boldsymbol{r^{-i}}) \equiv \prod_{j\in N\setminus\{i\}} p_j(r_j)$ is the occurrence probability of $\boldsymbol{r^{-i}}$.

Remark 1 Here notation is abused a little bit and the model considers the user costs $\lambda^i_{r_i}$ as functions whose exact definition depends on the other users' strategies: In the general case of a mixed strategies profile $\boldsymbol{p}$, (2) is valid and expresses the expected cost of user i wrt $\boldsymbol{p}$, conditioned on the event that i chooses path r_i. If the other users adopt a pure strategies profile $\boldsymbol{r^{-i}}$, we get the special form of (1) that expresses the exact cost of user i choosing action r_i.

A congestion game in which all users are indistinguishable (i. e., they have the same user cost functions) and have the same action set, is called *symmetric*. When each user's action set Π_i consists of sets of resources that comprise (simple) paths between a unique origin-destination pair of nodes (s_i, t_i) in a network $G = (V, E)$, the model refers to a *network congestion game*. If additionally all origin-destination pairs of the users coincide with a unique pair

(s, t) one gets a *single commodity network congestion game* and then all users share exactly the same action set. Observe that a single-commodity network congestion game is not necessarily symmetric because the users may have different demands and thus their cost functions will also differ.

Selfish Behavior

Fix an arbitrary (mixed in general) strategies profile $\boldsymbol{p}$ for a congestion game $((w_i)_{i\in N}, (\Pi_i)_{i\in N}, (d_e)_{e\in E})$. We say that $\boldsymbol{p}$ is a *Nash Equilibrium (NE)* if and only if $\forall i \in N, \forall r_i, \pi_i \in \Pi_i, p_i(r_i) > 0 \Rightarrow \lambda^i_{r_i}(\boldsymbol{p}) \leq \lambda^i_{\pi_i}(\boldsymbol{p})$. A configuration $\boldsymbol{r} \in \Pi$ is a *Pure Nash Equilibrium (PNE)* if and only if $(\forall i \in N, \forall \pi_i \in \Pi_i, \lambda_{r_i}(\boldsymbol{r}) \leq \lambda_{\pi_i}(\boldsymbol{r^{-i}} \oplus \pi_i)$ where, $\boldsymbol{r}^{-i} \oplus \pi_i$ is the same configuration with $\boldsymbol{r}$ except for user i that now chooses action π_i.

Key Results

In this section the article deals with the existence and tractability of PNE in weighted network congestion games. First, it is shown that it is not always the case that a PNE exists, even for a weighted single-commodity network congestion game with only linear and 2-wise linear (e. g., the maximum of two linear functions) resource delays. In contrast, it is well known ([1,6]) that any unweighted (not necessarily single-commodity, or even network) congestion game has a PNE, for any kind of nondecreasing delays. It should be mentioned that the same result has been independently proved also by [3].

Lemma 1 *There exist instances of weighted single–commodity network congestion games with resource delays being either linear or 2–wise linear functions of the loads, for which there is no PNE.*

Theorem 2 *For any weighted multi–commodity network congestion game with linear resource delays, at least one PNE exists and can be computed in pseudo-polynomial time.*

Proof Fix an arbitrary network $G = (V, E)$ with linear resource/edge delays $d_e(x) = a_e x + b_e$, $e \in E$, $a_e, b_e \geq 0$. Let $\boldsymbol{r} \in \Pi$ be an arbitrary configuration for the corresponding weighted multi–commodity congestion game on G. For the configuration r consider the potential $\Phi(\boldsymbol{r}) = C(\boldsymbol{r}) + W(\boldsymbol{r})$, where

$$C(\boldsymbol{r}) = \sum_{e\in E} d_e(\theta_e(\boldsymbol{r}))\theta_e(\boldsymbol{r}) = \sum_{e\in E}[a_e\theta_e^2(\boldsymbol{r}) + b_e\theta_e(\boldsymbol{r})],$$

and

$$W(\boldsymbol{r}) = \sum_{i=1}^{n}\sum_{e\in r_i} d_e(w_i)w_i = \sum_{e\in E}\sum_{i\in^- e(\boldsymbol{r})} d_e(w_i)w_i = \sum_{e\in E}\sum_{i\in^- e(\boldsymbol{r})}(a_e w_i^2 + b_e w_i)$$

one concludes that

$$\Phi(\boldsymbol{r'}) - \Phi(\boldsymbol{r}) = 2w_i[\lambda^i(\boldsymbol{r'}) - \lambda^i(\boldsymbol{r})],$$

Note that the potential is a global system function whose changes are proportional to selfish cost improvements of any user. The global minima of the potential then correspond to configurations in which no user can improve her cost acting unilaterally. Therefore, any weighted multi–commodity network congestion game with linear resource delays admits a PNE. □

Applications

In [5] many experiments have been conducted for several classes of pragmatic networks. The experiments show even faster convergence to pure Nash Equilibria.

Open Problems

The Potential function reported here is polynomial on the loads of the users. It is open whether one can find a purely combinatorial potential , which will allow strong polynomial time for finding Pure Nash equilibria.

Cross References

- ► Best Response Algorithms for Selfish Routing
- ► Computing Pure Equilibria in the Game of Parallel Links
- ► General Equilibrium

Recommended Reading

1. Fabrikant A., Papadimitriou C., Talwar K.: The complexity of pure nash equilibria. In: Proc. of the 36th ACM Symp. on Theory of Computing (STOC '04). ACM, Chicago (2004)
2. Fotakis, D., Kontogiannis, S., Spirakis, P.: Selfish unsplittable flows. J. Theoret. Comput. Sci. **348**, 226–239 (2005)
3. Libman L., Orda A.: Atomic resource sharing in noncooperative networks. Telecommun. Syst. **17**(4), 385–409 (2001)
4. Monderer D., Shapley L.: Potential games. Games Eco. Behav. **14**, 124–143 (1996)
5. Panagopoulou P., Spirakis P.: Algorithms for pure Nash Equilibrium in weighted congestion games. ACM J. Exp. Algorithms **11**, 2.7 (2006)
6. Rosenthal R.W.: A class of games possessing pure-strategy nash equilibria. Int. J. Game Theory **2**, 65–67 (1973)

Self-Stabilization

1974; Dijkstra

Ted Herman
Department of Computer Science, University of Iowa, Iowa City, IA, USA

Keywords and Synonyms

Autopoesis; Homeostasis; Autonomic system control

Problem Definition

An algorithm is self-stabilizing if it eventually manifests correct behavior regardless of initial state. The general problem is to devise self-stabilizing solutions for a specified task. The property of self-stabilization is now known to be feasible for a variety of tasks in distributed computing. Self-stabilization is important for distributed systems and network protocols subject to transient faults. Self-stabilizing systems automatically recover from faults that corrupt state.

The operational interpretation of self-stabilization is depicted in Fig. 1. Part (a) of the figure is an informal presentation of the behavior of a self-stabilizing system, with time on the x-axis and some informal measure of correctness on the y-axis. The curve illustrates a system trajectory, through a sequence of states, during execution. At the initial state, the system state is incorrect; later, the system enters a correct state, then returns to an incorrect state, and subsequently stabilizes to an indefinite period where all states are correct. This period of stability is disrupted by a transient fault that moves the system to an incorrect state, after which the scenario above repeats. Part (b) of the figure illustrates the scenario in terms of state predicates. The box represents the predicate *true*, which characterizes all possible states. Predicate C characterizes the correct states of the system, and $\mathcal{L} \subset C$ depicts the closed *legitimacy* predicate. Reaching a state in $\mathcal{L}$ corresponds to entering a period of stability in part (a). Given an algorithm A with this type of behavior, it is said that A self-stabilizes to $\mathcal{L}$; when $\mathcal{L}$ is implicitly understood, the statement is simplified to: A is self-stabilizing.

Problem [3]. The first setting for self-stabilization posed by Dijkstra is a ring of n processes numbered 0 through $n-1$. Let the state of process i be denoted by $x[i]$. Communication is unidirectional in the ring using a *shared state* model. An atomic step of process i can be expressed by a guarded assignment of the form $g(x[i \ominus 1], x[i]) \rightarrow x[i] := f(x[i \ominus 1], x[i])$. Here, $\ominus$ is subtraction modulo n, so that $x[i \ominus 1]$ is the state of the previous process in the ring with respect to process i. The guard g is a boolean expression; if $g(x[i \ominus 1], x[i])$ is *true*, then process i is said to be *privileged* (or enabled). Thus in one atomic step, privileged process i reads the state of the previous process and computes a new state. Execution scheduling is controlled by a *central daemon*, which fairly chooses one among all enabled processes to take the next step. The problem is to devise g and f so that, regardless of initial states of $x[i]$, $0 \leq i < n$, eventually there is one privilege and every process enjoys a privilege infinitely often.

Complexity Metrics

The complexity of self-stabilization is evaluated by measuring the resource needed for convergence from an arbitrary initial state. Most prominent in the literature of self-stabilization are metrics for worst-case time of convergence and space required by an algorithm solving the given task. Additionally, for reactive self-stabilizing algorithms, metrics are evaluated for the stable behavior of the algorithm, that is, starting from a legitimate state, and compared to non-stabilizing algorithms, to measure costs of self-stabilization.

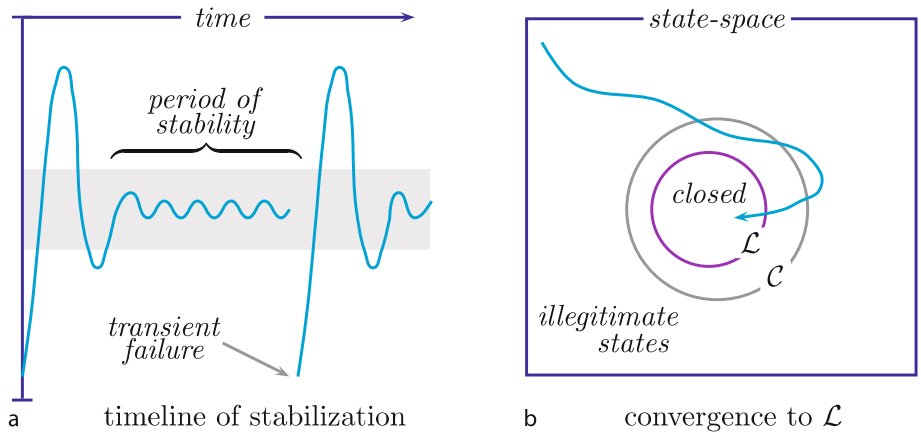
Key Results

Composition

Many self-stabilizing protocols have a layered construction. Let $\{ A_i \}_{i=0}^{m-1}$ be a set programs with the property that for every state variable x, if program A_i writes x, then no program A_j, for $j > i$, writes x. Programs in $\{ A_j \}_{j=i+1}^{m-1}$ may read variables written by A_i, that is, they use the output of A_i as input. Fair composition of programs B and C, written $B \,[]\, C$, assumes fair scheduling of steps of B and C. Let X_j be the set of variables read by A_j and possibly written by $\{ A_i \}_{i=0}^{j-1}$.

Theorem 1 (Fair Composition [4]) *Suppose A_i is self-stabilizing to $\mathcal{L}_i$ under the assumption that all variables in X_i remain constant throughout any execution; then $A_0 \,[]\, A_1 \,[]\, \cdots \,[]\, A_{m-1}$ self-stabilizes to $\{ \mathcal{L}_i \}_{i=0}^{m-1}$.*

Fair composition with a layered set $\{ A_i \}_{i=0}^{m-1}$ corresponds to sequential composition of phases in a distributed algorithm. For instance, let B be a self-stabilizing algorithm for mutual exclusion in a network that assumes the existence of a rooted, spanning tree and let algorithm C be a self-stabilizing algorithm to construct a rooted spanning tree in a connected network; then $B \,[]\, C$ is a self-stabilizing mutual exclusion algorithm for a connected network.



Self-Stabilization, Figure 1
Self-stabilization trajectories

Synchronization Tasks

One question related to the problem posed in Sect. "Problem Definition" is whether or not there can be a *uniform* solution, where all processes have identical algorithms. Dijkstra's result for the unidirectional ring is a semi-uniform solution (all but one process have the same algorithm), using n states per process. The state of each process is a counter: process 0 increments the counter modulo k, where $k \geq n$ suffices for convergence; the other processes copy the counter of the preceding process in the ring. At a legitimate state, each time process 0 increments the counter, the resulting value is different from all other counters in the ring. This ring algorithm turns out to be self-stabilizing for the *distributed daemon* (any subset of privileged processes may execute in parallel) when $k > n$. Subsequent results have established that mutual exclusion on a unidirection ring is $\Theta(1)$ space per process with a non-uniform solution. Deterministic uniform solutions to this task are generally impossible, with the exceptional case where n is and prime. Randomized uniform solutions are known for arbitrary n, using $O(\lg \alpha)$ space where α is the smallest number that does not divide n. Some lower bounds on space for uniform solutions are derived in [7]. Time complexity of Dijkstra's algorithm is $O(n^2)$ rounds, and some randomized solutions have been shown to have expected $O(n^2)$ convergence time.

Dijkstra also presented a solution to mutual exclusion for a linear array of processes, using $O(1)$ space per process [3]. This result was later generalized to a rooted tree of processes, but with mutual exclusion relaxed to having one privilege along any path from root to leaf. Subsequent research built on this theme, showing how tasks for distributed wave computations have self-stabilizing solutions. Tasks of phase synchronization and clock synchronization have also been solved. See reference [9] for an example of self-stabilizing mutual exclusion in a multiprocessor shared memory model.

Graph Algorithms

Communication networks are commonly represented with graph models and the need for distributed graph algorithms that tolerate transient faults motivates study of such tasks. Specific results in this area include self-stabilizing algorithms for spanning trees, center-finding, matching, planarity testing, coloring, finding independent sets, and so forth. Generally, all graph tasks can be solved by self-stabilizing algorithms: tasks that have network topology and possibly related factors, such as edge weights, for input, and define outputs to be a function of the inputs, can be solved by general methods for self-stabilization. These general methods require considerable space and time resource, and may also use stronger model assumptions than needed for specific tasks, for instance unique process identifiers and an assumed bound on network diameter. Therefore research continues on graph algorithms.

One discovery emerging from research on self-stabilizing graph algorithms is the difference between algorithms that terminate and those that continuously change state, even after outputs are stable. Consider the task of constructing a spanning tree rooted at process r. Some algorithms self-stabilize to the property that, for every $p \neq r$, the variable u_p refers to p's parent in the spanning tree and the state remains unchanged. Other algo-

rithms are self-stabilizing protocols for token circulation with the side-effect that the circulation route of the token establishes a spanning tree. The former type of algorithm has $O(\lg n)$ space per process, whereas the latter has $O(\lg \delta)$ where δ is the degree (number of neighbors) of a process. This difference was formalized in the notion of *silent* algorithms, which eventually stop changing any communication value; it was shown in [5] for the link register model that silent algorithms for many graph tasks have $\Omega(\lg n)$ space.

Transformation

The simple presentation of [3] is enabled by the abstract computation model, which hides details of communication, program control, and atomicity. Self-stabilization becomes more complicated when considering conventional architectures that have messages, buffers, and program counters. A natural question is how to transform or refine self-stabilizing algorithms expressed in abstract models to concrete models closer to practice. As an example, consider the problem of transforming algorithms written for the central daemon to the distributed daemon model. This transformation can be reduced to finding a self-stabilizing token-passing algorithm for the distributed daemon model such that, eventually, no two neighboring processes concurrently have a token; multiple tokens can increase the efficiency of the transformation.

General Methods

The general problem of constructing a self-stabilizing algorithm for an input nonreactive task can be solved using standard tools of distributed computing: snapshot, broadcast, system reset, and synchronization tasks are building blocks so that the global state can be continuously validated (in some fortunate cases $\mathcal{L}$ can be locally checked and corrected). These building blocks have self-stabilizing solutions, enabling the general approach.

Fault Tolerance

The connection between self-stabilization and transient faults is implicit in the definition. Self-stabilization is also applicable in executions that asynchronously change inputs, silently crash and restart, and perturb communication [10]. One objection to the mechanism of self-stabilization, particularly when general methods are applied, is that a small transient fault can lead to a system-wide correction. This problem has been investigated, for example in [8], where it is shown how convergence can be optimized for a limited number of faults. Self-stabilization has also been combined with other types of failure tolerance, though this is not always possible: the task of counting the number of processes in a ring has no self-stabilizing solution in the shared state model if a process may crash [1], unless a failure detector is provided.

Applications

Many network protocols are self-stabilizing by the following simple strategy: periodically, they discard current data and regenerate it from trusted information sources. This idea does not work in purely asynchronous systems; the availability of real-time clocks enables the simple strategy. Similarly, watchdogs with hardware clocks can provide an effective basis for self-stabilization [6].

Cross References

► Concurrent Programming, Mutual Exclusion

Recommended Reading

1. Anagnostou, E., Hadzilacos, V.: Tolerating Transient and Permanent Failures. In: Distributed Algorithms 7th International Workshop. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
2. Cournier, A., Datta, A.K., Petit, F., Villain, V.: Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In: Proceedings of the 22nd International Conference Distributed Computing Systems, pp. 199–206, Vienna, July 2002
3. Dijkstra, E.W.: Self Stabilizing Systems in Spite of Distributed Control. Commun. ACM **17**(11), 643–644 (1974). See also EWD391 (1973) In: Selected Writings on Computing: A Personal Perspective, pp. 41–46. Springer, New York (1982)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambrigde (2000)
5. Dolev, S., Gouda, M.G., Schneider, M.: Memory Requirements for Silent Stabilization. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, pp. 27–34, Philadelphia, May 1996
6. Dolev, S., Yagel, R.: Toward Self-Stabilizing Operating Systems. In: 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems, pp. 684–688, Zaragoza, August 2004
7. Israeli, A., Jalfon, M.: Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion. In: Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, pp. 119–131, Quebec City, August 1990
8. Kutten, S., Patt-Shamir, B.: Time-Adaptive Self Stabilization. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, pp. 149–158, Santa Barbara, August 1997
9. Lamport, L.: The Mutual Exclusion Problem: Part II-Statement and Solutions. J. ACM **33**(2), 327–348 (1986)
10. Varghese, G., Jayaram, M.: The Fault Span of Crash Failures. J. ACM **47**(2), 244–293 (2000)

Separators in Graphs

1998; Leighton, Rao
1999; Leighton, Rao

Goran Konjevod
Department of Computer Science and Engineering, Arizona State University, Tempe, AZ, USA

Keywords and Synonyms

Balanced cuts

Problem Definition

The (balanced) separator problem asks for a cut of minimum (edge)-weight in a graph, such that the two shores of the cut have approximately equal (node)-weight.

Formally, given an undirected graph $G = (V, E)$, with a nonnegative edge-weight function $c : E \to \mathbb{R}_+$, a nonnegative node-weight function $\pi : V \to \mathbb{R}_+$, and a constant $b \leq 1/2$, a cut $(S : V \setminus S)$ is said to be *b-balanced*, or a $(b, 1-b)$-*separator*, if $b\pi(V) \leq \pi(S) \leq (1-b)\pi(V)$ (where $\pi(S)$ stands for $\sum_{v \in S} \pi(v)$).

Problem 1 (*b*-balanced separator)
Input: *Edge- and node-weighted graph* $G = (V, E, c, \pi)$, *constant* $b \leq 1/2$.
Output: *A b-balanced cut* $(S : V \setminus S)$. *Goal: minimize the edge weight* $c(\delta(S))$.

Closely related is the *product sparsest cut problem*.

Problem 2 ((Product) Sparsest cut)
Input: *Edge- and node-weighted graph* $G = (V, E, c, \pi)$.
Output: *A cut* $(S : V \setminus S)$ *minimizing the ratio-cost* $\frac{(}{c}(\delta(S)))/(\pi(S)\pi(V \setminus S))$.

Problem 2 is the most general version of sparsest cut solved by Leighton and Rao. Setting all node weights are equal to 1 leads to the uniform version, Problem 3.

Problem 3 ((Uniform) Sparsest cut)
Input: *Edge-weighted graph* $G = (V, E, c)$.
Output: *A cut* $(S : V \setminus S)$ *minimizing the ratio-cost* $(c(\delta(S)))/(|S||V \setminus S|)$.

Sparsest cut arises as the (integral version of the) linear programming dual of *concurrent multicommodity flow* (Problem 4). An instance of a multicommodity flow problem is defined on an edge-weighted graph by specifying for each of k *commodities* a *source* $s_i \in V$, a *sink* $t_i \in V$, and a *demand* D_i. A feasible solution to the multicommodity flow problem defines for each commodity a flow function on E, thus routing a certain amount of flow from s_i to t_i. The edge weights represent capacities, and for each edge e, a capacity constraint is enforced: the sum of all commodities' flows through e is at most the capacity $c(e)$.

Problem 4 (Concurrent multicommodity flow)
Input: *Edge-weighted graph* $G = (V, E, c)$, *commodities* $(s_1, t_1, D_1), \ldots (s_k, t_k, D_k)$.
Output: *A multicommodity flow that routes* fD_i *units of commodity i from* s_i *to* t_i *for each i simultaneously, without violating the capacity of any edge. Goal: maximize f*.

Problem 4 can be solved in polynomial time by linear programming, and approximated arbitrarily well by several more efficient combinatorial algorithms (Sect. "Implementation"). The maximum value f for which there exists a multicommodity flow is called the *max-flow* of the instance. The *min-cut* is the minimum ratio $(c(\delta(S)))/(D(S, V \setminus S))$, where $D(S, V \setminus S) = \sum_{i:|\{s_i,t_i\} \cap S|=1} D_i$. This dual interpretation motivates the most general version of the problem, the *nonuniform sparsest cut* (Problem 5).

Problem 5 ((Nonuniform) Sparsest cut) Input: *Edge-weighted graph* $G = (V, E, c)$, *commodities* $(s_1, t_1, D_1), \ldots (s_k, t_k, D_k)$.
Output: *A min-cut* $(S : V \setminus S)$, *that is, a cut of minimum ratio-cost* $(c(\delta(S)))/(D(S, V \setminus S))$.

(Most literature focuses on either the uniform or the general nonuniform version, and both of these two versions are sometimes referred to as just the "sparsest cut" problem.)

Key Results

Even when all (edge- and node-) weights are equal to 1, finding a minimum-weight b-balanced cut is NP-hard (for $b = 1/2$, the problem becomes *graph bisection*). Leighton and Rao [23,24] give a pseudo-approximation algorithm for the general problem.

Theorem 1 *There is a polynomial-time algorithm that, given a weighted graph* $G = (V, E, c, \pi)$, $b \leq 1/2$ *and* $b' < \min\{b, 1/3\}$, *finds a* b'*-balanced cut of weight* $O((\log n)/(b - b'))$ *times the weight of the minimum b-balanced cut.*

The algorithm solves the sparsest cut problem on the given graph, puts aside the smaller-weight shore of the cut, and recurses on the larger-weight shore until both shores of the sparsest cut found have weight at most $(1 - b')\pi(G)$. Now the larger-weight shore of the last iteration's sparsest cut is returned as one shore of the balanced cut, and everything else as the other shore. Since the sparsest cut problem is

itself NP-hard, Leighton and Rao first required an approximation algorithm for this problem.

Theorem 2 *There is a polynomial-time algorithm with approximation ratio $O(\log p)$ for product sparsest cut (Problem 2), where p denotes the number of nonzero-weight nodes in the graph.*

This algorithm follows immediately from Theorem 3.

Theorem 3 *There is a polynomial-time algorithm that finds a cut $(S : V \setminus S)$ with ratio-cost $(c(\delta(S)))/(\pi(S)\pi(V \setminus S)) \in O(f \log p)$, where f is the max-flow for the product multicommodity flow and p the number of nodes with nonzero weight.*

The proof of Theorem 3 is based on solving a linear programming formulation of the multicommodity flow problem and using the solution to construct a sparse cut.

Related Results

Shahrokhi and Matula [27] gave a max-flow min-cut theorem for a special case of the multicommodity flow problem and used a similar LP-based approach to prove their result. An $O(\log n)$ upper bound for arbitrary demands was proved by Aumann and Rabani [6] and Linial et al. [26]. In both cases, the solution to the dual of the multicommodity flow linear program is interpreted as a finite metric and embedded into ℓ_1 with distortion $O(\log n)$, using an embedding due to Bourgain [10]. The resulting ℓ_1 metric is a convex combination of cut metrics, from which a cut can be extracted with sparsity ratio at least as good as that of the combination.

Arora et al. [5] gave an $O(\sqrt{\log n})$ pseudo-approximation algorithm for (uniform or product-weight) balanced separators, based on a semidefinite programming relaxation. For the nonuniform version, the best bound is $O(\sqrt{\log n} \log\log n)$ due to Arora et al. [4]. Khot and Vishnoi [18] showed that, for the nonuniform version of the problem, the semidefinite relaxation of [5] has an integrality gap of at least $(\log\log n)^{1/6-\delta}$ for any $\delta > 0$, and further, assuming their Unique Games Conjecture, that it is NP-hard to (pseudo)-approximate the balanced separator problem to within any constant factor. The SDP integrality gap was strengthened to $\Omega(\log\log n)$ by Krauthgamer and Rabani [20]. Devanur et al. [11] show an $\Omega(\log\log n)$ integrality gap for the SDP formulation even in the uniform case.

Implementation

The bottleneck in the balanced separator algorithm is solving the multicommodity flow linear program. There exists a substantial amount of work on fast approximate solutions to such linear programs [19,22,25]. In most of the following results, the algorithm produces a $(1+\epsilon)$-approximation, and its hidden constant depends on ϵ^{-2}. Garg and Könemann [15], Fleischer [14] and Karakostas [16] gave efficient approximation schemes for multicommodity flow and related problems, with running times $\tilde{O}((k+m)m)$ [15] and $\tilde{O}(m^2)$ [14,16]. Benczúr and Karger [7] gave an $O(\log n)$ approximation to sparsest cut based on randomized minimum cut and running in time $\tilde{O}(n^2)$. The current fastest $O(\log n)$ sparsest cut (balanced separator) approximation is based on a primal-dual approach to semidefinite programming due to Arora and Kale [3], and runs in time $O(m + n^{3/2})(\tilde{O}(m + n^{3/2})$, respectively). The same paper gives an $O(\sqrt{\log n})$ approximation in time $O(n^2)(\tilde{O}(n^2)$, respectively), improving on a previous $\tilde{O}(n^2)$ algorithm of Arora et al. [2]. If an $O(\log^2 n)$ approximation is sufficient, then sparsest cut can be solved in time $\tilde{O}(n^{3/2})$, and balanced separator in time $\tilde{O}(m + n^{3/2})$ [17].

Applications

Many problems can be solved by using a balanced separator or sparsest cut algorithm as a subroutine. The approximation ratio of the resulting algorithm typically depends directly on the ratio of the underlying subroutine. In most cases, the graph is recursively split into pieces of balanced size. In addition to the $O(\log n)$ approximation factor required by the balanced separator algorithm, this leads to another $O(\log n)$ factor due to the recursion depth. Even et al. [12] improved many results based on balanced separators by using *spreading metrics*, reducing the approximation guarantee to $O(\log n \log\log n)$ from $O(\log^2 n)$.

Some applications are listed here; where no reference is given, and for further examples, see [24].

- Minimum cut linear arrangement and minimum feedback arc set. One single algorithm provides an $O(\log^2 n)$ approximation for both of these problems.
- Minimum chordal graph completion and elimination orderings [1]. Elimination orderings are useful for solving sparse symmetric linear systems. The $O(\log^2 n)$ approximation algorithm of [1] for chordal graph completion has been improved to $O(\log n \log\log n)$ by Even et al. [12].
- Balanced node cuts. The cost of a balanced cut may be measured in terms of the weight of nodes removed from the graph. The balanced separator algorithm can be easily extended to this node-weighted case.
- VLSI layout. Bhatt and Leighton [8] studied several optimization problems in VLSI layout. Recursive par-

titioning by a balanced separator algorithm leads to polylogarithmic approximation algorithms for crossing number, minimum layout area and other problems.

- Treewidth and pathwidth. Bodlaender et al. [9] showed how to approximate treewidth within $O(\log n)$ and pathwidth within $O(\log^2 n)$ by using balanced node separators.
- Bisection. Feige and Krauthgamer [13] gave an $O(\alpha \log n)$ approximation for the minimum bisection, using any α-approximation algorithm for sparsest cut.

Experimental Results

Lang and Rao [21] compared a variant of the sparsest cut algorithm from [24] to methods used in graph decomposition for VLSI design.

Cross References

- ► Fractional Packing and Covering Problems
- ► Minimum Bisection
- ► Sparsest Cut

Recommended Reading

Further details and pointers to additional results may be found in the survey [28].

1. Agrawal, A., Klein, P.N., Ravi, R.: Cutting down on fill using nested dissection: provably good elimination orderings. In: Brualdi, R.A., Friedland, S., Klee, V. (eds.) Graph theory and sparse matrix computation. IMA Volumes in mathematics and its applications, pp. 31–55. Springer, New York (1993)
2. Arora, S., Hazan, E., Kale, S.: $O(\sqrt{\log n})$ approximation to sparsest cut in $\tilde{O}(n^2)$ time. In: FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04), pp. 238–247. IEEE Computer Society, Washington (2004)
3. Arora, S., Kale, S.: A combinatorial, primal-dual approach to semidefinite programs. In: STOC '07: Proceedings of the 39th Annual ACM Symposium on Theory of Computing, pp. 227–236. ACM (2007)
4. Arora, S., Lee, J.R., Naor, A.: Euclidean distortion and the sparsest cut. In: STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pp. 553–562. ACM Press, New York (2005)
5. Arora, S., Rao, S., Vazirani, U.: Expander flows, geometric embeddings and graph partitioning. In: STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, pp. 222–231. ACM Press, New York (2004)
6. Aumann, Y., Rabani, Y.: An (log) approximate min-cut max-flow theorem and approximation algorithm. SIAM J. Comput. **27**(1), 291–301 (1998)
7. Benczúr, A.A., Karger, D.R.: Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In: STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pp. 47–55. ACM Press, New York (1996)
8. Bhatt, S.N., Leighton, F.T.: A framework for solving vlsi graph layout problems. J. Comput. Syst. Sci. **28**(2), 300–343 (1984)
9. Bodlaender, H.L., Gilbert, J.R., Hafsteinsson, H., Kloks, T.: Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. J. Algorithms **18**(2), 238–255 (1995)
10. Bourgain, J.: On Lipshitz embedding of finite metric spaces in Hilbert space. Israel J. Math. **52**, 46–52 (1985)
11. Devanur, N.R., Khot, S.A., Saket, R., Vishnoi, N.K.: Integrality gaps for sparsest cut and minimum linear arrangement problems. In: STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, pp. 537–546. ACM Press, New York (2006)
12. Even, G., Naor, J.S., Rao, S., Schieber, B.: Divide-and-conquer approximation algorithms via spreading metrics. J. ACM **47**(4), 585–616 (2000)
13. Feige, U., Krauthgamer, R.: A polylogarithmic approximation of the minimum bisection. SIAM J. Comput. **31**(4), 1090–1118 (2002)
14. Fleischer, L.: Approximating fractional multicommodity flow independent of the number of commodities. SIAM J. Discret. Math. **13**(4), 505–520 (2000)
15. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In: FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, p. 300. IEEE Computer Society, Washington (1998)
16. Karakostas, G.: Faster approximation schemes for fractional multicommodity flow problems. In: SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 166–173. Society for Industrial and Applied Mathematics, Philadelphia (2002)
17. Khandekar, R., Rao, S., Vazirani, U.: Graph partitioning using single commodity flows. In: STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, pp. 385–390. ACM Press, New York (2006)
18. Khot, S., Vishnoi, N.K.: The unique games conjecture, integrality gap for cut problems and embeddability of negative type metrics into l_1. In: FOCS '07: Proceedings of the 46th Annual IEEE Symposium on Foundations and Computer Science, pp. 53–62. IEEE Computer Society (2005)
19. Klein, P.N., Plotkin, S.A., Stein, C., Tardos, É.: Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. SIAM J. Comput. **23**(3), 466–487 (1994)
20. Krauthgamer, R., Rabani, Y.: Improved lower bounds for embeddings into l_1. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 1010–1017. ACM Press, New York (2006)
21. Lang, K., Rao, S.: Finding near-optimal cuts: an empirical evaluation. In: SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, pp. 212–221. Society for Industrial and Applied Mathematics, Philadelphia (1993)
22. Leighton, F.T., Makedon, F., Plotkin, S.A., Stein, C., Stein, É., Tragoudas, S.: Fast approximation algorithms for multicommodity flow problems. J. Comput. Syst. Sci. **50**(2), 228–243 (1995)
23. Leighton, T., Rao, S.: An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science, pp. 422–431, IEEE Computer Society (1988)

24. Leighton, T., Rao, S.: Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. J. ACM **46**(6), 787–832 (1999)
25. Leong, T., Shor, P., Stein, C.: Implementation of a combinatorial multicommodity flow algorithm. In: Johnson, D.S., McGeoch, C.C. (eds.) Network flows and matching. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, pp. 387–406. AMS, Providence (1991)
26. Linial, N., London, E., Rabinovich, Y.: The geometry of graphs and some of its algorithmic applications. Comb. **15**(2), 215–245 (1995)
27. Shahrokhi, F., Matula, D.W.: The maximum concurrent flow problem. J. ACM **37**(2), 318–334 (1990)
28. Shmoys, D.B.: Cut problems and their applications to divide-and-conquer. In: Hochbaum, D.S. (ed.) Approximation algorithms for NP-hard problems, pp. 192–235. PWS Publishing Company, Boston, MA (1997)

Sequential Approximate String Matching

2003; Crochemore, Landau, Ziv-Ukelson 2004; Fredriksson, Navarro

Gonzalo Navarro
Department of Computer Science, University of Chile, Santiago, Chile

Keywords and Synonyms

String matching allowing errors or differences; Inexact string matching; Semiglobal or semilocal sequence similarity

Problem Definition

Given a *text string* $T = t_1 t_2 \ldots t_n$ and a *pattern string* $P = p_1 p_2 \ldots p_m$, both being sequences over an alphabet Σ of size σ, and given a *distance function* among strings d and a *threshold* k, the *approximate string matching (ASM)* problem is to find all the text positions that finish a so-called *approximate occurrence* of P in T, that is, compute the set $\{j, \exists i, 1 \leq i \leq j, d(P, t_i \ldots t_j) \leq k\}$. In the sequential version of the problem T, P, and k are given together, whereas the algorithm can be tailored for a specific d.

The solutions to the problem vary widely depending on the distance d used. This entry focuses on a very popular one, called *Levenshtein distance* or *edit distance*, defined as the minimum number of character insertions, deletions, and substitutions necessary to convert one string into the other. It will also pay some attention to other common variants such as *indel distance*, where only insertions and deletions are permitted and is the dual of the *longest common subsequence lcs* ($d(A, B) = |A| + |B| - 2 \cdot lcs(A, B)$); and *Hamming distance*, where only substitutions are permitted.

A popular generalization of all the above is the *weighted edit distance*, where the operations are given positive real-valued weights and the distance is the minimum sum of weights of a sequence of operations converting one string into the other. The weight of deleting a character c is written $w(c \to \epsilon)$, that of inserting c is written $w(\epsilon \to c)$, and that of substituting c by $c' \neq c$ is written $w(c \to c')$. It is assumed $w(c \to c) = 0$ and the triangle inequality, that is, $w(x \to y) + w(y \to z) \geq w(x \to z)$ for any $x, y, z \in \Sigma \cup \{\epsilon\}$. As the distance may now be asymmetric, it is fixed that $d(A, B)$ is the cost of converting A into B. Of course any result for weighted edit distance applies to edit, Hamming and indel distances (collectively termed *unit-cost edit distances*) as well, but other reductions are not immediate.

Both worst- and average-case complexity are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from Σ. For simplicity and practicality, $m = o(n)$ is assumed in this entry.

Key Results

The most ancient and versatile solution to the problem [13] builds over the process of computing weighted edit distance. Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two strings. Let $C[0 \ldots m, 0 \ldots n]$ be a matrix such that $C[i, j] = d(a_1 \ldots a_i, b_1 \ldots b_j)$. Then it holds $C[0, 0] = 0$ and

$$C[i, j] = \min(C[i-1, j] + w(a_i \to \epsilon), C[i, j-1] + w(\epsilon \to b_j), C[i-1, j-1] + w(a_i \to b_j)),$$

where $C[i, -1] = C[-1, j] = \infty$ is assumed. This matrix is computed in $O(mn)$ time and $d(A, B) = C[m, n]$. In order to solve the approximate string matching problem, one takes $A = P$ and $B = T$, and sets $C[0, j] = 0$ for all j, so that the above formula is used only for $i > 0$.

Theorem 1 (Sellers 1980 [13]) *There exists an $O(mn)$ worst-case time solution to the ASM problem under weighted edit distance.*

The space is $O(m)$ if one realizes that C can be computed column-wise and only column $j - 1$ is necessary to compute column j. As explained, this immediately implies that searching under unit-cost edit distances can be done in $O(mn)$ time as well. In those cases, it is quite easy to com-

pute only part of matrix C so as to achieve $O(kn)$ average-time algorithms [14].

Yet, there exist algorithms with lower worst-case complexity for weighted edit distance. By applying a Ziv-Lempel parsing to P and T, it is possible to identify regions of matrix C corresponding to substrings of P and T that can be computed from other previous regions corresponding to similar substrings of P and T [5].

Theorem 2 (Crochemore et al. 2003 [5]) *There exists an $O(n + mn/\log_\sigma n)$ worst-case time solution to the ASM problem under weighted edit distance. Moreover, the time is $O(n + mnh/\log n)$, where $0 \le h \le \log\sigma$ is the entropy of T.*

This result is very general, also holding for computing weighted edit distance and local similarity (see section on applications). For the case of edit distance and exploiting the unit-cost RAM model, it is possible to do better. On one hand, one can apply a four-Russian technique: All the possible blocks (submatrices of C) of size $t \times t$, for $t = O(\log_\sigma n)$, are precomputed and matrix C is computed block-wise [9]. On the other hand, one can represent each cell in matrix C using a constant number of bits (as it can differ from neighboring cells by ± 1) so as to store and process several cells at once in a single machine word [10]. This latter technique is called *bit-parallelism* and assumes a machine word of $\Theta(\log n)$ bits.

Theorem 3 (Masek and Paterson 1980 [9]; Myers 1999 [10]) *There exist $O(n + mn/(\log_\sigma n)^2)$ and $O(n + mn/\log n)$ worst-case time solutions to the ASM problem under edit distance.*

Both complexities are retained for indel distance, yet not for Hamming distance.

For unit-cost edit distances, the complexity can depend on k rather than on m, as $k < m$ for the problem to be nontrivial and usually k is a small fraction of m (or even $k = o(m)$). A classic technique [8] computes matrix C by processing in constant time diagonals $C[i + d, j + d]$, $0 \le d \le s$, along which cell values do not change. This is possible by preprocessing the suffix trees of T and P for Lowest Common Ancestor queries.

Theorem 4 (Landau and Vishkin 1989 [8]) *There exists an O(kn) worst-case time solution to the ASM problem under unit-cost edit distances.*

Other solutions exist which are better for small k, achieving time $O(n(1 + k^4/m))$ [4]. For the case of Hamming distance, one can achieve improved results using convolutions [1].

Theorem 5 (Amir et al. 2004 [1]) *There exist $O(n\sqrt{k \log k})$ and $O(n(1 + k^3/m)\log k)$ worst-case time solution to the ASM problem under Hamming distance.*

The last result for edit distance [4] achieves $O(n)$ time if k is small enough ($k = O(m^{1/4})$). It is also possible to achieve $O(n)$ time on unit-cost edit distances at the expense of an exponential additive term on m or k: The number of different columns in C is independent of n, so the transition from every possible column to the next can be precomputed as a finite-state machine.

Theorem 6 (Ukkonen 1985 [14]) *There exists an $O(n + m\min(3^m, m(2m\sigma)^k))$ worst-case time solution to the ASM problem under edit distance.*

Similar results apply for Hamming and indel distance, where the exponential term reduces slightly according to the particularities of the distances.

The worst-case complexity of the ASM problem is of course $\Omega(n)$, but it is not known if this can be attained for any m and k. Yet, the average-case complexity of the problem is known.

Theorem 7 (Chang and Marr 1994 [3]) *The average-case complexity of the ASM problem is $\Theta(n(k + \log_\sigma m)/m)$ under unit-cost edit distances.*

It is not hard to prove the lower bound as an extension to Yao's bound for exact string matching [15]. The lower bound was reached in the same paper [3], for $k/m < 1/3 - O(1/\sqrt{\sigma})$. This was improved later to $k/m < 1/2 - O(1/\sqrt{\sigma})$ [6] using a slightly different idea. The approach is to precompute the minimum distance to match every possible text substring (block) of length $O(\log_\sigma m)$ inside P. Then, a text window is scanned backwards, block-wise, adding up those minimum precomputed distances. If they exceed k before scanning all the window, then no occurrence of P with k errors can contain the scanned blocks and the window can be safely slid over the scanned blocks, advancing in T. This is an example of a *filtration* algorithm, which discards most text areas and applies an ASM algorithm only over those areas that cannot be discarded.

Theorem 8 (Fredriksson and Navarro 2004 [6]) *There exists an optimal-on-average solution to the ASM problem under edit distance, for any $k/m \le \frac{1-e/\sqrt{\sigma}}{2-e/\sqrt{\sigma}} = 1/2 - O(1/\sqrt{\sigma})$.*

The result applies verbatim to indel distance. The same complexity is achieved for Hamming distance, yet the limit on k/m improves to $1 - 1/\sigma$. Note that, when the limit k/m is reached, the average complexity is already $\Theta(n)$. It

is not clear up to which k/m limit could one achieve linear time on average.

Applications

The problem has many applications in computational biology (to compare DNA and protein sequences, recovering from experimental errors, so as to spot mutations or predict similarity of structure or function), text retrieval (to recover from spelling, typing or automatic recognition errors), signal processing (to recover from transmission and distortion errors), and several others. See [11] for a more detailed discussion.

Many extensions of the ASM problem exist, particularly in computational biology. For example, it is possible to substitute whole substrings by others (called *generalized edit distance*), swap characters in the strings (*string matching with swaps or transpositions*), reverse substrings (*reversal distance*), have variable costs for insertions/deletions when they are grouped (*similarity with gap penalties*), and look for any pair of substrings of both strings that are sufficiently similar (*local similarity*). See for example Gusfield's book [7], where many related problems are discussed.

Open Problems

The worst-case complexity of the problem is not fully understood. For unit-cost edit distances it is $\Theta(n)$ if $m = O(\min(\log n, (\log_\sigma n)^2))$ or $k = O(\min(m^{1/4}, \log_{m\sigma} n))$. For weighted edit distance the complexity is $\Theta(n)$ if $m = O(\log_\sigma n)$. It is also unknown up to which k/m value can one achieve $O(n)$ average time; up to now this has been achieved up to $k/m = 1/2 - O(1/\sqrt{\sigma})$.

Experimental Results

A thorough survey on the subject [11] presents extensive experiments. Nowadays, the fastest algorithms for edit distance are in practice filtration algorithms [6,12] combined with bit-parallel algorithms to verify the candidate areas [2,10]. Those filtration algorithms work well for small enough k/m, otherwise the bit-parallel algorithms should be used stand-alone. Filtration algorithms are easily extended to handle multiple patterns searched simultaneously.

URL to Code

Well-known packages offering efficient ASM are *agrep* (http://webglimpse.net/download.html, top-level subdirectory `agrep/`) and *nrgrep* (http://www.dcc.uchile.cl/~gnavarro/software).

Cross References

- Approximate Regular Expression Matching is the more complex case where P can be a regular expression;
- Indexed Approximate String Matching refers to the case where the text can be preprocessed;
- Local Alignment (with Concave Gap Weights) refers to a more complex weighting scheme of interest in computational biology.
- Sequential Exact String Matching is the simplified version where no errors are permitted;

Recommended Reading

1. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms **50**(2), 257–275 (2004)
2. Baeza-Yates, R., Navarro, G.: Faster approximate string matching. Algorithmica **23**(2), 127–158 (1999)
3. Chang, W., Marr, T.: Approximate string matching and local similarity. In: Proc. 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94). LNCS, vol. 807, pp. 259–273. Springer, Berlin, Germany (1994)
4. Cole, R., Hariharan, R.: Approximate string matching: A simpler faster algorithm. SIAM J. Comput. **31**(6), 1761–1782 (2002)
5. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. **32**(6), 1654–1673 (2003)
6. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. ACM J. Exp. Algorithms **9**(1.4) (2004)
7. Gusfield, D.: Algorithms on strings, trees and sequences. Cambridge University Press, Cambridge (1997)
8. Landau, G., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms **10**, 157–169 (1989)
9. Masek, W., Paterson, M.: A faster algorithm for computing string edit distances. J. Comput. Syst. Sci. **20**, 18–31 (1980)
10. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic progamming. J. ACM **46**(3), 395–415 (1999)
11. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
12. Navarro, G., Baeza-Yates, R.: Very fast and simple approximate string matching. Inf. Proc. Lett. **72**, 65–70 (1999)
13. Sellers, P.: The theory and computation of evolutionary distances: pattern recognition. J. Algorithms **1**, 359–373 (1980)
14. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms **6**, 132–137 (1985)
15. Yao, A.: The complexity of pattern matching for a random string. SIAM J. Comput. **8**, 368–387 (1979)

Sequential Circuit Technology Mapping

1998; Pan, Liu

Peichen Pan
Magma Design Automation, Inc., Los Angeles, CA, USA

Keywords and Synonyms

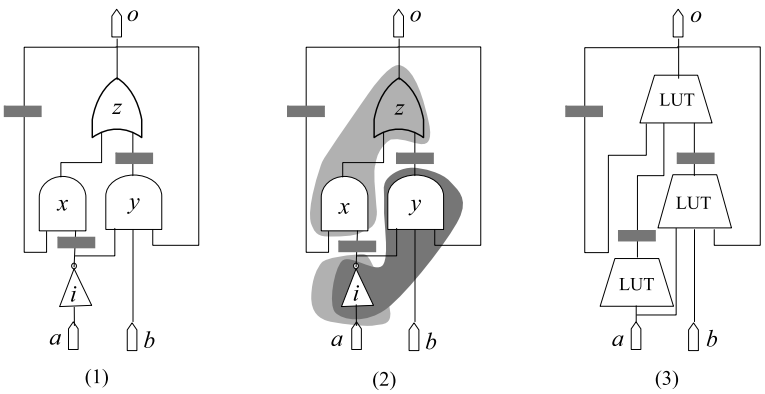
Integrated retiming and technology mapping; Technology mapping with retiming

Problem Definition

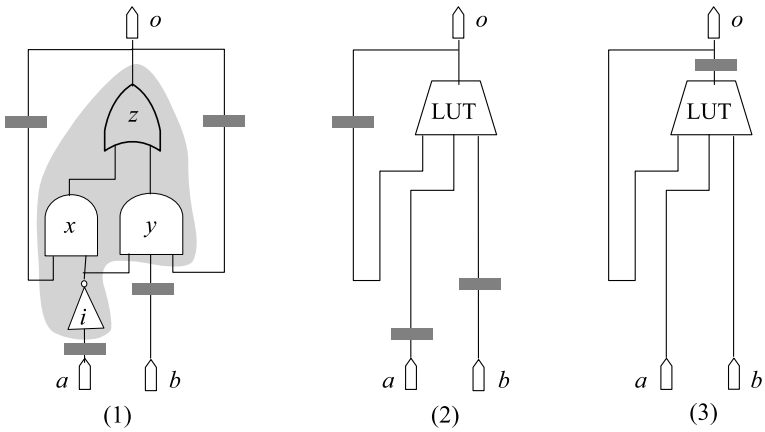
One of the key steps in VLSI design flow is technology mapping which converts a Boolean network of technology-independent logic gates and edge-triggered D-flipflops (FFs) into an equivalent one comprised of cells from a target technology cell library [1,3,5]. Technology mapping can be formulated as a covering problem in where logic gates are covered by cells from the technology library. For ease of discussion, it is assumed that the cell library contains only one cell, a K-input lookup table (K-LUT) with one unit of delay. A K-LUT can realize any Boolean function with up to K inputs as is the case in high performance field-programmable gate arrays (FPGAs).

Figure 1 shows an example of technology mapping. The original network in (1) with three FFs and four gates, is covered by three 3-input cones as indicated in (2). The corresponding mapping solution using 3-LUTs is shown in (3). Note that gate i is covered by two cones. The mapping solution in (3) has a *cycle time* (or *clock period*) of two units, which is the total delay of a longest path between FFs, from primary inputs (PIs) to FFs, and from FFs to primary outputs (POs).

Retiming is a transformation that relocates FFs of a design while preserving its functionality [4]. Retiming can affect technology mapping. Figure 2 (1) shows a design obtained from the one in Fig. 1 (1) by retiming the FFs at the output of y and i to their inputs. It can be covered with just one 3-input cone as indicated in (1). The corresponding mapping solution shown in (2) is better in both timing and area than the functionally-equivalent solution in Fig. 1 (3) obtained without retiming.



Sequential Circuit Technology Mapping, Figure 1
Technology mapping: (1) Original network, (2) covering, (3) mapping solution



Sequential Circuit Technology Mapping, Figure 2
Retiming and mapping: (1) Retiming and covering, (2) mapping solution, (3) retimed solution

FindAllCuts(N, K)
  foreach node v in N **do** $C(v) \Leftarrow \{\{v^0\}\}$
  while (*new cuts discovered*) **do**
    foreach node v in N **do** $C(v) \Leftarrow merge(C(u_1), ..., C(u_t))$

Sequential Circuit Technology Mapping, Figure 3
Cut enumeration procedure

iter	a	b	i	x	y	z	o
0	$\{a^0\}$	$\{b^0\}$	$\{i^0\}$	$\{x^0\}$	$\{y^0\}$	$\{z^0\}$	$\{o^0\}$
1			$\{a^0\}$	$\{i^1, z^1\}$ $\{a^1, z^1\}$	$\{i^0, b^0, z^0\}$ $\{a^0, b^0, z^0\}$	$\{x^0, y^1\}$ $\{i^1, z^1, b^1\}$ $\{a^1, z^1, b^1\}$ $\{i^1, z^1, y^1\}$ $\{a^1, z^1, y^1\}$	$\{z^0\}$
2				$\{i^1, x^1, y^2\}$ $\{a^1, x^1, y^2\}$			

Sequential Circuit Technology Mapping, Figure 4
Cut enumeration example

A *K-bounded* network is one in which each gate has at most K inputs. The sequential circuit technology mapping problem can be defined as follows: *Given a K-bounded Boolean network N and a target cycle time φ , find a mapping solution with a cycle time of φ , assuming FFs can be repositioned using retiming.*

Key Results

The first polynomial time algorithm for the problem was proposed in [8,9]. An improved algorithm was proposed in [2] to reduce runtime. Both algorithms are based on min-cost flow computation.

In [7], another algorithm was proposed to take advantage of the fact that K is a small integer usually between 3 and 6 in practice. The algorithm enumerates all K-input cones for each gate. It can incorporate other optimization objectives (e. g., area and power) and can be apllied to standard cells libraries.

Cut Enumeration

A Boolean network can be represented as an edge-weighted directed graph where the nodes denote logic gates, PIs, and POs. There is a directed edge (u, v) with weight d if u, after going through d FFs, drives v.

A logic cone for a node can be captured by a *cut* consisting of inputs to the cone. An element in a cut for v consists of the driving node u and the total weight d on the paths from u to v, denoted by u^d. If u reaches v on several paths with different FF counts, u will appear in the cut multiple times with different d's. As an example, for the cone for z in Fig. 2 (2), the corresponding cut is $\{z^1, a^1, b^1\}$. A cut of size K is called a K-cut.

Let (u_i, v) be an edge in N with weight d_i, and $C(u_i)$ be a set of K-cuts for u_i, for $i = 1, \ldots, t$. Let $merge(C(u_1), \ldots, C(u_t))$ denote the following set operation:

$$\{\{v^0\}\} \cup \{c_1^{d_1} \cup \ldots \cup c_t^{d_t} | c_1 \in C(u_1), \ldots, c_t \in C(u_t), \; |c_1^{d_1} \cup \ldots \cup c_t^{d_t}| \leq K\}$$

where $c_i^{d_i} = \{u^{d+d_i} | u^d \in c_i\}$ for $i = 1, \ldots, t$. It is obvious that $merge(C(u_1), \ldots, C(u_t))$ is a set of K-cuts for v.

If the network N does not contain cycles, the K-cuts of all nodes can be determined using the merge operation in

FindMinLabels(N)
  foreach node v in N **do** $l(v) \Leftarrow -w_v \cdot \phi$
  while(*there are updates in labels*) **do**
    foreach node v in N **do**
      $l(v) \Leftarrow \min_{c \in C(v)}\{\max\{l(u) - d \cdot \phi + 1 | u^d \in c\}\}$
    if v is a PO and $l(v) > \phi$, **return** failure
  return success

Sequential Circuit Technology Mapping, Figure 5
Labeling procedure

iter	*a*	*b*	*i*	*x*	*y*	*z*	*o*
0	$\{a^0\}:0$	$\{b^0\}:0$	$\{i^0\}:0$	$\{x^0\}:-1$	$\{y^0\}:\ \ 0$	$\{z^0\}:-1$	$\{o^0\}:-1$
1			$\{a^0\}:1$	$\{a^1,z^1\}:0$	$\{a^0,b^0,z^0\}:1$	$\{a^1,z^1,b^1\}:0$	$\{z^0\}:0$

Sequential Circuit Technology Mapping, Figure 6
Labeling example

a topological order starting from the PIs. For general networks, Fig. 3 outlines the iterative cut computation procedure proposed in [7].

Figure 4 depicts the iterations in enumerating 3-cuts for the design in Fig. 1 (1) when cuts are merged in the order *i*, *x*, *y*, *z*, and *o*. At the beginning, every node has its trivial cut formed by itself. Row 1 shows the new cuts discovered in the first iteration. In second iteration, two more cuts are discovered (for *x*). After that, the procedure stops as further merging does not yield any new cut.

Lemma 1 *After at most Kn iterations, the cut enumeration procedure will find the K-cuts for all nodes in N.*

Techniques have been proposed to speed up the procedure [7]. With those techniques, all 4-cuts for each of the ISCAS89 benchmark designs can be found in at most five iterations.

Labeling Phase

After obtaining all K-cuts, the algorithm evaluates the cuts based on sequential arrival times (or l-values), which is an extension of traditional arrival times, to consider the effect of retiming [6,8].

The labeling procedure tries to find a label for each node as outlined in Fig. 5, where w_v denotes the weight of shortest paths from PIs to node v.

Figure 6 shows the iterations for label computation for the design in Fig. 1 (1) assuming the target cycle time $\phi = 1$ and the nodes are evaluated in the order of *i*, *x*, *y*, *z*, and *o*. In the table, the current label as well as a corresponding cut for each node is listed. In this example, after first iteration, none of the labels will change and the procedure stops.

It can be shown that the labeling procedure will stop after at most $n(n-1)$ iterations [9]. The following lemma relates labels to mapping:

Lemma 2 *N has a mapping solution with cycle time φ iff the labeling procedure returns "success".*

Mapping Phase

Once the labels for all nodes are computed successfully, a mapping solution can be constructed starting from POs. At each node v, the procedure selects a cut that realizes the label of the node, and then moves on to select a cut for u if u^d is in the cut selected for v. On the edge from the LUT for u to the LUT for v, d FFs are added. For the design in Fig. 1 (1), the mapping solution generated based on the labels found in Fig. 6 is exactly the network in Fig. 2 (2).

To obtain a mapping solution with the target cycle time φ, the LUT for v can be retimed by $\lceil l(v)/\phi \rceil - 1$. For the design in Fig. 1 (1), the final mapping solution after retiming is shown in Fig. 2 (3).

Applications

The algorithm can be used to map a technology-independent Boolean network to a network consisting of cells from a target technology library. The concepts and framework are general enough to be adapted to study other circuit optimizations such as sequential circuit clustering and sequential circuit restructuring.

Cross References

- ► Circuit Retiming
- ► FPGA Technology Mapping
- ► Technology Mapping

Recommended Reading

1. Cong, J., Ding, Y.: FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. IEEE Trans. on Comput. Aided Des. of Integr. Circuits and Syst., **13**(1), 1–12 (1994)
2. Cong, J., Wu, C.: FPGA Synthesis with Retiming and Pipelining for Clock Period Minimization of Sequential Circuits. ACM/IEEE Design Automation Conference (1997)
3. Keutzer, K.: DAGON: Technology Binding and Local Optimization by DAG Matching. ACM/IEEE Design Automation Conference (1987)
4. Leiserson, C.E., Saxe, J.B.: Retiming Synchronous Circuitry. Algorithmica **6**, 5–35 (1991)
5. Mishchenko, A., Chatterjee, S., Brayton, R., Ciesielski, M.: An integrated technology mapping environment. International Workshop on Logic Synthesis (2005)
6. Pan, P.: Continuous Retiming: Algorithms and Applications. IEEE International Conference on Computer Design, pp. 116–121. (1997)
7. Pan, P., Lin, C.C.: A New Retiming-based Technology Mapping Algorithm for LUT-based FPGAs. ACM International Symposium on Field-Programmable Gate Arrays (1998)

8. Pan, P., Liu, C.L.: Optimal Clock Period FPGA Technology Mapping for Sequential Circuits. ACM/IEEE Design Automation Conference, June (1996)
9. Pan, P., Liu, C.L.: Optimal Clock Period FPGA Technology Mapping for Sequential Circuits. ACM Trans. on Des. Autom. of Electron. Syst., **3**(3), 437–462 (1998)

Sequential Exact String Matching

1994; Crochemore, Czumaj, Gąsieniec, Jarominek, Lecroq, Plandowski, Rytter

MAXIME CROCHEMORE[1], THIERRY LECROQ[2]
[1] Laboratory of Computer Science, University of Paris-East, Descartes, France
[2] Computer Science Department and LITIS Faculty of Science, University of Rouen, Rouen, France

Keywords and Synonyms

Exact pattern matching

Problem Definition

Given a *pattern string* $P = p_1 p_2 \dots p_m$ and a *text string* $T = t_1 t_2 \dots t_n$, both being sequences over an alphabet Σ of size σ, the *exact string matching (ESM)* problem is to find one or, more generally, all the text positions where P occurs in T, that is, compute the set $\{j \mid 1 \le j \le n-m+1$ and $P = t_j t_{j+1} \dots t_{j+m-1}\}$. The pattern is assumed to be given first and is then to be searched for in several texts.

Both worst- and average-case complexity are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from Σ. For simplicity and practicality the assumption $m = o(n)$ is set in this entry.

Key Results

Most algorithms that solve the ESM problem proceed in two steps: a preprocessing phase of the pattern P followed by a searching phase over the text T. The preprocessing phase serves to collect information on the pattern in order to speed up the searching phase.

The searching phase of string-matching algorithms works as follows: it first aligns the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern – this specific work is called an attempt or a scan – and after a whole match of the pattern or after a mismatch it shifts the pattern to the right. It repeats the same procedure again until the right end of the pattern goes beyond the right end of the text. The scanning part can be viewed as operating on the text through a window, which size is most often the length of the pattern. This processing manner is called the scan and shift mechanism. Different scanning strategies of the window lead to algorithms having specific properties and advantages.

The brute force algorithm for the ESM problem consists in checking if P occurs at each position j on T, with $1 \le j \le n - m + 1$. It does not need any preprocessing phase. It runs in quadratic time $O(mn)$ with constant extra space and performs $O(n)$ character comparisons on average. This is to be compared with the following bounds.

Theorem 1 (Cole et al. 1995 [3]) *The minimum number of character comparisons to solve the ESM problem in the worst case is* $\ge n + 9/(4m)(n - m)$, *and can be made* $\le n + 8/(3(m + 1))(n - m)$.

Theorem 2 (Yao 1979 [15]) *The ESM problem can be solved in optimal expected time* $O((\log m/m) \times n)$.

On-Line Text Parsing

The first linear ESM algorithm appears in the 1970's. The preprocessing phase consists in computing the periods of the pattern prefixes, or equivalently the length of the longest border for all the prefixes of the pattern. A border of a string is both a prefix and a suffix of it distinct from the string itself. Let $next[i]$ be the length of the longest border of $p_1 \dots p_{i-1}$. Consider an attempt at position j, when the pattern $p_1 \dots p_m$ is aligned with the segment $t_j \dots t_{j+m-1}$ of the text. Assume that the first mismatch (during a left to right scan) occurs between symbols p_i and t_{i+j} for $1 \le i \le m$. Then, $p_1 \dots p_{i-1} = t_j \dots t_{i+j-1} = u$ and $a = p_i \ne t_{i+j} = b$. When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. Doing so, after a shift, the comparisons can resume between $p_{next[i]}$ and t_{i+j} without missing any occurrence of P in T and having to backtrack on the text. There exists two variants, depending on whether $p_{next[i]}$ has to be different from p_i or not.

Theorem 3 (Knuth, Morris and Pratt 1977 [11]) *The text searching can be done in time O(n) and space O(m). Preprocessing the pattern can be done in time O(m).*

The search can be realized using an implementation with successor by default of the deterministic automaton $\mathcal{D}(P)$ recognizing the language $\Sigma^* P$. The size of the implementation is $O(m)$ independent of the alphabet size, due to the fact that $\mathcal{D}(P)$ possesses $m + 1$ states, m forward arcs, and at most m backward arcs. Using the automaton for searching a text leads to an algorithm having an efficient delay (maximum time for processing a character of the text).

Theorem 4 (Hancart 1993 [10]) *Searching for the pattern P can be done with a delay of* $O(\min\{\sigma, \log_2 m\})$ *letter comparisons.*

Note that for most algorithms the pattern preprocessing is not necessarily done before the text parsing as it can be performed on the fly during the parsing.

Practically-Efficient Algorithms

The Boyer–Moore algorithm is among the most efficient ESM algorithms. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the window from right to left beginning with its rightmost symbol. In case of a mismatch (or a complete match of the pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations. Assume that a mismatch occurs between character $p_i = a$ of the pattern and character $t_{i+j} = b$ of the text during an attempt at position j. Then, $p_{i+1} \dots p_m = t_{i+j+1} \dots t_{j+m} = u$ and $p_i \neq t_{i+j}$. The good-suffix shift consists in aligning the segment $t_{i+j+1} \dots t_{j+m}$ with its rightmost occurrence in P that is preceded by a character different from p_i. Another variant called the *best-suffix shift* consists in aligning the segment $t_{i+j} \dots t_{j+m}$ with its rightmost occurrence in P. Both variants can be computed in time and space $O(m)$ independent of the alphabet size. If there exists no such segment, the shift consists in aligning the longest suffix v of $t_{i+j+1} \dots t_{j+m}$ with a matching prefix of x. The bad-character shift consists in aligning the text character t_{i+j} with its rightmost occurrence in $p_1 \dots p_{m-1}$. If t_{i+j} does not appear in the pattern, no occurrence of P in T can overlap the symbol t_{i+j}, then the left end of the pattern is aligned with the character at position $i + j + 1$. The search can then be done in $O(n/m)$ in the best case.

Theorem 5 (Cole 1994 (see [5,14])) *During the search for a non-periodic pattern P of length m (such that the length of the longest border of P is less than m/2) in a text T of length n, the Boyer-Moore algorithm performs at most 3n comparisons between letters of P and of T.*

Yao's bound can be reached using an indexing structure for the reverse pattern. This is done by the Reverse Factor algorithm also called BDM (for Backward Dawg Matching).

Theorem 6 (Crochemore et al. 1994 [4]) *The search can be done in optimal expected time* $O((\log m/m) \times n)$ *using the suffix automaton or the suffix tree of the reverse pattern.*

A factor oracle can be used instead of an index structure, this is made possible since the only string of length m accepted by the factor oracle of a string w of length m is w itself. This is done by the Backward Oracle Matching (BOM) algorithm of Allauzen, Crochemore and Raffinot [1]. Its behavior in practice is similar to the one of the BDM algorithm.

Time-Space Optimal Algorithms

Algorithms of this type run in linear time (for both preprocessing and searching) and need only constant space in addition to the inputs.

Theorem 7 (Galil and Seiferas 1983 [8]) *The search can be done optimally in time O(n) and constant extra space.*

After Galil and Seiferas' first solution, other solutions are by Crochemore-Perrin [6] and by Rytter [13]. Algorithms rely on a partition of the pattern in two parts; they first search for the right part of the pattern from left to right, and then, if no mismatch occurs, they search for the left part. The partition can be: the perfect factorization [8], the critical factorization [6], or based on the lexicographically maximum suffix of the pattern [13]. Another solution by Crochemore (see [2]) is a variant of KMP [11]: it computes lower bounds of pattern prefixes periods on the fly and requires no preprocessing.

Bit-Parallel Solution

It is possible to use the bit-parallelism technique for ESM.

Theorem 8 (Baeza-Yates & Gonnet 1992; Wu & Manber 1992 (see [5,14])) *If the length m of the string P is smaller than the number of bits of a machine word, the preprocessing phase can be done in time and space* $\Theta(\sigma)$*. The searching phase executes in time* $\Theta(n)$*.*

It is even possible to use this bit-parallelism technique to simulate the BDM algorithm. This is realized by the BNDM (Backward Non-deterministic Dawg Matching) algorithm (see [2,12]).

In practice, when scanning the window from right to left during an attempt, it is sometimes more efficient to only use the bad-character shift. This was first done by the Horspool algorithm (see [2,12]). Other practical efficient algorithms are the Quick Search by Sunday (see [2,12]) and the Tuned Boyer-Moore by Hume and Sunday (see [2,12]).

There exists another method that uses the bit-parallelism technique that is optimal on the average though it consists actually of a filtration method. It con-

siders sparse q-grams and thus avoids to scan a lot of text positions. It is due to Fredriksson and Grabowski [7].

Applications

The methods which are described here apply to the treatment of the natural language, the treatment and analysis of genetic sequences and of musical sequences, the problems of safety related to data flows like virus detection, and the management of textual data bases, to quote only some immediate applications.

Open Problems

There remain only a few open problems on this question. It is still unknown if it is possible to design an average optimal time constant space string matching algorithm. The exact size of the Boyer-Moore automaton is still unknown (see [5]).

Experimental Results

The book of G. Navarro and M. Raffinot [12] is a good introduction and presents an experimental map of ESM algorithms for different alphabet sizes and pattern lengths. Basically, the Shift-Or algorithm is efficient for small alphabets and short patterns, the BNDM algorithm is efficient for medium size alphabets and medium length patterns, the Horspool algorithm is efficient for large alphabets, and the BOM algorithm is efficient for long patterns.

URL to Code

The site monge.univ-mlv.fr/~lecroq/string presents a large number of ESM algorithms (see also [2]). Each algorithm is implemented in C code and a Java applet is given.

Cross References

- ▶ Indexed approximate string matching refers to the case where the text is preprocessed;
- ▶ Regular expression matching is the more complex case where P can be a regular expression.
- ▶ Sequential approximate string matching is the version where errors are permitted;
- ▶ Sequential multiple string matching is the version where a finite set of patterns is searched in a text;

Recommended Reading

1. Allauzen, C., Crochemore, M., Raffinot, M.: Factor oracle: a new structure for pattern matching. In: SOFSEM'99. LNCS, vol. 1725, pp. 291–306. Springer, Berlin (1999)
2. Charras, C., Lecroq, T.: Handbook of exact string matching algorithms. King's College London Publications, London (2004)
3. Cole, R., Hariharan, R., Paterson, M., Zwick, U.: Tighter lower bounds on the exact complexity of string matching. SIAM J. Comput. **24**(1), 30–45 (1995)
4. Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string matching algorithms. Algorithmica **12**(4/5), 247–267 (1994)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press, New York (2007)
6. Crochemore, M., Perrin, D.: Two-way string matching. J. ACM **38**(3), 651–675 (1991)
7. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Proceedings of SPIRE'2005. LNCS, vol. 3772, pp. 374–385. Springer, Berlin (2005)
8. Galil, Z., Seiferas, J.: Time-space optimal string matching. J. Comput. Syst. Sci. **26**(3), 280–294 (1983)
9. Gusfield, D.: Algorithms on strings, trees and sequences. Cambridge University Press, Cambridge, UK (1997)
10. Hancart, C.: On Simon's string searching algorithm. Inf. Process. Lett. **47**(2), 95–99 (1993)
11. Knuth, D.E., Morris, J.H. Jr., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(1), 323–350 (1977)
12. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge, Uk (2002)
13. Rytter, W.: On maximal suffixes and constant-space linear-time versions of KMP algorithm. Theor. Comput. Sci. **299**(1–3), 763–774 (2003)
14. Smyth, W.F.: Computing Patterns in Strings. Addison Wesley Longman, Harlow, UK (2002)
15. Yao, A.: The complexity of pattern matching for a random string. SIAM J. Comput. **8**, 368–387 (1979)

Sequential Multiple String Matching

1999; Crochemore, Czumaj, Gąsieniec, Lecroq, Plandowski, Rytter

MAXIME CROCHEMORE[1,2], THIERRY LECROQ[3]

[1] Department of Computer Science, Kings College London, London, UK

[2] Laboratory of Computer Science, University of Paris-East, Paris, France

[3] Computer Science Department and LITIS Faculty of Science, University of Rouen, Rouen, France

Keywords and Synonyms

Dictionary matching

Problem Definition

Given a finite set of k *pattern strings* $\mathcal{P} = \{P^1, P^2, \ldots, P^k\}$ and a *text string* $T = t_1 t_2 \ldots t_n$, T and the P^is being sequences over an alphabet Σ of size σ, the *multiple string matching (MSM)* problem is to find one or, more generally, all the text positions where a P^i occurs in T. More

precisely the problem is to compute the set $\{j \mid \exists i, P^i = t_j t_{j+1} \ldots t_{j+|P^i|-1}\}$, or equivalently the set $\{j \mid \exists i, P^i = t_{j-|P^i|+1} t_{j-|P^i|+2} \ldots t_j\}$. Note that reporting all the occurrences of the patterns may lead to a quadratic output (for example, when P^is and T are drawn from a one-letter alphabet). The length of the shortest pattern in $\mathcal{P}$ is denoted by ℓmin. The patterns are assumed to be given first and are then to be searched for in several texts. This problem is an extension of the exact string matching problem.

Both worst- and average-case complexities are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from Σ. For simplicity and practicality the assumption $|P^i| = o(n)$ is set, for $1 \leq i \leq k$, in this entry.

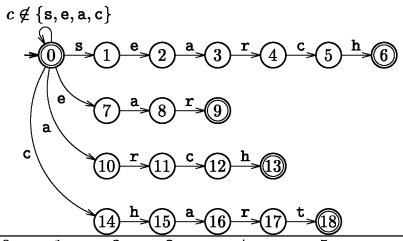
Key Results

A first solution to the multiple string matching problem consists in applying an exact string matching algorithm for locating each pattern in $\mathcal{P}$. This solution has an $O(kn)$ worst case time complexity. There are more efficient solutions along two main approaches. The first one, due to Aho and Corasick [1], is an extension of the automaton-based solution for matching a single string. The second approach, initiated by Commentz-Walter [3], extends the Boyer–Moore algorithm to several patterns.

The Aho–Corasick algorithm first builds a trie $T(\mathcal{P})$, a digital tree recognizing the patterns of $\mathcal{P}$. The trie $T(\mathcal{P})$ is a tree whose edges are labeled by letters and whose branches spell the patterns of $\mathcal{P}$. A node p in the trie $T(\mathcal{P})$ is associated with the unique word w spelled by the path of $T(\mathcal{P})$ from its root to p. The root itself is identified with the empty word ε. Notice that if w is a node in $T(\mathcal{P})$ then w is a prefix of some $P^i \in \mathcal{P}$. If in addition $a \in \Sigma$ then $child(w, a)$ is equal to wa if wa is a node in $T(\mathcal{P})$; it is equal to NIL otherwise.

During a second phase, when patterns are added to the trie, the algorithm initializes an output function *out*. It associates the singleton $\{P^i\}$ with the nodes P^i $(1 \leq i \leq k)$, and associates the empty set with all other nodes of $T(\mathcal{P})$.

Finally, the last phase of the preprocessing consists in building a failure link for each node of the trie, and simultaneously completing the output function. The failure function *fail* is defined on nodes as follows (w is a node): $fail(w) = u$ where u is the longest proper suffix of w that belongs to $T(\mathcal{P})$. Computation of failure links is done during a breadth-first traversal of $T(\mathcal{P})$. Completion of the output function is done while computing the failure function *fail* using the following rule: if $fail(w) = u$ then $out(w) = out(w) \cup out(u)$.



state	0	1	2	3	4	5	6
prefix	ε	s	se	sea	sear	searc	search
fail	0	0	7	8	9	12	13
out					{ear}		{search, arch}
state	7	8	9	10	11	12	13
prefix	e	ea	ear	a	ar	arc	arch
fail	0	10	11	0	0	14	15
out		{ear}				{arch}	
state	14	15	16	17	18	19	
prefix	c	ch	cha	char	chart		
fail	0	0	10	11	0		
out					{chart}		

Sequential Multiple String Matching, Figure 1
The Pattern Matching Machine or Aho–Corasick automaton for the set of strings {search, ear, arch, chart}

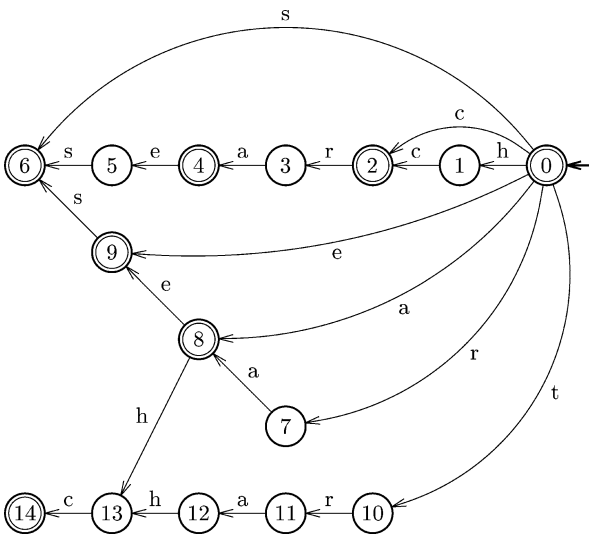
To stop going back with failure links during the computation of the failure links, and also to overpass text characters for which no transition is defined from the root during the searching phase, a loop is added on the root of the trie for these symbols. This finally produces what is called a Pattern Matching Machine or an Aho–Corasick automaton (see Fig. 1).

After the preprocessing phase is completed, the searching phase consists in parsing the text T with $T(\mathcal{P})$. This starts at the root of $T(\mathcal{P})$ and uses failure links whenever a character in T does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

Theorem 1 (Aho and Corasick [1]) *After preprocessing $\mathcal{P}$, searching for the occurrences of the strings of $\mathcal{P}$ in a text T can be done in time $O(n \times \log \sigma)$. The running time of the associated preprocessing phase is $O(|\mathcal{P}| \times \log \sigma)$. The extra memory space required for both operations is $O(|\mathcal{P}|)$.*

The Aho–Corasick algorithm is actually a generalization to a finite set of strings of the Morris–Pratt exact string matching algorithm.

Commentz-Walter [3] generalized the Boyer–Moore exact string matching algorithm to Multiple String Matching. Her algorithm builds a trie for the reverse patterns in $\mathcal{P}$ together with two shift tables, and applies a right to left scan strategy. However it is intricate to implement and has a quadratic worst-case time complexity.



Sequential Multiple String Matching, Figure 2
An example of DAWG, index structure used for matching the set of strings {`search`, `ear`, `arch`, `chart`}. The automaton accepts the reverse prefixes of the strings

The DAWG-match algorithm [4] is a generalization of the BDM exact string matching algorithm. It consists in building an exact indexing structure for the reverse strings of $\mathcal{P}$ such as a factor automaton or a generalized suffix tree, instead as just a trie as in the previous solution (see Fig. 2). The overall algorithm can be made optimal by using both an indexing structure for the reverse patterns and an Aho–Corasick automaton for the patterns. Then, searching involves scanning some portions of the text from left to right and some other portions from right to left. This enables to skip large portions of the text T.

Theorem 2 (Crochemore et al. [4]) *The DAWG-match algorithm performs at most 2n symbol comparisons. Assuming that the sum of the length of the patterns in $\mathcal{P}$ is less than ℓmin^k, the DAWG-match algorithm makes on average $O((n \log \ell min)/\ell min)$ inspections of text characters.*

The bottleneck of the DAWG-match algorithm is the construction time and space consumption of the exact indexing structure. This can be avoided by replacing the exact indexing structure by a factor oracle for a set of strings. When the factor oracle is used alone, it gives the Set Backward Oracle Matching (SBOM) algorithm [2]. It is an exact algorithm that behaves almost as well as the DAWG-match algorithm.

The bit-parallelism technique can be used to simulate the DAWG-match algorithm. It gives the MultiBNDM algorithm of Navarro and Raffinot [7]. This strategy is efficient when $k \times \ell min$ bits fit in a few computer words. The prefixes of strings of $\mathcal{P}$ of length ℓmin are packed together in a bit vector. Then, the search is similar to the BNDM exact string matching and is performed for all the prefixes at the same time.

The use of the generalization of the bad-character shift alone as done in the Horspool exact string matching algorithm gives poor performances for the MSM problem due to the high probability of finding each character of the alphabet in one of the strings of $\mathcal{P}$.

The algorithm of Wu and Manber [11] considers blocks of length ℓ. Blocks of such a length are hashed using a function h into values less than *maxvalue*. Then $shift[h(B)]$ is defined as the minimum between $|P^i| - j$ and $\ell min - \ell + 1$ with $B = p^i_{j-\ell+1} \dots p^i_j$ for $1 \le i \le k$ and $1 \le j \le |P^i|$. The value of ℓ varies with the minimum length of the strings in $\mathcal{P}$ and the size of the alphabet. The value of *maxvalue* varies with the memory space available.

The searching phase of the algorithm consists in reading blocks B of length ℓ. If $shift[h(B)] > 0$ then a shift of length $shift[h(B)]$ is applied. Otherwise, when $shift[h(B)] = 0$ the patterns ending with block B are examined one by one in the text. The first block to be scanned is $t_{\ell min-\ell+1} \dots t_{\ell min}$. This method is incorporated in the *agrep* command [10].

Applications

MSM algorithms serve as basis for multidimensional pattern matching and approximate pattern matching with wildcards. The problem has many applications in computational biology, database search, bibliographic search, virus detection in data flows, and several others.

Experimental Results

The book of G. Navarro and M. Raffinot [8] is a good introduction to the domain. It presents experimental graphics that report experimental evaluation of multiple string matching algorithms for different alphabet sizes, pattern lengths, and sizes of pattern set.

URL to Code

Well-known packages offering efficient MSM are *agrep* (http://webglimpse.net/download.html, top-level subdirectory `agrep/`) and *grep* with the -F option (http://www.gnu.org/software/grep/grep.html).

Cross References

- ▶ Indexed String Matching refers to the case where the text can be preprocessed;

- ▶ Multidimensional String Matching is the case where the text dimension is greater than one.
- ▶ Regular Expression Matching is the more complex case where the pattern can be a regular expression;
- ▶ Sequential Exact String Matching is the version where a single pattern is searched for in a text;

Recommended Reading

Further information can be found in the four following books: [5,6,8] and [9].

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. C. ACM **18**(6), 333–340 (1975)
2. Allauzen, C., Crochemore, M., Raffinot, M.: Factor oracle: a new structure for pattern matching. In: SOFSEM'99. LNCS, vol. 1725, pp. 291–306. Springer, Berlin (1999)
3. Commentz-Walter, B.: A string matching algorithm fast on the average. In: Proceedings of ICALP'79. LNCS, vol. 71, pp. 118–132. Springer, Berlin (1979)
4. Crochemore, M., Czumaj, A., Gąsieniec, L., Lecroq, T., Plandowski, W., Rytter, W.: Fast practical multi-pattern matching. Inf. Process. Lett. **71**(3–4), 107–113 (1999)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press, Cambridge (2007)
6. Gusfield, D.: Algorithms on strings, trees and sequences. Cambridge University Press, Cambridge (1997)
7. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM J. Exp. Algorithm **5**, 4 (2000)
8. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge (2002)
9. Smyth, W.F.: Computing Patterns in Strings. Addison Wesley Longman (2002)
10. Wu, S., Manber, U.: Agrep – a fast approximate pattern-matching tool. In: Proceedings of USENIX Winter (1992) Technical Conference, pp. 153–162. USENIX Association, Berkeley (1992)
11. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ (1994)

Set Agreement

1993; Chaudhuri

Michel Raynal
IRISA, University of Rennes 1,
Rennes, France

Keywords and Synonyms

Distributed coordination

Problem Definition

Short History

The k-set agreement problem is a paradigm of coordination problems. Defined in the setting of systems made up of processes prone to failures, it is a simple generalization of the consensus problem (that corresponds to the case $k = 1$). That problem was introduced in 1993 by Chaudhuri [2] to investigate how the number of choices (k) allowed for the processes is related to the maximum number of processes that can crash. (After it has crashed, a process executes no more steps: a crash is a premature halting.)

Definition

Let S be a system made up of n processes where up to t can crash and where each process has an input value (called a *proposed* value). The problem is defined by the three following properties (i. e., any algorithm that solves that problem has to satisfy these properties):

1. Termination. Every nonfaulty process decides a value.
2. Validity. A decided value is a proposed value.
3. Agreement. At most k different values are decided.

The Trivial Case

It is easy to see that this problem can be trivially solved if the upper bound on the number of process failures t is smaller than the allowed number of choices k, also called the *coordination degree*. (The trivial solution consists in having $t + 1$ predetermined processes that send their proposed values to all the processes, and a process deciding the first value it ever receives.) So, $k \leq t$ is implicitly assumed in the following.

Key Results

Key Results in Synchronous Systems

The Synchronous Model In this computation model, each execution consists of a sequence of rounds. These are identified by the successive integers 1, 2, etc. For the processes, the current round number appears as a global variable whose global progress entails their own local progress.

During a round, a process first broadcasts a message, then receives messages, and finally executes local computation. The fundamental synchrony property the a synchronous system provides the processes with is the following: a message sent during a round r is received by its destination process during the very same round r. If during a round, a process crashes while sending a message, an arbitrary subset (not known in advance) of the processes receive that message.

```
Function k-set_agreement (v_i)
(1) est_i ← v_i;
(2) when r = 1, 2, ..., ⌊t/k⌋ + 1 do % r: round number %
(3) begin_round
(4)       send (est_i) to all; % including p_i itself %
(5)       est_i ← min({est_j values received during
                                    the current round r});
(6) end_round;
(7) return (est_i)
```

Set Agreement, Figure 1
A simple *k*-set agreement synchronous algorithm (code for p_i)

Main Results The k-set agreement problem can always be solved in a synchronous system. The main result is for the minimal number of rounds (R_t) that are needed for the nonfaulty processes to decide in the worst-case scenario (this scenario is when exactly k processes crash in each round). It was shown in [3] that $R_t = \lfloor \frac{t}{k} \rfloor + 1$. A very simple algorithm that meets this lower bound is described in Fig. 1.

Although failures do occur, they are rare in practice. Let f denote the number of processes that crash in a given run, $0 \leq f \leq t$. We are interested in synchronous algorithms that terminate in at most R_t rounds when t processes crash in the current run, but that allow the nonfaulty processes to decide in far fewer rounds when there are few failures. Such algorithms are called *early-deciding* algorithms. It was shown in [4] that, in the presence of f process crashes, any early-deciding k-set agreement algorithm has runs in which no process decides before the round $R_f = \min(\lfloor \frac{f}{k} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1)$. This lower bound shows an inherent tradeoff linking the coordination degree k, the maximum number of process failures t, the actual number of process failures f, and the best time complexity that can be achieved. Early-deciding k-set agreement algorithms for the synchronous model can be found in [4,12].

Other Failure Models In the send omission failure model, a process is faulty if it crashes or forgets to send messages. In the general omission failure model, a process is faulty if it crashes, forgets to send messages, or forgets to receive messages. (A send omission failure models the failure of an output buffer, while a receive omission failure models the failure of an input buffer.) These failure models were introduced in [11].

The notion of *strong* termination for set agreement problems was introduced in [13]. Intuitively, that property requires that as many processes as possible decide. Let a *good* process be a process that neither crashes nor commits receive omission failures. A set agreement algorithm is strongly terminating if it forces all the good processes to decide. (Only the processes that crash during the execution of the algorithm, or that do not receive enough messages, can be prevented from deciding.)

An early-deciding k-set agreement algorithm for the general omission failure model was described in [13]. That algorithm, which requires $t < n/2$, directs a good process to decide and stop in at most $R_f = \min(\lfloor \frac{f}{k} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1)$ rounds. Moreover, a process that is not a good process executes at most $R_f(not\_good) = \min(\lceil \frac{f}{k} \rceil + 2, \lfloor \frac{t}{k} \rfloor + 1)$ rounds.

As R_f is a lower bound for the number of rounds in the crash failure model, the previous algorithm shows that R_f is also a lower bound for the nonfaulty processes to decide in the more severe general omission failure model. Proving that $R_f(not\_good)$ is an upper bound for the number of rounds that a nongood process has to execute remains an open problem.

It was shown in [13] that, for a given coordination degree k, $t < \frac{k}{k+1}n$ is an upper bound on the number of process failures when one wants to solve the k-set agreement problem in a synchronous system prone to process general omission failures. A k-set agreement algorithm that meets this bound was described in [13]. That algorithm requires the processes execute $R = t + 2 - k$ rounds to decide. Proving (or disproving) that R is a lower bound when $t < \frac{k}{k+1}n$ is an open problem. Designing an early-deciding k-set agreement algorithm for $t < \frac{k}{k+1}n$ and $k > 1$ is another problem that remains open.

Key Results in Asynchronous Systems

Impossibility A fundamental result of distributed computing is the impossibility to design a deterministic algorithm that solves the k-set agreement problem in asynchronous systems when $k \leq$ t [1,7,15]. Compared with the impossibility of solving asynchronous consensus despite one process crash, that impossibility is based on deep combinatorial arguments. This impossibility has opened new research directions for the connection between distributed computing and topology. This topology approach has allowed the discovery of links relating asynchronous k-set agreement with other distributed computing problems such as the *renaming* problem [5].

Circumventing the Impossibility Several approaches have been investigated to circumvent the previous impossibility. These approaches are the same as those that have been used to circumvent the impossibility of asynchronous consensus despite process crashes.

One approach consists in replacing the "deterministic algorithm" by a "randomized algorithm." In that case, the termination property becomes "the probability for a correct process to decide tends to 1 when the number of rounds tends to $+\infty$." That approach was investigated in [9].

Another approach that has been proposed is based on failure detectors. Roughly speaking, a failure detector provides each process with a list of processes suspected to have crashed. As an example, the class of failure detectors denoted $\diamond S_x$ includes all the failure detectors such that, after some finite (but unknown) time, (1) any list contains the crashed processes and (2) there is a set Q of x processes such that Q contains one correct process and that correct process is no longer suspected by the processes of Q (let us observe that correct processes can be suspected intermittently or even forever). Tight bounds for the k-set agreement problem in asynchronous systems equipped with such failure detectors, conjectured in [9], were proved in [6]. More precisely, such a failure detector class allows the k-set agreement problem to be solved for $k \geq t - x + 2$ [9], and cannot solve it when $k < t - x + 2$ [6].

Another approach that has been investigated is the combination of failure detectors and conditions [8]. A condition is a set of input vectors, and each input vector has one entry per process. The entries of the input vector associated with a run contain the values proposed by the processes in that run. Basically, such an approach guarantees that the nonfaulty processes always decide when the actual input vector belongs to the condition the k-set algorithm has been instantiated with.

Applications

The set agreement problem was introduced to study how the number of failures and the synchronization degree are related in an asynchronous system; hence, it is mainly a theoretical problem. That problem is used as a canonical problem when one is interested in asynchronous computability in the presence of failures. Nevertheless, one can imagine practical problems the solutions of which are based on the set agreement problem (e. g., allocating a small shareable resources—such as broadcast frequencies—in a network).

Cross References

- ► Asynchronous Consensus Impossibility
- ► Failure Detectors
- ► Renaming
- ► Topology Approach in Distributed Computing

Recommended Reading

1. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computation, California, 1993, pp. 91–100
2. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Inf. Comput. **105**, 132–158 (1993)
3. Chaudhuri, S., Herlihy, M., Lynch, N., Tuttle, M.: Tight Bounds for k-Set Agreement. J. ACM **47**(5), 912–943 (2000)
4. Gafni, E., Guerraoui, R., Pochon, B.: From a Static Impossibility to an Adaptive Lower Bound: The Complexity of Early Deciding Set Agreement. In: Proc. 37th ACM Symposium on Theory of Computing (STOC 2005), pp. 714–722. ACM Press, New York (2005)
5. Gafni, E., Rajsbaum, S., Herlihy, M.: Subconsensus Tasks: Renaming is Weaker than Set Agreement. In: Proc. 20th Int'l Symposium on Distributed Computing (DISC'06). LNCS, vol. 4167, pp. 329–338. Springer, Berlin (2006)
6. Herlihy, M.P., Penso, L.D.: Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. Distrib. Comput. **18**(2), 157–166 (2005)
7. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. J. ACM **46**(6), 858–923 (1999)
8. Mostefaoui, A., Rajsbaum, S., Raynal, M.: The Combined Power of Conditions and Failure Detectors to Solve Asynchronous Set Agreement. In: Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC'05), pp. 179–188. ACM Press, New York (2005)
9. Mostefaoui, A., Raynal, M.: k-Set Agreement with Limited Accuracy Failure Detectors. In: Proc. 19th ACM Symposium on Principles of Distributed Computing, pp. 143–152. ACM Press, New York (2000)
10. Mostefaoui, A., Raynal, M.: Randomized Set Agreement. In: Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01), Hersonissos (Crete) pp. 291–297. ACM Press, New York (2001)
11. Perry, K.J., Toueg, S.: Distributed Agreement in the Presence of Processor and Communication Faults. IEEE Trans. Softw. Eng. **SE-12**(3), 477–482 (1986)
12. Raipin Parvedy, P., Raynal, M., Travers, C.: Early-stopping k-set agreement in synchronous systems prone to any number of process crashes. In: Proc. 8th Int'l Conference on Parallel Computing Technologies (PaCT'05). LNCS, vol. 3606, pp. 49–58. Springer, Berlin (2005)
13. Raipin Parvedy, P., Raynal, M., Travers, C.: Strongly-terminating early-stopping k-set agreement in synchronous systems with general omission failures. In: Proc. 13th Colloquium on Structural Information and Communication Complexity (SIROCCO'06). LNCS, vol. 4056, pp. 182–196. Springer, Berlin (2006)
14. Raynal, M., Travers, C.: Synchronous set agreement: a concise guided tour (including a new algorithm and a list of open problems). In: Proc. 12th Int'l IEEE Pacific Rim Dependable Computing Symposium (PRDC'2006), pp. 267–274. IEEE Society Computer Press, Los Alamitos (2006)
15. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM J. Comput. **29**(5), 1449–1483 (2000)

Set Cover with Almost Consecutive Ones

2004; Mecke, Wagner

Michael Dom
Department of Mathematics and Computer Science, University of Jena, Jena, Germany

Keywords and Synonyms

Hitting set

Problem Definition

The Set Cover problem has as input a set R of m items, a set C of n subsets of R and a weight function $w\colon C \to \mathbb{Q}$. The task is to choose a subset $C' \subseteq C$ of minimum weight whose union contains all items of R.

The sets R and C can be represented by an $m \times n$ binary matrix A that consists of a row for every item in R and a column for every subset of R in C, where an entry $a_{i,j}$ is 1 iff the ith item in R is part of the jth subset in C. Therefore, the Set Cover problem can be formulated as follows.

> **Input**: An $m \times n$ binary matrix A and a weight function w on the columns of A.
> **Task:** Select some columns of A with minimum weight such that the submatrix A' of A that is induced by these columns has at least one 1 in every row.

While Set Cover is NP-hard in general [4], it can be solved in polynomial time on instances whose columns can be permuted in such a way that in every row the ones appear consecutively, that is, on instances that have the *consecutive ones property* (*C1P*).[1]

Motivated by problems arising from railway optimization, Mecke and Wagner [7] consider the case of Set Cover instances that have "almost the C1P". Having almost the C1P means that the corresponding matrices are similar to matrices that have been generated by starting with a matrix that has the C1P and replacing randomly a certain percentage of the 1's by 0's [7]. For Ruf and Schöbel [8], in contrast, having almost the C1P means that the average number of blocks of consecutive 1's per row is much smaller than the number of columns of the matrix. This entry will also mention some of their results.

[1]The C1P can be defined symmetrically for columns; this article focuses on rows. Set Cover on instances with the C1P can be solved in polynomial time, e. g., with a linear programming approach, because the corresponding coefficient matrices are totally unimodular (see [9]).

Notation

Given an instance (A, w) of Set Cover, let R denote the row set of A and C its column set. A column c_j *covers* a row r_i, denoted by $r_i \in c_j$, if $a_{i,j} = 1$.

A binary matrix has the *strong C1P* if (without any column permutation) the 1's appear consecutively in every row. A *block of consecutive 1's* is a maximal sequence of consecutive 1's in a row. It is possible to determine in linear time if a matrix has the C1P, and if so, to compute a column permutation that yields the strong C1P [2,3,6]. However, note that it is NP-hard to permute the columns of a binary matrix such that the number of blocks of consecutive 1's in the resulting matrix is minimized [1,4,5].

A *data reduction rule* transforms in polynomial time a given instance I of an optimization problem into an instance I' of the same problem such that $|I'| < |I|$ and the optimal solution for I' has the same value (e. g., weight) as the optimal solution for I. Given a set of data reduction rules, *to reduce* a problem instance means to repeatedly apply the rules until no rule is applicable; the resulting instance is called *reduced*.

Key Results

Data Reduction Rules

For Set Cover there exist well-known data reduction rules:

Row domination rule: If there are two rows $r_{i_1}, r_{i_2} \in R$ with $\forall c \in C\colon r_{i_1} \in c$ implies $r_{i_2} \in c$, then r_{i_2} is *dominated* by r_{i_1}. Remove row r_{i_2} from A.

Column domination rule: If there are two columns $c_{j_1}, c_{j_2} \in C$ with $w(c_{j_1}) \geq w(c_{j_2})$ and $\forall r \in R\colon r \in c_{j_1}$ implies $r \in c_{j_2}$, then c_{j_1} is *dominated* by c_{j_2}. Remove c_{j_1} from A.

In addition to these two rules, a column $c_{j_1} \in C$ can also be dominated by a subset $C' \subseteq C$ of the columns instead of a single column: If there is a subset $C' \subseteq C$ with $w(c_{j_1}) \geq \sum_{c \in C'} w(c)$ and $\forall r \in R\colon r \in c_{j_1}$ implies $(\exists c \in C'\colon r \in c)$, then remove c_{j_1} from A. Unfortunately, it is NP-hard to find a dominating subset C' for a given set c_{j_1}. Mecke and Wagner [7], therefore, present a restricted variant of this generalized column domination rule.

For every row $r \in R$, let $c_{\min}(r)$ be a column in C that covers r and has minimum weight under this property. For two columns $c_{j_1}, c_{j_2} \in C$, define $X(c_{j_1}, c_{j_2}) := \{c_{\min}(r) \mid r \in c_{j_1} \wedge r \notin c_{j_2}\}$. The new data reduction rule then reads as follows.

Advanced column domination rule: If there are two columns $c_{j_1}, c_{j_2} \in C$ and a row that is covered by both c_{j_1}

and c_{j_2}, and if $w(c_{j_1}) \geq w(c_{j_2}) + \sum_{c \in X(c_{j_1}, c_{j_2})} w(c)$, then c_{j_1} is *dominated* by $\{c_{j_2}\} \cup X(c_{j_1}, c_{j_2})$. Remove c_{j_1} from A.

Theorem 1 ([7]) *A matrix A can be reduced in $O(Nn)$ time with respect to the column domination rule, in $O(Nm)$ time with respect to the row domination rule, and in $O(Nmn)$ time with respect to all three data reduction rules described above, when N is the number of 1's in A.*

In the databases used by Ruf and Schöbel [8], matrices are represented by the column indices of the first and last 1's of its blocks of consecutive 1's. For such matrix representations, a fast data reduction rule is presented [8], which eliminates "unnecessary" columns and which, in the implementations, replaces the column domination rule. The new rule is faster than the column domination rule (a matrix can be reduced in $O(mn)$ time with respect to the new rule), but not as powerful: Reducing a matrix A with the new rule can result in a matrix that has more columns than the matrix resulting from reducing A with the column domination rule.

Algorithms

Mecke and Wagner [7] present an algorithm that solves SET COVER by enumerating all feasible solutions.

Given a row r_i of A, a *partial solution for the rows* $r_1, \ldots, r_i$ is a subset $C' \subseteq C$ of the columns of A such that for each row r_j with $j \in \{1, \ldots, i\}$ there is a column in C' that covers row r_j.

The main idea of the algorithm is to find an optimal solution by iterating over the rows of A and updating in every step a data structure S that keeps *all* partial solutions for the rows considered so far. More exactly, in every iteration step the algorithm considers the first row of A and updates the data structure S accordingly. Thereafter, the first row of A is deleted. The following code shows the algorithm.

```
1 Repeat m times: {
2   for every partial solution C' in S that does not cover
                                        the first row of A: {
3     for every column c of A that covers the first row
                                                  of A: {
4       Add {c} ∪ C' to S; }
5     Delete C' from S; }
6   Delete the first row of A; }
```

This straightforward enumerative algorithm could create a set S of exponential size. Therefore, the data reduction rules presented above are used to delete after each iteration step partial solutions that are not needed any more. To this end, a matrix B is associated with the set S, where every row corresponds to a row of A and every column corresponds to a partial solution in S—an entry $b_{i,j}$ of B is 1 iff the jth partial solution of B contains a column of A that covers the row r_i. The algorithm uses the matrix $C := \left(\begin{array}{c|c} A & B \\ 0 \ldots 0 & 1 \ldots 1 \end{array}\right)$, which is updated together with S in every iteration step.[2] Line 6 of the code shown above is replaced by the following two lines:

```
6   Delete the first row of the matrix C;
7   Reduce the matrix C and update S accordingly; }
```

At the end of the algorithm, S contains exactly one solution, and this solution is optimal. Moreover, if the SET COVER instance is nicely structured, the algorithm has polynomial running time:

Theorem 2 ([7]) *If A has the strong C1P, is reduced, and its rows are sorted in lexicographic order, then the algorithm has a running time of $O(M^{3n})$ where M is the maximum number of 1's per row and per column.*

Theorem 3 ([7]) *If the distance between the first and the last 1 in every column is at most k, then at any time throughout the algorithm the number of columns in the matrix B is $O(2^{kn})$, and the running time is $O(2^{2k}kmn^2)$.*

Ruf and Schöbel [8] present a branch and bound algorithm for SET COVER instances that have a small average number of blocks of consecutive 1's per row.

The algorithm considers in each step a row r_i of the current matrix (which has been reduced with data reduction rules before) and branches into bl_i cases, where bl_i is the number of blocks of consecutive 1's in r_i. In each case, one block of consecutive 1's in row r_i is selected, and the 1's of all other blocks in this row are replaced by 0's. Thereafter, a lower and an upper bound on the weight of the solution for each resulting instance is computed. If a lower bound differs by a factor of more than $1 + \epsilon$, for a given constant ε, from the best upper bound achieved so far, the corresponding instance is subjected to further branchings. Finally, the best upper bound that was found is returned.

In each branching step, the bl_i instances that are newly generated are "closer" to have the (strong) C1P than the instance from which they descend. If an instance has the C1P, the lower and upper bound can easily be computed by exactly solving the problem. Otherwise, standard heuristics are used.

[2] The last row of C allows to distinguish the columns belonging to A from those belonging to B.

Applications

SET COVER instances occur e. g. in railway optimization, where the task is to determine where new railway stations should be built. Each row then corresponds to an existing settlement, and each column corresponds to a location on the existing trackage where a railway station could be build. A column c covers a row r, if the settlement corresponding to r lies within a given radius around the location corresponding to c.

If the railway network consisted of one straight line rail track only, the corresponding SET COVER instance would have the C1P; instances arising from real world data are close to have the C1P [7,8].

Experimental Results

Mecke and Wagner [7] make experiments on real-world instances as described in the Applications section and on instances that have been generated by starting with a matrix that has the C1P and replacing randomly a certain percentage of the 1's by 0's. The real-world data consists of a railway graph with 8200 nodes and 8700 edges, and 30 000 settlements. The generated instances consist of 50–50 000 rows with 10–200 1's per row. Up to 20% of the 1's are replaced by 0's.

In the real-world instances, the data reduction rules decrease the number of 1's to between 1% and 25% of the original number of 1's without and to between 0.2% and 2.5% with the advanced column reduction rule. In the case of generated instances that have the C1P, the number of 1's is decreased to about 2% without and to 0.5% with the advanced column reduction rule. In instances with 20% perturbation, the number of 1's is decreased to 67% without and to 20% with the advanced column reduction rule.

The enumerative algorithm has a running time that is almost linear for real-world instances and most generated instances. Only in the case of generated instances with 20% perturbation, the running time is quadratic.

Ruf and Schöbel [8] consider three instance types: real-world instances, instances arising from Steiner triple systems, and randomly generated instances. The latter have a size of 100×100 and contain either 1–5 blocks of consecutive 1's in each row, each one consisting of between one and nine 1's, or they are generated with a probability of 3% or 5% for any entry to be 1.

The data reduction rules used by Ruf and Schöbel turn out to be powerful for the real-world instances (reducing the matrix size from about 1100×3100 to 100×800 in average), whereas for all other instance types the sizes could not be reduced noticeably.

The branch and bound algorithm could solve almost all real-world instances up to optimality within a time of less than a second up to one hour. In all cases where an optimal solution has been found, the first generated subproblem had already provided a lower bound equal to the weight of the optimal solution.

Cross References

► Greedy Set-Cover Algorithms

Recommended Reading

1. Atkins, J.E., Middendorf, M.: On physical mapping and the consecutive ones property for sparse matrices. Discret. Appl. Math. **71**(1–3), 23–40 (1996)
2. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Comput. Syst. Sci. **13**, 335–379 (1976)
3. Fulkerson, D.R., Gross, O.A.: Incidence matrices and interval graphs. Pac. J. Math. **15**(3), 835–855 (1965)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
5. Goldberg, P.W., Golumbic, M.C., Kaplan, H., Shamir, R.: Four strikes against physical mapping of DNA. J. Comput. Biol. **2**(1), 139–152 (1995)
6. Hsu, W.L., McConnell, R.M.: PC trees and circular-ones arrangements. Theor. Comput. Sci. **296**(1), 99–116 (2003)
7. Mecke, S., Wagner, D.: Solving geometric covering problems by data reduction. In: Proceedings of the 12th Annual European Symposium on Algorithms (ESA '04). LNCS, vol. 3221, pp. 760–771. Springer, Berlin (2004)
8. Ruf, N., Schöbel, A.: Set covering with almost consecutive ones property. Discret. Optim. **1**(2), 215–228 (2004)
9. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)

Shortest Elapsed Time First Scheduling

2003; Bansal, Pruhs

NIKHIL BANSAL
IBM Research, IBM, Yorktown Heights, NY, USA

Keywords and Synonyms

Sojourn time; Response time; Scheduling with unknown job sizes; MLF algorithm; Feedback Queues

Problem Definition

The problem is concerned with scheduling dynamically arriving jobs in the scenario when the processing require-

ments of jobs are unknown to the scheduler. The lack of knowledge of how long a job will take to execute is a particularly attractive assumption in real systems where such information might be difficult or impossible to obtain. The goal is to schedule jobs to provide good quality of service to the users. In particular the goal is to design algorithms that have good average performance and are also fair in the sense that no subset of users experiences substantially worse performance than others.

Notations

Let $\mathcal{J} = \{1, 2, \ldots, n\}$ denote the set of jobs in the input instance. Each job j is characterized by its release time r_j and its processing requirement p_j. In the online setting, job j is revealed to the scheduler only at time r_j. A further restriction is the *non-clairvoyant* setting, where only the existence of job j is revealed at r_j, in particular the scheduler does not know p_j until the job meets its processing requirement and leaves the system. Given a schedule, the completion time c_j of a job is the earliest time at which job j receives p_j amount of service. The flow time f_j of j is defined as $c_j - r_j$. The stretch of a job is defined the ratio of its flow time divided by its size. Stretch is also referred to as normalized flow time or slowdown, and is a natural measure of fairness as it measures the waiting time of a job per unit of service received. A schedule is said to be preemptive, if a job can be interrupted arbitrarily, and its execution can be resumed later from the point of interruption without any penalty. It is well known that preemption is necessary to obtain reasonable guarantees for flow time even in the offline setting [5].

Recall that the online Shortest Remaining Processing Time (SRPT) algorithm, that at any time works on the job with the least remaining processing, is optimum for minimizing average flow time. However, a common critique of SRPT is that it may lead to starvation of jobs, where some jobs may be delayed indefinitely. For example, consider the sequence where a job of size 3 arrives at time $t = 0$, and one job of size 1 arrives every unit of time starting $t = 1$ for a long time. Under SRPT the size 3 job will be delayed until the size 1 jobs stop arriving. On the other hand, if the goal is to minimize the maximum flow time, then it is easily seen that First in First out (FIFO) is the optimum algorithm. However, FIFO can perform very poorly with respect to average flow time (for example, many small jobs could be stuck behind a very large job that arrived just earlier). A natural way to balance both the average and worst case performance is to consider the ℓ_p norms of flow time and stretch, where the ℓ_p norm of the sequence $x_1, \ldots, x_n$ is defined as $(\sum_i x_i^p)^{1/p}$.

The Shortest Elapsed Time First (SETF) is a non-clairvoyant algorithm that at any time works on the job that has received the least amount of service thus far. This is a natural way to favor short jobs given the lack of knowledge of job sizes. In fact, SETF is the continuous version of the Multi-Level Feedback (MLF) algorithm. Unfortunately, SETF (or any other deterministic non-clairvoyant algorithm) performs poorly in the framework of competitive analysis, where an algorithm is called c-competitive if for every input instance, its performance is no worse than c times that of the optimum offline (clairvoyant) solution for that instance [7]. However, competitive analysis can be overly pessimistic in its guarantee. A way around this problem was proposed by Kalyanasundaram and Pruhs [6] who allowed the online scheduler a slightly faster processor to make up for its lack of knowledge of future arrivals and job sizes. Formally, an algorithm Alg is said to be s-speed, c-speed competitive where c is worst case ratio over all instance I, of $Alg_s(I)/Opt_1(I)$, where Alg_s is the value of solution produced by Alg when given an s speed processor, and Opt_1 is the optimum value using a speed 1 processor. Typically the most interesting results are those where c is small and $s = (1 + \epsilon)$ for any arbitrary $\epsilon > 0$.

Key Results

In their seminal paper [6], Kalyanasundaram and Pruhs showed the following.

Theorem 1 ([6]) *SETF is a $(1 + \epsilon)$-speed, $(1 + 1/\epsilon)$-competitive non-clairvoyant algorithm for minimizing the average flow time on a single machine with preemptions.*

For minimizing the average stretch, Muthukrishnan, Rajaraman, Shaheen and Gehrke [8] considered the clairvoyant setting and showed that SRPT is 2-competitive for a single machine and 14 competitive for multiple machines. The non-clairvoyant setting was consider by Bansal, Dhamdhere, Konemann and Sinha [1]. They showed that

Theorem 2 ([1]) *SETF is a $(1 + \epsilon)$-speed, $O(\log^2 P)$-competitive for minimizing average stretch, where P is the ratio of the maximum to minimum job size. On the other hand, even with $O(1)$-speed, any non-clairvoyant algorithm is at least $\Omega(\log P)$-competitive. Interestingly, in terms of n, any non-clairvoyant algorithm must be $\Omega(n)$-competitive even with $O(1)$-speedup. Moreover, SETF is $O(n)$ competitive (even without extra speedup).*

For the special case when all jobs arrive at time 0, SETF is optimum up to constant factors. It is $O(\log P)$-competitive (without any extra speedup). Moreover, any

non-clairvoyant must be $\Omega(\log P)$ competitive even with factor O(1) speedup.

The key idea of the above result was a connection between SETF and SRPT. First, at the expense of $(1+\epsilon)$-speedup it can be seen that SETF is no worse than MLF where the thresholds are powers of $(1+\epsilon)$. Second, the behavior of MLF on an instance I can be related to the behavior of Shortest Job First (SJF) algorithm on another instance I' that is obtained from I by dividing each job into logarithmically many jobs with geometrically increasing sizes. Finally, the performance of SJF is related to SRPT using another $(1+\epsilon)$ factor speedup.

Bansal and Pruhs [2] considered the problem of minimizing the ℓ_p norms of flow time and stretch on a single machine. They showed the following.

Theorem 3 ([2]) *In the clairvoyant setting, SRPT and SJF are $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive for minimizing the ℓ_p norms of both flowtime and stretch. On the other hand, for $1 < p < \infty$, no online algorithm (possibly clairvoyant) can be O(1) competitive for minimizing ℓ_p norms of stretch or flow time without speedup. In particular, any randomized online algorithm is at least $\Omega(n^{(p-1)/3p^2})$-competitive for ℓ_p norms of stretch, and is at least $\Omega(n^{(p-1)/p(3p-1)})$-competitive for ℓ_p norms of flow time.*

The above lower bounds are somewhat surprising, since SRPT and FIFO are optimum for the case $p = 1$ and $p = \infty$ for flow time.

Bansal and Pruhs [2] also consider the non-clairvoyant case.

Theorem 4 ([2]) *In the non-clairvoyant setting, SETF is $(1+\epsilon)$-speed, $O(1/\epsilon^{2+2/p})$-competitive for minimizing the ℓ_p norms of flow time. For minimizing ℓ_p norms of stretch, SETF is $(1+\epsilon)$-speed, $O(1/\epsilon^{3+1/p} \cdot \log^{1+1/p} P)$-competitive.*

Finally, Bansal and Pruhs also consider Round Robin (RR) or Processor Sharing that at any time splits the processor equally among the unfinished jobs. RR is considered to be an ideal fair strategy since it treats all unfinished jobs equally. However, they show that

Theorem 5 *For any $p \geq 1$, there is an $\epsilon > 0$ such that even with a $(1+\epsilon)$ times faster processor, RR is not $n^{o(1)}$ competitive for minimizing the ℓ_p norms of flow time. In particular, for $\epsilon < 1/2p$, RR is $(1+\epsilon)$-speed, $\Omega(n^{(1-2\epsilon p)/p})$-competitive. For ℓ_p norms of stretch, RR is $\Omega(n)$ competitive as is in fact any randomized non-clairvoyant algorithm.*

The results above have been extended in a couple of directions. Bansal and Pruhs [3] extend these results to *weighted* ℓ_p norms of flow time and stretch. Chekuri, Khanna, Kumar and Goel [4] have extended these results to the multiple machines case. Their algorithms are particularly elegant: Each job is assigned to some machine at random and all jobs at a particular machine are processed using SRPT or SETF (as applicable).

Applications

SETF and its variants such as MLF are widely used in operating systems [9,10]. Note that SETF is not really practical since each job could be preempted infinitely often. However, variants of SETF with fewer preemptions are quite popular.

Open Problems

It would be interesting to explore other notions of fairness in the dynamic scheduling setting. In particular, it would be interesting to consider algorithms that are both fair and have a good average performance.

An immediate open problem is whether the gap between $O(\log^2 P)$ and $\Omega(\log P)$ can be closed for minimizing the average stretch in the non-clairvoyant setting.

Cross References

- ► Flow Time Minimization
- ► Minimum Flow Time
- ► Multi-level Feedback Queues

Recommended Reading

1. Bansal, N., Dhamdhere, K., Könemann, J., Sinha, A.: Non-Clairvoyant Scheduling for Minimizing Mean Slowdown. Algorithmica **40**(4), 305–318 (2004)
2. Bansal, N., Pruhs, K.: Server scheduling in the Lp norm: a rising tide lifts all boat. In: Symposium on Theory of Computing, STOC, pp. 242–250 (2003)
3. Bansal, N., Pruhs, K.: Server scheduling in the weighted Lp norm. In: LATIN, pp. 434–443 (2004)
4. Chekuri, C., Goel, A., Khanna, S., Kumar, A.: Multi-processor scheduling to minimize flow time with epsilon resource augmentation. In: Symposium on Theory of Computing, STOC, pp. 363–372 (2004)
5. Kellerer, H., Tautenhahn, T., Woeginger, G.J.: Approximability and Nonapproximability Results for Minimizing Total Flow Time on a Single Machine. SIAM J. Comput. **28**(4), 1155–1166 (1999)
6. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. J. ACM **47**(4), 617–643 (2000)
7. Motwani, R., Phillips, S., Torng, E.: Non-Clairvoyant Scheduling. Theor. Comput. Sci. **130**(1), 17–47 (1994)
8. Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.: Online Scheduling to Minimize Average Stretch. SIAM J. Comput. **34**(2), 433–452 (2004)
9. Nutt, G.: Operating System Projects Using Windows NT. Addison Wesley, Reading (1999)

10. Tanenbaum, A.S.: Modern Operating Systems. Prentice-Hall Inc., Englewood Cliffs (1992)

Shortest Path

- ▶ Algorithms for Spanners in Weighted Graphs
- ▶ All Pairs Shortest Paths via Matrix Multiplication
- ▶ Maximum-scoring Segment with Length Restrictions
- ▶ Routing in Road Networks with Transit Nodes

Shortest Paths Approaches for Timetable Information

2004; Pyrga, Schulz, Wagner, Zaroliagis

Riko Jacob
Institute of Computer Science,
Technical University of Munich, Munich, Germany

Keywords and Synonyms

Passenger information system; Timetable lookup; Journey planner; Trip planner

Problem Definition

Consider the route-planning task for passengers of scheduled public transportation. Here, the running example is that of a train system, but the discussion applies equally to bus, light-rail and similar systems. More precisely, the task is to construct a timetable information system that, based upon the detailed schedules of all trains, provides passengers with good itineraries, including the transfer between different trains.

Solutions to this problem consist of a model of the situation (e. g. can queries specify a limit on the number of transfers?), an algorithmic approach, its mathematical analysis (does it always return the best solution? Is it guaranteed to work fast in all settings?), and an evaluation in the real world (Can travelers actually use the produced itineraries? Is an implementation fast enough on current computers and real data?).

Key Results

The problem is discussed in detail in a recent survey article [6].

Modeling

In a simplistic model, it is assumed that a transfer between trains does not take time. A more realistic model specifies a certain minimum transfer time per station. Furthermore, the objective of the optimization problem needs to be defined. Should the itinerary be as fast as possible, or as cheap as possible, or induce the least possible transfers? There are different ways to resolve this as surveyed in [6], all originating in multi-objective optimization, like resource constraints or Pareto-optimal solutions. From a practical point of view, the preferences of a traveler are usually difficult to model mathematically, and one might want to let the user choose the best option among a set of reasonable itineraries himself. For example, one can compute all itineraries that are not inferior to some other itinerary in all considered aspects. As it turns out, in real timetables the number of such itineraries is not too big, such that this approach is computationally feasible and useful for the traveler [5]. Additionally, the fare structure of most railways is fairly complicated [4], mainly because fares usually are not additive, i. e., are not the sum of fares of the parts of a trip.

Algorithmic Models

The current literature establishes two main ideas how to transform the situation into a shortest path problem on a graph. As an example, consider the simplistic modeling where transfer takes no time, and where queries specify starting time and station to ask for an itinerary that achieves the earliest arrival time at the destination.

In the time-expanded model [11], every arrival and departure event of the timetable is a vertex of the directed graph. The arcs of the graph represent consecutive events at one station, and direct train connections. The length of an arc is given by the time difference of its end vertices. Let s be the vertex at the source station whose time is directly after the starting time. Now, a shortest path from s to any vertex of the destination station is an optimal itinerary.

In the time-dependent model [3,7,9,10], the vertices model stations, and the arcs stand for the existence of a direct (non-stop) train connection. Instead of edge length, the arcs are labeled with edge-traversal functions that give the arrival time at the end of the arc in dependence on the time a passenger starts at the beginning of the arc, reflecting the times when trains actually run. To solve this time-dependent shortest path problem, a modification of Dijkstra's algorithm can be used. Further exploiting the structure of this situation, the graph can be represented in a way that allows constant time evaluation of the link traversal functions [3]. To cope with more realistic transfer models, a more complicated graph can be used.

Additionally, many of the speed-up techniques for shortest path computations can be applied to the resulting graph queries.

Applications

The main application are timetable information systems for scheduled transit (buses, trains, etc.). This extends to route planning where trips in such systems are allowed, as for example in the setting of fine-grained traffic simulation to compute fastest itineraries [2].

Open Problems

Improve computation speed, in particular for fully integrated timetables and the multi-criteria case. Extend the problem to the dynamic case, where the current real situation is reflected, i. e., delayed or canceled trains, and otherwise temporarily changed timetables are reflected.

Experimental Results

In the cited literature, experimental results usually are part of the contribution [2,4,5,6,7,8,9,10,11]. The time-dependent approach can be significantly faster than the time-expanded approach. In particular for the simplistic models speed-ups in the range 10–45 are observed [8,10]. For more detailed models, the performance of the two approaches becomes comparable [6].

Cross References

- ► Implementation Challenge for Shortest Paths
- ► Routing in Road Networks with Transit Nodes
- ► Single-Source Shortest Paths

Acknowledgments

I want to thank Matthias Müller-Hannemann, Dorothea Wagner, and Christos Zaroliagis for helpful comments on an earlier draft of this entry.

Recommended Reading

1. Gerards, B., Marchetti-Spaccamela, A. (eds.): Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03) 2003. Electronic Notes in Theoretical Computer Science, vol. 92. Elsevier (2004)
2. Barrett, C.L., Bisset, K., Jacob, R., Konjevod, G., Marathe, M.V.: Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In: Algorithms – ESA 2002: 10th Annual European Symposium, Rome, Italy, 17–21 September 2002. Lecture Notes Computer Science, vol. 2461, pp. 126–138. Springer, Berlin (2002)
3. Brodal, G.S., Jacob, R.: Time-dependent networks as models to achieve fast exact time-table queries. In: Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03), 2003, [1], pp. 3–15
4. Müller-Hannemann, M., Schnee, M.: Paying less for train connections with MOTIS. In: Kroon, L.G., Möhring, R.H. (eds.) Proceedings of the 5th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'05), Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany 2006. Dagstuhl Seminar Proceedings, no. 06901
5. Müller-Hannemann, M., Schnee, M.: Finding all attractive train connections by multi-criteria pareto search. In: Geraets, F., Kroon, L.G., Schöbel, A., Wagner, D., Zaroliagis, C.D. (eds.) Algorithmic Methods for Railway Optimization, International Dagstuhl Workshop, Dagstuhl Castle, Germany, June 20–25, 2004, 4th International Workshop, ATMOS 2004, Bergen, September 16–17, 2004, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4359, pp. 246–263. Springer, Berlin (2007)
6. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.D.: Timetable information: Models and algorithms. In: Geraets, F., Kroon, L.G., Schöbel, A., Wagner, D., Zaroliagis, C.D. (eds.) Algorithmic Methods for Railway Optimization, International Dagstuhl Workshop, Dagstuhl Castle, Germany, June 20–25, 2004, 4th International Workshop, ATMOS 2004, Bergen, September 16–17, 2004, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4359, pp. 67–90. Springer (2007)
7. Nachtigall, K.: Time depending shortest-path problems with applications to railway networks. Eur. J. Oper. Res. **83**, 154–166 (1995)
8. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Experimental comparison of shortest path approaches for timetable information. In: Proceedings 6th Workshop on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics, 2004, pp. 88–99
9. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Towards realistic modeling of time-table information through the time-dependent approach. In: Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03), 2003, [1], pp. 85–103
10. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient models for timetable information in public transportation systems. J. Exp. Algorithmics **12**, 2.4 (2007)
11. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. J. Exp. Algorithmics **5** 1–23 (2000)

Shortest Paths in Planar Graphs with Negative Weight Edges

2001; Fakcharoenphol, Rao

Jittat Fakcharoenphol[1], Satish Rao[2]
[1] Department of Computer Engineering, Kasetsart University, Bangkok, Thailand
[2] Department of Computer Science, University of California at Berkeley, Berkeley, CA, USA

Keywords and Synonyms

Shortest paths in planar graphs with general arc weights; Shortest paths in planar graphs with arbitrary arc weights

Problem Definition

This problem is to find shortest paths in planar graphs with general edge weights. It is known that shortest paths exist only in graphs that contain no negative weight cycles. Therefore, algorithms that work in this case must deal with the presence of negative cycles, i. e., they must be able to detect negative cycles.

In general graphs, the best known algorithm, the Bellman-Ford algorithm, runs in time $O(mn)$ on graphs with n nodes and m edges, while algorithms on graphs with no negative weight edges run much faster. For example, Dijkstra's algorithm implemented with the Fibonacchi heap runs in time $O(m + n \log n)$, and, in case of integer weights Thorup's algorithm runs in linear time. Goldberg [5] also presented an $O(m\sqrt{n} \log L)$-time algorithm where L denotes the absolute value of the most negative edge weights. Note that his algorithm is weakly polynomial.

Notations

Given a directed graph $G = (V, E)$ and a weight function $w\colon E \to \mathbb{R}$ on its directed edges, a *distance labeling* for a source node s is a function $d\colon V \to \mathbb{R}$ such that $d(v)$ is the minimum length over all s-to-v paths, where the *length of path* P is $\sum_{e \in P} w(e)$.

Problem 1 (Single-Source-Shortest-Path)
INPUT: *A directed graph* $G = (V, E)$, *weight function* $w\colon E \to \mathbb{R}$, *source node* $s \in V$.
OUTPUT: *If* G *does not contain negative length cycles, output a distance labeling* d *for source node* s. *Otherwise, report that the graph contains some negative length cycle.*

The algorithm by Fakcharoenphol and Rao [4] deals with the case when G is planar. They gave an $O(n \log^3 n)$-time algorithm, improving on an $O(n^{3/2})$-time algorithm by Lipton, Rose, and Tarjan [9] and an $O(n^{4/3} \log nL)$-time algorithm by Henzinger, Klein, Rao, and Subramanian [6].

Their algorithm, as in all previous algorithms, uses a recursive decomposition and constructs a data structure called a dense distance graph, which shall be defined next.

A *decomposition* of a graph is a set of subsets P_1, $P_2, \ldots, P_k$ (not necessarily disjoint) such that the union of all the sets is V and for all $e = (u, v) \in E$, there is a unique P_i that contains e. A node v is a *border node* of a set P_i if $v \in P_i$ and there exists an edge $e = (v, x)$ where $x \notin P_i$. The subgraph induced on a subset P_i is referred to as a *piece* of the decomposition.

The algorithm works with a *recursive decomposition* where at each level, a piece with n nodes and r border nodes is divided into two subpieces such that each subpiece has no more than $2n/3$ nodes and at most $2r/3 + c\sqrt{n}$ border nodes, for some constant c. In this recursive context, a border node of a subpiece is defined to be any border node of the original piece or any new border node introduced by the decomposition of the current piece.

With this recursive decomposition, the *level of a decomposition* can be defined in the natural way, with the entire graph being the only piece in the level 0 decomposition, the pieces of the decomposition of the entire graph being the level 1 pieces in the decomposition, and so on.

For each piece of the decomposition, the all-pair shortest path distances between all its border nodes along paths that lie entirely inside the piece are recursively computed. These all-pair distances form the edge set of a non-planar graph representing shortest paths between border nodes. The dense distance graph of the planar graph is the union of these graphs over all the levels.

Using the dense distance graph, the shortest distance queries between pairs of nodes can be answered.

Problem 2 (Shortest-Path-Distance-Data-Structure)
INPUT: *A directed graph* $G = (V, E)$, *weight function* $w\colon E \to \mathbb{R}$, *source node* $s \in V$.
OUTPUT: *If* G *does not contain negative length cycles, output a data structure that support distance queries between pairs of nodes. Otherwise, report that the graph contains some negative length cycle.*

The algorithm of Fakcharoenphol and Rao relies heavily on planarity, i. e., it exploits properties regarding how shortest paths on each piece intersect. Therefore, unlike previous algorithms that require only that the graph can be recursively decomposed with small numbers of border nodes [10], their algorithm also requires that each piece has a nice embedding.

Given an embedding of the piece, a *hole* is a bounded face where all adjacent nodes are border nodes. Ideally, one would hope that there is a planar embedding of any piece in the recursive decomposition where all the border nodes are on a single face and are circularly ordered, i. e., there is no holes in each piece. Although this is not always true, the algorithm works with any decomposition with a constant number of holes in each piece. This decomposition can be found in $O(n \log n)$ time using the simple cycle separator algorithm by Miller [12].

Key Results

Theorem 1 *Given a recursive decomposition of a planar graph such that each piece of the decomposition contains at most a constant number of holes, there is an algorithm that constructs the dense distance graph is $O(n \log^3 n)$ time.*

Given the procedure that constructs the dense distance graph, the shortest paths from a source s can be computed by first adding s as a border node in every piece of the decomposition, computing the dense distance graph, and then extending the distances into all internal nodes on every piece. This can be done in time $O(n \log^3 n)$.

Theorem 2 *The single-source shortest path problem for an n-node planar graph with negative weight edges can be solved in time $O(n \log^3 n)$.*

The dense distance graph can be used to answer distance queries between pairs of nodes.

Theorem 3 *Given the dense distance graph, the shortest distance between any pair of nodes can be found in $O(\sqrt{n} \log^2 n)$ time.*

It can also be used as a dynamic data structure that answers shortest path queries and allows edge cost updates.

Theorem 4 *For planar graphs with only non-negative weight edges, there is a dynamic data structure that supports distance queries and update operations that change edge weights in amortized $O(n^{2/3} \log^{7/3} n)$ time per operation. For planar graph with negative weight edges, there is a dynamic data structures that supports the same set of operations in amortized $O(n^{4/5} \log^{13/5} n)$ time per operation.*

Note that the dynamic data structure does not support edge insertions and deletions, since these operations might destroy the recursive decomposition.

Applications

The shortest path problem has long been studied and continues to find applications in diverse areas. There are a many problems that reduce to the shortest path problem where negative weight edges are required, for example the minimum-mean length directed circuit. For planar graphs, the problem has wide application even when the underlying graph is a grid. For example, there are recent image segmentation approaches that use negative cycle detection [2,3]. Some of other applications for planar graphs include separator algorithms [13] and multi-source multi-sink flow algorithms [11].

Open Problems

Klein [8] gives a technique that improves the running time of the construction of the dense distance graph to $O(n \log^2 n)$ when all edge weights are non-negative; this also reduces the amortized running time for the dynamic case down to $O(n^{2/3} \log^{5/3} n)$. Also, for planar graphs with no negative weight edges, Cabello [1] gives a faster algorithm for computing the shortest distances between k pairs of nodes. However, the problem for improving the bound of $O(n \log^3 n)$ for finding shortest paths in planar graphs with general edge weights remains opened.

It is not known how to handle edge insertions and deletions in the dynamic data structure. A new data structure might be needed instead of the dense distance graph, because the dense distance graph is determined by the decomposition.

Cross References

- ▶ All Pairs Shortest Paths in Sparse Graphs
- ▶ All Pairs Shortest Paths via Matrix Multiplication
- ▶ Approximation Schemes for Planar Graph Problems
- ▶ Decremental All-Pairs Shortest Paths
- ▶ Fully Dynamic All Pairs Shortest Paths
- ▶ Implementation Challenge for Shortest Paths
- ▶ Negative Cycles in Weighted Digraphs
- ▶ Planarity Testing
- ▶ Shortest Paths Approaches for Timetable Information
- ▶ Single-Source Shortest Paths

Recommended Reading

1. Cabello, S.: Many distances in planar graphs. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 1213–1220. ACM Press, New York (2006)
2. Cox, I.J., Rao, S. B., Zhong, Y.: 'Ratio Regions': A Technique for Image Segmentation. In: Proceedings International Conference on Pattern Recognition, IEEE, pp. 557–564, August (1996)
3. Geiger, L.C.D., Gupta, A., Vlontzos, J.: Dynamic programming for detecting, tracking and matching elastic contours. IEEE Trans. On Pattern Analysis and Machine Intelligence (1995)
4. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. **72**, 868–889 (2006)
5. Goldberg, A.V.: Scaling algorithms for the shortest path problem. SIAM J. Comput. **21**, 140–150 (1992)
6. Henzinger, M.R., Klein, P.N., Rao, S., Subramanian, S.: Faster Shortest-Path Algorithms for Planar Graphs. J. Comput. Syst. Sci. **55**, 3–23 (1997)
7. Johnson, D.: Efficient algorithms for shortest paths in sparse networks. J. Assoc. Comput. Mach. **24**, 1–13 (1977)
8. Klein, P.N.: Multiple-source shortest paths in planar graphs. In: Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, pp. 146–155 (2005)

9. Lipton, R., Rose, D., Tarjan, R.E.: Generalized nested dissection. SIAM. J. Numer. Anal. **16**, 346–358 (1979)
10. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM. J. Appl. Math. **36**, 177–189 (1979)
11. Miller, G., Naor, J.: Flow in planar graphs with multiple sources and sinks. SIAM J. Comput. **24**, 1002–1017 (1995)
12. Miller, G.L.: Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci. **32**, 265–279 (1986)
13. Rao, S.B.: Faster algorithms for finding small edge cuts in planar graphs (extended abstract). In: Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing, pp. 229–240, May (1992)
14. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. J. ACM **51**, 993–1024 (2004)

Shortest Route

► All Pairs Shortest Paths in Sparse Graphs
► Rectilinear Steiner Tree
► Single-Source Shortest Paths

Shortest Vector Problem

1982; Lenstra, Lenstra, Lovasz

Daniele Micciancio
Department of Computer Science, University of California, San Diego, La Jolla, CA, USA

Keywords and Synonyms

Lattice basis reduction; LLL algorithm; Closest vector problem; Nearest vector problem; Minimum distance problem

Problem Definition

A *point lattice* is the set of all integer linear combinations

$$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_1, \ldots, x_n \in \mathbb{Z} \right\}$$

of n linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^m$ in m-dimensional Euclidean space. For computational purposes, the lattice vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ are often assumed to have integer (or rational) entries, so that the lattice can be represented by an integer matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{m \times n}$ (called *basis*) having the generating vectors as columns. Using matrix notation, lattice points in $\mathcal{L}(\mathbf{B})$ can be conveniently represented as $\mathbf{Bx}$ where $\mathbf{x}$ is an integer vector. The integers m and n are called the *dimension* and *rank* of the lattice respectively. Notice that any lattice admits multiple bases, but they all have the same rank and dimension.

The main computational problems on lattices are the *Shortest Vector Problem*, which asks to find the shortest nonzero vector in a given lattice, and the *Closest Vector Problem*, which asks to find the lattice point closest to a given target. Both problems can be defined with respect to any norm, but the Euclidean norm $\|\mathbf{v}\| = \sqrt{\sum_i v_i^2}$ is the most common. Other norms typically found in computer science applications are the ℓ_1 norm $\|\mathbf{v}\|_1 = \sum_i |v_i|$ and the *max* norm $\|\mathbf{v}\|_\infty = \max_i |v_i|$. This entry focuses on the Euclidean norm.

Since no efficient algorithm is known to solve SVP and CVP exactly in arbitrary high dimension, the problems are usually defined in their approximation version, where the approximation factor $\gamma \geq 1$ can be a function of the dimension or rank of the lattice.

Definition 1 (Shortest Vector Problem, SVP$_\gamma$) Given a lattice $\mathcal{L}(\mathbf{B})$, find a nonzero lattice vector $\mathbf{Bx}$ (where $\mathbf{x} \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$) such that $\|\mathbf{Bx}\| \leq \gamma \cdot \|\mathbf{By}\|$ for any $\mathbf{y} \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$.

Definition 2 (Closest Vector Problem, CVP$_\gamma$) Given a lattice $\mathcal{L}(\mathbf{B})$ and a target point $\mathbf{t}$, find a lattice vector $\mathbf{Bx}$ (where $\mathbf{x} \in \mathbb{Z}^n$) such that $\|\mathbf{Bx} - \mathbf{t}\| \leq \gamma \cdot \|\mathbf{By} - \mathbf{t}\|$ for any $\mathbf{y} \in \mathbb{Z}^n$.

Lattices have been investigated by mathematicians for centuries in the equivalent language of quadratic forms, and are the main object of study in the *geometry of numbers*, a field initiated by Minkowski as a bridge between geometry and number theory. For a mathematical introduction to lattices see [3]. The reader is referred to [6,12] for an introduction to lattices with an emphasis on computational and algorithmic issues.

Key Results

The problem of finding an efficient (polynomial time) solution to SVP$_\gamma$ for lattices in arbitrary dimension was first solved by the celebrated *lattice reduction* algorithm of Lenstra, Lenstra and Lovász [11], commonly known as the *LLL* algorithm.

Theorem 1 *There is a polynomial time algorithm to solve* SVP$_\gamma$ *for* $\gamma = (2/\sqrt{3})^n$, *where n is the rank of the input lattice.*

The LLL algorithm achieves more than just finding a relatively short lattice vector: it finds a so-called *reduced basis* for the input lattice, i. e., an entire basis of relatively short lattice vectors. Shortly after the discovery of the LLL algorithm, Babai [2] showed that reduced bases can be used to efficiently solve CVP$_\gamma$ as well within similar approximation factors.

Corollary 1 *There is a polynomial time algorithm to solve* CVP_γ *for* $\gamma = O(2/\sqrt{3})^n$*, where* n *is the rank of the input lattice.*

The reader is referred to the original papers [2,11] and [12, chap. 2] for details. Introductory presentations of the LLL algorithm can also be found in many other texts, e. g., [5, chap. 16] and [15, chap. 27]. It is interesting to note that CVP is at least as hard as SVP (see [12, chap 2]) in the sense that any algorithm that solves CVP_γ can be efficiently adapted to solve SVP_γ within the same approximation factor.

Both SVP_γ and CVP_γ are known to be NP-hard in their exact ($\gamma = 1$) or even approximate versions for small values of γ, e. g., constant γ independent of the dimension. (See [13, chaps. 3 and 4] and [4,10] for the most recent results.) So, no efficient algorithm is likely to exist to solve the problems exactly in arbitrary dimension. For any fixed dimension n, both SVP and CVP can be solved exactly in polynomial time using an algorithm of Kannan [9]. However, the dependency of the running time on the lattice dimension is $n^{O(n)}$. Using randomization, exact SVP can be solved probabilistically in $2^{O(n)}$ time and space using the *sieving* algorithm of Ajtai, Kumar and Sivakumar [1].

As for approximate solutions, the LLL lattice reduction algorithm has been improved both in terms of running time and approximation guarantee. (See [14] and references therein.) Currently, the best (randomized) polynomial time approximation algorithm achieves approximation factor $\gamma = 2^{O(n \log\log n/\log n)}$.

Applications

Despite the large (exponential in n) approximation factor, the LLL algorithm has found numerous applications and lead to the solution of many algorithmic problems in computer science. The number and variety of applications is too large to give a comprehensive list. Some of the most representative applications in different areas of computer science are mentioned below.

The first motivating applications of lattice basis reduction were the solution of integer programs with a fixed number of variables and the factorization of polynomials with rationals coefficients. (See [11] [8], and [5, chap. 16].) Other classic applications are the solution of random instances of low-density subset-sum problems, breaking (truncated) linear congruential pseudorandom generators, simultaneous Diophantine approximation, and the disproof of Mertens' conjecture. (See [8] and [5, chap. 17].)

More recently, lattice basis reduction has been extensively used to solve many problems in cryptanalysis and coding theory, including breaking several variants of the RSA cryptosystem and the DSA digital signature algorithm, finding small solutions to modular equations, and list decoding of CRT (Chinese Reminder Theorem) codes. The reader is referred to [7,13] for a survey of recent applications, mostly in the area of cryptanalysis.

One last class of applications of lattice problems is the design of cryptographic functions (e. g., collision resistant hash functions, public key encryption schemes, etc.) based on the apparent intractability of solving SVP_γ within small approximation factors. The reader is referred to [12, chap. 8] and [13] for a survey of such applications, and further pointers to relevant literature. One distinguishing feature of many such lattice based cryptographic functions is that they can be proved to be hard to break *on the average*, based on a *worst-case* intractability assumption about the underlying lattice problem.

Open Problems

The main open problems in the computational study of lattices is to determine the complexity of approximate SVP_γ and CVP_γ for approximation factors $\gamma = n^c$ polynomial in the rank of the lattice. Specifically,

- Are there polynomial time algorithm that solve SVP_γ or CVP_γ for polynomial factors $\gamma = n^c$? (Finding such algorithms even for very large exponent c would be a major breakthrough in computer science.)
- Is there an $\epsilon > 0$ such that approximating SVP_γ or CVP_γ to within $\gamma = n^\epsilon$ is NP-hard? (The strongest known inapproximability results [4] are for factors of the form $n^{O(1/\log\log n)}$ which grow faster than any polylogarithmic function, but slower than any polynomial.)

There is theoretical evidence that for large polynomials factors $\gamma = n^c$, SVP_γ and CVP_γ are not NP-hard. Specifically, both problems belong to complexity class coAM for approximation factor $\gamma = O(\sqrt{n/\log n})$. (See [12, chap. 9].) So, the problems cannot be NP-hard within such factors unless the polynomial hierarchy PH collapses.

URL to Code

The LLL lattice reduction algorithm is implemented in most library and packages for computational algebra, e. g.,

- GAP (http://www.gap-system.org)
- LiDIA (http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/)
- Magma (http://magma.maths.usyd.edu.au/magma/)
- Maple (http://www.maplesoft.com/)
- Mathematica (http://www.wolfram.com/products/mathematica/index.html)
- NTL (http://shoup.net/ntl/).

NTL also includes an implementation of Block Korkine-Zolotarev reduction that has been extensively used for cryptanalysis applications.

Cross References

- ▶ Cryptographic Hardness of Learning
- ▶ Knapsack
- ▶ Learning Heavy Fourier Coefficients of Boolean Functions
- ▶ Quantum Algorithm for the Discrete Logarithm Problem
- ▶ Quantum Algorithm for Factoring
- ▶ Sphere Packing Problem

Recommended Reading

1. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the thirty-third annual ACM symposium on theory of computing – STOC 2001, Heraklion, Crete, Greece, July 2001, pp 266–275. ACM, New York (2001)
2. Babai, L.: On Lovasz' lattice reduction and the nearest lattice point problem. Combinatorica **6**(1), 1–13 (1986). Preliminary version in STACS 1985
3. Cassels, J.W.S.: An introduction to the geometry of numbers. Springer, New York (1971)
4. Dinur, I., Kindler, G., Raz, R., Safra, S.: Approximating CVP to within almost-polynomial factors is NP-hard. Combinatorica **23**(2), 205–243 (2003). Preliminary version in FOCS 1998
5. von zur Gathen, J., Gerhard, J.: Modern Comptuer Algebra, 2nd edn. Cambridge (2003)
6. Grotschel, M., Lovász, L., Schrijver, A.: Geometric algorithms and combinatorial optimization. Algorithms and Combinatorics, vol. 2, 2nd edn. Springer (1993)
7. Joux, A., Stern, J.: Lattice reduction: A toolbox for the cryptanalyst. J. Cryptolo. **11**(3), 161–185 (1998)
8. Kannan, R.: Annual reviews of computer science, vol. 2, chap. "Algorithmic geometry of numbers", pp. 231–267. Annual Review Inc., Palo Alto, California (1987)
9. Kannan, R.: Minkowski's convex body theorem and integer programming. Math. Oper. Res. **12**(3), 415–440 (1987)
10. Khot, S.: Hardness of Approximating the Shortest Vector Problem in Lattices. J. ACM **52**(5), 789–808 (2005). Preliminary version in FOCS 2004
11. Lenstra, A.K., Lenstra, Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math Ann. **261**, 513–534 (1982)
12. Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems: A Cryptographic Perspective. The Kluwer International Series in Engineering and Computer Science, vol. 671. Kluwer Academic Publishers, Boston, Massachusetts (2002)
13. Nguyen, P., Stern, J.: The two faces of lattices in cryptology. In: J. Silverman (ed.) Cryptography and lattices conference – CaLC 2001, Providence, RI, USA, March 2001. Lecture Notes in Computer Science, vol. 2146, pp. 146–180. Springer, Berlin (2001)
14. Schnorr, C.P.: Fast LLL-type lattice reduction. Inform. Comput. **204**(1), 1–25 (2006)
15. Vazirani, V.V.: Approximation Algorithms. Springer (2001)

Similarity between Compressed Strings

2005; Kim, Amir, Landau, Park

Jin Wook Kim[1], Amihood Amir[2], Gad M. Landau[3], Kunsoo Park[4]

[1] HM Research, Seoul, Korea

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

[3] Department of Computer Science, University of Haifa, Haifa, Israel

[4] School of Computer Science and Engineering, Seoul National University, Seoul, Korea

Keywords and Synonyms

Similarity between compressed strings; Compressed approximate string matching; Alignment between compressed strings

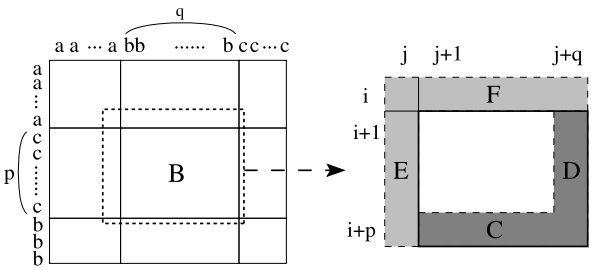
Problem Definition

The problem of computing similarity between two strings is concerned with comparing two strings using some scoring metric. There exist various scoring metrics and a popular one is the Levenshtein distance (or edit distance) metric. The standard solution for the Levenshtein distance metric was proposed by Wagner and Fischer [13], which is based on dynamic programming. Other widely used scoring metrics are the longest common subsequence metric, the weighted edit distance metric, and the affine gap penalty metric. The affine gap penalty metric is the most general, and it is a quite complicated metric to deal with. Table 1 shows the differences between the four metrics.

The problem considered in this entry is the similarity between two compressed strings. This problem is concerned with efficiently computing similarity without decompressing two strings. The compressions used for this

Similarity between Compressed Strings, Table 1
Various scoring metrics

Metric	Match	Mismatch	Indel	Indel of k characters
Longest common subsequence	1	0	0	0
Levenshtein distance	0	1	1	k
Weighted edit distance	0	δ	μ	$k\mu$
Affine gap penalty	1	$-\delta$	$-\gamma-\mu$	$-\gamma-k\mu$



Similarity between Compressed Strings, Figure 1
Dynamic programming table for strings $a^r c^p b^t$ and $a^s b^q c^u$ is divided into 9 blocks. For one of the blocks, e. g., *B*, only the bottom row *C* and the rightmost column *D* are computed from *E* and *F*

problem in the literature are run-length encoding and Lempel-Ziv (LZ) compression [14].

Run-Length Encoding

A string S is run-length encoded if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i", each consisting of an alphabet symbol, σ, and an integer, i. Each pair corresponds to a *run* in S, consisting of i consecutive occurrences of σ. For example, the string *aaabbbbbaccccbb* can be encoded $a^3b^4a^1c^4b^2$ or, equivalently, $(a,3)(b,4)(a,1)(c,4)(b,2)$. Let A and B be two strings with lengths n and m, respectively. Let A' and B' be the run-length encoded strings of A and B, and n' and m' be the lengths of A' and B', respectively.

Problem 1
INPUT: *Two run-length encoded strings A' and B', a scoring metric d.*
OUTPUT: *The similarity between A' and B' using d.*

LZ Compression

Let X and Y be two strings with length $O(n)$. Let X' and Y' be the LZ compressed strings of X and Y, respectively. Then the lengths of X' and Y' are $O(hn/\log n)$, where $h \leq 1$ is the entropy of strings X and Y.

Problem 2
INPUT: *Two LZ compressed strings X' and Y', a scoring metric d.*
OUTPUT: *The similarity between X' and Y' using d.*

Block Computation

To compute similarity between compressed strings efficiently, one can use a block computation method. Dynamic programming tables are divided into submatrices, which are called "*blocks*". For run-length encoded strings, a block is a submatrix made up of two runs – one of A and one of B. For LZ compressed strings, a block is a submatrix made up of two phrases – one phrase from each string. See [5] for more details. Then, blocks are computed from left to right and from top to bottom. For each block, only the bottom row and the rightmost column are computed. Figure 1 shows an example of block computation.

Key Results

The problem of computing similarity of two run-length encoded strings, A' and B', has been studied for various scoring metrics. Bunke and Csirik [4] presented the first solution to Problem 1 using the longest common subsequence metric. The algorithm is based on block computation of the dynamic programming table.

Theorem 1 (Bunke and Csirik 1995 [4]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the Levenshtein distance metric, Arbell, Landau, and Mitchell [2] and Mäkinen, Navarro, and Ukkonen [10] presented $O(nm' + n'm)$ time algorithms, independently. These algorithms are extensions of the algorithm of Bunke and Csirik.

Theorem 2 (Arbell, Landau, and Mitchell 2002 [2], Mäkinen, Navarro, and Ukkonen [10]) *The Levenshtein distance between run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the weighted edit distance metric, Crochemore, Landau, and Ziv-Ukelson [6] and Mäkinen, Navarro, and Ukkonen [11] gave $O(nm' + n'm)$ time algorithms using techniques completely different from each other. The algorithm of Crochemore, Landau, and Ziv-Ukelson [6] is based on the technique which is used in the LZ compressed pattern matching algorithm [6], and the algorithm of Mäkinen, Navarro, and Ukkonen [11] is an extension of the algorithm for the Levenshtein distance metric.

Theorem 3 (Crochemore, Landau, and Ziv-Ukelson 2003 [6] Mäkinen, Navarro, and Ukkonen [11]) *The weighted edit distance between run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the affine gap penalty metric, Kim, Amir, Landau, and Park [8] gave an $O(nm' + n'm)$ time algorithm. To compute similarity in this metric efficiently, the problem is converted into a path problem on a directed acyclic graph and some properties of maximum paths in this graph are used. It is not necessary to build the graph explicitly since they came up with new recurrences using the properties of the graph.

Theorem 4 (Kim, Amir, Landau, and Park 2005 [8]) *The similarity between run-length encoded strings A' and B' in the affine gap penalty metric can be computed in $O(nm' + n'm)$ time.*

The above results show that comparison of run-length encoded strings using the longest common subsequence metric is successfully extended to more general scoring metrics.

For the longest common subsequence metric, there exist improved algorithms. Apostolico, Landau, and Skiena [1] gave an $O(n'm' \log(n'm'))$ time algorithm. This algorithm is based on tracing specific optimal paths.

Theorem 5 (Apostolico, Landau, and Skiena 1999 [1]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(n'm' \log(n' + m'))$ time.*

Mitchell [12] obtained an $O((d + n' + m') \log(d + n' + m'))$ time algorithm, where d is the number of matches of compressed characters. This algorithm is based on computing geometric shortest paths using special convex distance functions.

Theorem 6 (Mitchell 1997 [12]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O((d + n' + m') \log(d + n' + m'))$ time, where d is the number of matches of compressed characters.*

Mäkinen, Navarro, and Ukkonen [11] conjectured an $O(n'm')$ time algorithm on average under the assumption that the lengths of the runs are equally distributed in both strings.

Conjecture 1 (Mäkinen, Navarro, and Ukkonen 2003 [11]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(n'm')$ time on average.*

For Problem 2, Crochemore, Landau, and Ziv-Ukelson [6] presented a solution using the additive gap penalty metric. The additive gap penalty metric consists of 1 for match, $-\delta$ for mismatch, and $-\mu$ for indel, which is almost the same as the weighted edit distance metric.

Theorem 7 (Crochemore, Landau, and Ziv-Ukelson 1993 [6]) *The similarity between LZ compressed strings X' and Y' in the additive gap penalty metric can be computed in $O(hn^2/\log n)$ time, where $h \leq 1$ is the entropy of strings X and Y.*

Applications

Run-length encoding serves as a popular image compression technique, since many classes of images (e. g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels. Approximate matching on images can be a useful tool to handle distortions. Even a one-dimensional compressed approximate matching algorithm would be useful to speed up two-dimensional approximate matching allowing mismatches and even rotations [3,7,9].

Open Problems

The worst-case complexity of the problem is not fully understood. For the longest common subsequence metric, there exist some results whose time complexities are better than $O(nm' + n'm)$ to compute the similarity of two run-length encoded strings [1,11,12]. It remains open to extend these results to the Levenshtein distance metric, the weighted edit distance metric and the affine gap penalty metric.

In addition, for the longest common subsequence metric, it is an open problem to prove Conjecture 1.

Cross References

- ▶ Compressed Pattern Matching
- ▶ Local Alignment (with Affine Gap Weights)
- ▶ Sequential Approximate String Matching

Recommended Reading

1. Apostolico, A., Landau, G.M., Skiena, S.: Matching for Run Length Encoded Strings. J. Complex. **15**(1), 4–16 (1999)
2. Arbell, O., Landau, G.M., Mitchell, J.: Edit Distance of Run-Length Encoded Strings. Inf. Proc. Lett. **83**(6), 307–314 (2002)
3. Baeza-Yates, R., Navaro, G.: New Models and Algorithms for Multidimensional Approximate Pattern Matching. J. Discret. Algorithms **1**(1), 21–49 (2000)
4. Bunke, H., Csirik, H.: An Improved Algorithm for Computing the Edit Distance of Run Length Coded Strings. Inf. Proc. Lett. **54**, 93–96 (1995)
5. Crochemore, M., Landau, G.M., Schieber, B., Ziv-Ukelson, M.: Re-Use Dynamic Programming for Sequence Alignment: An Algorithmic Toolkit. In: Iliopoulos, C.S., Lecroq, T. (eds.) String Algorithmics, pp. 19–59. King's College London Publications, London (2005)
6. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. SIAM J. Comput. **32**(6), 1654–1673 (2003)
7. Fredriksson, K., Navarro, G., Ukkonen, E.: Sequential and Indexed Two-Dimensional Combinatorial Template Matching Allowing Rotations. Theor. Comput. Sci. **347**(1–2), 239–275 (2005)
8. Kim, J.W., Amir, A., Landau, G.M., Park, K.: Computing Similarity of Run-Length Encoded Strings with Affine Gap Penalty. In: Proc. 12th Symposium on String Processing and Information Retrieval (SPIRE'05). LNCS, vol. 3772, pp. 440–449 (2005)

9. Krithivasan, K., Sitalakshmi, R.: Efficient Two-Dimensional Pattern Matching in The Presence of Errors. Inf. Sci. **43**, 169–184 (1987)
10. Mäkinen, V., Navarro, G., Ukkonen, E.: Approximate Matching of Run-Length Compressed Strings. In: Proc. 12th Symposium on Combinatorial Pattern Matching (CPM'01). LNCS, vol. 2089, pp. 31–49 (2001)
11. Mäkinen, V., Navarro, G., Ukkonen, E.: Approximate Matching of Run-Length Compressed Strings. Algorithmica **35**, 347–369 (2003)
12. Mitchell, J.: A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings. Technical Report, Dept. of Applied Mathematics, SUNY Stony Brook (1997)
13. Wagner, R.A., Fischer, M.J.: The String-to-String correction Problem. J. ACM **21**(1), 168–173 (1974)
14. Ziv, J., Lempel, A.: Compression of Individual Sequences via Variable Rate Coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)

Single-Source Fully Dynamic Reachability

2005; Demetrescu, Italiano

CAMIL DEMETRESCU, GIUSEPPE F. ITALIANO
Department of Computer & Systems Science, University of Rome, Rome, Italy

Keywords and Synonyms

Single-source fully dynamic transitive closure

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. An 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.

Given a graph with n vertices and m edges, the *transitive closure* (or *reachability*) problem consists of building an $n \times n$ Boolean matrix M such that $M[x, y] = 1$ if and only if there is a directed path from vertex x to vertex y in the graph. The fully dynamic version of this problem can be defifined as follows:

Definition 1 (Fully dynamic reachability problem) The *fully dynamic reachability 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.
- `reachable`(x,y): return *true* if there is a directed path from vertex x to vertex y, and *false* otherwise.

This entry addresses the *single-source* version of the fully-dynamic reachability problem, where one is only interested in queries with a fixed source vertex s. The problem is defined as follows:

Definition 2 (Single-source fully dynamic reachability problem) The *fully dynamic single-source reachability 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.
- `reachable`(y): return *true* if there is a directed path from the source vertex s to vertex y, and *false* otherwise.

Approaches

A simple-minded solution to the problem of Definition would be to keep explicit reachability information from the source to all other vertices and update it by running any graph traversal algorithm from the source s after each insert or delete. This takes $O(m + n)$ time per operation, and then reachability queries can be answered in constant time.

Another simple-minded solution would be to answer queries by running a point-to-point reachability computation, without the need to keep explicit reachability information up to date after each insertion or deletion. This can be done in $O(m + n)$ time using any graph traversal algorithm. With this approach, queries are answered in $O(m + n)$ time and updates require constant time. Notice that the time required by the slowest operation is $O(m+n)$ for both approaches, which can be as high as $O(n^2)$ in the case of dense graphs.

The first improvement upon these two basic solutions is due to Demetrescu and Italiano, who showed how to support update operations in $O(n^{1.575})$ time and reachability queries in $O(1)$ time [1] in a directed acyclic graph. The result is based on a simple reduction of the single-source problem of Definition to the all-pairs problem of Definition. Using a result by Sankowski [2], the bounds above can be extended to the case of general directed graphs.

Key Results

This Section presents a simple reduction presented in [1] that allows it to keep explicit single-source reachability information up to date in subquadratic time per operation

in a directed graph subject to an intermixed sequence of edge insertions and edge deletions. The bounds reported in this entry were originally presented for the case of directed acyclic graphs, but can be extended to general directed graphs using the following theorem from [2]:

Theorem 1 *Given a general directed graph with n vertices, there is a data structure for the fully dynamic reachability problem that supports each insertion/deletion in $O(n^{1.575})$ time and each reachability query in $O(n^{0.575})$ time. The algorithm is randomized with one-sided error.*

The idea described in [1] is to maintain reachability information from the source vertex s to all other vertices explicitly by keeping a Boolean array R of size n such that $R[y] = 1$ if and only if there is a directed path from s to y. An instance D of the data structure for fully dynamic reachability of Theorem is also maintained. After each insertion or deletion, it is possible to update D in $O(n^{1.575})$ time and then rebuild R in $O(n \cdot n^{0.575}) = O(n^{1.575})$ time by letting $R[y] \leftarrow D.\texttt{reachable}(s,y)$ for each vertex y. This yields the following bounds for the single-source fully dynamic reachability problem:

Theorem 2 *Given a general directed graph with n vertices, there is a data structure for the single-source fully dynamic reachability problem that supports each insertion/deletion in $O(n^{1.575})$ time and each reachability query in $O(1)$ time.*

Applications

The graph reachability problem is particularly relevant to the field of databases for supporting transitivity queries on dynamic graphs of relations [3]. The problem also arises in many other areas such as compilers, interactive verification systems, garbage collection, and industrial robotics.

Open Problems

An important open problem is whether one can extend the result described in this entry to maintain fully dynamic single-source shortest paths in subquadratic time per operation.

Cross References

► Trade-Offs for Dynamic Graph Problems

Recommended Reading

1. Demetrescu, C., Italiano, G.: Trade-offs for fully dynamic reachability on dags: Breaking through the $O(n^2)$ barrier. J. Assoc. Comput. Machin. (JACM) **52**, 147–156 (2005)
2. 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)
3. 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

Single-Source Shortest Paths

1999; Thorup

Seth Pettie
Department of Computer Science,
University of Michigan, Ann Arbor, MI, USA

Keywords and Synonyms

Shortest route; Quickest route

Problem Definition

The *single source* shortest path problem (SSSP) is, given a graph $G = (V, E, \ell)$ and a *source* vertex $s \in V$, to find the shortest path from s to every $v \in V$. The difficulty of the problem depends on whether the graph is directed or undirected and the assumptions placed on the length function ℓ. In the most general situation $\ell : E \to \mathbb{R}$ assigns arbitrary (positive & negative) real lengths. The algorithms of Bellman-Ford and Edmonds [1,4] may be applied in this situation and have running times of roughly $O(mn)$,[1] where $m = |E|$ and $n = |V|$ are the number of edges and vertices. If ℓ assigns only *non-negative* real edge lengths then the algorithms of Dijkstra and Pettie-Ramachandran [4,14] may be applied on directed and undirected graphs, respectively. These algorithms include a *sorting bottleneck* and, in the worst case, take $\Omega(m + n \log n)$ time.[2]

A common assumption is that ℓ assigns *integer* edge lengths in the range $\{0, \ldots, 2^w - 1\}$ or $\{-2^{w-1}, \ldots, 2^{w-1} - 1\}$ and that the machine is a w-bit *word RAM*; that is, each edge length fits in one register. For general integer edge lengths the best SSSP algorithms improve on Bellman-Ford and Edmonds by a factor of roughly $\sqrt{n}$ [7]. For non-negative integer edge lengths the best SSSP algorithms are faster than Dijkstra and Pettie-Ramachandran

[1] Edmonds's algorithm works for undirected graphs and presumes that there are no negative length simple cycles.

[2] The [14] algorithm actually runs in $O(m + n \log \log n)$ time if the ratio of any two edge lengths is polynomial in n.

by up to a logarithmic factor. They are frequently based on integer priority queues [10].

Key Results

Thorup's primary result [17] is an optimal linear time SSSP algorithm for undirected graphs with integer edge lengths. This is the first and only linear time shortest path algorithm that does not make serious assumptions on the class of input graphs.

Theorem 1 *There is a SSSP algorithm for integer-weighted undirected graphs that runs in $O(m)$ time.*

Thorup avoids the sorting bottleneck inherent in Dijkstra's algorithm by precomputing (in linear time) a *component hierarchy*. The algorithm of [17] operates in a manner similar to Dijkstra's algorithm [4] but uses the component hierarchy to identify groups of vertices that can be visited in any order. In later work, Thorup [18] extended this approach to work when the edge lengths are floating-point numbers.[3]

Thorup's hierarchy-based approach has since been extended to directed and/or real-weighted graphs, and to solve the *all pairs* shortest path (APSP) problem [12,13,14]. The generalizations to related SSSP problems are summarized by below. See [12,13] for hierarchy-based APSP algorithms.

Theorem 2 *(Hagerup [9], 2000) A component hierarchy for a directed graph $G = (V, E, \ell)$, where $\ell : E \to \{0, \ldots, 2^w - 1\}$, can be constructed in $O(m \log w)$ time. Thereafter SSSP from any source can be computed in $O(m + n \log\log n)$ time.*

Theorem 3 *(Pettie and Ramachandran [14], 2005) A component hierarchy for an undirected graph $G = (V, E, \ell)$, where $\ell : E \to \mathbb{R}^+$, can be constructed in $O(m\alpha(m, n) + \min\{n \log\log r, n \log n\})$ time, where r is the ratio of the maximum-to-minimum edge length. Thereafter SSSP from any source can be computed in $O(m \log\alpha(m, n))$ time.*

The algorithms of Hagerup [9] and Pettie-Ramachandran [14] take the same basic approach as Thorup's algorithm: use some kind of component hierarchy to identify groups of vertices that can safely be visited in any order. However, the assumption of directed graphs [9] and real edge lengths [14] renders Thorup's hierarchy inapplicable or inefficient. Hagerup's component hierarchy is based on a directed analogue of the minimum spanning tree. The Pettie-Ramachandran algorithm enforces a certain degree of balance in its component hierarchy and, when computing SSSP, uses a specialized priority queue to take advantage of this balance.

[3]There is some flexibility in the definition of *shortest path* since floating-point addition is neither commutative nor associative.

Applications

Shortest path algorithms are frequently used as a subroutine in other optimization problems, such as flow and matching problems [1] and facility location [19]. A widely used commercial application of shortest path algorithms is finding efficient routes on road networks, e. g., as provided by Google Maps, MapQuest, or Yahoo Maps.

Open Problems

Thorup's SSSP algorithm [17] runs in linear time and is therefore optimal. The main open problem is to find a linear time SSSP algorithm that works on *real*-weighted *directed* graphs. For real-weighted undirected graphs the best running time is given in Theorem 3. For integer-weighted directed graphs the fastest algorithms are based on Dijkstra's algorithm (not Theorem 2) and run in $O(m\sqrt{\log\log n})$ time (randomized) and deterministically in $O(m + n \log\log n)$ time.

Problem 1 *Is there an $O(m)$ time SSSP algorithm for integer-weighted directed graphs?*

Problem 2 *Is there an $O(m) + o(n \log n)$ time SSSP algorithm for real-weighted graphs, either directed or undirected?*

The complexity of SSSP on graphs with positive & negative edge lengths is also open.

Experimental Results

Asano and Imai [2] and Pettie et al. [15] evaluated the performance of the hierarchy-based SSSP algorithms [14,17]. There have been a number of studies of SSSP algorithms on integer-weighted directed graphs; see [8] for the latest and references to many others. The trend in recent years is to find practical preprocessing schemes that allow for very quick point-to-point shortest path queries. See [3,11,16] for recent work in this area.

Data Sets

See [5] for a number of US and European road networks.

URL to Code

See [6] and [5].

Cross References

► All Pairs Shortest Paths via Matrix Multiplication

Recommended Reading

1. Ahuja, R.K., Magnati, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
2. Asano, Y., Imai, H.: Practical efficiency of the linear-time algorithm for the single source shortest path problem. J. Oper. Res. Soc. Jpn. **43**(4), 431–447 (2000)
3. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant shortest-path queries in road networks. In: Proceedings 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
5. Demetrescu, C., Goldberg, A.V., Johnson, D.: 9th DIMACS Implementation Challege—Shortest Paths. http://www.dis.uniroma1.it/~challenge9/ (2006)
6. Goldberg, A.V.: AVG Lab. http://www.avglab.com/andrew/
7. Goldberg, A.V.: Scaling algorithms for the shortest paths problem. SIAM J. Comput. **24**(3), 494–504 (1995)
8. Goldberg, A.V.: Shortest path algorithms: Engineering aspects. In: Proc. 12th Int'l Symp. on Algorithms and Computation (ISAAC). LNCS, vol. 2223, pp. 502–513. Springer, Berlin (2001)
9. Hagerup, T.: Improved shortest paths on the word RAM. In: Proc. 27th Int'l Colloq. on Automata, Languages, and Programming (ICALP). LNCS vol. 1853, pp. 61–72. Springer, Berlin (2000)
10. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proc. 43rd Symp. on Foundations of Computer Science (FOCS), 2002, pp. 135–144
11. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Proceedings 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007
12. Pettie, S.: On the comparison-addition complexity of all-pairs shortest paths. In: Proc. 13th Int'l Symp. on Algorithms and Computation (ISAAC), 2002, pp. 32–43
13. Pettie, S.: A new approach to all-pairs shortest paths on real-weighted graphs. Theor. Comput. Sci. **312**(1), 47–74 (2004)
14. Pettie, S., Ramachandran, V.: A shortest path algorithm for real-weighted undirected graphs. SIAM J. Comput. **34**(6), 1398–1431 (2005)
15. Pettie, S., Ramachandran, V., Sridhar, S.: Experimental evaluation of a new shortest path algorithm. In: Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX), 2002, pp. 126–142
16. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Proc. 14th European Symposium on Algorithms (ESA), 2006, pp. 804–816
17. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. J. ACM **46**(3), 362–394 (1999)
18. Thorup, M.: Floats, integers, and single source shortest paths. J. Algorithms **35** (2000)
19. Thorup, M.: Quick and good facility location. In: Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 178–185

Ski Rental Problem

1990; Karlin, Manasse, McGeogh, Owicki

Mark S. Manasse
Microsoft Research, Mountain View, CA, USA

Index Terms

Ski-rental problem, Competitive algorithms, Deterministic and randomized algorithms, On-line algorithms

Keywords and Synonyms

Oblivious adversaries, Worst-case approximation, Metrical task systems

Problem Definition

The ski rental problem was developed as a pedagogical tool for understanding the basic concepts in some early results in on-line algorithms.[1] The ski rental problem considers the plight of one consumer who, in order to socialize with peers, is forced to engage in a variety of athletic activities, such as skiing, bicycling, windsurfing, rollerblading, sky diving, scuba-diving, tennis, soccer, and ultimate Frisbee, each of which has a set of associated apparatus, clothing, or protective gear.

In all of these, it is possible either to purchase the accoutrements needed, or to rent them. For the purpose of this problem, it is assumed that one-time rental is less expensive than purchasing. It is also assumed that purchased items are durable, and suitable for reuse for future activities of the same type without further expense, until the items wear out (which occurs at the same rate for all users), are outgrown, become unfashionable, or are disposed of

[1] In the interest of full disclosure, the earliest presentations of these results described the problem as the wedding-tuxedo-rental problem. Objections were presented that this was a gender-biased name for the problem, since while groomsmen can rent their wedding apparel, bridesmaids usually cannot. A further complication, owing to the difficulty of instantaneously producing fitted garments or ski equipment outlined below, suggests that some complications could have been avoided by focusing on the dilemma of choosing between daily lift passes or season passes, although this leads to the pricing complexities of purchasing season passes well in advance of the season, as opposed to the higher cost of purchasing them at the mountain during the ski season. A similar problem could be derived from the question as to whether to purchase the daily newspaper at a newsstand or to take a subscription, after adding the challenge that one's peers will treat one contemptuously if one has not read the news on days on which they have.

to make room for other purchased items. The social consumer must make the decision to rent or buy for each event, although it is assumed that the consumer is sufficiently parsimonious as to abjure rental if already in possession of serviceable purchased equipment. Whether purchases are as easy to arrange as rentals, or whether some advance planning is required (to mount bindings on a ski, say) is a further detail considered in this problem. It is assumed that the social consumer has no particular independent interest in these activities, and engages in these activities only to socialize with peers who choose to engage in these activities disregarding the consumer's desires.

These putative peers are more interested in demonstrating the superiority of their financial acumen to that of the social consumer in question than they are in any particular activity. To that end, the social consumer is taunted mercilessly based on the ratio of his/her total expenses on rentals and purchases to theirs. Consequently, the peers endeavor to invite the social consumer to engage in events while they are costly to him/her, and once the activities are free to the social consumer, if continued activity would be costly to them, cease. But, to present an illusion of fairness, skis, both rented and purchased, have the same cost for the peers as they do for the social consumer in question. The ski rental problem takes a very restricted setting. It assumes that purchased ski equipment never needs replacement, and that there are no costs to a ski trip other than the skis (thus, no cost for the gasoline, for the lift and/or speeding tickets, for the hot chocolates during skiing, or for the après-ski liqueurs and meals). It is assumed that the social consumer experiences no physical disabilities preventing him/her from skiing, and has no impending restrictions to his/her participation in ski trips (obviously, a near-term-fatal illness or an anticipated conviction leading to confinement for life in a penitentiary would eliminate any potential interest in purchasing alpine equipment—when the ratio of purchase to rental exceeds the maximum need for equipment, one should always rent). It is assumed that the social consumer's peers have disavowed any interest in activities other than skiing, and that the closet, basement, attic, garage, or storage locker included in the social consumer's rent or mortgage (or necessitated by other storage needs) has sufficient capacity to hold purchased ski equipment without entailing the disposal of any potentially useful items. Bringing these complexities into consideration brings one closer to the hardware-based problems which initially inspired this work.

The impact of invitations issued with sufficient time allowed for purchasing skis, as well as those without, will be considered.

Given all of that, what ratio of expenses can the social consumer hope to attain? What ratio can the social consumer not expect to beat? These are the basic questions of competitive analysis.

The impact of keeping secrets from one's peers is further considered. Rather than a fixed strategy for when to purchase skis, the social consumer may introduce an element of chance into the process. If the peers are able to observe his/her ski equipment and notice when it changes from rented skis to purchased skis, and change their schedule for alpine recreation in light of this observation, randomness provides no advantages. If, on the other hand, the social consumer announces to the peers, in advance of the first trip, how he/she will decide when the time is right for purchasing skis, including any use of probabilistic techniques, and they then decide on the schedule for ski trips for the coming winter, a deterministic decision procedure generally produces a larger competitive ratio than does a randomized procedure.

Key Results

Given an unbounded sequence of skiing trips, one should eventually purchase skis if the cost of renting skis, r, is positive. In particular, let the cost of purchasing skis be some number $p \geq r$. If one never intends to make a purchase, one's cost for the season will be rn, where n is the number of ski trips in which one participates. If n exceeds p/r, one's cost will exceed the price of purchasing skis; as n continues to increase, the ratio of one's costs to those of one's peers increases to nr/p, which grows unboundedly with n, since your peers, knowing that n exceeds p/r, will have purchased skis prior to the first trip.

On the other hand, if one rushes out to purchase skis upon being told that the ski season is approaching, one's peers will decide that this season looks inopportune, and that skiing is passé, leaving their costs at zero, and one's costs at p, leaving an infinite ratio between one's costs and theirs; if one chooses to defer the purchase until after one's first ski trip, this produces the less unfavorable ratio p/r or $1 + p/r$, depending on whether the invitation left one time to purchase skis before the first trip or not.

Suppose one chooses, instead, to defer one's purchase until after one has made k rentals, but before ski trip $k + 1$. One's costs are then bounded by $kr + p$. After k ski trips, the cost to one's peers will be the lesser of kr and p (as one's peers will have decided whether to rent or buy for the season upon knowing one's plans, which in this case amounts to knowing k), for a ratio equal to the larger of $1 + kr/p$ and $1 + p/kr$. Were they to choose to terminate the activity earlier (so $n < k$), the ratio would be only the

greater of kr/p and 1, which is guaranteed to be less than the sum of the two—one's peers would be shirking their opportunity to make one's behavior look foolish were they to allow one to stop skiing prior to one's purchase of a pair of skis!

It is certain, since kr/p and p/kr are reciprocals, that one of them is at least equal to 1, ensuring that one will be compelled to spend at least twice as much as one's peers.

The analysis above applies to the case where ski trips are announced without enough warning to leave one time to buy skis. Purchases in that case are not instantaneous; in contrast, if one is able to purchase skis on demand, the cost to one's peers changes to the lesser of $(k+1)\,r$ and p. The overall results are not much different; the ratio choices become the larger of $1 + kr/p$ and $1 + \left(p - r\right)/\left((k+1)\,r\right)$.

When probabilistic algorithms are considered with oblivious frenemies (those who know the way in which random choices will affect one's purchasing decisions, but who do not take time to notice that one's skis are no longer marked with the name and phone number of a rental agency), one can appear more thrifty.

A randomized algorithm can be viewed as a distribution over deterministic algorithms. No good algorithm can purchase skis prior to the first invitation, lest it exhibit infinite regrettability (some positive cost compared to zero). A good algorithm must purchase skis by the time one's peers will have, otherwise one's cost ratio continues to increase with the number of ski trips. Moreover, the ratio should be the same after every ski trip; if not, then there is an earliest ratio not equal to the largest, and probabilities can be adjusted to change this earliest ratio to be closer to the largest while decreasing all larger ratios.

Consider, for example, the case of $p = 2r$, with purchases allowed at the time of an invitation. The best deterministic ratio in this case is 1.5. It is only necessary to choose a probability q, the probability of purchasing at the time of the first invitation. The cost after one trip is then $\left(1-q\right)r + 2qr = r\left(1+q\right)$, for a ratio of $1+q$, and after two trips the costs is $q\,(2r) + \left(1-q\right)(3r) = \left(3-q\right)r$, producing a ratio of $\left(3-q\right)/2$. Setting these to be equal yields $q = 1/3$, for a ratio of 4/3.

If insufficient time is allowed for purchases before skiing, the best deterministic ratio is 2. Purchasing after the first ski trip with probability q (and after the second with probability $1-q$) leads to expected costs of $\left(1-q\right)r + 3qr = r\left(1+2q\right)$ after the first trip, and $\left(1-q\right)(2+2)\,r + 3qr = r\left(4-q\right)$, leading to a ratio of $2 - q/2$. Setting $1 + 2q = 2 - q/2$ yields $q = 2/5$, for a ratio of 9/5.

More careful analysis, for which readers are referred to the references and the remainder of this volume, shows that the best achievable ratio approaches $\epsilon/(\epsilon-1) \approx 1.58197$ as p/r increases, approaching the limit from below if sufficient warning time is offered, and from above otherwise.

Applications

The primary initial results were directed towards problems of computer architecture; in particular, design questions for capacity conflicts in caches, and shared memory design in the presence of a shared communication channel. The motivation for these analyses was to find designs which would perform reasonably well on as-yet-unknown workloads, including those to be designed by competitors who may have chosen alternative designs which favor certain cases. While it is probably unrealistic to assume that precisely the least-desirable workloads will occur in ordinary practice, it is not unreasonable to assume that extremal workloads favoring either end of a decision will occur.

History and Further Reading

This technique of finding algorithms with bounded worst-case performance ratios is common in analyzing approximation algorithms. The initial proof techniques used for such analyses (the method of amortized analysis) were first presented by Sleator and Tarjan.

The reader is advised to consult the remainder of this volume for further extensions and applications of the principles of competitive on-line algorithms.

Cross References

- ▶ Algorithm DC-Tree for k Servers on Trees
- ▶ Metrical Task Systems
- ▶ Online List Update
- ▶ Online Paging and Caching
- ▶ Paging
- ▶ Work-Function Algorithm for k Servers

Recommended Reading

1. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive Snoopy Caching. Algorithmica **3**, 77–119 (1988) (Conference version: FOCS 1986, pp. 244–254)
2. Karlin, A.R., Manasse, M.S., McGeoch, L.A., Owicki, S.S.: Competitive Randomized Algorithms for Nonuniform Problems. Algorithmica **11**(6), 542–571 (1994) (Conference version: SODA 1990, pp. 301–309)
3. Reingold, N., Westbrook, J., Sleator, D.D.: Randomized Competitive Algorithms for the List Update Problem. Algorithmica **11**(1), 15–32 (1994) (Conference version included author Irani, S.: SODA 1991, pp. 251–260)

Slicing Floorplan Orientation

1983; Stockmeyer

Evangeline F. Y. Young
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

Keywords and Synonyms

Shape curve computation

Problem Definition

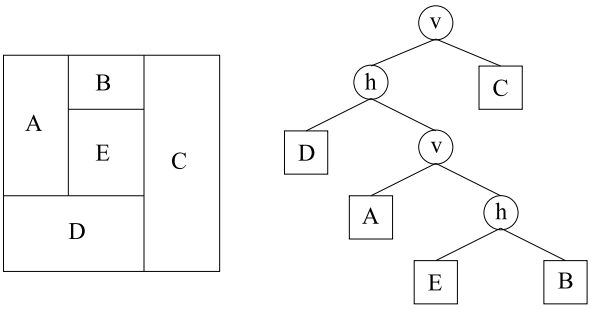
This problem is about finding the optimal orientations of the cells in a slicing floorplan to minimize the total area. In a floorplan, cells represent basic pieces of the circuit which are regarded as indivisible. After performing an initial placement, for example, by repeated application of a min-cut partitioning algorithm, the relative positions between the cells on a chip are fixed. Various optimization can then be done on this initial layout to optimize different cost measures such as chip area, interconnect length, routability, etc. One such optimization, as mentioned in Lauther [3], Otten [4], and Zibert and Saal [13], is to determine the best orientation of each cell to minimize the total chip area. This work by Stockmeyer [8] gives a polynomial time algorithm to solve the problem optimally in a special type of floorplans called *slicing floorplans* and shows that this orientation optimization problem in general non-slicing floorplans is NP-complete.

Slicing Floorplan

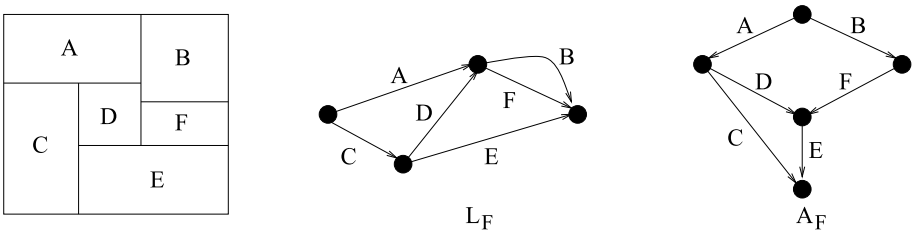
A floorplan consists of an enclosing rectangle subdivided by horizontal and vertical line segments into a set of non-overlapping *basic rectangles*. Two different line segments can meet but not cross. A floorplan F is characterized by a pair of planar acyclic directed graphs A_F and L_F defined as follows. Each graph has one source and one sink. The graph A_F captures the "above" relationships and has a vertex for each horizontal line segment, including the top and the bottom of the enclosing rectangle. For each basic rectangle R, there is an edge e_R directed from segment σ to segment σ' if and only if σ (or part of σ) is the top of R and σ' (or part of σ') is the bottom of R. There is a one-to-one correspondence between the basic rectangles and the edges in A_F. The graph L_F is defined similarly for the "left" relationships of the vertical segments. An example is shown in Fig. 1. Two floorplans F and G are equivalent if and only if $A_F = A_G$ and $L_F = L_G$. A floorplan F is slicing if and only if both its A_F and L_F are series parallel.

Slicing Tree

A slicing floorplan can also be described naturally by a rooted binary tree called *slicing tree*. In a slicing tree, each internal node is labeled by either an h or a v, indicating a horizontal or a vertical slice respectively. Each leaf corresponds to a basic rectangle. An example is shown in Fig. 2. There can be several slicing trees describing the same slicing floorplan but this redundancy can be removed by requiring the label of an internal node to differ from that of its right child [12]. For the algorithm presented in this work, a tree of smallest depth should be chosen and this depth minimization process can be done in $O(n \log n)$ time using the algorithm by Golumbic [2].



Slicing Floorplan Orientation, Figure 2
A slicing floorplan *F* and its slicing tree representation



Slicing Floorplan Orientation, Figure 1
A floorplan *F* and its A_F and L_F representing the above and left relationships

Orientation Optimization

In optimization of a floorplan layout, some freedom in moving the line segments and in choosing the dimensions of the rectangles are allowed. In the input, each basic rectangle R has two positive integers a_R and b_R, representing the dimensions of the cell that will be fit into R. Each cell has two possible orientations resulting in either the side of length a_R or b_R being horizontal. Given a floorplan F and an orientation ρ, each edge e in A_F and L_F is given a label $l(e)$ representing the height or the width of the cell corresponding to e depending on its orientation. Define an (F, ρ)-*placement* to be a labeling l of the vertices in A_F and L_F such that (i) the sources are labeled by zero, and (ii) if e is an edge from vertex σ to σ', $l(\sigma') \geq l(\sigma) + l(e)$. Intuitively, if σ is a horizontal segment, $l(\sigma)$ is the distance of σ from the top of the enclosing rectangle and the inequality constraint ensures that the basic rectangle corresponding to e is tall enough for the cell contained in it, and similarly for the vertical segments. Now, $h_F(\rho)$ (resp. $w_F(\rho)$) is defined to be the minimum label of the sink in $A_F(\rho)$ (resp. $L_F(\rho)$) over all (F, ρ)-placements, where $A_F(\rho)$ (resp. $L_F(\rho)$) is obtained from A_F (resp. L_F) by labeling the edges and vertices as described above. Intuitively, $h_F(\rho)$ and $w_F(\rho)$ give the minimum height and width of a floorplan F given an orientation ρ of all the cells such that each cell fits well into its associated basic rectangle. The orientation optimization problem can be defined formally as follows:

Problem 1 (Orientation Optimization Problem for Slicing Floorplan) INPUT: *A slicing floorplan F of n cells described by a slicing tree T, the widths and heights of the cells a_i and b_i for $i = 1 \ldots n$ and a cost function $\psi(h, w)$.*
OUTPUT: *An orientation ρ of all the cells that minimizes the objective function $\psi(h_F(\rho), w_F(\rho))$ over all orientations ρ.*

For this problem, Lauther [3] has suggested a greedy heuristic. Zibert and Saal [13] use integer programming methods to do rotation optimization and several other optimization simultaneously for general floorplans. In the following sections, an efficient algorithm will be given to solve the problem optimally in $O(nd)$ time where n is the number of cells and d is the depth of the given slicing tree.

Key Results

In the following algorithm, $F(u)$ denotes the floorplan described by the subtree rooted at u in the given slicing tree T and let $L(u)$ be the set of leaves in that subtree. For each node u of T, the algorithm constructs recursively a list of pairs:

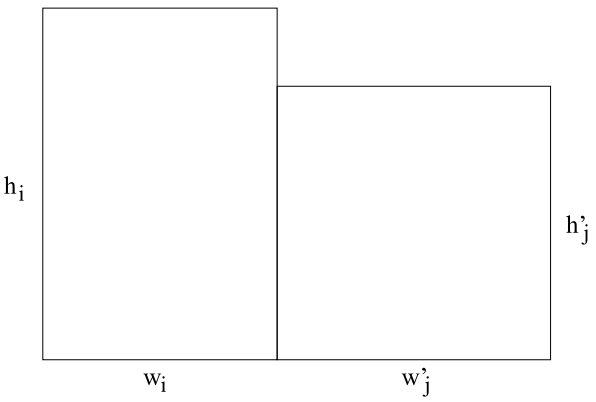
$$\{(h_1, w_1), (h_2, w_2), \ldots, (h_m, w_m)\}$$

where (1) $m \leq |L(u)| + 1$, (2) $h_i > h_{i+1}$ and $w_i < w_{i+1}$ for $i = 1 \ldots m - 1$, (3) there is an orientation ρ of the cells in $L(u)$ such that $(h_i, w_i) = (h_{F(u)}(\rho), w_{F(u)}(\rho))$ for each $i = 1 \ldots m$, and (4) for each orientation ρ of the cells in $L(u)$, there is a pair (h_i, w_i) in the list such that $h_i \leq h_{F(u)}(\rho)$ and $w_i \leq w_{F(u)}(\rho)$.

$L(u)$ is thus a non-redundant list of all possible dimensions of the floorplan described by the subtree rooted at u. Since the cost function ψ is non-decreasing, it can be minimized over all orientations by finding the minimum $\psi(h_i, w_i)$ over all the pairs (h_i, w_i) in the list constructed at the root of T. At the beginning, a list is constructed at each leaf node of T representing the possible dimensions of the cell. If a leaf cell has dimensions a and b with $a > b$, the list is $\{(a, b), (b, a)\}$. If $a = b$, there will just be one pair (a, b) in the list. (If the cell has a fixed orientation, there will also be just one pair as defined by the fixed orientation.) Notice that the condition (1) above is satisfied in these leaf node lists. The algorithm then works its way up the tree and constructs the list at each node recursively. In general, assume that u is an internal node with children v and v' and u represents a vertical slice. Let $\{(h_1, w_1) \ldots (h_k, w_k)\}$ and $\{(h'_1, w'_1) \ldots (h'_m, w'_m)\}$ be the lists at v and v' respectively where $k \leq |L(v)| + 1$ and $m \leq |L(v')| + 1$. A pair (h_i, w_i) from v can be put together by a vertical slice with a pair (h'_j, w'_j) from v' to give a pair:

$$join((h_i, w_i), (h'_j, w'_j)) = (\max(h_i, h'_j), w_i + w'_j)$$

in the list of u (see Fig. 3). The key fact is that most of the km pairs are sub-optimal and do not need to be considered. For example, if $h_i > h'_j$, there is no need to join



Slicing Floorplan Orientation, Figure 3
An illustration of the merging step

(h_i, w_i) with (h'_z, w'_z) for any $z > j$ since

$$\max(h_i, h'_z) = \max(h_i, h'_j) = h_i \,,$$
$$w_i + w'_z > w_i + w'_j$$

Similarly, if node u represents a horizontal slice, the *join* operation will be:

$$join((h_i, w_i), (h'_j, w'_j)) = (h_i + h'_j, \max(w_i, w'_j))$$

The algorithm also keeps two pointers for each element in the lists in order to construct back the optimal orientation at the end. The algorithm is summarized by the following pseudocode:

Pseudocode *Stockmeyer()*

1. Initialize the list at each leaf node.
2. Traverse the tree in postorder. At each internal node u with children v and v', construct a list at node u as follows:
3. Let $\{(h_1, w_1) \ldots (h_k, w_k)\}$ and $\{(h'_1, w'_1) \ldots (h'_m, w'_m)\}$ be the lists at v and v' respectively.
4. Initialize i and j to one.
5. If $i > k$ or $j > m$, the whole list at u is constructed.
6. Add $join((h_i, w_i), (h'_j, w'_j))$ to the list with pointers pointing to (h_i, w_i) and (h'_j, w'_j) in $L(v)$ and $L(v')$ respectively.
7. If $h_i > h'_j$, increment i by 1.
8. If $h_i < h'_j$, increment j by 1.
9. If $h_i = h'_j$, increment both i and j by 1.
10. Go to step 5.
11. Compute $\psi(h_i, w_i)$ for each pair (h_i, w_i) in the list L_r at the root r of T.
12. Return the minimum $\psi(h_i, w_i)$ for all (h_i, w_i) in L_r and construct back the optimal orientation by following the pointers.

Correctness

The algorithm is correct since at each node u, a list is constructed that records all the possible non-redundant dimensions of the floorplan described by the subtree rooted at u. This can be proved easily by induction starting from the leaf nodes and working up the tree recursively. Since the cost function ψ is non-decreasing, it can be minimized over all orientations of the cells by finding the minimum $\psi(h_i, w_i)$ over all the pairs (h_i, w_i) in the list L_r constructed at the root r of T.

Runtime

At each internal node u with children v and v'. If the lengths of the lists at v and v' are k and m respectively, the time spent at u to combine the two lists is $O(k + m)$. Each possible dimension of a cell will thus invoke one unit of execution time at each node on its path up to the root in the post-order traversal. The total runtime is thus $O(d \times N)$ where N is the total number of realizations of all the n cells, which is equal to $2n$ in the orientation optimization problem. Therefore, the runtime of this algorithm is $O(nd)$.

Theorem 1 *Let $\psi(h, w)$ be non-decreasing in both arguments, i. e., if $h \le h'$ and $w \le w'$, $\psi(h, w) \le \psi(h', w')$, and computable in constant time. For a slicing floorplan F described by a binary slicing tree T, the problem of minimizing $\psi(h_F(\rho), w_F(\rho))$ over all orientations ρ can be solved in time $O(nd)$ time, where n is the number of leaves of T (equivalently, the number of cells of F) and d is the depth of T.*

Applications

Floorplan design is an important step in the physical design of VLSI circuits. Stockmeyer's optimal orientation algorithm [8] has been generalized to solve the area minimization problem in slicing floorplans [7], in hierarchical non-slicing floorplans of order five [6,9] and in general floorplans [5]. The floorplan area minimization problem is similar except that each *soft cell* now has a number of possible realizations, instead of just two different orientations. The same technique can be applied immediately to solve optimally the area minimization problem for slicing floorplans in $O(nd)$ time where n is the total number of realizations of all the cells in a given floorplan F and d is the depth of the slicing tree of F. Shi [7] has further improved this result to $O(n \log n)$ time. This is done by storing the list of non-redundant pairs at each node in a balanced binary search tree structure called *realization tree* and using a new merging algorithm to combine two such trees to create a new one. It is also proved in [7] that this $O(n \log n)$ time complexity is the lower bound for this area minimization problem in slicing floorplans.

For hierarchical non-slicing floorplans, Pan et al. [6] prove that the problem is NP-complete. Branch-and-bound algorithms are developed by Wang and Wong [9], and pseudopolynomial time algorithms are developed by Wang and Wong [10], and Pan et al. [6]. For general floorplans, Stockmeyer [8] has shown that the problem is strongly NP-complete. It is therefore unlikely to have any pseudopolynomial time algorithm. Wimer et al. [11], and Chong and Sahni [1] propose branch-and-bound algorithms. Pan et al. [5] develop algorithms for general floorplans that are approximately slicing.

Recommended Reading

1. Chong, K., Sahni, S.: Optimal Realizations of Floorplans. In: IEEE Trans. Comput. Aided Des. **12**(6), 793–901 (1993)
2. Golumbic, M.C.: Combinatorial Merging. IEEE Trans. Comput. **C-25**, 1164–1167 (1976)
3. Lauther, U.: A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation. J. Digital Syst. **4**, 21–34 (1980)
4. Otten, R.H.J.M.: Automatic Floorplan Design. In: Proceedings of the 19th Design Automation Conference, pp. 261–267 (1982)
5. Pan, P., Liu, C.L.: Area Minimization for Floorplans. In: IEEE Trans. Comput. Aided Des. **14**(1), 123–132 (1995)
6. Pan, P., Shi, W., Liu, C.L.: Area Minimization for Hierarchical Floorplans. In: Algorithmica **15**(6), 550–571 (1996)
7. Shi, W.: A Fast Algorithm for Area Minimization of Slicing Floorplan. In: IEEE Trans. Comput. Aided Des. **15**(12), 1525–1532 (1996)
8. Stockmeyer, L.: Optimal Orientations of Cells in Slicing Floorplan Designs. Inf. Control **59**, 91–101 (1983)
9. Wang, T.C., Wong, D.F.: Optimal Floorplan Area Optimization. In: IEEE Trans. Comput. Aided Des. **11**(8), 992–1002 (1992)
10. Wang, T.C., Wong, D.F.: A Note on the Complexity of Stockmeyer's Floorplan Optimization Technique. In: Algorithmic Aspects of VLSI Layout, Lecture Notes Series on Computing, vol. 2, pp. 309–320 (1993)
11. Wimer, S., Koren, I., Cederbaum, I.: Optimal Aspect Ratios of Building Blocks in VLSI. IEEE Trans. Comput. Aided Des. **8**(2), 139–145 (1989)
12. Wong, D.F., Liu, C.L.: A New Algorithm for Floorplan Design. Proceedings of the 23rd ACM/IEEE Design Automation Conference, pp. 101–107 (1986)
13. Zibert, K., Saal, R.: On Computer Aided Hybrid Circuit Layout. Proceedings of the IEEE Intl. Symp. on Circuits and Systems, pp. 314–318 (1974)

Snapshots in Shared Memory

1993; Afek, Attiya, Dolev, Gafni, Merritt, Shavit

Eric Ruppert
Department of Computer Science and Engineering, York University, Toronto, ON, Canada

Keywords and Synonyms

Atomic scan

Problem Definition

Implementing a snapshot object is an abstraction of the problem of obtaining a consistent view of several shared variables while other processes are concurrently updating those variables.

In an asynchronous shared-memory distributed system, a collection of n processes communicate by accessing shared data structures, called *objects*. The system provides basic types of shared objects; other needed types must be built from them. One approach uses locks to guarantee exclusive access to the basic objects, but this approach is not fault-tolerant, risks deadlock or livelock, and causes delays when a process holding a lock runs slowly. Lock-free algorithms avoid these problems but introduce new challenges. For example, if a process reads two shared objects, the values it reads may not be consistent if the objects were updated between the two reads.

A *snapshot object* stores a vector of m values, each from some domain D. It provides two operations: scan and update(i, v), where $1 \leq i \leq m$ and $v \in D$. If the operations are invoked sequentially, an update(i, v) operation changes the value of the ith component of the stored vector to v, and a scan operation returns the stored vector.

Correctness when snapshot operations by different processes overlap in time is described by the *linearizability* condition, which says operations should appear to occur instantaneously. More formally, for every execution, one can choose an instant of time for each operation (called its *linearization point*) between the invocation and the completion of the operation. (An incomplete operation may either be assigned no linearization point or given a linearization point at any time after its invocation.) The responses returned by all completed operations in the execution must return the same result as they would if all operations were executed sequentially in the order of their linearization points.

An implementation must also satisfy a progress property. *Wait-freedom* requires that each process completes each scan or update in a finite number of its own steps. The weaker *non-blocking* progress condition says the system cannot run forever without some operation completing.

This article describes implementations of snapshots from more basic types, which are also linearizable, without locks. Two types of snapshots have been studied. In a *single-writer* snapshot, each component is owned by a process, and only that process may update it. (Thus, for single-writer snapshots, $m = n$.) In a *multi-writer* snapshot, any process may update any component. There also exist algorithms for *single-scanner* snapshots, where only one process may scan at a time [10,13,14,16]. Snapshots were introduced by Afek et al. [1], Anderson [2] and Aspnes and Herlihy [4].

Space complexity is measured by the number of basic objects used and their size (in bits). Time complexity is measured by the maximum number of steps a process must do to finish a scan or update, where a step is an access to a basic shared object. (Local computation and local memory accesses are usually not counted.) Complexity

bounds will be stated in terms of n, m, $d = \log |D|$ and k, the number of operations invoked in an execution. Ordinarily, there is no bound on k.

Most of the algorithms below use read-write registers, the most elementary shared object type. A *single-writer* register may only be written by one process. A *multi-writer* register may be written by any process. Some algorithms using stronger types of basic objects are discussed in Sect. "Wait-Free Implementations from Small, Stronger Objects".

Key Results

A Simple Non-blocking Implementation from Small Registers

Suppose each component of a single-writer snapshot object is represented by a single-writer register. Process i does an update(i, v) by writing v and a sequence number into register i, and incrementing its sequence number. Performing a scan operation is more difficult than merely reading each of the m registers, since some registers might change while these reads are done. To scan, a process repeatedly reads all the registers. A sequence of reads of all the registers is called a *collect*. If two collects return the same vector, the scan returns that vector (with the sequence numbers stripped away). The sequence numbers ensure that, if the same value is read in a register twice, the register had that value during the entire interval between the two reads. The scan can be assigned a linearization point between the two identical collects, and updates are linearized at the write. This algorithm is non-blocking, since a scan continues running only if at least one update operation is completed during each collect. A similar algorithm, with process identifiers appended to the sequence numbers, implements a non-blocking multi-writer snapshot from m multi-writer registers.

Wait-Free Implementations from Large Registers

Afek et al. [1] described how to modify the non-blocking single-writer snapshot algorithm to make it wait-free using scans embedded within the updates. An update(i, v) first does a scan and then writes a triple containing the scan's result, v and a sequence number into register i. While a process P is repeatedly performing collects to do a scan, either two collects return the same vector (which P can return) or P will eventually have seen three different triples in the register of some other process. In the latter case, the third triple that P saw must contain a vector that is the result of a scan that started after P's scan, so P's scan outputs that vector. Updates and scans that terminate after seeing two identical collects are assigned linearization points as before. If one scan obtains its output from an embedded scan, the two scans are given the same linearization point. This is a wait-free single-writer snapshot implementation from n single-writer registers of $(n + 1)d + \log k$ bits each. Operations complete within $O(n^2)$ steps. Afek et al. [1] also describe how to replace the unbounded sequence numbers with handshaking bits. This requires $n\Theta(nd)$-bit registers and n^2 1-bit registers. Operations still complete in $O(n^2)$ steps.

The same idea can be used to build multi-writer snapshots from multi-writer registers. Using unbounded sequence numbers yields a wait-free algorithm that uses m registers storing $\Theta(nd + \log k)$ bits each, in which each operation completes within $O(mn)$ steps. (This algorithm is given explicitly in [9].) No algorithm can use fewer than m registers if $n \geq m$ [9]. If handshaking bits are used instead, the multi-writer snapshot algorithm uses n^2 1-bit registers, $m(d + \log n)$-bit registers and n (md)-bit registers, and each operation uses $O(nm + n^2)$ steps [1].

Guerraoui and Ruppert [12] gave a similar wait-free multi-writer snapshot implementation that is anonymous, *i. e.*, it does not use process identifiers and all processes are programmed identically.

Anderson [3] gave an implementation of a multi-writer snapshot from a single-writer snapshot. Each process stores its latest update to each component of the multi-writer snapshot in the single-writer snapshot, with associated timestamp information computed by scanning the single-writer snapshot. A scan is done using just one scan of the single-writer snapshot. An update requires scanning and updating the single-writer snapshot twice. The implementation involves some blow-up in the size of the components, *i. e.*, to implement a multi-writer snapshot with domain D requires a single-writer snapshot with a much larger domain D'. If the goal is to implement multi-writer snapshots from single-writer registers (rather than multi-writer registers), Anderson's construction gives a more efficient solution than that of Afek et al.

Attiya, Herlihy and Rachman [7] defined the *lattice agreement* object, which is very closely linked to the problem of implementing a single-writer snapshot when there is a known upper bound on k. Then, they showed how to construct a single-writer snapshot (with no bound on k) from an infinite sequence of lattice agreement objects. Each snapshot operation accesses the lattice agreement object twice and does $O(n)$ additional steps. Their implementations of lattice agreement are discussed in Sect. "Wait-Free Implementations from Small, Stronger Objects".

Attiya and Rachman [8] used a similar approach to give a single-writer snapshot implementation from large single-writer registers using $O(n \log n)$ steps per operation. Each update has an associated sequence number. A scanner traverses a binary tree of height $\log k$ from root to leaf (here, a bound on k is required). Each node has an array of n single-writer registers. A process arriving at a node writes its current vector into a single-writer register associated with the node and then gets a new vector by combining information read from all n registers. It proceeds to the left or right child depending on the sum of the sequence numbers in this vector. Thus, all scanners can be linearized in the order of the leaves they reach. Updates are performed by doing a similar traversal of the tree. The bound on k can be removed as in [7]. Attiya and Rachman also give a more direct implementation that achieves this by recycling the snapshot object that assumes a bound on k. Their algorithm has also been adapted to solve condition-based consensus [15].

Attiya, Fouren and Gafni [6] described how to adapt the algorithm of Attiya and Rachman [8] so that the number of steps required to perform an operation depends on the number of processes that actually access the object, rather than the number of processes in the system.

Attiya and Fouren [5] solve lattice agreement in $O(n)$ steps. (Here, instead of using the terminology of lattice agreement, the algorithm is described in terms of implementing a snapshot in which each process does at most one snapshot operation.) The algorithm uses, as a data structure, a two-dimensional array of $O(n^2)$ *reflectors*. A reflector is an object that can be used by two processes to exchange information. Each reflector is built from two large single-writer registers. Each process chooses a path through the array of reflectors, so that at most two processes visit each reflector. Each reflector in column i is used by process i to exchange information with one process $j < i$. If process i reaches the reflector first, process j learns about i's update (if any). If process j reaches it first, then process i learns all the information that j has already gathered. (If both reach it at about the same time, both processes learn the information described above.) As the processes move from column $i-1$ to column i, a process that enters column i at some row r will have gathered all the information that has been gathered by any process that enters column i below row r (and possibly more). This invariant is maintained by ensuring that if process i passes information to any process $j < i$ in row r of column i, it also passes that information to all processes that entered column i above row r. Furthermore, process i exits column i at a row that matches the amount of information it learns while traveling through the column. When processes have reached the rightmost column of the array, the ones in higher rows know strictly more than the ones in lower rows. Thus, the linearization order of their scans is the order in which they exit the rightmost column, from bottom to top. The techniques of Attiya, Herlihy and Rachman [7,8], mentioned above, can be used to remove the restriction that each process performs at most one operation. The number of steps per operation is still $O(n)$.

Wait-Free Implementations from Small, Stronger Objects

All of the wait-free implementations described above use registers that can store $\Omega(m)$ bits each, and are therefore not practical when m is large. Some implementations from smaller objects equipped with stronger synchronization operations, rather than just reads and writes, are described in this section. An object is considered to be small if it can store $O(d + \log n + \log k)$ bits. This means that it can store a constant number of component values, process identifiers and sequence numbers.

Attiya, Herlihy and Rachman [7] gave an elegant divide-and-conquer recursive solution to the lattice agreement problem. The division of processes into groups for the recursion can be done dynamically using test&set objects. This provides a snapshot algorithm that runs in $O(n)$ time per operation, and uses $O(kn^2 \log n)$ small single-writer registers and $O(kn \log^2 n)$ test&set objects. (This requires modifying their implementation to replace those registers that are large, which are written only once, by many small registers.) Using randomization, each test&set object can be replaced by single-writer registers to give a snapshot implementation from registers only with $O(n)$ expected steps per operation.

Jayanti [13] gave a multi-writer snapshot implementation from $O(mn^2)$ small compare&swap objects where updates take $O(1)$ steps and scans take $O(m)$ steps. He began with a very simple single-scanner, single-writer snapshot implementation from registers that uses a secondary array to store a copy of recent updates. A scan clears that array, collects the main array, and then collects the secondary array to find any overlooked updates. Several additional mechanisms are introduced for the general, multi-writer, multi-scanner snapshot. In particular, compare&swap operations are used instead of writes to coordinate writers updating the same component and multiple scanners coordinate with one another to simulate a single scanner. Jayanti's algorithm builds on an earlier paper by Riany, Shavit and Touitou [16], which gave an implementation that achieved similar complexity, but only for a single-writer snapshot.

Applications

Applications of snapshots include distributed databases, storing checkpoints or backups for error recovery, garbage collection, deadlock detection, debugging distributed programmes and obtaining a consistent view of the values reported by several sensors. Snapshots have been used as building blocks for distributed solutions to randomized consensus and approximate agreement. They are also helpful as a primitive for building other data structures. For example, consider implementing a counter that stores an integer and provides increment, decrement and read operations. Each process can store the number of increments it has performed minus the number of its decrements in its own component of a single-writer snapshot object, and the counter may be read by summing the values from a scan. See [10] for references on many of the applications mentioned here.

Open Problems

Some complexity lower bounds are known for implementations from registers [9], but there remain gaps between the best known algorithms and the best lower bounds. In particular, it is not known whether there is an efficient wait-free implementation of snapshots from small registers.

Experimental Results

Riany, Shavit and Touitou gave performance evaluation results for several implementations [16].

Cross References

- ► Implementing Shared Registers in Asynchronous Message-Passing Systems
- ► Linearizability
- ► Registers

Recommended Reading

See also Fich's survey paper on the complexity of implementing snapshots [11].

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. Assoc. Comput. Mach. **40**, 873–890 (1993)
2. Anderson, J.H.: Composite registers. Distrib. Comput. **6**, 141–154 (1993)
3. Anderson, J.H.: Multi-writer composite registers. Distrib. Comput. **7**, 175–195 (1994)
4. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures, Crete, July 1990. pp. 340–349. ACM, New York, 1990
5. Attiya, H., Fouren, A.: Adaptive and efficient algorithms for lattice agreement and renaming. SIAM J. Comput. **31**, 642–664 (2001)
6. Attiya, H., Fouren, A., Gafni, E.: An adaptive collect algorithm with applications. Distrib. Comput. **15**, 87–96 (2002)
7. Attiya, H., Herlihy, M., Rachman, O.: Atomic snapshots using lattice agreement. Distrib. Comput. **8**, 121–132 (1995)
8. Attiya, H., Rachman, O.: Atomic snapshots in $O(n \log n)$ operations. SIAM J. Comput. **27**, 319–340 (1998)
9. Ellen, F., Fatourou, P., Ruppert, E.: Time lower bounds for implementations of multi-writer snapshots. J. Assoc. Comput. Mach. **54**(6) article 30 (2007)
10. Fatourou, P., Kallimanis, N.D.: Single-scanner multi-writer snapshot implementations are fast! In: Proc. 25th ACM Symposium on Principles of Distrib. Comput. Colorado, July 2006 pp. 228–237. ACM, New York (2006)
11. Fich, F.E.: How hard is it to take a snapshot? In: SOFSEM 2005: Theory and Practice of Computer Science. Liptovský Ján, January 2005, LNCS, vol. 3381, pp. 28–37. Springer (2005)
12. Guerraoui, R., Ruppert, E.: Anonymous and fault-tolerant shared-memory computing. Distrib. Comput. **20**(3) 165–177 (2007)
13. Jayanti, P.: An optimal multi-writer snapshot algorithm. In: Proc. 37th ACM Symposium on Theory of Computing. Baltimore, May 2005, pp. 723–732. ACM, New York (2005)
14. Kirousis, L.M., Spirakis, P., Tsigas, P.: Simple atomic snapshots: A linear complexity solution with unbounded time-stamps. Inf. Process. Lett. **58**, 47–53 (1996)
15. Mostéfaoui, A., Rajsbaum, S., Raynal, M., Roy, M.: Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. Distrib. Comput. **17**, 1–20 (2004)
16. Riany, Y., Shavit, N., Touitou, D.: Towards a practical snapshot algorithm. Theor. Comput. Sci. **269**, 163–201 (2001)

Sojourn Time

- ► Minimum Flow Time
- ► Shortest Elapsed Time First Scheduling

Sorting of Multi-Dimensional Keys

- ► String Sorting

Sorting Signed Permutations by Reversal (Reversal Distance)

2001; Bader, Moret, Yan

David A. Bader
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Keywords and Synonyms

Sorting by reversals; Inversion distance; Reversal distance

Problem Definition

This entry describes algorithms for finding the minimum number of steps needed to sort a signed permutation (also known as: inversion distance, reversal distance). This is a real-world problem and for example is used in computational biology.

Inversion distance is a difficult computational problem that has been studied intensively in recent years [1,4,6,7,8,9,10]. Finding the inversion distance between unsigned permutations is NP-hard [7], but with signed ones, it can be done in linear time [1].

Key Results

Bader et al. [1] present the first worst-case linear-time algorithm for computing the reversal distance that is simple and practical and runs faster than previous methods. Their key innovation is a new technique to compute connected components of the overlap graph using only a stack, which results in the simple linear-time algorithm for computing the inversion distance between two signed permutations. Bader et al. provide ample experimental evidence that their linear-time algorithm is efficient in practice as well as in theory: they coded it as well as the algorithm of Berman and Hannenhalli, using the best principles of algorithm engineering to ensure that both implementations would be as efficient as possible, and compared their running times on a large range of instances generated through simulated evolution.

Bafna and Pevzner introduced the cycle graph of a permutation [3], thereby providing the basic data structure for inversion distance computations. Hannenhalli and Pevzner then developed the basic theory for expressing the inversion distance in easily computable terms (number of breakpoints minus number of cycles plus number of hurdles plus a correction factor for a fortress [3,15]—hurdles and fortresses are easily detectable from a connected component analysis). They also gave the first polynomial-time algorithm for sorting signed permutations by reversals [9]; they also proposed a $O(n^4)$ implementation of their algorithm which runs in quadratic time when restricted to distance computation. Their algorithm requires the computation of the connected components of the overlap graph, which is the bottleneck for the distance computation. Berman and Hannenhalli later exploited some combinatorial properties of the cycle graph to give a $O(n\alpha(n))$ algorithm to compute the connected components, leading to a $O(n^2\alpha(n))$ implementation of the sorting algorithm [6], where α is the inverse Ackerman function. (The later Kaplan–Shamir–Tarjan (KST) algorithm [10] reduces the time needed to compute the shortest sequence of inversions, but uses the same algorithm for computing the length of that sequence.)

No algorithm that actually builds the overlap graph can run in linear time, since that graph can be of quadratic size. Thus, Bader's key innovation is to construct an *overlap forest* such that two vertices belong to the same tree in the forest exactly when they belong to the same connected component in the overlap graph. An overlap forest (the composition of its trees is unique, but their structure is arbitrary) has exactly one tree per connected component of the overlap graph and is thus of linear size. The linear-time step for computing the connected components scans the permutation twice. The first scan sets up a trivial forest in which each node is its own tree, labeled with the beginning of its cycle. The second scan carries out an iterative refinement of this first forest, by adding edges and so merging trees in the forest; unlike a Union-Find, however, this algorithm does not attempt to maintain the trees within certain shape parameters. This step is the key to Bader's linear-time algorithm for computing the reversal distance between signed permutations.

Applications

Some organisms have a single chromosome or contain single-chromosome organelles (such as mitochondria or chloroplasts), the evolution of which is largely independent of the evolution of the nuclear genome. Given a particular strand from a single chromosome, whether linear or circular, we can infer the ordering and directionality of the genes, thus representing each chromosome by an ordering of oriented genes. In many cases, the evolutionary process that operates on such single-chromosome organisms consists mostly of inversions of portions of the chromosome; this finding has led many biologists to reconstruct phylogenies based on gene orders, using as a measure of evolutionary distance between two genomes the inversion distance, i. e., the smallest number of inversions needed to transform one signed permutation into the other [11,12,14].

The linear-time algorithm is in wide-use (as it has been cited nearly 200 times within the first several years of its publication). Examples include the handling multichromosomal genome rearrangements [16], genome comparison [5], parsing RNA secondary structure [13], and phylogenetic study of the HIV-1 virus [2].

Open Problems

Efficient algorithms for computing minimum distances with weighted inversions, transpositions, and inverted transpositions, are open.

Experimental Results

Bader et al. give experimental results in [1].

URL to Code

An implementation of the linear-time algorithm is available as C code from www.cc.gatech.edu/~bader. Two other dominated implementations are available that are designed to compute the shortest sequence of inversions as well as its length; one, due to Hannenhalli that implements his first algorithm [9], which runs in quadratic time when computing distances, while the other, a Java applet written by Mantin (http://www.math.tau.ac.il/~rshamir/GR/) implements the KST algorithm [10], but uses an explicit representation of the overlap graph and thus also takes quadratic time. The implementation due to Hannenhalli is very slow and implements the original method of Hannenhalli and Pevzner and not the faster one of Berman and Hannenhalli. The KST applet is very slow as well since it explicitly constructs the overlap graph.

Cross References

For finding the actual sorting sequence, see the entry:

- ► Sorting Signed Permutations by Reversal (Reversal Sequence)

Recommended Reading

1. Bader, D.A., Moret, B.M.E., Yan, M.: A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. J. Comput. Biol. **8**(5), 483–491 (2001) An earlier version of this work appeared In: the Proc. 7th Int'l Workshop on Algorithms and Data Structures (WADS 2001)
2. Badimo, A., Bergheim, A., Hazelhurst, S., Papathanasopolous, M., Morris, L.: The stability of phylogenetic tree construction of the HIV-1 virus using genome-ordering data versus env gene data. In: Proc. ACM Ann. Research Conf. of the South African institute of computer scientists and information technologists on enablement through technology (SAICSIT 2003), vol. 47, pp. 231–240, Fourways, ACM, South Africa, September 2003
3. Bafna, V., Pevzner, P.A.: Genome rearrangements and sorting by reversals. In: Proc. 34th Ann. IEEE Symp. Foundations of Computer Science (FOCS93), pp. 148–157. IEEE Press (1993)
4. Bafna, V., Pevzner, P.A.: Genome rearrangements and sorting by reversals. SIAM J. Comput. **25**, 272–289 (1996)
5. Bergeron, A., Stoye, J.: On the similarity of sets of permutations and its applications to genome comparison. J. Comput. Biol. **13**(7), 1340–1354 (2006)
6. Berman, P., Hannenhalli, S.: Fast sorting by reversal. In: Hirschberg, D.S., Myers, E.W. (eds.) Proc. 7th Ann. Symp. Combinatorial Pattern Matching (CPM96). Lecture Notes in Computer Science, vol. 1075, pp. 168–185. Laguna Beach, CA, June 1996. Springer (1996)
7. Caprara, A.: Sorting by reversals is difficult. In: Proc. 1st Conf. Computational Molecular Biology (RECOMB97), pp. 75–83. ACM, Santa Fe, NM (1997)
8. Caprara, A.: Sorting permutations by reversals and Eulerian cycle decompositions. SIAM J. Discret. Math. **12**(1), 91–110 (1999)
9. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In: Proc. 27th Ann. Symp. Theory of Computing (STOC95), pp. 178–189. ACM, Las Vegas, NV (1995)
10. Kaplan, H., Shamir, R., Tarjan, R.E.: A faster and simpler algorithm for sorting signed permutations by reversals. SIAM J. Comput. **29**(3), 880–892 (1999) First appeared In: Proc.8th Ann. Symp. Discrete Algorithms (SODA97), pp. 344–351. ACM Press, New Orleans, LA
11. Olmstead, R.G., Palmer, J.D.: Chloroplast DNA systematics: a review of methods and data analysis. Am. J. Bot. **81**, 1205–1224 (1994)
12. Palmer, J.D.: Chloroplast and mitochondrial genome evolution in land plants. In: Herrmann, R. (ed.) Cell Organelles, pp. 99–133. Springer, Vienna (1992)
13. Rastegari, B., Condon, A.: Linear time algorithm for parsing RNA secondary structure. In: Casadio, R., Myers, E.: (eds.) Proc. 5th Workshop Algs. in Bioinformatics (WABI'05). Lecture Notes in Computer Science, vol. 3692, pp. 341–352. Springer, Mallorca, Spain (2005)
14. Raubeson, L.A., Jansen, R.K.: Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants. Science **255**, 1697–1699 (1992)
15. Setubal, J.C., Meidanis, J.: Introduction to Computational Molecular Biology. PWS, Boston, MA (1997)
16. Tesler, G.: Efficient algorithms for multichromosomal genome rearrangements. J. Comput. Syst. Sci. **63**(5), 587–609 (2002)

Sorting Signed Permutations by Reversal (Reversal Sequence)

2004; Tannier, Sagot

Eric Tannier
NRIA Rhone-Alpes, University of Lyon, Lyon, France

Keywords and Synonyms

Sorting by inversions

Problem Definition

A *signed permutation* π of size n is a permutation over $\{-n, \ldots, -1, 1 \ldots n\}$, where $\pi_{-i} = -\pi_i$ for all i.

The *reversal* $\rho = \rho_{i,j}$ $(1 \leq i \leq j \leq n)$ is an operation that reverses the order and flips the signs of the elements

$\pi_i, \ldots, \pi_j$ in a permutation π:

$$\pi \cdot \rho = (\pi_1, \ldots, \pi_{i-1}, -\pi_j, \ldots, -\pi_i, \pi_{j+1}, \ldots, \pi_n) .$$

If $\rho_1, \ldots, \rho_k$ is a sequence of reversals, it is said to *sort* a permutation π if $\pi \cdot \rho_1 \cdots \rho_k = Id$, where $Id = (1, \ldots, n)$ is the identity permutation. The length of a shortest sequence of reversals sorting π is called the *reversal distance* of π, and is denoted by $d(\pi)$.

If the computation of $d(\pi)$ is solved in linear time [2] (see the entry "reversal distance"), the computation of a sequence of size $d(\pi)$ that sorts π is more complicated and no linear algorithm is known so far. The best complexity is currently achieved by the solution of Tannier and Sagot [17], which has later been improved papers by Tannier, Bergeron and Sagot [18] and Han [8].

Key Results

Recall there is a linear algorithm to compute the reversal distance thanks to the formula $d(\pi) = n + 1 - c(\pi) + t(\pi)$ (notation from [4]), where $c(\pi)$ is the number of cycles in the breakpoint graph, and $t(\pi)$ is computed from the unoriented components of the permutation (see the entry "reversal distance"). Once this is known, there is a trivial algorithm that computes a sequence of size $d(\pi)$: try every possible reversal ρ at one step, until you find one such that $d(\pi \cdot \rho) = d(\pi) - 1$. Such a reversal is called *safe*. This necessitates $O(n)$ computations for every possible reversal (they are at most $(n + 1)(n + 2)/2 = O(n^2)$), and iterating this to find a sequence yields an $O(n^4)$ algorithm.

The first polynomial algorithm by Hannenhalli and Pevzner [9] was not achieving a better complexity and the algorithmic study of finding shortest sequences of reversals began its history.

The Scenario of Reversals

All the published solutions for the computations of a sorting sequence are divided into two, following the division of the distance formula into two parameters: a first part computes a sequence of reversals so that the resulting permutation has no unoriented component, and a second part sorts all oriented components.

The first part was given its best solution by Kaplan, Shamir and Tarjan [10], whose algorithm runs in linear time when coupled with the linear distance computation [2], and it is based on Hannenhalli and Pevzner's [9] early results.

The second part is the bottleneck of the whole procedure. At this point, if there is no unoriented component, the distance is $d(\pi) = n + 1 - c(\pi)$, so a safe reversal is one that increases $c(\pi)$ and do not create unoriented components (that would increase $t(\pi)$).

A reversal that increases $c(\pi)$ is called *oriented*. Finding an oriented reversal is an easy part: any two consecutive numbers that have different signs in the permutation define one. The hard part is to make sure it does not increase the number of unoriented components.

The quadratic algorithms designed on one side by Berman and Hannenhalli [5] and on the other by Kaplan, Shamir and Tarjan [10] are based on the linear recognition of safe reversals. No better algorithm is known so far to recognize safe reversals, and it seemed that a lower bound had been reached, as witnessed by a survey of Ozery-Flato and Shamir [14] in which they wrote that "a central question in the study of genome rearrangements is whether one can obtain a subquadratic algorithm for sorting by reversals". This was obtained by Tannier and Sagot [17], who proved that the recognition of safe reversal at each step is not necessary, but only the recognition of oriented reversals.

The algorithm is based on the following theorem, taken from [18]. A sequence of oriented reversals $\rho_1, \ldots, \rho_k$ is said to be *maximal* if there is no oriented reversal in $\pi \cdot \rho_1 \cdots \rho_k$. In particular a sorting sequence is maximal, but the converse is not true.

Theorem 1 *If S is a maximal but not a sorting sequence of oriented reversals for a permutation, then there exists a nonempty sequence S' of oriented reversals such that S may be split into two parts $S = S_1, S_2$, and S_1, S', S_2 is a sequence of oriented reversal.*

This allows to construct sequences of oriented reversals instead of safe reversals, and increase their size by adding reversals inside the sequence instead of at the end, and obtain a sorting sequence.

This algorithm, with a classical data structure to represent permutations (as an array for example) has still an $O(n^2)$ complexity, because at each step it has to test the presence of an oriented reversal, and apply it to the permutation.

The slight modification of a data structure invented by Kaplan and Verbin [11] allows to pick and apply an oriented reversal in $O(\sqrt{n \log n})$, and using this, Tannier and Sagot's algorithm achieves $O(n^{3/2}\sqrt{\log n})$ time complexity.

Recently, Han [8] announced another data structure that allows to pick and apply an oriented reversal in $O(\sqrt{n})$ time, and a similar slight modification can probably decrease the complexity of the overall method to $O(n^{3/2})$.

The Space of all Optimal Solutions

Almost all the studies on sorting sequences of reversals were devoted to giving only one sequence, though it has been remarked that there are often plenty of them (it may be over several millions even for $n \leq 10$). A few studies have tried to fill this deficiency.

An algorithm to enumerate all safe reversals at one step has been designed and implemented by Siepel [16]. A structure of the space of optimal solutions has been discovered by Chauve et al. [3], and the algorithmics related to this structure are studied in [6].

Applications

The motivation as well as the main application of this problem is in computational biology. Signed permutations are an adequate object to model the relative position and orientation of homologous blocks of DNA in two species. A generalization of this problem to multichromosomal models has been solved by and applied in mammalian genomes [15] to argue for a model of evolution where reversals do not occur randomly.

Ajana et al. [1] used a random exploration in the space of solutions to test the hypothesis that in bacteria, reversals occur mainly around an origin or terminus of replication.

Generalizations to the comparison of more than two genomes has been the subject of an abundant literature, and applied to reconstruct evolutionary events and the organization of the genomes of common ancestors of living species, or to infer gene orthology from their positions, and they are based on heuristic principles guided by the theory of sorting signed permutations by reversals [12,13].

Open Problems

- Finding a better complexity than $O(n^{3/2})$. It could be achieved by a smarter data structure, or changing the principle of the algorithm, so that there is no need to apply at each step a sorting reversal to be able to compute the next ones.
- The efficient representation and enumeration of the whole set of solutions (see some advances in [3,6]).
- Finding, among the solutions, the ones that fit some biological constraints, as preserving some common groups of genes or favoring small inversions (see some advances in [7]).

Experimental Results

The algorithm of Tannier, Bergeron and Sagot [18] has been implemented in its quadratic version (without any special data structure, which are probably worth only for very big sizes of permutations) by Diekmann (biomserv.univ-lyon1.fr/~tannier/PSbR/), but no implementation of the data structures nor experiments on the complexity are reported.

URL to Code

- www.cse.ucsd.edu/groups/bioinformatics/GRIMM/
  In Pevzner's group, Tesler has put online an implementation of the multicromosomal generalization of the algorithm of Kaplan, Shamir, and Tarjan [10], that he has called GRIMM, for "Genome Rearrangements In Man and Mouse".
- www.cs.unm.edu/~moret/GRAPPA/
  GRAPPA stands for "Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms". It contains the distance computation, and the algorithm to find all safe reversals at one step. It has been developed in Moret's team.
- www.math.tau.ac.il/~rshamir/GR/
  An applet written by Mantin implementing the algorithm of Kaplan, Shamir and Tarjan [10].
- biomserv.univ-lyon1.fr/~tannier/PSbR/
  A program by Diekmann to find a scenario of reversals with additional constraints for signed permutations, implementing the algorithm of Tannier and Sagot [17].
- www.geocities.com/mdvbraga/baobabLuna.html
  A program by Braga for the manipulation of permutations, and in particular sorting signed permutations by reversals, and giving a condensed representation of all optimal sorting sequences, implementing an algorithm of [6].

Cross References

► Sorting Signed Permutations by Reversal (Reversal Distance)

Recommended Reading

1. Ajana, Y., Lefebvre, J.-F., Tillier, E., El-Mabrouk, N.: Exploring the Set of All Minimal Sequences of Reversals – An Application to Test the Replication-Directed Reversal Hypothesis, Proceedings of the Second Workshop on Algorithms in Bioinformatics. Lecture Notes in Computer Science, vol. 2452, pp. 300–315. Springer, Berlin (2002)
2. Bader, D.A., Moret, B.M.E., Yan, M.: A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study. J. Comput. Biol. **8**(5), 483–491 (2001)
3. Bergeron, A., Chauve, C., Hartman, T., St-Onge, K.: On the properties of sequences of reversals that sort a signed permutation. Proceedings of JOBIM'02, 99–108 (2002)

4. Bergeron, A., Mixtacki, J., Stoye, J.: The inversion distance problem. In: Gascuel, O. (ed.) Mathematics of evolution and phylogeny. Oxford University Press, USA (2005)
5. Berman, P., Hannenhalli, S.: Fast Sorting by Reversal, proceedings of CPM '96. Lecture notes in computer science **1075**, 168–185 (1996)
6. Braga, M.D.V., Sagot, M.F., Scornavacca, C., Tannier, E.: The Solution Space of Sorting by Reversals. In: Proceedings of ISBRA'07. Lect. Notes Comp. Sci. **4463**, 293–304 (2007)
7. Diekmann, Y., Sagot, M.F., Tannier, E.: Evolution under Reversals: Parsimony and Conversation of Common Intervals. IEEE/ACM Transactions in Computational Biology and Bioinformatics, **4**, 301–309, 1075 (2007)
8. Han, Y.: Improving the Efficiency of Sorting by Reversals, Proceedings of The 2006 International Conference on Bioinformatics and Computational Biology. Las Vegas, Nevada, USA (2006)
9. Hannenhalli, S., Pevzner, P.: Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). J. ACM **46**, 1–27 (1999)
10. Kaplan, H., Shamir, R., Tarjan, R.E.: Faster and simpler algorithm for sorting signed permutations by reversals. SIAM J. Comput. **29**, 880–892 (1999)
11. Kaplan, H., Verbin, E.: Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In: Proceedings of CPM'03. Lecture Notes in Computer Science **2676**, 170–185
12. Moret, B.M.E., Tang, J., Warnow, T.: Reconstructing phylogenies from gene-content and gene-order data. In: Gascuel, O. (ed.) Mathematics of Evolution and Phylogeny. pp. 321–352, Oxford Univ. Press, USA (2005)
13. Murphy, W., et al.: Dynamics of Mammalian Chromosome Evolution Inferred from Multispecies Comparative Maps. Science **309**, 613–617 (2005)
14. Ozery-Flato, M., Shamir, R.: Two notes on genome rearrangement. J. Bioinf. Comput. Biol. **1**, 71–94 (2003)
15. Pevzner, P., Tesler, G.: Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution. PNAS **100**, 7672–7677 (2003)
16. Siepel, A.C.: An algorithm to enumerate sorting reversals for signed permutations. J. Comput. Biol. **10**, 575–597 (2003)
17. Tannier, E., Sagot, M.-F.: Sorting by reversals in subquadratic time. In: Proceedings of CPM'04. Lecture Notes Comput. Sci. **3109**, 1–13
18. Tannier, E., Bergeron, A., Sagot, M.-F.: Advances on Sorting by Reversals. Discret. Appl. Math. **155**, 881–888 (2006)

Sorting by Transpositions and Reversals (Approximate Ratio 1.5)

2004; Hartman, Sharan

Chin Lung Lu
Institute of Bioinformatics & Department of Biological Science and Technology, National Chiao Tung University, Hsinchu, Taiwan

Keywords and Synonyms

Genome rearrangements

Problem Definition

One of the most promising ways to determine evolutionary distance between two organisms is to compare the order of appearance of identical (e. g., orthologous) genes in their genomes. The resulting genome rearrangement problem calls for finding a shortest sequence of rearrangement operations that sorts one genome into the other. In this work [8], Hartman and Sharan provide a 1.5-approximation algorithm for the problem of sorting by transpositions, transreversals and revrevs, improving on a previous 1.75 ratio for this problem. Their algorithm is also faster than current approaches and requires $O(n^{3/2}\sqrt{\log n})$ time for n genes.

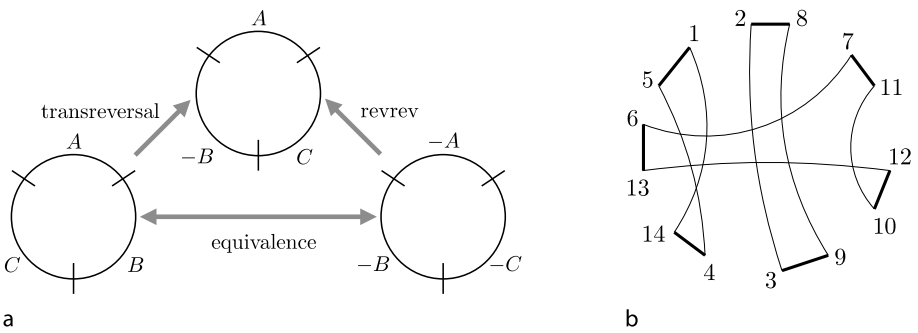
Notations and Definition

A *signed permutation* $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ on $n(\pi) \equiv n$ elements is a permutation in which each element is labeled by a sign of plus or minus. A *segment* of π is a sequence of consecutive elements $\pi_i, \pi_{i+1}, \ldots, \pi_k$, where $1 \le i \le k \le n$. A *reversal* ρ is an operation that reverses the order of the elements in a segment and also flips their signs. Two segments $\pi_i, \pi_{i+1}, \ldots, \pi_k$ and $\pi_j, \pi_{j+1}, \ldots, \pi_l$ are said to be *contiguous* if $j = k + 1$ or $i = l + 1$. A *transposition* τ is an operation that exchanges two contiguous (disjoint) segments. A *transreversal* $\tau\rho_{A,B}$ (respectively, $\tau\rho_{B,A}$) is a transposition that exchanges two segments A and B and also reverses A (respectively, B). A *revrev* operation $\rho\rho$ reverses each of the two contiguous segments (without transposing them). The problem of finding a shortest sequence of transposition, transreversal and revrev operations that transforms a permutation into the identity permutation is called *sorting by transpositions, transreversals and revrevs*. The *distance* of a permutation π, denoted by $d(\pi)$, is the length of the shortest sorting sequence.

Key Results

Linear vs. Circular Permutations

An operation is said to *operate* on the affected segments as well as on the elements in those segments. Two operations μ and μ' are *equivalent* if they have the same rearrangement result, i. e., $\mu \cdot \pi = \mu' \cdot \pi$ for all π. In this work [8], Hartman and Sharan showed that for an element x of a circular permutation π, if μ is an operation that operates on x, then there exists an equivalent oper-



Sorting by Transpositions and Reversals (Approximate Ratio 1.5), Figure 1

a The equivalence of transreversal and revrev on circular permutations. b The breakpoint graph $G(\pi)$ of the permutation $\pi = [1, -4, 6, -5, 2, -7, -3]$, for which $f(\pi) = [1, 2, 8, 7, 11, 12, 10, 9, 3, 4, 14, 13, 6, 5]$. It is convenient to draw $G(\pi)$ on a circle such that *black edges* (i. e., *thick lines*) are on the circumference and *gray edges* (i. e., *thin lines*) are chords

ation μ' that does not operate on x. Based on this property, they further proved that the problem of sorting by transpositions, transreversals and revrevs is equivalent for linear and circular permutations. Moreover, they observed that revrevs and transreversals are equivalent operations for circular permutations (as illustrated in Fig. 1a), implying that the problem of sorting a linear/circular permutation by transpositions, transreversals and revrevs can be reduced to that of sorting a circular permutation by transpositions and transreversals only.

The Breakpoint Graph

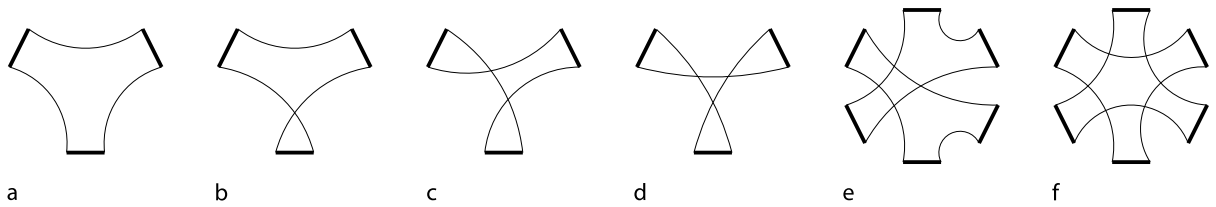
Given a signed permutation π on $\{1, 2, \ldots, n\}$ of n elements, it is transformed into an unsigned permutation $f(\pi) = \pi' = [\pi'_1, \pi'_2, \ldots, \pi'_{2n}]$ on $\{1, 2, \ldots, 2n\}$ of $2n$ elements by replacing each positive element i with two elements $2i - 1, 2i$ (in this order) and each negative element $-i$ with $2i, 2i - 1$. The extended $f(\pi)$ is considered here as a circular permutation by identifying $2n + 1$ and 1 in both indices and elements. To ensure that every operation on $f(\pi)$ can be mimicked by an operation on π, only operations that cut before odd position are allowed for $f(\pi)$. The *breakpoint graph* $G(\pi)$ is an edge-colored graph on $2n$ vertices $\{1, 2, \ldots, 2n\}$, in which for every $1 \le i \le n$, π'_{2i} is joined to π'_{2i+1} by a black edge and $2i$ is joined to $2i + 1$ by a gray edge (see Fig. 1b for an example). Since the degree of each vertex in $G(\pi)$ is exactly 2, $G(\pi)$ uniquely decomposes into cycles. A *k-cycle* (i. e., a cycle of *length* k) is a cycle with k black edges, and it is *odd* if k is odd. The number of odd cycles in $G(\pi)$ is denoted by $c_{\text{odd}}(\pi)$. It is not hard to verify that $G(\pi)$ consists of n 1-cycles and hence $c_{\text{odd}}(\pi) = n$, if π is an identity permutation $[1, 2, \ldots, n]$. Gu et al. [5] have shown that $c_{\text{odd}}(\mu \cdot \pi) \le c_{\text{odd}}(\pi) + 2$ for all linear permutations π and operations μ. In this work [8], Hartman and Sharan further noted that the above result holds also for circular permutations and proved that the lower bound of $d(\pi)$ is $(n(\pi) - c_{\text{odd}}(\pi))/2$.

Transformation into 3-Permutations

A permutation is called *simple* if its breakpoint graph contains only k-cycle, where $k \le 3$. A simple permutation is also called a *3-permutation* if it contains no 2-cycles. A transformation from π to $\hat{\pi}$ is said to be *safe* if $n(\pi) - c_{\text{odd}}(\pi) = n(\hat{\pi}) - c_{\text{odd}}(\hat{\pi})$. It has been shown that every permutation π can be transformed into a simple one π' by safe transformations and, moreover, every sorting of π' mimics a sorting of π with the same number of operations [6,11]. Here, Hartman and Sharan [8] further showed that every simple permutation π' can be transformed into a 3-permutation $\hat{\pi}$ by safe paddings (of transforming those 2-cycles into 1-twisted 3-cycles) and, moreover, every sorting of $\hat{\pi}$ mimics a sorting of π' with the same number of operations. Hence, based on these two properties, an arbitrary permutation π can be transformed into a 3-permutation $\hat{\pi}$ such that every sorting of $\hat{\pi}$ mimics a sorting of π with the same number of operations, suggesting that one can restrict attention to circular 3-permutations only.

Cycle Types

An operation that cuts some black edges is said to *act* on these edges. An operation is further called a *k-operation* if it increases the number of odd cycles by k. A $(0, 2, 2)$-*sequence* is a sequence of three operations, of which the first is a 0-operation and the next two are 2-operations. An odd cycle is called *oriented* if there is a 2-operation that acts on three of its black edges; otherwise, it is *unori-*



Sorting by Transpositions and Reversals (Approximate Ratio 1.5), Figure 2
Configurations of 3-cycles. a Unoriented, 0-twisted 3-cycle. b Unoriented, 1-twisted 3-cycle. c Oriented, 2-twisted 3-cycle. d Oriented, 3-twisted 3-cycle. e A pair of intersecting 3-cycles. f A pair of interleaving 3-cycles

ented. A *configuration* of cycles is a subgraph of the breakpoint graph that contains one ore more cycles. As shown in Fig. 2a–d, there are four possible configurations of single 3-cycles. A black edge is called *twisted* if its two adjacent gray edges cross each other in the circular breakpoint graph. A cycle is k-twisted if k of its black edges are twisted. For example, the 3-cycles in Fig. 2a–d are 0-, 1-, 2- and 3-twisted, respectively. Hartman and Sharan observed that a 3-cycle is oriented if and only if it is 2- or 3-twisted.

Cycle Configurations

Two pairs of black edges are called *intersecting* if they alternate in the order of their occurrence along the circle. A pair of black edges *intersects* with cycle C, if it intersects with a pair of black edges that belong to C. Cycles C and D *intersect* if there is a pair of black edges in C that intersects with D (see Fig. 2e). Two intersecting cycles are called *interleaving* if their black edges alternate in their order of occurrence along the circle (see Fig. 2f). Clearly, the relation between two cycles is one of (1) non-intersecting, (2) intersecting but non-interleaving and (3) interleaving. A pair of black edges is *coupled* if they are connected by a gray edge and when reading the edges along the cycle, they are read in the same direction. For example, all pairs of black edges in Fig. 2a are coupled. Gu et al. [5] have shown that given a pair of coupled black edges (b_1, b_2), there exists a cycle C that intersects with (b_1, b_2). A *1-twisted pair* is a pair of 1-twisted cycles, whose twists are consecutive on the circle in a configuration that consists of these two cycles only. A 1-twisted cycle is called *closed* in a configuration if its two coupled edges intersect with some other cycle in the configuration. A configuration is *closed* if at least one of its 1-twisted cycles is closed; otherwise, it is called *open*.

The Algorithm

The basic ideas of the Hartman and Sharan's 1.5-approximation algorithm [8] for the problem of sorting by transpositions, transreversals and revrevs are as follows. Hartman and Sharan reduced the problem to that of sorting a circular 3-permutation by transpositions and transreversals only and then focused on transforming the 3-cycles into 1-cycles in the breakpoint graph of this 3-permutation. By definition, an oriented (i. e., 2- or 3-twisted) 3-cycle admits a 2-operation and, therefore, they continued to consider unoriented (i. e., 0- or 1-twisted) 3-cycles only. Since configurations involving only 0-twisted 3-cycles were handled with $(0, 2, 2)$-sequences in [7], Hartman and Sharan restricted their attention to those configurations that consist of 0- and 1-twisted 3-cycles. They showed that these configurations are all closed and that it can be sorted by a $(0, 2, 2)$-sequence of operations for each of the following five possible closed configurations: (1) a closed configuration with two unoriented, interleaving 3-cycles that do not form a 1-twisted pair, (2) a closed configuration with two intersecting, 0-twisted 3-cycles, (3) a closed configuration with two intersecting, 1-twisted 3-cycles, (4) a closed configuration with a 0-twisted 3-cycles that intersects with the coupled edges of a 1-twisted 3-cycle, and (5) a closed configuration that contains $k \geq 2$ mutually interleaving 1-twisted 3-cycles such that all their twists are consecutive on the circle and k is maximal with this property. As a result, the sequence of operations used by Hartman and Sharan in their algorithm contains only 2-operations and $(0, 2, 2)$-sequences. Since every sequence of three operations increases the number of odd cycles by at least 4 out of 6 possible in 3 steps, the ratio of their approximation algorithm is 1.5. Furthermore, Hartman and Sharan showed that their algorithm can be implemented in $O(n^{3/2}\sqrt{\log n})$ time using the data structure of Kaplan and Verbin [10], where n is the number of elements in the permutation.

Theorem 1 *The problem of sorting linear permutations by transpositions, transreversals and revrevs is linearly equiv-*

alent to the problem of sorting circular permutations by transpositions, transreversals and revrevs.

Theorem 2 *There is a 1.5-approximation algorithm for sorting by transpositions, transreversals and revrevs, which runs in* $O(n^{3/2}\sqrt{\log n})$ *time.*

Applications

When trying to determine evolutionary distance between two organisms using genomic data, biologists may wish to reconstruct the sequence of evolutionary events that have occurred to transform one genome into the other. One of the most promising ways to do this phylogenetic study is to compare the order of appearance of identical (e. g., orthologous) genes in two different genomes [9,12]. This comparison of computing global rearrangement events (such as reversals, transpositions and transreversals of genome segments) may provide more accurate and robust clues to the evolutionary process than the analysis of local point mutations (i. e., substitutions, insertions and deletions of nucleotides/amino acids). Usually, the two genomes being compared are represented by signed permutations, with each element standing for a gene and its sign representing the (transcriptional) direction of the corresponding gene on a chromosome. Then the goal of the resulting genome rearrangement problem is to find a shortest sequence of rearrangement operations that transforms (or, equivalently, *sorts*) one permutation into the other. Previous work focused on the problem of sorting a permutation by reversals. This problem has been shown by Capara [2] to be NP-hard, if the considered permutation is unsigned. However, for signed permutations, this problem becomes tractable and Hannenhalli and Pevzer [6] gave the first polynomial-time algorithm for it. On the other hand, there has been less progress on the problem of sorting by transpositions. Thus far, the complexity of this problem is still open, although several 1.5-approximation algorithms [1,3,7] have been proposed for it. Recently, the approximation ratio of sorting by transpositions was further improved to 1.375 by Elias and Hartman [4]. Gu et al. [5] and Lin and Xue [11] gave quadratic-time 2-approximation algorithms for sorting signed, linear permutations by transpositions and transreversals. In [11], Lin and Xue considered the problem of sorting signed, linear permutations by transpositions, transreversals and revrevs, and proposed a quadratic-time 1.75-approximation algorithm for it. In this work [8], Hartman and Sharan further showed that this problem is equivalent for linear and circular permutations and can be reduced to that of sorting signed, circular permutations by transpositions and transreversals only. In addition, they provided a 1.5-approximation algorithm that can be implemented in $O(n^{3/2}\sqrt{\log n})$ time.

Cross References

- ▶ Sorting Signed Permutations by Reversal (Reversal Distance)
- ▶ Sorting Signed Permutations by Reversal (Reversal Sequence)

Recommended Reading

1. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM J. Discret. Math. **11**, 224–240 (1998)
2. Caprara, A.: Sorting permutations by reversals and Eulerian cycle decompositions. SIAM J. Discret. Math. **12**, 91–110 (1999)
3. Christie, D.A.: Genome Rearrangement Problems. Ph. D. thesis, Department of Computer Science. University of Glasgow, U.K. (1999)
4. Elias, I., Hartman, T.: A 1.375-approximation algorithm for sorting by transpositions. IEEE/ACM Transactions on Computational Biology and Bioinformatics **3**, 369–379 (2006)
5. Gu, Q.P., Peng, S., Sudborough, H.: A 2-approximation algorithm for genome rearrangements by reversals and transpositions. Theor. Comput. Sci. **210**, 327–339 (1999)
6. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J. Assoc. Comput. Mach. **46**, 1–27 (1999)
7. Hartman, T., Shamir, R.: A simpler and faster 1.5-approximation algorithm for sorting by transpositions. Inf. Comput. **204**, 275–290 (2006)
8. Hartman, T., Sharan, R.: A 1.5-approximation algorithm for sorting by transpositions and transreversals. In: Proceedings of the 4th Workshop on Algorithms in Bioinformatics (WABI'04), pp. 50–61. Bergen, Norway, 17–21 Sep (2004)
9. Hoot, S.B., Palmer, J.D.: Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera. J. Mol. Evol. **38**, 274–281 (1994)
10. Kaplan, H., Verbin, E.: Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03), pp. 170–185. Morelia, Michocán, Mexico, 25–27 Jun (2003)
11. Lin, G.H., Xue, G.: Signed genome rearrangements by reversals and transpositions: models and approximations. Theor. Comput. Sci. **259**, 513–531 (2001)
12. Palmer, J.D., Herbon, L.A.: Tricircular mitochondrial genomes of Brassica and Raphanus: reversal of repeat configurations by inversion. Nucleic Acids Res. **14**, 9755–9764 (1986)

Spanning Ratio

- ▶ Algorithms for Spanners in Weighted Graphs
- ▶ Dilation of Geometric Networks
- ▶ Geometric Dilation of Geometric Networks

Sparse Graph Spanners

2004; Elkin, Peleg

Michael Elkin
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel

Keywords and Synonyms

$(1+\epsilon,\beta)$-spanners; Almost additive spanners

Problem Definition

For a pair of numbers α, β, $\alpha \geq 1$, $\beta \geq 0$, a subgraph $G' = (V, H)$ of an unweighted undirected graph $G = (V, E)$, $H \subseteq E$, is an (α, β)-spanner of G if for every pair of vertices $u, w \in V$, $\mathrm{dist}_{G'}(u, w) \leq \alpha \cdot \mathrm{dist}_G(u, w) + \beta$, where $\mathrm{dist}_G(u, w)$ stands for the distance between u and w in G. It is desirable to show that for every n-vertex graph there exists a sparse (α, β)-spanner with as small values of α and β as possible. The problem is to determine asymptotic tradeoffs between α and β on one hand, and the sparsity of the spanner on the other.

Key Results

The main result of Elkin and Peleg [6] establishes the existence and efficient constructibility of $(1+\epsilon,\beta)$-spanners of size $O(\beta n^{1+1/\kappa})$ for every n-vertex graph G, where $\beta = \beta(\epsilon, \kappa)$ is constant whenever κ and ϵ are. The specific dependence of β on κ and ϵ is $\beta(\kappa, \epsilon) = \kappa^{\log\log\kappa - \log\epsilon}$.

An important ingredient of the construction of [6] is a partition of the graph G into regions of small diameter in such a way that the super-graph induced by these regions is sparse. The study of such partitions was initiated by Awerbuch [2], that used them for network synchronization. Peleg and Schäffer [8] were the first to employ such partitions for constructing spanners. Specifically, they constructed $(O(\kappa), 1)$-spanners with $O(n^{1+1/\kappa})$ edges. Althofer et al. [1] provided an alternative proof of the result of Peleg and Schäffer that uses an elegant greedy argument. This argument also enabled Althofer et al. to extend the result to weighted graphs, to improve the constant hidden by the O-notation in the result of Peleg and Schäffer, and to obtain related results for planar graphs.

Applications

Efficient algorithms for computing sparse $(1+\epsilon,\beta)$-spanners were devised in [5] and [11]. The algorithm of [5] was used in [5,7,10] for computing almost shortest paths in centralized, distributed, streaming, and dynamic centralized models of computations. The basic approach used in these results is to construct a sparse spanner, and then to compute exact shortest paths on the constructed spanner. The sparsity of the latter guarantees that the computation of shortest paths in the spanner is far more efficient than in the original graph.

Open Problems

The main open question is whether it is possible to achieve similar results with $\epsilon = 0$. More formally, the question is: Is it true that for any $\kappa \geq 1$ and any n-vertex graph G there exists $(1, \beta(\kappa))$-spanner of G with $O(n^{1+1/\kappa})$ edges? This question was answered in affirmitive for κ equal to 2 and 3 [3,4,6]. Some lower bounds were recently proved by Woodruff [12].

A less challenging problem is to improve the dependence of β on ϵ and κ. Some progress in this direction was achieved by Thorup and Zwick [11], and very recently by Pettie [9].

Cross References

► Synchronizers, Spanners

Recommended Reading

1. Althofer, I., Das, G., Dobkin, D.P., Joseph, D., Soares, J.: On Sparse Spanners of Weighted Graphs. Discret. Comput. Geom. **9**, 81–100 (1993)
2. Awerbuch, B.: Complexity of network synchronization. J. ACM **4**, 804–823 (1985)
3. Baswana, S., Kavitha, T., Mehlhorn, K., Pettie, S.: New Constructions of (alpha, beta)-spanners and purely additive spanners. In: Proc. of Symp. on Discrete Algorithms, Vancouver, Jan 2005, pp. 672–681
4. Dor, D., Halperin, S., Zwick, U.: All Pairs Almost Shortest Paths. SIAM J. Comput. **29**, 1740–1759 (2000)
5. Elkin, M.: Computing Almost Shortest Paths. Trans. Algorithms **1**(2), 283–323 (2005)
6. Elkin, M., Peleg, D.: $(1+\epsilon,\beta)$-Spanner Constructions for General Graphs. SIAM J. Comput. **33**(3), 608–631 (2004)
7. Elkin, M., Zhang, J.: Efficient Algorithms for Constructing $(1+\epsilon,\beta)$-spanners in the Distributed and Streaming Models. Distrib. Comput. **18**(5), 375–385 (2006)
8. Peleg, D., Schäffer, A.: Graph spanners. J. Graph Theory **13**, 99–116 (1989)
9. Pettie, S.: Low-Distortion Spanners. In: 34th International Colloquium on Automata Languages and Programm, Wroclaw, July 2007, pp. 78–89
10. Roditty, L., Zwick, U.: Dynamic approximate all-pairs shortest paths in undirected graphs. In: Proc. of Symp. on Foundations of Computer Science, Rome, Oct. 2004, pp. 499–508
11. Thorup, M., Zwick, U.: Spanners and Emulators with sublinear distance errors. In: Proc. of Symp. on Discrete Algorithms, Miami, Jan. 2006, pp. 802–809

12. Woodruff, D.: Lower Bounds for Additive Spanners, Emulators, and More. In: Proc. of Symp. on Foundations of Computer Science, Berkeley, Oct. 2006, pp. 389–398

Sparsest Cut

2004; Arora, Rao, Vazirani

Shuchi Chawla
Department of Computer Science, University of Wisconsin–Madison, Madison, WI, USA

Keywords and Synonyms

Minimum ratio cut

Problem Definition

In the Sparsest Cut problem, informally, the goal is to partition a given graph into two or more large pieces while removing as few edges as possible. Graph partitioning problems such as this one occupy a central place in the theory of network flow, geometric embeddings, and Markov chains, and form a crucial component of divide-and-conquer approaches in applications such as packet routing, VLSI layout, and clustering.

Formally, given a graph $G = (V, E)$, the *sparsity* or *edge expansion* of a non-empty set $S \subset V$, $|S| \leq \frac{1}{2}|V|$, is defined as follows:

$$\alpha(S) = \frac{|E(S, V \setminus S)|}{|S|} .$$

The sparsity of the graph, $\alpha(G)$, is then defined as follows:

$$\alpha(G) = \min_{S \subset V, |S| \leq \frac{1}{2}|V|} \alpha(S) .$$

The goal in the Sparsest Cut problem is to find a subset $S \subset V$ with the minimum sparsity, and to determine the sparsity of the graph.

The first approximation algorithm for the Sparsest Cut problem was developed by Leighton and Rao in 1988 [13]. Employing a linear programming relaxation of the problem, they obtained an $O(\log n)$ approximation, where n is the size of the input graph. Subsequently Arora, Rao and Vazirani [4] obtained an improvement over Leighton and Rao's algorithm using a semi-definite programming relaxation, approximating the problem to within an $O(\sqrt{\log n})$ factor.

In addition to the Sparsest Cut problem, Arora et al. also consider the closely related Balanced Separator problem. A partition $(S, V \setminus S)$ of the graph G is called a c-balanced separator for $0 < c \leq \frac{1}{2}$, if both S and $V \setminus S$ have at least $c|V|$ vertices. The goal in the Balanced Separator problem is to find a c-balanced partition with the minimum sparsity. This sparsity is denoted $\alpha_c(G)$.

Key Results

Arora et al. provide an $O(\sqrt{\log n})$ *pseudo-approximation* to the balanced-separator problem using semi-definite programming. In particular, given a constant $c \in (0, \frac{1}{2}]$, they produce a separator with balance c' that is slightly worse than c (that is, $c' < c$), but sparsity within an $O(\sqrt{\log n})$ factor of the sparsity of the optimal c-balanced separator.

Theorem 1 *Given a graph $G = (V, E)$, let $\alpha_c(G)$ be the minimum edge expansion of a c-balanced separator in this graph. Then for every fixed constant $a < 1$, there exists a polynomial-time algorithm for finding a c'-balanced separator in G, with $c' \geq ac$, that has edge expansion at most $O(\sqrt{\log n}\alpha_c(G))$.*

Extending this theorem to include unbalanced partitions, Arora et al. obtain the following:

Theorem 2 *Let $G = (V, E)$ be a graph with sparsity $\alpha(G)$. Then there exists a polynomial-time algorithm for finding a partition $(S, V \setminus S)$, with $S \subset V$, $S \neq \emptyset$, having sparsity at most $O(\sqrt{\log n}\alpha(G))$.*

An important contribution of Arora et al. is a new geometric characterization of vectors in n-dimensional space endowed with the squared-Euclidean metric. This result is of independent significance and has lead to or inspired improved approximation factors for several other partitioning problems (see, for example, [1,5,6,7,11]).

Informally, the result says that if a set of points in n-dimensional space is randomly projected on to a line, a good separator on the line is, with high probability, a good separator (in terms of squared-Euclidean distance) in the original high-dimensional space. Separation on the line is related to separation in the original space via the following definition of stretch.

Definition 1 (Def. 4 in [4]) Let $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ be a set of n points in $\mathcal{R}^n$, equipped with the squared-Euclidean metric $d(x, y) = \|x - y\|_2^2$. The set of points is said to be (t, γ, β)-*stretched at scale* ℓ, if for at least a γ fraction of all the n-dimensional unit vectors u, there is a partial matching $M_u = \{(x_i, y_i)\}_i$ among these points, with $|M_u| \geq \beta n$, such that for all $(x, y) \in M_u$, $d(x, y) \leq \ell^2$ and $\langle u, \vec{x} - \vec{y} \rangle \geq t\ell/\sqrt{n}$. Here $\langle \cdot, \cdot \rangle$ denotes the dot product of two vectors.

Theorem 3 *For any $\gamma, \beta > 0$, there is a constant*

$C = C(\gamma, \beta)$ such that if $t > C \log^{1/3} n$, then no set of n points in $\mathcal{R}^n$ can be (t, γ, β)-stretched for any scale ℓ.

In addition to the SDP-rounding algorithm, Arora et al. provide an alternate algorithm for finding approximate sparsest cuts, using the notion of expander flows. This result leads to fast (quadratic time) implementations of their approximation algorithm [3].

Applications

One of the main applications of balanced separators is in improving the performance of divide and conquer algorithms for a variety of optimization problems.

One example is the Minimum Cut Linear Arrangement problem. In this problem, the goal is to order the vertices of a given n vertex graph G from 1 through n in such a way that the capacity of the largest of the cuts $(\{1, 2, \cdots, i\}, \{i+1, \cdots, n\})$, $i \in [1, n]$, is minimized. Given a ρ-approximation to the balanced separator problem, the following divide and conquer algorithm gives an $O(\rho \log n)$-approximation to the Minimum Cut Linear Arrangement problem: find a balanced separator in the graph, then recursively order the two parts, and concatenate the orderings. The approximation follows by noting that if the graph has a balanced separator with expansion $\alpha_c(G)$, only $O(\rho n \alpha_n(G))$ edges are cut at every level, and given that a balanced separator is found at every step, the number of levels of recursion is at most $O(\log n)$.

Similar approaches can be used for problems such as VLSI layout and Gaussian elimination. (See the survey by Shmoys [14] for more details on these topics.)

The Sparsest Cut problem is also closely related to the problem of embedding squared-Euclidean metrics into the Manhattan (ℓ_1) metric with low distortion. In particular, the integrality gap of Arora et al.'s semi-definite programming relaxation for Sparsest Cut (generalized to include weights on vertices and capacities on edges) is exactly equal to the worst-case distortion for embedding a squared-Euclidean metric into the Manhattan metric. Using the technology introduced by Arora et al., improved embeddings from the squared-Euclidean metric into the Manhattan metric have been obtained [5,7].

Open Problems

Hardness of approximation results for the Sparsest Cut problem are fairly weak. Recently Chuzhoy and Khanna [9] showed that this problem is APX-hard, that is, there exists a constant $\epsilon > 0$, such that a $(1+\epsilon)$-approximation algorithm for Sparsest Cut would imply P=NP. It is conjectured that the weighted version of the problem is NP-hard to approximate better than $O((\log \log n)^c)$ for some constant c, but this is only known to hold true assuming a version of the so-called Unique Games conjecture [8,12]. On the other hand, the semi-definite programming relaxation of Arora et al. is known to have an integrality gap of $\Omega(\log \log n)$ even in the unweighted case [10]. Proving an unconditional super-constant hardness result for weighted or unweighted Sparsest Cut, or obtaining $o(\sqrt{\log n})$-approximations for these problems remain open.

The directed version of the Sparset Cut problem has also been studied, and is known to be hard to approximate within a $2^{\Omega(\log^{1-\epsilon} n)}$ factor [9]. On the other hand, the best approximation known for this problem only achieves a polynomial factor of approximation—a factor of $O(n^{11/23} \log^{O(1)} n)$ due to Aggarwal, Alon and Charikar [2].

Recommended Reading

1. Agarwal, A., Charikar, M., Makarychev, K., Makarychev, Y.: $O(\sqrt{\log n})$ approximation algorithms for Min UnCut, Min 2CNF Deletion, and directed cut problems. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), Baltimore, May 2005, pp. 573–581
2. Aggarwal, A., Alon, N., Charikar, M.: Improved approximations for directed cut problems. In: Proceedings of the 39th ACM Symposium on Theory of Computing (STOC), San Diego, June 2007, pp. 671–680
3. Arora, S., Hazan, E., Kale, S.: An $O(\sqrt{\log n})$ approximation to SPARSEST CUT in $\tilde{O}(n^2)$ time. In: Proceedings of the 45th IEEE Symposium on Foundations of Computer Science (FOCS), Rome, ITALY, 17–19 October 2004, pp. 238–247
4. Arora, S., Rao, S., Vazirani, U.: Expander Flows, Geometric Embeddings, and Graph Partitionings. In: Proceedings of the 36th ACM Symposium on Theory of Computing (STOC), Chicago, June 2004, pp. 222–231
5. Arora, S., Lee, J., Naor, A.: Euclidean Distortion and the Sparsest Cut. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), Baltimore, May 2005, pp. 553–562
6. Arora, S., Chlamtac, E., Charikar, M.: New approximation guarantees for chromatic number. In: Proceedings of the 38th ACM Symposium on Theory of Computing (STOC), Seattle, May 2006, pp. 215–224
7. Chawla, S., Gupta, A., Räcke, H.: Embeddings of Negative-type Metrics and An Improved Approximation to Generalized Sparsest Cut. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), Vancouver, January 2005, pp. 102–111
8. Chawla, S., Krauthgamer, R., Kumar, R., Rabani, Y., Sivakumar, D.: On the Hardness of Approximating Sparsest Cut and Multicut. In: Proceedings of the 20th IEEE Conference on Computational Complexity (CCC), San Jose, June 2005, pp. 144–153
9. Chuzhoy, J., Khanna, S.: Polynomial flow-cut gaps and hardness of directed cut problems. In: Proceedings of the 39th ACM Symposium on Theory of Computing (STOC), San Diego, June 2007 pp. 179–188

10. Devanur, N., Khot, S., Saket, R., Vishnoi, N.: Integrality gaps for Sparsest Cut and Minimum Linear Arrangement Problems. In: Proceedings of the 38th ACM Symposium on Theory of Computing (STOC), Seattel, May 2006, pp. 537–546
11. Feige, U., Hajiaghayi, M., Lee, J.: Improved approximation algorithms for minimum-weight vertex separators. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), Baltimore, May 2005, pp. 563–572
12. Khot, S., Vishnoi, N.: The Unique Games Conjecture, Integrality Gap for Cut Problems and the Embeddability of Negative-Type Metrics into ℓ_1. In: Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS), Pittsburgh, October 2005, pp. 53–62
13. Leighton, F.T., Rao, S.B.: An Approximate Max-Flow Min-Cut Theorem for Uniform Multicommodity Flow Problems with Applications to Approximation Algorithms. In: Proceedings of the 29th IEEE Symposium on Foundations of Computer Science (FOCS), White Plains, October 1988, pp. 422–431
14. Shmoys, D.B.: Cut problems and their application to divide-and-conquer. In: Hochbaum, D.S. (ed.) Approximation Algorithms for NP-hard Problems, pp. 192–235. PWS Publishing, Boston (1997)

Spatial Databases and Search

► Quantum Algorithm for Search on Grids
► R-Trees

Speed Scaling

1995; Yao, Demers, Shenker

Kirk Pruhs
Department of Computer Science,
University of Pittsburgh, Pittsburgh, PA, USA

Keywords and Synonyms

Speed scaling; Voltage scaling; Frequency scaling

Problem Definition

Speed scaling is a power management technique in modern processor that allows the processor to run at different speeds. There is a power function $P(s)$ that specifies the power, which is energy used per unit of time, as a function of the speed. In CMOS-based processors, the cube-root rule states that $P(s) \approx s^3$. This is usually generalized to assume that $P(s) = s^\alpha$ form some constant α. The goals of power management are to reduce temperature and/or to save energy. Energy is power integrated over time. Theoretical investigations to date have assumed that there is a fixed ambient temperature and that the processor cools according to Newton's law, that is, the rate of cooling is proportional to the temperature difference between the processor and the environment.

In the resulting scheduling problems, the scheduler must not only have a job-selection policy to determine the job to run at each time, but also a speed scaling policy to determine the speed at which to run that job. The resulting problems are generally dual objective optimization problems. One objective is some quality of service measure for the schedule, and the other objective is temperature or energy.

We will consider problems where jobs arrive at the processor over time. Each job i has a release time r_i when it arrives at the processor, and a work requirement w_i. A job i run at speed s takes w_i/s units of time to complete.

Key Results

[5] initiated the theoretical algorithmic investigation of speed scaling problems. [5] assumed that each job i had a deadline d_i, and that the quality of service measure was deadline feasibility (each job completes by its deadline). [5] gives a greedy algorithm YDS to find the minimum energy feasible schedule. The job selection policy for YDS is to run the job with the earliest deadline. To understand the speed scaling policy for YDS, define the intensity of a time interval to be the work that must be completed in this time interval divided by the length of the time interval. YDS then finds the maximum intensity interval, runs the jobs that must be run in this interval at constant speed, eliminates these jobs and this time interval from the instance, and proceeds recursively. [5] gives two online algorithms: OA and AVR. In OA the speed scaling policy is the speed that YDS would run at, given the current state and given that no more jobs will be released in the future. In AVR, the rate at which each job is completed is constant between the time that a job is released and the deadline for that job. [5] showed that AVR is $2^{\alpha-1}\alpha^\alpha$-competitive with respect to energy.

The results in [5] were extended in [2]. [2] showed that OA is α^α-competitive with respect to energy. [2] proposed another online algorithm, BKP. BKP runs at the speed of the maximum intensity interval containing the current time, taking into account only the work that has been released by the current time. They show that the competitiveness of BKP with respect to energy is at most $2(\alpha/(\alpha-1))^\alpha e^\alpha$. They also show that BKP is e-competitive with respect to the maximum speed.

[2] initiated the theoretical algorithmic investigation of speed scaling to manage temperature. [2] showed that the deadline feasible schedule that minimizes maximum temperature can in principle be computed in poly-

nomial time. [2] showed that the competitiveness of BKP with respect to maximum temperature is at most $2^{\alpha+1}e^{\alpha}(6(\alpha/(\alpha-1))^{\alpha}+1)$.

[4] initiated the theoretical algorithmic investigation into speed scaling when the quality-of-service objective is average/total flow time. The flow time of a job is the delay from when a job is released until it is completed. [4] give a rather complicated polynomial-time algorithm to find the optimal flow time schedule for unit work jobs, given a bound on the energy available. It is easy to see that no $O(1)$-competitive algorithm exists for this problem.

[1] introduce the objective of minimizing a linear combination of energy used and total flow time. This has a natural interpretation if one imagines the user specifying how much energy he is willing to use to increase the flow time of a job by a unit amount. [1] give an $O(1)$-competitive online algorithm for the case of unit work jobs. [3] improves upon this result and gives a 4-competitive online algorithm. The speed scaling policies of the online algorithms in [1] and [3] essentially run as power equal to the number of unfinished jobs (in each case modified in a particular way to facilitate analysis of the algorithm). [3] extend these results to apply to jobs with arbitrary work, and even arbitrary weight. The speed scaling policy is essentially to run at power equal to the weight of the unfinished work. The expression for the resulting competitive ratio is a bit complicated but is approximately 8 when the cube-root rule holds.

The analysis of the online algorithms in [2] and [3] heavily relied on amortized local competitiveness. An online algorithm is locally competitive for a particular objective if for all times the rate of increase of that objective for the online algorithm, plus the rate of change of some potential function, is at most the competitive ratio times the rate of increase of the objective in any other schedule.

Applications

None

Open Problems

The outstanding open problem is probably to determine if there is an efficient algorithm to compute the optimal flow time schedule given a fixed energy bound.

Recommended Reading

1. Albers, S., Fujiwara, H.: Energy-efficient algorithms for flow time minimization. In: STACS. Lecture Notes in Computer Science, vol. 3884, pp. 621–633. Springer, Berlin (2006)
2. Bansal, N., Kimbrel, T., Pruhs, K.: Speed scaling to manage energy and temperature. J. ACM **54**(1) (2007)
3. Bansal, N., Pruhs, K., Stein, C.: Speed scaling for weighted flow. In: ACM/SIAM Symposium on Discrete Algorithms, 2007
4. Pruhs, K., Uthaisombut, P., Woeginger, G.: Getting the Best Response for Your Erg. In: Scandanavian Workshop on Algorithms and Theory, 2004
5. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced CPU energy. In: IEEE Syposium on Foundations of Computer Science, 1995, p. 374

Sphere Packing Problem

2001; Chen, Hu, Huang, Li, Xu

Danny Z. Chen
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

Keywords and Synonyms

Ball packing; Disk packing

Problem Definition

The sphere packing problem seeks to pack spheres into a given geometric domain. The problem is an instance of geometric packing. Geometric packing is a venerable topic in mathematics. Various versions of geometric packing problems have been studied, depending on the shapes of packing domains, the types of packing objects, the position restrictions on the objects, the optimization criteria, the dimensions, etc. It also arises in numerous applied areas. The sphere packing problem under consideration here finds applications in radiation cancer treatment using Gamma Knife systems. Unfortunately, even very restricted versions of geometric packing problems (e. g., regular-shaped objects and domains in lower dimensional spaces) have been proved to be NP-hard. For example, for *congruent packing* (i. e., packing copies of the same object), it is known that the 2-D cases of packing fixed-sized congruent squares or disks in a simple polygon are NP-hard [7]. Baur and Fekete [2] considered a closely related *dispersion problem* of packing k congruent disks in a polygon of n vertices such that the radius of the disks is maximized; they proved that the dispersion problem cannot be approximated arbitrarily well in polynomial time unless P = NP, and gave a $\frac{2}{3}$-approximation algorithm for the L_∞ disk case with a time bound of $O(n^{38})$.

Chen et al. [4] proposed a practically efficient heuristic scheme, called *pack-and-shake*, for the **congruent sphere packing** problem, based on computational geometry techniques. The problem is defined as follows.

The Congruent Sphere Packing Problem

Given a d-D polyhedral region $R(d = 2, 3)$ of n vertices and a value $r > 0$, find a packing SP of R using spheres of radius r, such that (i) each sphere is contained in R, (ii) no two distinct spheres intersect each other in their interior, and (iii) the ratio (called the packing density) of the covered volume in R by SP over the total volume of R is maximized.

In the above problem, one can view the spheres as "solid" objects. The region R is also called the *domain* or *container*. Without loss of generality, let $r = 1$.

Much work on congruent sphere packing studied the case of packing spheres into an unbounded domain or even the whole space [5]. There are also results on packing congruent spheres into a bounded region. Hochbaum and Maass [8] presented a unified and powerful *shifting technique* for designing pseudo-polynomial time approximation schemes for packing congruent squares into a rectilinear polygon. But, the high time complexities associated with the resulting algorithms restrict their applicability in practice. Another approach is to formulate a packing problem as a non-linear optimization problem, and resort to an available optimization software to generate packings; however, this approach works well only for small problem sizes and regular-shaped domains.

To reduce the running time yet achieve a dense packing, a common idea is to consider objects that form a certain lattice or double-lattice. A number of results were given on lattice packing of congruent objects in the whole (especially high dimensional) space [5]. For a bounded rectangular 2-D domain, Milenkovic [10] adopted a method that first finds the densest translational lattice packing for a set of polygonal objects in the whole plane, and then uses some heuristics to extract the actual bounded packing.

Key Results

The *pack-and-shake* scheme of Chen et al. [4] for packing congruent spheres in an irregular-shaped 2-D or 3-D bounded domain R consists of three phases. In the first phase, the d-D domain R is partitioned into a set of convex subregions (called *cells*). The resulting set of cells defines a dual graph G_D, such that each vertex v of G_D corresponds to a cell $C(v)$ and an edge connects two vertices if and only if their corresponding cells share a $(d-1)$-D face. In the second phase, the algorithm repeats the following *trimming and packing* process until $G_D = \emptyset$: Remove the lowest degree vertex v from G_D and pack the cell $C(v)$. In the third phase, a *shake* procedure is applied to globally adjust the packing to obtain a denser one.

The objective of the trimming and packing procedure is that after each cell is packed, the remaining "packable" subdomain R' of R is always kept as a connected region. The rationale for maintaining the connectivity of R' is as follows. To pack spheres in a bounded domain R, two typical approaches have been used: (a) packing spheres layer by layer going from the boundary of R towards its interior [9], and (b) packing spheres starting from the "center" of R, such as its medial axis, towards its boundary [3,13,14]. Due to the shape irregularity of R, both approaches may fragment the remaining "packable" subdomain R' into more and more disconnected regions; however, at the end of packing each such region, a small "unpackable" area may eventually remain that allows no further packing. It could fit more spheres if the "packable" subdomain R' is lumped together instead of being divided into fragments, which is what the trimming and packing procedure aims to achieve.

Due to the packing of its adjacent cells that have been done by the trimming and packing procedure, the boundary of a cell $C(v)$ that is to be packed may consist of both line segments and arcs (from packed spheres). Hence, a key problem is to pack spheres in a cell bounded by curves of low degrees. Chen et al.'s algorithms [4] for packing each cell are based on certain lattice structures and allow the cell to both translate and rotate. Their algorithms have fairly low time bounds. In certain cases, they even run in nearly linear time.

An interesting feature of the cell packings generated by the trimming and packing procedure is that the resulted spheres cluster together in the middle of the cells of the domain R, leaving some small unpackable areas scattered along the boundary of R. The "shake" procedure in [4] thus seeks to collect these small areas together by "pushing" the spheres towards the boundary of R, in the hope of obtaining some "packable" region in the middle of R.

The approach in [4] is to first obtain a densest lattice unit sphere packing $LSP(C)$ for each cell C of R, and then use a "shake" procedure to globally adjust the resulting packing of R to generate a denser packing SP in R. Suppose the plane P is already packed by infinitely many unit spheres whose center points form a lattice (e. g., the hexagonal lattice). To obtain a densest packing $LSP(C)$ for a cell C from the lattice packing of the plane P, a position and orientation of C on P need to be computed such that C contains the maximum number of spheres from the lattice packing of P. There are two types of algorithms in [4] for computing an optimal placement of C on P: translational algorithms that allow C to be translated only, and translational/rotational algorithms that allow C to be both translated and rotated.

Let $n = |C|$, the number of bounding curves of C, and m be the number of spheres along the boundary of C in a sought optimal packing of C.

Theorem 1 *Given a polygonal region C bounded by n algebraic curves of constant degrees, a densest lattice unit sphere packing of C based only on translational motion can be computed in $O(N \log N + K)$ time, where $N = f(n, m)$ is a function of n and m, and K is the number of intersections between N planar algebraic curves of constant degrees that are derived from the packing instance.*

Note: In the worst case, $N = f(n, m) = n \times m$. But in practice, N may be much smaller. The N planar algebraic curves in Theorem 1 form a structure called *arrangement*. Since all these curves are of a constant degree, any two such curves can intersect each other at most a constant number of times. In the worst case, the number K of intersections between the N algebraic curves, which is also the *size* of the arrangement, is $O(N^2)$. The arrangement of these curves can be computed by the algorithms [1,6] in $O(N \log N + K)$ time.

Theorem 2 *Given a polygonal region C bounded by n algebraic curves of constant degrees, a densest lattice unit sphere packing of C based on both translational and rotational motions can be computed in $O(T(n) + (N + K') \log N)$ time, where $N = f(n, m)$ is a function of n and m, K' is the size of the arrangement of N pseudo-plane surfaces in 3-D that are derived from the packing instance, and T(n) is the time for solving $O(n^2)$ quadratic optimization problem instances associated with the packing instance.*

In Theorem 2, $K' = O(N^3)$ in the worst case. In practice, K' can be much smaller.

The results on 2-D sphere packing in [4] can be extended to d-D for any constant integer $d \geq 3$, so long as a good d-D lattice packing of the d-D space is available.

Applications

Recent interest in the considered congruent sphere packing problem was motivated by medical applications in Gamma Knife radiosurgery [4,11,12]. Radiosurgery is a minimally invasive surgical procedure that uses radiation to destroy tumors inside human body while sparing the normal tissues. The Gamma Knife is a radiosurgical system that consists of 201 Cobalt-60 sources [3,14]; the gamma-rays from these sources are all focused on a common center point, thus creating a spherical volume of radiation field. The Gamma Knife treatment normally applies high radiation dose. In this setting, overlapping spheres may result in overdose regions (called *hot spots*) in the target treatment domain, while a low packing density may cause underdose regions (called *cold spots*) and a non-uniform dose distribution. Hence, one may view the spheres used in Gamma Knife packing as "solid" spheres. Therefore, a key geometric problem in Gamma Knife treatment planning is to fit multiple spheres into a 3-D irregular-shaped tumor [3,13,14]. The total treatment time crucially depends on the number of spheres used. Subject to a given packing density, the minimum number of spheres used in the packing (i. e., treatment) is desired. The Gamma Knife currently produces spheres of four different radii (4 mm, 8 mm, 14 mm, and 18 mm), and hence the Gamma Knife sphere packing is in general not congruent. In practice, a commonly used approach is to pack larger spheres first, and then fit smaller spheres into the remaining subdomains, in the hope of reducing the total number of spheres involved and thus shortening the treatment time. Therefore, congruent sphere packing can be used as a key subroutine for such a common approach.

Open Problems

An open problem is to analyze the quality bounds of the resulting packing for the algorithms in [4]; such packing quality bounds are currently not yet known. Another open problem is to reduce the running time of the packing algorithms in [4], since these algorithms, especially for sphere packing problems in higher dimensions, are still very time-consuming. In general, it is highly desirable to develop efficient sphere packing algorithms in d-D ($d \geq 2$) with guaranteed good packing quality.

Experimental Results

Some experimental results of the 2-D pack-and-shake sphere packing algorithms were given in [4]. The planar hexagonal lattice was used for the lattice packing. On packings whose sizes are in the hundreds, the C++ programs of the algorithms in [4] based only on translational motion run very fast (a few minutes), while those of the algorithms based on both translation and rotation take much longer time (hours), reflecting their respective theoretical time bounds, as expected. On the other hand, the packing quality of the translation-and-rotation based algorithms is a little better than the translation based algorithms. The packing densities of all the algorithms in the experiments are well above 70% and some are even close to or above 80%. Comparing with the nonconvex programming methods, the packing algorithms in [4] seemed to run faster based on the experiments.

Cross References

► Local Approximation of Covering and Packing Problems

Recommended Reading

1. Amato, N.M., Goodrich, M.T., Ramos, E.A.: Computing the arrangement of curve segments: Divide-and-conquer algorithms via sampling. In: Proc. 11th Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 705–706 (2000)
2. Baur, C., Fekete, S.P.: Approximation of geometric dispersion problems. Algorithmica **30**(3), 451–470 (2001)
3. Bourland, J.D., Wu, Q.R.: Use of shape for automated, optimized 3D radiosurgical treatment planning. SPIE Proc. Int. Symp. on Medical Imaging, pp. 553–558 (1996)
4. Chen, D.Z., Hu, X., Huang, Y., Li, Y., Xu, J.: Algorithms for congruent sphere packing and applications. Proc. 17th Annual ACM Symp. on Computational Geometry, pp. 212–221 (2001)
5. Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups. Springer, New York (1988)
6. Edelsbrunner, H., Guibas, L.J., Pach, J., Pollack, R., Seidel, R., Sharir, M.: Arrangements of curves in the plane: Topology, combinatorics, and algorithms. Theor. Comput. Sci. **92**, 319–336 (1992)
7. Fowler, R.J., Paterson, M.S., Tanimoto, S.L.: Optimal packing and covering in the plane are NP-complete. Inf. Process. Lett. **12**(3), 133–137 (1981)
8. Hochbaum, D.S., Maass, W.: Approximation schemes for covering and packing problems in image processing and VLSI. J. ACM **32**(1), 130–136 (1985)
9. Li, X.Y., Teng, S.H., Üngör, A.: Biting: Advancing front meets sphere packing. Int. J. Num. Methods Eng. **49**(1–2), 61–81 (2000)
10. Milenkovic, V.J.: Densest translational lattice packing of non-convex polygons. Proc. 16th ACM Annual Symp. on Computational Geometry, 280–289 (2000)
11. Shepard, D.M., Ferris, M.C., Ove, R., Ma, L.: Inverse treatment planning for Gamma Knife radiosurgery. Med. Phys. **27**(12), 2748–2756 (2000)
12. Sutou, A., Dai, Y.: Global optimization approach to unequal sphere packing problems in 3D. J. Optim. Theor. Appl. **114**(3), 671–694 (2002)
13. Wang, J.: Medial axis and optimal locations for min-max sphere packing. J. Combin. Optim. **3**, 453–463 (1999)
14. Wu, Q.R.: Treatment planning optimization for Gamma unit radiosurgery. Ph. D. Thesis, The Mayo Graduate School (1996)

Squares and Repetitions

1999; Kolpakov, Kucherov

Maxime Crochemore[1,2], Wojciech Rytter[3]

[1] Department of Computer Science, King's College London, London, UK

[2] Laboratory of Computer Science, University of Paris-East, Paris, France

[3] Institute of Informatics, Warsaw University, Warsaw, Poland

Keywords and Synonyms

Powers; Runs; Tandem repeats

Problem Definition

Periodicities and repetitions in strings have been extensively studied and are important both in theory and practice (combinatorics of words, pattern-matching, computational biology). The words of the type ww and www, where w is a nonempty primitive (not of the form u^k for an integer $k > 1$) word, are called squares and cubes, respectively. They are well-investigated objects in combinatorics on words [16] and in string-matching with small memory [5].

A string w is said to be periodic iff $period(w) \leq |w|/2$, where $period(w)$ is the smallest positive integer p for which $w[i] = w[i + p]$ whenever both sides of the equality are defined. In particular each square and cube is periodic.

A repetition in a string $x = x_1 x_2 \ldots x_n$ is an interval $[i \,..\, j] \subseteq [1 \,..\, n]$ for which the associated factor $x[i \,..\, j]$ is periodic. It is an occurrence of a periodic word $x[i \,..\, j]$, also called a positioned repetition. A word can be associated with several repetitions, see Fig. 1.

Initially people investigated mostly positioned squares, but their number is $\Omega(n \log n)$ [2], hence algorithms computing all of them cannot run in linear time, due to the potential size of the output. The optimal algorithms reporting all positioned squares or just a single square were designed in [1,2,3,19]. Unlike this, it is known that only $O(n)$ (un-positioned) squares can appear in a string of length n [8].

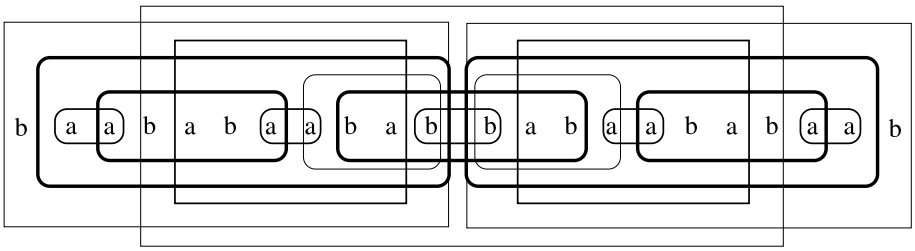
The concept of maximal repetitions, called runs (equivalent terminology) in [14], has been introduced to represent all repetitions in a succinct manner. The crucial property of runs is that there are only $O(n)$ runs in a word of length n [15,21].

A *run* in a string x is an interval $[i \,..\, j]$ such that both the associated string $x[i \,..\, j]$ has period $p \leq (j - i + 1)/2$, and the periodicity cannot be extended to the right nor to the left: $x[i - 1] \neq x[x + p - 1]$ and $x[j - p + 1] \neq x[j + 1]$ when the elements are defined. The set of runs of x is denoted by $RUNS(x)$. An example is displayed in Fig. 1.

Key Results

The main results concern fast algorithms for computing positioned squares and runs, as well as combinatorial estimation on the number of corresponding objects.

Theorem 1 (Crochemore [1], Apostolico-Preparata [2], Main-Lorentz [19]) *There exists an $O(n \log n)$ worst-case*



Squares and Repetitions, Figure 1
The structure of *RUNS*(*x*) where *x* = baababaababbabaababaab = bz²(z^R)²b, for *z* = aabab. The operation ·^R is reversing the string

Squares and Repetitions, Figure 2
The f-factorization of the example string *x* = baababaababbabaababaab and the set of its internal runs; all other runs overlap factorization points

time algorithm for computing all the occurrences of squares in a string of length n.

Techniques used to design the algorithms are based on partitioning, suffix trees, and naming segments. A similar result has been obtained by Franek, Smyth, and Tang using suffix arrays [11]. The key component in the next algorithm is the function described in the following lemma.

Lemma 2 (Main-Lorentz [19]) *Given two square-free strings u and v, reporting if uv contains a square centered in u can be done in worst-case time* $O(|u|)$.

Using suffix trees or suffix automata together with the function derived from the lemma, the following fact has been shown.

Theorem 3 (Crochemore [3], Main-Lorentz [19]) *Testing the square-freeness of a string of length n can be done in worst-case time* $O(n \log a)$*, where a is the size of the alphabet of the string.*

As a consequence of the algorithms and of the estimation on the number of squares, the most important result related to repetitions can be formulated as follows.

Theorem 4 (Kolpakov-Kucherov [15], Rytter [21], Crochemore-Ilie [4])
(1) All runs in a string can be computed in linear time (on a fixed-size alphabet).
(2) The number of all runs is linear in the length of the string.

The point (2) is very intricate, it is of purely combinatorial nature and has nothing to do with the algorithm. We sketch shortly the basic components in the constructive proof of the point (1). The main idea is to use, as for the previous theorem, the f-factorization (see [3]): a string x is decomposed into factors $u_1, u_2, \ldots, u_k$, where u_i is the longest segment which appears before (possibly with overlap) or is a single letter if the segment is empty.

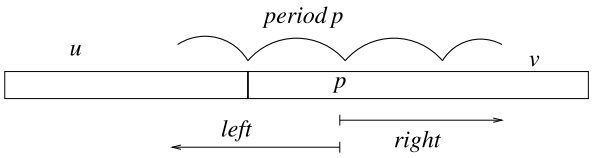
The runs which fit in a single factor are called internal runs, other runs are called here overlapping runs. There are three crucial facts:

- all overlapping runs can be computed in linear time,
- each internal run is a copy of an earlier overlapping run,
- the f-factorization can be computed in linear time (on a fixed-size alphabet) if we have the suffix tree or suffix automaton of the string. Figure 2 shows f-factorization and internal runs of an example string.

It follows easily from the definition of the f-factorization that if a run overlaps two (consecutive) factors u_{k-1} and u_k then its size is at most twice the total size of these two factors.

Figure 3 shows the basic idea for computing runs that overlap $u\,v$ in time $O(|u| + |v|)$. Using similar tables as in the Morris–Pratt algorithm (border and prefix tables), see [6], we can test the continuation of a period p from position p in v to the left and to the right. The corresponding tables can be constructed in linear time in a preprocessing phase. After computing all overlapping runs the internal runs can be copied from their earlier occurrences by processing the string from left to right.

Another interesting result concerning periodicities is the following lemma and its fairly immediate corollary.



Squares and Repetitions, Figure 3
If an overlapping run with period *p* starts in *u*, ends in *v*, and its part in *v* is of size at least *p* then it is easily detectable by computing continuations of the periodicity *p* in two directions: left and right

Lemma 5 (Three Prefix Squares, Crochemore-Rytter [5]) *If u, v, and w are three primitive words satisfying:* $|u| < |v| < |w|$, *uu is a prefix of vv, and vv is a prefix of ww, then* $|u| + |v| \leq |w|$

Corollary 1 *Any nonempty string x possesses less than* $\log_{\Phi} |y|$ *prefixes that are squares.*

In the configuration of the lemma, a second consequence is that *uu* is a prefix of *w*. Therefore, a position in a string *x* cannot be the largest position of more than two squares, which yields the next corollary. A simple direct proof of it is by Ilie [13], see also [17].

Corollary 2 (Fraenkel and Simpson [8]) *Any string x contains at most* $2|x|$ *(different) squares, that is:* $card\{u \mid u \text{ primitive and } u^2 \text{ factor of } y\} \leq 2|x|$.

The structure of all squares and of un-positioned runs has been also computed within the same time complexities as above in [18] and [12].

Applications

Detecting repetitions in strings is an important element of several questions: pattern matching, text compression, and computational biology to quote a few. Pattern-matching algorithms have to cope with repetitions to be efficient as these are likely to slow down the process; the large family of dictionary-based text compression methods use a weaker notion of repeats (like the software gzip); repetitions in genomes, called satellites, are intensively studied because, for example, some over-repeated short segments are related to genetic diseases; some satellites are also used in forensic crime investigations.

Open Problems

The most intriguing question remains the asymptotically tight bound for the maximum number $\rho(n)$ of runs in a string of size *n*. The first proof (by painful induction) was quite difficult and has not produced any *concrete* constant coefficient in the $O(n)$ notation. This subject has been studied in [9,10,22,23]. The best-known lower bound of approximately $0.927\,n$ is from [10]. The exact number of runs has been considered for special strings: *Fibonacci words* and (more generally) *Sturmian words* [7,14,20]. It is proved in a structural and intricate manner in the full version of [21] that $\rho(n) \leq 3.44\,n$, by introducing a *sparse-neighbors technique*. The neighbors are runs for which both the distance between their starting positions is small and the difference between their periods is also proportionally small (according to some fixed coefficient of proportionality). The occurrences of neighbors satisfy certain *sparsity* properties which imply the linear upper bound. Several variations for the definitions of neighbors and sparsity are possible. Considering runs having close centers the bound has been lowered to $1.6\,n$ in [4].

As a conclusion, we believe that the following fact is valid.

Conjecture: *A string of length n contains less than n runs, i. e.,* $|\text{RUNS}|(n) < n$.

Cross References

Elements of the present entry are of main importance for run-length compression as well as for ► Run-length Compressed Pattern Matching. They are also related to the ► Approximate Tandem Repeats entries because "tandem repeat" is a synonym of repetition and "power."

Recommended Reading

1. Apostolico, A., Preparata, F.P.: Optimal off-line detection of repetitions in a string. Theor. Comput. Sci. **22**(3), 297–315 (1983)
2. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inform. Process. Lett. **12**(5), 244–250 (1981)
3. Crochemore, M. : Transducers and repetitions. Theor. Comput. Sci. **45**(1), 63–86 (1986)
4. Crochemore, M., Ilie, L.: Analysis of maximal repetitions in strings. J. Comput. Sci. (2007)
5. Crochemore, M., Rytter, W.: Squares, cubes, and time-space efficient string searching. Algorithmica **13**(5), 405–425 (1995)
6. Crochemore, M., Rytter, W.: Jewels of stringology. World Scientific, Singapore (2003)
7. Franek, F., Karaman, A., Smyth, W.F.: Repetitions in Sturmian strings. Theor. Comput. Sci. **249**(2), 289–303 (2000)
8. Fraenkel, A.S., Simpson, R.J.: How many squares can a string contain? J. Comb. Theory Ser. A **82**, 112–120 (1998)
9. Fraenkel, A.S., Simpson, R.J.: The Exact Number of Squares in Fibonacci Words. Theor. Comput. Sci. **218**(1), 95–106 (1999)
10. Franek, F., Simpson, R.J. , and Smyth, W.F.: The maximum number of runs in a string. In: Proc. 14-th Australian Workshop on Combinatorial Algorithms, pp. 26–35. Curtin University Press, Perth (2003)

11. Franek, F., Smyth, W.F., Tang, Y.: Computing all repeats using suffix arrays. J. Autom. Lang. Comb. **8**(4), 579–591 (2003)
12. Gusfield, D.and Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. **69**(4), 525–546 (2004)
13. Ilie, L.: A simple proof that a word of length n has at most 2n distinct squares. J. Combin. Theory, Ser. A **112**(1), 163–164 (2005)
14. Iliopoulos, C., Moore, D., Smyth, W.F.: A characterization of the squares in a Fibonacci string. Theor. Comput. Sci. **172** 281–291 (1997)
15. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proceedings of the 40th Symposium on Foundations of Computer Science, pp. 596–604. IEEE Computer Society Press, Los Alamitos (1999)
16. Lothaire, M. (ed.): Algebraic Combinatorics on Words. Cambridge University Press, Cambridge (2002)
17. Lothaire, M. (ed.): Applied Combinatorics on Words. Cambridge University Press, Cambridge (2005)
18. Main, M.G.: Detecting leftmost maximal periodicities. Discret. Appl. Math. **25**, 145–153 (1989)
19. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms **5**(3), 422–432 (1984)
20. Rytter, W.: The structure of subword graphs and suffix trees of Fibonacci words. In: Implementation and Application of Automata, CIAA 2005. Lecture Notes in Computer Science, vol. 3845, pp. 250–261. Springer, Berlin (2006)
21. Rytter, W.: The Number of Runs in a String: Improved Analysis of the Linear Upper Bound. In: Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, vol. 3884, pp. 184–195. Springer, Berlin (2006)
22. Smyth, W.F.: Repetitive perhaps, but certainly not boring. Theor. Comput. Sci. **249**(2), 343–355 (2000)
23. Smyth, W.F.: Computing patterns in strings. Addison-Wesley, Boston, MA (2003)

Stable Marriage

1962; Gale, Shapley

Robert W. Irving
Department of Computing Science,
University of Glasgow, Glasgow, UK

Keywords and Synonyms

Stable matching

Problem Definition

The objective in *stable matching problems* is to match together pairs of elements of a set of participants, taking into account the preferences of those involved, and focusing on a stability requirement. The stability property ensures that no pair of participants would both prefer to be matched together rather than to accept their allocation in the matching. Such problems have widespread application, for example in the allocation of medical students to hospital posts, students to schools or colleges, etc.

An instance of the classical *Stable Marriage problem* (SM), introduced by Gale and Shapley [2], involves a set of $2n$ participants comprising n men $\{m_1, \ldots, m_n\}$ and n women $\{w_1, \ldots, w_n\}$. Associated with each participant is a *preference list*, which is a total order over the participants of the opposite sex. A man m_i *prefers* woman w_j to woman w_k if w_j precedes w_k on the preference list of m_i, and similarly for the women. A *matching* M is a bijection between the sets of men and women, in other words a set of man-woman pairs so that each man and each woman belongs to exactly one pair of M. For a man m_i, $M(m_i)$ denotes the *partner* of m_i in M, i. e., the unique woman w_j such that (m_i, w_j) is in M. Similarly, $M(w_j)$ denotes the partner of woman w_j in M. A matching M is *stable* if there is no *blocking pair*, namely a pair (m_i, w_j) such that m_i prefers w_j to $M(m_i)$ and w_j prefers m_i to $M(w_j)$.

Relaxing the requirements that the numbers of men and women are equal, and that each participant should rank *all* of the members of the opposite sex, gives the *Stable Marriage problem with Incomplete lists* (SMI). So an instance of SMI comprises a set of n_1 men $\{m_1, \ldots, m_{n_1}\}$ and a set of n_2 women $\{w_1, \ldots, w_{n_2}\}$, and each participant's preference list is a total order over a *subset* of the participants of the opposite sex. The implication is that if woman w_j does not appear on the list of man m_i then she is not an acceptable partner for m_i, and vice versa. A man-woman pair is *acceptable* if each member of the pair is on the preference list of the other, and a matching M is now a set of acceptable pairs such that each man and each woman is in *at most* one pair of M. In this context, a blocking pair for matching M is an acceptable pair (m_i, w_j) such that m_i is either unmatched in M or prefers w_j to $M(m_i)$, and likewise, w_j is either unmatched or prefers m_i to $M(w_j)$. A matching is stable if it has no blocking pair. So in an instance of SMI, a stable matching need not match all of the participants.

Gale and Shapley also introduced a many-one version of stable marriage, which they called the *College Admissions problem*, but which is now more usually referred to as the ► Hospitals/Residents Problem (HR) because of its well-known applications in the medical employment field. This problem is covered in detail in Entry 150 of this volume.

A comprehensive treatment of many aspects of the Stable Marriage problem, as of 1989, appears in the monograph of Gusfield and Irving [5].

Key Results

Theorem 1 *For every instance of SM or SMI there is at least one stable matching.*

Theorem 1 was proved constructively by Gale and Shapley [2] as a consequence of the algorithm that they gave to find a stable matching.

Theorem 2

(i) For a given instance of SM involving n men and n women, there is a $O(n^2)$ time algorithm that finds a stable matching.

(ii) For a given instance of SMI in which the combined lengths of all the preference lists is a, there is a $O(a)$ time algorithm that finds a stable matching.

The algorithm for SMI is a simple extension of that for SM. Each can be formulated in a variety of ways, but is most usually expressed in terms of a sequence of 'proposals' from the members of one sex to the members of the other. A pseudocode version of the SMI algorithm appears in Fig. 1, in which the traditional approach of allowing men to make proposals is adopted.

The complexity bound of Theorem 2(i) first appeared in Knuth's monograph on Stable Marriage [11]. The fact that this algorithm is asymptotically optimal was subsequently established by Ng and Hirschberg [15] via an adversary argument. On the other hand, Wilson [19] proved that the average running time, taken over all possible instances of SM, is $O(n \log n)$.

The algorithm of Fig. 1, in its various guises, has come to be known as the Gale–Shapley algorithm. The variant of the algorithm given here is called *man-oriented*, because men have the advantage of proposing. Reversing the roles of men and women gives the *woman-oriented* variant. The 'advantage' of proposing is remarkable, as spelled out in the next theorem.

Theorem 3 *The man-oriented version of the Gale–Shapley algorithm for SM or SMI yields the man-optimal stable matching in which each man has the best partner that he can have in any stable matching, but in which each woman has her worst possible partner. The woman-oriented version yields the woman-optimal stable matching, which has analogous properties favoring the women.*

The optimality property of Theorem 3 was established by Gale and Shapley [2], and the corresponding 'pessimality' property was first observed by McVitie and Wilson [14].

As observed earlier, a stable matching for an instance of SMI need not match all of the participants. But the following striking result was established by Gale and Sotomayor [3] and Roth [17] (in the context of the more general HR problem).

Theorem 4 *In an instance of SMI, all stable matchings have the same size and match exactly the same subsets of the men and women.*

For a given instance of SM or SMI, there may be many different stable matchings. Indeed Knuth [11] showed that the maximum possible number of stable matchings grows exponentially with the number of participants. He also pointed out that the set of stable matchings forms a distributive lattice under a natural dominance relation, a result attributed to Conway. This powerful algebraic structure that underlies the set of stable matchings can be exploited algorithmically in a number of ways. For example,

```
M = ∅;
assign each person to be free;        /* i. e., not a member of a pair in M */
while (some man m is free and has not proposed to every woman on his list)
    m proposes to w, the first woman on his list to whom he has not proposed;
    if (w is free)
        add (m, w) to M;              /* w accepts m */
    else if (w prefers m to her current partner m′)
        remove (m′, w) from M;        /* w rejects m′, setting m′ free */
        add (m, w) to M;              /* w accepts m */
    else
        M remains unchanged;          /* w rejects m */
return M;
```

Stable Marriage, Figure 1
The Gale–Shapley Algorithm

Gusfield [4] showed how all k stable matchings for an instance of SM can be generated in $O(n^2 + kn)$ time. ► Optimal Stable Marriage.

Extensions of these problems that are important in practice, so-called SMT and SMTI (extensions of SM and SMI respectively), allow the presence of *ties* in the preference lists. In this context, three different notions of stability have been defined [7] – *weak*, *strong* and *super-stability*, depending on whether the definition of a blocking pair requires that both members should improve, or at least one member improves and the other is no worse off, or merely that neither member is worse off. The following theorem summarizes the basic algorithmic results for these three varieties of stable matchings.

Theorem 5 *For a given instance of SMT or SMTI:*

(i) A weakly stable matching is guaranteed to exist, and can be found in $O(n^2)$ or $O(a)$ time, respectively;

(ii) A super-stable matching may or may not exist; if one does exist it can be found in $O(n^2)$ or $O(a)$ time respectively;

(iii) A strongly stable matching may or may not exist; if one does exist it can be found in $O(n^3)$ or $O(na)$ time, respectively.

Theorem 5 parts (i) and (ii) are due to Irving [7] (for SMT) and Manlove [12] (for SMTI). Part (iii) is due to Mehlhorn et al. [10], who improved earlier algorithms of Irving and Manlove.

It turns out that, in contrast to the situation described by Theorem 4(i), weakly stable matchings in SMTI can have different sizes. The natural problem of finding a maximum cardinality weakly stable matching, even under severe restrictions on the ties, is NP-hard [13]. ► Stable Marriage with Ties and Incomplete Lists explores this problem further.

The Stable Marriage problem is an example of a *bipartite* matching problem. The extension in which the bipartite requirement is dropped is the so-called *Stable Roommates* (SR) problem.

Gale and Shapley had observed that, unlike the case of SM, an instance of SR may or may not admit a stable matching, and Knuth [11] posed the problem of finding an efficient algorithm for SR, or proving it NP-complete. Irving [6] established the following theorem via a non-trivial extension of the Gale–Shapley algorithm.

Theorem 6 *For a given instance of SR, there exists a $O(n^2)$ time algorithm to determine whether a stable matching exists, and if so to find such a matching.*

Variants of SR may be defined, as for SM, in which preference lists may be incomplete and/or contain ties – these are denoted by SRI, SRT and SRTI – and in the presence of ties, the three flavors of stability, weak, strong and super, are again relevant.

Theorem 7 *For a given instance of SRT or SRTI:*

(i) A weakly stable matching may or may not exist, and it is an NP-complete problem to determine whether such a matching exists;

(ii) A super-stable matching may or may not exist; if one does exist it can be found in $O(n^2)$ or $O(a)$ time respectively;

(iii) A strongly stable matching may or may not exist; if one does exist it can be found in $O(n^4)$ or $O(a^2)$ time, respectively.

Theorem 7 part (i) is due to Ronn [16], part (ii) is due to Irving and Manlove [9], and part (iii) is due to Scott [18].

Applications

Undoubtedly the best known and most important applications of stable matching algorithms are in centralized matching schemes in the medical and educational domains. ► Hospitals/Residents Problem includes a summary of some of these applications.

Open Problems

The parallel complexity of stable marriage remains open. The best known parallel algorithm for SMI is due to Feder, Megiddo and Plotkin [1] and has $O(\sqrt{a}\log^3 a)$ running time using a polynomially bounded number of processors. It is not known whether the problem is in NC, but nor is there a proof of P-completeness.

One of the open problems posed by Knuth in his early monograph on stable marriage [11] was that of determining the maximum possible number x_n of stable matchings for any SM instance involving n men and n women. This problem remains open, although Knuth himself showed that x_n grows exponentially with n. Irving and Leather [8] conjecture that, when n is a power of 2, this function satisfies the recurrence

$$x_n = 3x_{n/2}^2 - 2x_{n/4}^4 .$$

Many open problems remain in the setting of weak stability, such as finding a good approximation algorithm for a maximum cardinality weakly stable matching – see ► Stable Marriage with Ties and Incomplete Lists – and enumerating all weakly stable matchings efficiently.

Cross References

- ▶ Hospitals/Residents Problem
- ▶ Optimal Stable Marriage
- ▶ Ranked Matching
- ▶ Stable Marriage and Discrete Convex Analysis
- ▶ Stable Marriage with Ties and Incomplete Lists
- ▶ Stable Partition Problem

Recommended Reading

1. Feder, T., Megiddo, N., Plotkin, S.A.: A sublinear parallel algorithm for stable matching. Theor. Comput. Sci. **233**(1–2), 297–308 (2000)
2. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. Am. Math. Monthly **69**, 9–15 (1962)
3. Gale, D., Sotomayor, M.: Some remarks on the stable matching problem. Discret. Appl. Math. **11**, 223–232 (1985)
4. Gusfield, D.: Three fast algorithms for four problems in stable marriage. SIAM J. Comput. **16**(1), 111–128 (1987)
5. Gusfield, D., Irving, R.W.: The Stable Marriage Problem: Structure and Algorithms. MIT Press, Cambridge (1989)
6. Irving, R.W.: An efficient algorithm for the stable roommates problem. J. Algorithms **6**, 577–595 (1985)
7. Irving, R.W.: Stable marriage and indifference. Discret. Appl. Math. **48**, 261–272 (1994)
8. Irving, R.W., Leather, P.: The complexity of counting stable marriages. SIAM J. Comput. **15**(3), 655–667 (1986)
9. Irving, R.W., Manlove, D.F.: The stable roommates problem with ties. J. Algorithms **43**, 85–105 (2002)
10. Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.: Strongly stable matchings in time O(nm), and extension to the H/R problem. In: Proceedings of STACS 2004: the 21st Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, vol. 2996, pp. 222–233. Springer, Berlin (2004)
11. Knuth, D.E.: Mariages Stables. Les Presses de L'Université de Montréal, Montréal (1976)
12. Manlove, D.F.: Stable marriage with ties and unacceptable partners. Technical Report TR-1999-29, University of Glasgow, Department of Computing Science, January (1999)
13. Manlove, D.F., Irving, R.W., Iwama, K., Miyazaki, S., Morita, Y.: Hard variants of stable marriage. Theor. Comput. Sci. **276**(1–2), 261–279 (2002)
14. McVitie, D., Wilson, L.B.: The stable marriage problem. Commun. ACM **14**, 486–490 (1971)
15. Ng, C., Hirschberg, D.S.: Lower bounds for the stable marriage problem and its variants. SIAM J. Comput. **19**, 71–77 (1990)
16. Ronn, E.: NP-complete stable matching problems. J. Algorithms **11**, 285–304 (1990)
17. Roth, A.E.: The evolution of the labor market for medical interns and residents: a case study in game theory. J. Polit. Econ. **92**(6), 991–1016 (1984)
18. Scott, S.: A study of stable marriage problems with ties. Ph. D. thesis, University of Glasgow, Department of Computing Science (2005)
19. Wilson, L.B.: An analysis of the stable marriage assignment algorithm. BIT **12**, 569–575 (1972)

Stable Marriage and Discrete Convex Analysis

2000; Eguchi, Fujishige, Tamura, Fleiner

AKIHISA TAMURA
Department of Mathematics, Keio University, Yokohama, Japan

Keywords and Synonyms

Stable matching

Problem Definition

In the stable marriage problem first defined by Gale and Shapley [7], there are one set each of men and women having the same size, and each person has a strict preference order on persons of the opposite gender. The problem is to find a matching such that there is no pair of a man and a woman who prefer each other to their partners in the matching. Such a matching is called a *stable marriage* (or *stable matching*). Gale and Shapley showed the existence of a stable marriage and gave an algorithm for finding one. Fleiner [4] extended the stable marriage problem to the framework of matroids, and Eguchi, Fujishige, and Tamura [3] extended this formulation to a more general one in terms of discrete convex analysis, which was developed by Murota [8,9]. Their formulation is described as follows.

Let M and W be sets of men and women who attend a dance party at which each person dances a waltz T times and the number of times that he/she can dance with the same person of the opposite gender is unlimited. The problem is to find an "agreeable" allocation of dance partners, in which each person is assigned at most T persons of the opposite gender with possible repetition. Let $E = M \times W$, i. e., the set of all man-woman pairs. Also define $E_{(i)} = \{i\} \times W$ for all $i \in M$ and $E_{(j)} = M \times \{j\}$ for all $j \in W$. Denoting by $x(i, j)$ the number of dances between man i and woman j, an allocation of dance partners can be described by a vector $x = (x(i, j) : i \in M, j \in W) \in \mathbf{Z}^E$, where $\mathbf{Z}$ denotes the set of all integers. For each $y \in \mathbf{Z}^E$ and $k \in M \cup W$, denote by $y_{(k)}$ the restriction of y on $E_{(k)}$. For example, for an allocation $x \in \mathbf{Z}^E$, $x_{(k)}$ represents the allocation of person k with respect to x. Each person k describes his/her preferences on allocations by using a value function $f_k : \mathbf{Z}^{E_{(k)}} \to \mathbf{R} \cup \{-\infty\}$, where $\mathbf{R}$ denotes the set of all reals and $f_k(y) = -\infty$ means that allocation $y \in \mathbf{Z}^{E_{(k)}}$ is unacceptable for k. Note that the valuation of each person on allocations is determined only by his/her allocations. Let $\mathrm{dom}\, f_k = \{y \mid f_k(y) \in \mathbf{R}\}$. Assume

that each value function f_k satisfies the following assumption:

(A) dom f_k is bounded and hereditary, and has $\mathbf{0}$ as the minimum point, where $\mathbf{0}$ is the vector of all zeros and heredity means that for any $y, y' \in \mathbf{Z}^{E(k)}, \mathbf{0} \le y' \le y \in \text{dom } f_k$ implies $y' \in \text{dom } f_k$.

For example, the following value functions with $M = \{1\}$ and $W = \{2, 3\}$

$$f_1(x(1,2), x(1,3)) = \begin{cases} 10(x(1,2)+x(1,3)) - x(1,2)^2 - x(1,3)^2 & \text{if } x(1,2), x(1,3) \ge 0 \\ & \text{and } x(1,2)+x(1,3) \le 3 \\ -\infty & \text{otherwise,} \end{cases}$$

$$f_j(x(1,j)) = \begin{cases} x(1,j) & \text{if } x(1,j) \in \{0,1,2,3\} (j = 2,3) \\ -\infty & \text{otherwise} \end{cases}$$

represent the case where (1) everyone wants to dance as many times, up to three, as possible, and (2) man 1 wants to divide his dances between women 2 and 3 as equally as possible. Allocations $(x(1,2), x(1,3)) = (1,2)$ and (2,1) are stable in the sense below.

A vector $x \in \mathbf{Z}^E$ is called a *feasible allocation* if $x_{(k)} \in \text{dom } f_k$ for all $k \in M \cup W$. An allocation x is said to satisfy *incentive constraints* if each person has no incentive to unilaterally decrease the current units of x, that is if it satisfies

$$f_k(x_{(k)}) = \max\{f_k(y) \mid y \le x_{(k)}\} \qquad (\forall k \in M \cup W). \quad (1)$$

An allocation x is called *unstable* if it does not satisfy incentive constraints or there exist $i \in M$, $j \in W$, $y' \in \mathbf{Z}^{E(i)}$ and $y'' \in \mathbf{Z}^{E(j)}$ such that

$$f_i(x_{(i)}) < f_i(y'), \quad (2)$$

$$y'(i,j') \le x(i,j') \qquad (\forall j' \in W \setminus \{j\}), \quad (3)$$

$$f_j(x_{(j)}) < f_j(y''), \quad (4)$$

$$y''(i',j) \le x(i',j) \qquad (\forall i' \in M \setminus \{i\}), \quad (5)$$

$$y'(i,j) = y''(i,j). \quad (6)$$

Conditions (2) and (3) say that man i can strictly increase his valuation by changing the current number of dances with j without increasing the numbers of dances with other women, and (4) and (5) describe a similar situation for women. Condition (6) requires that i and j agree on the number of dances between them. An allocation x is called *stable* if it is not unstable.

Problem 1 *Given disjoint sets M and W, and value functions $f_k : \mathbf{Z}^{E(k)} \to \boldsymbol{R} \cup \{-\infty\}$ for $k \in M \cup W$ satisfying assumption* (A)*, find a stable allocation x.*

Remark 1 A time schedule for a given feasible allocation can be given by a famous result on graph coloring, namely, "any bipartite graph can be edge-colorable with the maximum degree colors."

Key Results

The work of Eguchi, Fujishige, and Tamura [3] gave a solution to Problem 1 in the case where each value function f_k is $\mathrm{M}^\natural$-concave.

Discrete Convex Analysis: $\mathrm{M}^\natural$-Concave Functions

Let V be a finite set. For each $S \subseteq V$, e_S denotes the characteristic vector of S defined by: $e_S(v) = 1$ if $v \in S$ and $e_S(v) = 0$ otherwise. Also define e_0 as the zero vector in $\mathbf{Z}^V$. For a vector $x \in \mathbf{Z}^V$, its positive support $\text{supp}^+(x)$ and negative support $\text{supp}^-(x)$ is defined by $\text{supp}^+(x) = \{u \in V \mid x(u) > 0\}$ and $\text{supp}^-(x) = \{u \in V \mid x(u) < 0\}$. A function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ is called *$\mathrm{M}^\natural$-concave* if it satisfies the following condition $\forall x, y \in \text{dom } f$, $\forall u \in \text{supp}^+(x-y), \exists v \in \text{supp}^-(x-y) \cup \{0\}$:

$$f(x) + f(y) \le f(x - e_u + e_v) + f(y + e_u - e_v).$$

The above condition says that the sum of the function values at two points does not decrease as the points symmetrically move one or two steps closer to each other on the set of integral lattice points of $\mathbf{Z}^V$. This is a discrete analogue of the fact that for an ordinary concave function the sum of the function values at two points does not decrease as the points symmetrically move closer to each other on the straight line segment between the two points.

Example 1 A nonempty family $\mathcal{T}$ of subsets of V is called a *laminar family* if $X \cap Y = \emptyset$, $X \subseteq Y$ or $Y \subseteq X$ holds for every $X, Y \in \mathcal{T}$. For a laminar family $\mathcal{T}$ and a family of univariate concave functions $f_Y : \mathbf{R} \to \mathbf{R} \cup \{-\infty\}$ indexed by $Y \in \mathcal{T}$, the function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ defined by

$$f(x) = \sum_{Y \in \mathcal{T}} f_Y\left(\sum_{v \in Y} x(v)\right) \qquad (\forall x \in \mathbf{Z}^V)$$

is $\mathrm{M}^\natural$-concave. The stable marriage problem can be formulated as Problem 1 by using value functions of this type.

Example 2 For the independence family $\mathcal{I} \subseteq 2^V$ of a matroid on V and $w \in \mathbf{R}^V$, the function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ defined by

$$f(x) = \begin{cases} \sum_{u \in X} w(u) & \text{if } x = e_X \text{ for some } X \in \mathcal{I} \\ -\infty & \text{otherwise} \end{cases} \qquad (\forall x \in \mathbf{Z}^V)$$

is $\mathrm{M}^\natural$-concave. Fleiner [4] showed that there always exists a stable allocation for value functions of this type.

Theorem 1 ([6]) *Assume that the value functions f_k $(k \in M \cup W)$ are $\mathrm{M}^\natural$-concave satisfying **(A)**. Then a feasible allocation x is stable if and only if there exist $z_M = (z_{(i)} \mid i \in M) \in (\mathbf{Z} \cup \{+\infty\})^E$ and $z_W = (z_{(j)} \mid j \in W) \in (\mathbf{Z} \cup \{+\infty\})^E$ such that*

$$x_{(i)} \in \arg\max\{f_i(y) \mid y \le z_{(i)}\} \qquad (\forall i \in M), \tag{7}$$

$$x_{(j)} \in \arg\max\{f_j(y) \mid y \le z_{(j)}\} \qquad (\forall j \in W), \tag{8}$$

$$z_M(e) = +\infty \text{ or } z_W(e) = +\infty \qquad (\forall e \in E), \tag{9}$$

where $\arg\max\{f_i(y) \mid y \le z_{(i)}\}$ denotes the set of all maximizers of f_i under the constraints $y \le z_{(i)}$.

Theorem 2 ([3]) *Assume that the value functions f_k $(k \in M \cup W)$ are $\mathrm{M}^\natural$-concave satisfying **(A)**. Then there always exists a stable allocation.*

Eguchi, Fujishige, and Tamura [3] proved Theorem 2 by showing that the following algorithm finds a feasible allocation x, and z_M, z_W satisfying (7), (8), and (9).

Algorithm EXTENDED-GS
Input: $\mathrm{M}^\natural$-concave functions f_M, f_W with $f_M(x) = \sum_{i \in M} f_i(x_{(i)})$ and $f_W(x) = \sum_{j \in W} f_j(x_{(j)})$;
Output: (x, z_M, z_W) satisfying (7), (8), and (9);
$z_M := (+\infty, \cdots, +\infty)$, $z_W := x_W := \mathbf{0}$;
repeat{
let x_M be any element in
$\arg\max\{f_M(y) \mid x_W \le y \le z_M\}$;
let x_W be any element in
$\arg\max\{f_W(y) \mid y \le x_M\}$;
for each $e \in E$ with $x_M(e) > x_W(e)$ {
$z_M(e) := x_W(e)$;
$z_W(e) := +\infty$;
};
} **until** $x_M = x_W$;
return $(x_M, z_M, z_W \vee x_M)$.

Here $z_W \vee x_M$ is defined by $(z_W \vee x_M)(e) = \max\{z_W(e), x_M(e)\}$ for all $e \in E$.

Applications

Abraham, Irving, and Manlove [1] dealt with a student-project allocation problem which is a concrete example of models in [4] and [3], and discussed the structure of stable allocations.

Fleiner [5] generalized the stable marriage problem and its extension in [4] to a wide framework, and showed the existence of a stable allocation by using a fixed point theorem.

Fujishige and Tamura [6] proposed a common generalization of the stable marriage problem and the assignment game defined by Shapley and Shubik [10] by utilizing $\mathrm{M}^\natural$-concave functions, and gave a constructive proof of the existence of a stable allocation.

Open Problems

Algorithm EXTENDED-GS solves the maximization problem of an $\mathrm{M}^\natural$-concave function in each iteration. A maximization problem of an $\mathrm{M}^\natural$-concave function f on E can be solved in polynomial time in $|E|$ and $\log L$, where $L = \max\{||x - y||_\infty \mid x, y \in \mathrm{dom}\, f\}$, provided that the function value $f(x)$ can be calculated in constant time for each x [11,12]. Eguchi, Fujishige, and Tamura [3] showed that EXTENDED-GS terminates after at most L iterations, where L is defined by $\{||x||_\infty \mid x \in \mathrm{dom}\, f_M\}$ in this case, and there exist a series of instances in which EXTENDED-GS requires numbers of iterations proportional to L. On the other hand, Baïou and Balinski [2] gave a polynomial time algorithm in $|E|$ for the special case where f_M and f_W are linear on rectangular domains. Whether a stable allocation for the general case can be found in polynomial time in $|E|$ and $\log L$ or not is open.

Cross References

- Assignment Problem
- Hospitals/Residents Problem
- Optimal Stable Marriage
- Stable Marriage
- Stable Marriage with Ties and Incomplete Lists

Recommended Reading

1. Abraham, D.J., Irving, R.W., Manlove, D.F.: Two Algorithms for the Student-Project Allocation Problem. J. Discret. Algorithms **5**, 73–90 (2007)
2. Baïou, M., Balinski, M.: Erratum: The Stable Allocation (or Ordinal Transportation) Problem. Math. Oper. Res. **27**, 662–680 (2002)
3. Eguchi, A., Fujishige, S., Tamura, A.: A generalized Gale-Shapley algorithm for a discrete-concave stable-marriage model. In:

Ibaraki, T., Katoh, N., Ono, H. (eds.) Algorithms and Computation: 14th International Symposium, ISAAC2003. LNCS, vol. 2906, pp. 495–504. Springer, Berlin (2003)
4. Fleiner, T.: A matroid generalization of the stable matching polytope. In: Gerards, B., Aardal K. (eds.) Integer Programming and Combinatorial Optimization: 8th International IPCO Conference. LNCS, vol. 2081, pp. 105–114. Springer, Berlin (2001)
5. Fleiner, T.: A Fixed Point Approach to Stable Matchings and Some Applications. Math. Oper. Res. **28**, 103–126 (2003)
6. Fujishige, S., Tamura, A.: A Two-Sided Discrete-Concave Market with Bounded Side Payments: An Approach by Discrete Convex Analysis. Math. Oper. Res. **32**, 136–155 (2007)
7. Gale, D., Shapley, S.L.: College admissions and the stability of marriage. Am. Math. Mon. **69**, 9–15 (1962)
8. Murota, K.: Discrete Convex Analysis. Math. Program. **83**, 313–371 (1998)
9. Murota, K.: Discrete Convex Analysis. Soc. Ind. Appl. Math. Philadelphia (2003)
10. Shapley, S.L., Shubik, M.: The Assignment Game I: The Core. Int. J. Game. Theor. **1**, 111–130 (1971)
11. Shioura, A.: Fast Scaling Algorithms for M-convex Function Minimization with Application to the Resource Allocation Problem. Discret. Appl. Math. **134**, 303–316 (2004)
12. Tamura, A.: Coordinatewise Domain Scaling Algorithm for M-convex Function Minimization. Math. Program. **102**, 339–354 (2005)

Stable Marriage with Ties and Incomplete Lists

2007; Iwama, Miyazaki, Yamauchi

Kazuo Iwama[1], Shuichi Miyazaki[2]
[1] School of Informatics, Kyoto University, Kyoto, Japan
[2] Academic Center for Computing and Media Studies, Kyoto University, Kyoto, Japan

Keywords and Synonyms

Stable matching problem

Problem Definition

In the original setting of the stable marriage problem introduced by Gale and Shapley [2], each preference list has to include all members of the other party, and furthermore, each preference list must be totally ordered (see entry ► Stable Marriage also).

One natural extension of the problem is then to allow persons to include ties in preference lists. In this extension, there are three variants of the stability definition, super-stability, strong stability, and weak stability (see below for definitions). In the first two stability definitions, there are instances that admit no stable matching, but there is a polynomial-time algorithm in each case that determines if a given instance admits a stable matching, and finds one if exists [8]. On the other hand, in the case of weak stability, there always exists a stable matching and one can be found in polynomial time.

Another possible extension is to allow persons to declare unacceptable partners, so that preference lists may be incomplete. In this case, every instance admits at least one stable matching, but a stable matching may not be a perfect matching. However, if there are two or more stable matchings for one instance, then all of them have the same size [3].

The problem treated in this entry allows both extensions simultaneously, which is denoted as SMTI (Stable Marriage with Ties and Incomplete lists).

Notations

An instance I of SMTI comprises n men, n women and each person's preference list that may be incomplete and may include ties. If a man m includes a woman w in his list, w is *acceptable* to m. $w_i \succ_m w_j$ means that m strictly prefers w_i to w_j in I. $w_i =_m w_j$ means that w_i and w_j are tied in m's list (including the case $w_i = w_j$). The statement $w_i \succeq_m w_j$ is true if and only if $w_i \succ_m w_j$ or $w_i =_m w_j$. Similar notations are used for women's preference lists. A matching M is a set of pairs (m, w) such that m is acceptable to w and vice versa, and each person appears at most once in M. If a man m is matched with a woman w in M, it is written as $M(m) = w$ and $M(w) = m$.

A man m and a woman w are said to form a *blocking pair for weak stability* for M if they are not partners in M but by matching them, both become better off, namely, (i) $M(m) \neq w$ but m and w are acceptable to each other, (ii) $w \succ_m M(m)$ or m is single in M, and (iii) $m \succ_w M(w)$ or w is single in M.

Two persons x and y are said to form a *blocking pair for strong stability* for M if they are not partners in M but by matching them, one becomes better off, and the other does not become worse off, namely, (i) $M(x) \neq y$ but x and y are acceptable to each other, (ii) $y \succ_x M(x)$ or x is single in M, and (iii) $x \succeq_y M(y)$ or y is single in M.

A man m and a woman w are said to form a *blocking pair for super-stability* for M if they are not partners in M but by matching them, neither become worse off, namely, (i) $M(m) \neq w$ but m and w are acceptable to each other, (ii) $w \succeq_m M(m)$ or m is single in M, and (iii) $m \succeq_w M(w)$ or w is single in M.

A matching M is called *weakly stable* (*strongly stable* and *super-stable*, respectively) if there is no blocking pair for weak (strong and super, respectively) stability for M.

Problem 1 (SMTI)
INPUT: *n men, n women, and each person's preference list.*
OUTPUT: *A stable matching.*

Problem 2 (MAX SMTI)
INPUT: *n men, n women, and each person's preference list.*
OUTPUT: *A stable matching of maximum size.*

The following problem is a restriction of MAX SMTI in terms of the length of preference lists:

Problem 3 ((p, q)-MAX SMTI)
INPUT: *n men, n women, and each person's preference list, where each man's preference list includes at most p women, and each woman's preference list includes at most q men.*
OUTPUT: *A stable matching of maximum size.*

Definition of Approximation Ratios

A goodness measure of an approximation algorithm T for a maximization problem is defined as follows: the *approximation ratio* of T is $\max\{opt(x)/T(x)\}$ over all instances x of size N, where $opt(x)$ and $T(x)$ are the size of the optimal and the algorithm's solution, respectively.

Key Results

SMTI and MAX SMTI in Super-Stability and Strong Stability

Theorem 1 ([16]) *There is an $O(n^2)$-time algorithm that determines if a given SMTI instance admits a super-stable matching, and finds one if exists.*

Theorem 2 ([15]) *There is an $O(n^3)$-time algorithm that determines if a given SMTI instance admits a strongly stable matching, and finds one if exists.*

It is shown that all stable matchings for a fixed instance are of the same size [16]. So, the above theorems imply that MAX SMTI can also be solved in the same time complexity.

SMTI and MAX SMTI in Weak Stability

In the case of weak stability, every instance admits at least one stable matching, but one instance can have stable matchings of different sizes. If the size is not important, a stable matching can be found in polynomial time by breaking ties arbitrarily and applying the Gale-Shapley algorithm.

Theorem 3 *There is an $O(n^2)$-time algorithm that finds a weakly stable matching for a given SMTI instance.*

However, if larger stable matchings are required, the problem becomes hard.

Theorem 4 ([5,7,12,17]) *MAX SMTI is NP-hard, and cannot be approximated within $21/19 - \epsilon$ for any positive constant ϵ, unless $P = NP$. ($21/19 \simeq 1.105$)*

The current best approximation algorithm is a local search type algorithm.

Theorem 5 ([13]) *There is a polynomial-time approximation algorithm for MAX SMTI, whose approximation ratio is at most $15/8 (= 1.875)$.*

There are a couple of approximation algorithms for restricted inputs.

Theorem 6 ([6]) *There is a polynomial-time randomized approximation algorithm for MAX SMTI whose expected approximation ratio is at most $10/7 (\simeq 1.429)$, if in a given instance, ties appear in only one side and the length of each tie is two.*

Theorem 7 ([6]) *There is a polynomial-time randomized approximation algorithm for MAX SMTI whose expected approximation ratio is at most $7/4 (= 1.75)$, if in a given instance, the length of each tie is two.*

Theorem 8 ([7]) *There is a polynomial-time approximation algorithm for MAX SMTI whose approximation ratio is at most $2/(1 + L^{-2})$, if in a given instance, ties appear in only one side and the length of each tie is at most L.*

Theorem 9 ([7]) *There is a polynomial-time approximation algorithm for MAX SMTI whose approximation ratio is at most $13/7 (\simeq 1.858)$, if in a given instance, the length of each tie is two.*

(p, q)-MAX SMTI in Weak Stability

Irving et al. show the boundary between P and NP in terms of the length of preference lists.

Theorem 10 ([11]) *$(2, \infty)$-MAX SMTI is solvable in time $O(n^{\frac{3}{2}} \log n)$.*

Theorem 11 ([11]) *(3,4)-MAX SMTI is NP-hard, and cannot be approximated within some constant $\delta (> 1)$, unless $P = NP$.*

Recently, Manlove proved NP-hardness of (3,3)-MAX SMTI [18].

Applications

One of the most famous applications of the stable marriage problem is a centralized assignment system between

medical students (residents) and hospitals. This is an extension of the stable marriage problem to a many-one variant: Each hospital declares the number of residents it can accept, which may be more than one, while each resident has to be assigned to at most one hospital. Actually, there are several applications in the world, known as NRMP in the US [4], CaRMS in Canada [1], SPA in Scotland [9,10], and JRMP in Japan [14]. One of the optimization criteria is clearly the number of matched residents. In a real-world application such as the above residents matching, hospitals and residents tend to submit short preference lists that include ties, in which case, the problem can be naturally considered as MAX SMTI.

Open Problems

One apparent open problem is to narrow the gap of approximability of MAX SMTI in weak stability, namely, between 15/8(= 1.875) and 21/19($\simeq$ 1.105) for general case. The same problem can be considered for restricted instances. The reduction shown in [7] creates instances where ties appear in only one side, and the length of ties is two. So, considering Theorem 8 for $L = 2$, there is a gap between 8/5(= 1.6) and 21/19($\simeq$ 1.105) in this case. It is shown in [7] that if the $2 - \epsilon$ lower bound (for any positive constant ϵ) on the approximability of Minimum Vertex Cover is derived, the same reduction shows the $5/4 - \delta$ lower bound (for any positive constant δ) on the approximability of MAX SMTI.

Cross References

- ► Assignment Problem
- ► Hospitals/Residents Problem
- ► Optimal Stable Marriage
- ► Ranked Matching
- ► Stable Marriage
- ► Stable Marriage and Discrete Convex Analysis
- ► Stable Partition Problem

Recommended Reading

1. Canadian Resident Matching Service (CaRMS) http://www.carms.ca/. Accessed 27 Feb 2008, JST
2. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. Am. Math. Monthly **69**, 9–15 (1962)
3. Gale, D., Sotomayor, M.: Some remarks on the stable matching problem. Discret. Appl. Math. **11**, 223–232 (1985)
4. Gusfield, D., Irving, R.W.: The Stable Marriage Problem: Structure and Algorithms. MIT Press, Boston, MA (1989)
5. Halldórsson, M.M., Irving, R.W., Iwama, K., Manlove, D.F., Miyazaki, S., Morita, Y., Scott, S.: Approximability results for stable marriage problems with ties. Theor. Comput. Sci. **306**, 431–447 (2003)
6. Halldórsson, M.M., Iwama, K., Miyazaki, S., Yanagisawa, H.: Randomized approximation of the stable marriage problem. Theor. Comput. Sci. **325**(3), 439–465 (2004)
7. Halldórsson, M.M., Iwama, K., Miyazaki, S., Yanagisawa, H.: Improved approximation of the stable marriage problem. Proc. ESA 2003. LNCS 2832, pp. 266–277. (2003)
8. Irving, R.W.: Stable marriage and indifference. Discret. Appl. Math. **48**, 261–272 (1994)
9. Irving, R.W.: Matching medical students to pairs of hospitals: a new variation on a well-known theme. Proc. ESA 98. LNCS 1461, pp. 381–392. (1998)
10. Irving, R.W., Manlove, D.F., Scott, S.: The hospitals/residents problem with ties. Proc. SWAT 2000. LNCS 1851, pp. 259–271. (2000)
11. Irving, R.W., Manlove, D.F., O'Malley, G.: Stable marriage with ties and bounded length preference lists. Proc. the 2nd Algorithms and Complexity in Durham workshop, Texts in Algorithmics, College Publications (2006)
12. Iwama, K., Manlove, D.F., Miyazaki, S., Morita, Y.: Stable marriage with incomplete lists and ties. Proc. ICALP 99. LNCS 1644, pp. 443–452. (1999)
13. Iwama, K., Miyazaki, S., Yamauchi, N.: A 1.875-approximation algorithm for the stable marriage problem. Proc, SODA 2007, pp. 288–297. (2007)
14. Japanese Resident Matching Program (JRMP) http://www.jrmp.jp/
15. Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.: Strongly stable matchings in time $O(nm)$ and extension to the hospitals-residents problem. Proc. STACS 2004. LNCS (2996), pp. 222–233. (2004)
16. Manlove, D.F.: Stable marriage with ties and unacceptable partners. Technical Report no. TR-1999-29 of the Computing Science Department of Glasgow University (1999)
17. Manlove, D.F., Irving, R.W., Iwama, K., Miyazaki, S., Morita, Y.: Hard variants of stable marriage. Theor. Comput. Sci. **276**(1–2), 261–279 (2002)
18. Manlove, D.F.: private communication (2006)

Stable Matching

- ► Market Games and Content Distribution
- ► Stable Marriage
- ► Stable Marriage and Discrete Convex Analysis
- ► Stable Marriage with Ties and Incomplete Lists

Stable Partition Problem

2002; Cechlárová, Hajduková

KATARÍNA CECHLÁROVÁ
Faculty of Science, Institute of Mathematics,
P.J. Šafárik University, Košice, Slovakia

Keywords and Synonyms

In the economists community these models are often referred to as *Coalition formation games* [4,7], or *Hedonic*

games [3,6,16]; some variants correspond to the *Directed cycle cover* problems [1]. Important special cases are the *Stable Matching Problems* [17]. .

Problem Definition

In the Stable Partition Problem a set of participants has to be split into several disjoint sets called *coalitions*. The resulting partition should fulfill some stability requirements that take into account the preferences of participants.

Various variants of this problem arise if the participants are required to express their preferences over all the possible coalitions to which they could belong or when only preferences over other players are given and those are then extended to preferences over coalitions. Sometimes one seeks rather a permutation of players and the partition is given by the cycles of the permutation [1, 19].

Notation

An instance of the Stable Partition Problem (SPP for short) is a pair $(N,\mathcal{P})$, where N is a finite set of participants and $\mathcal{P}$ the collection of their preferences, called the *preference profile*. If the preferences of participants are given as linearly ordered lists of the coalitions to which a particular participant can belong (i. e. participant i writes a list of subsets of N that contain i), we say that the instance of the SPP is in the *LC form* (list of coalitions). A special case of the SPP in the LC form is obtained when participants do not care about the actual content of the coalitions, only about their sizes. Preferences are then called *anonymous*.

A more succinct representation is obtained when each participant i linearly orders only individual participants, or more precisely, a subset of them – these are *acceptable* for i. In this case the SPP is in the *LP form* (list of participants). With the exception of Stable Matchings, when the obtained partitions are allowed to contain only singletons or a two-element sets, preferences over participants have to be extended to preferences over coalitions. Algorithmically, the most intensively studied are the following extensions:

$\mathcal{B}$-preferences – a participant orders coalitions first on the basis of the most preferred (briefly best) member of the coalition, and if those are equal or tied, the coalition with smaller cardinality is preferred;

$\mathcal{W}$-preferences – a participant orders coalitions on the basis of the least preferred (briefly worst) member of the coalition;

$\mathcal{BW}$-preferences – a participant orders coalitions first on the basis of the best member of the coalition, and if those are equal or tied, the coalition with a more preferred worst member is preferred.

The above preferences are said to be *strict*, if the original preferences over individuals are strict linear orders and they are called *dichotomous* if all acceptable participants are tied in each preference list. The presence of ties very often leads to different computational results compared to the case with strict preferences.

In *additively separable* preferences it is supposed that for each $i \in N$ there exists a function $v_i : N \to \mathbb{R}$ such that i prefers a coalition S to coalition T if and only if $\sum_{j\in S} v_i(j) > \sum_{j\in T} v_i(j)$. Additively separable preferences and their various variants are studied in [7].

Another approach is presented in [16]. The authors call these preferences *simple* and it is supposed that for each participant i a set F_i of friends and a set E_i of enemies are given. A participant i has *appreciation of friends* when he prefers a coalition S to a coalition T if $|S \cap F_i| > |T \cap F_i|$ and he has *aversion against enemies* when he prefers a coalition S to a coalition T if $|S \cap E_i| < |T \cap E_i|$.

Stability Definitions

Let $M(i)$ denote the set of partition M that contains participant i.

Definition 1 A set $Z \subseteq N$ is called *blocking* for partition M, if each participant $i \in Z$ prefers Z to $M(i)$. A set $Z \subseteq N$ is called *weakly blocking* for partition M, if each participant $i \in Z$ prefers Z to $M(i)$ or is indifferent between Z and $M(i)$ and at least one participant $j \in Z$ prefers Z to $M(j)$.

A participant i is said to be *covered* if $|M(i)| \geq 2$.

In the literature, several different stability definitions were studied, including *Nash stability, individual stability, contractual individual stability, Pareto optimality* etc. An interested reader can consult [4] or [6]. Algorithmically, the most deeply studied notions are the core and the strong core.

Definition 2 A partition M is called a *core* partition, if there is no blocking set for M. A partition M is called a *strong core* partition, if there is no weakly blocking set for M.

Problems

Several decision or computational problems arise in the context of the SPP:

- StabilityTest: Given $(N,\mathcal{P})$ and a partition M of N, is M stable?

- EXISTENCE: Does a stable partition for a given $(N,\mathcal{P})$ exist?
- CONSTRUCTION: If a stable partition for a given $(N,\mathcal{P})$ exists, find one.
- STRUCTURE: Describe the structure of stable partitions for a given $(N,\mathcal{P})$.

Their complexity depends on the particular type of preferences used.

Key Results

SPP in LC Form

EXISTENCE for core partitions is NP-complete even when the given preferences over coalitions are strict or anonymous [3].

$\mathcal{W}$-preferences

The SPP with strict $\mathcal{W}$-preferences has many features similar to the Stable Roommates Problem [17]. First, each core partition set contains at most two participants and if a blocking set exists, then there is a blocking set of size at most 2, hence STABILITYTEST is polynomial. EXISTENCE and CONSTRUCTION are polynomial in the strict preferences case [11], which can be shown using an extension of Irving's Stable Roommates Algorithm (discussed in detail in [17]). This algorithm can also be used to derive some results for STRUCTURE. In the case of ties, EXISTENCE is NP-complete and a complete solution to STRUCTURE is not available [11].

$\mathcal{B}$-preferences

A polynomial algorithm for STABILITYTEST is given in [9]. For strict $\mathcal{B}$-preferences a core as well as strong core partition always exists and one can be found by the Top Trading Cycles algorithm attributed to Gale in [19] (an implementation of this algorithm of time complexity $O(m)$, where m is the total length of the preference lists of all participants, was described in [2]). However, if preferences of participants contain ties, EXISTENCE is NP-complete for both core and strict core [10]. In the dichotomous case, a core partition can be constructed in polynomial time, but EXISTENCE for strong core is NP-complete [8].

Very little is known about the STRUCTURE. Several questions about the existence of core partitions with special properties are shown to be NP-hard even for the strict preferences case [15]:

- Does a core partition M exist, such that $|M(i)| < |T(i)|$ for each participant i, where T is the partition obtained by the Top Trading Cycles algorithm?
- Does a core partition M exist, such that $|M(i)| \leq 3$ for each participant i?
- Does a core partition M exist that covers all participants?

Moreover, the maximum number of participants covered by a core partition is not approximable within $n^{1-\varepsilon}$ [5].

$\mathcal{BW}$-preferences

In the strict preferences case a core partition always exists and one can be obtained by the Top Trading Cycles algorithm. However, if preferences contain ties, EXISTENCE is NP-hard [12]. STABILITYTEST remains open.

Simple Preferences

If all the participants have aversion to enemies, a core partition always exists, but CONSTRUCTION is NP-hard. In the appreciation-of-friends case, a strong core partition always exists and CONSTRUCTION can be solved in $O(n^3)$ time, where n is the number of participants [16].

Applications

Stable partitions give rise to various economic and game theoretical models. They appear in the study of exchange economies with discrete commodities [19], in barter exchange markets [20], or in the study of formation of countries [14]. A recent application concerns exchange of kidneys for transplantation between willing donors and their incompatible intended recipients [18]. In this context, the use of $\mathcal{B}$-preferences was suggested in [8], as they express the wish of each patient for the best suitable kidney as well as his desire for the shortest possible exchange cycle.

Open Problems

Because of the great number of variants, a lot of open problems exist. In almost all cases, STRUCTURE is not satisfactorily solved. For instances with no stable partition, one may seek one that minimizes the number of participants who have an incentive to deviate. Parallel algorithms were also not studied.

Experimental Results

In the context of kidney exchange, Roth et al. in [18] performed extensive experiments with the Top Trading Cycles algorithm on simulated patients' data. The number of covered participants and sizes of the obtained partition sets were recorded. The structure of core partitions for $\mathcal{B}$-preferences was studied in [15]. Two heuristics were tested. The starting point was the stable partition obtained

by the Top Trading Cycles algorithm. Heuristic Cut-Cycle tried to split at least one of the obtained partition sets, Cut-and-Add tried to add an uncovered participant to an existing partition set on condition that the new partition remained in the core. It was shown that as the total number of participants grows, the percentage of participants uncovered in the Top Trading Cycles partition decreases and the percentage of successes of both heuristics grows.

Cross References

- ▶ Hospitals/Residents Problem
- ▶ Optimal Stable Marriage
- ▶ Ranked Matching
- ▶ Stable Marriage
- ▶ Stable Marriage with Ties and Incomplete Lists

Recommended Reading

1. Abraham, D., Blum, A., Sandholm, T.: Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges. EC'07, June 11–15, 2007, San Diego, California
2. Abraham, D., Cechlárová, K., Manlove, D., Mehlhorn, K.: Pareto-optimality in house allocation problems. In: Fleischer, R., Trippen, G. (eds.) Lecture Notes in Comp. Sci. Vol. 3341/2004, Algorithms and Computation, 14th Int. Symposium ISAAC 2004, pp. 3–15. Hong Kong, December 2004
3. Ballester, C.: NP-completeness in Hedonic Games. Games. Econ. Behav. **49**(1), 1–30 (2004)
4. Banerjee, S., Konishi, H., Sönmez, T.: Core in a simple coalition formation game. Soc. Choice. Welf. **18**, 135–153 (2001)
5. Biró, P., Cechlárová, K.: Inapproximability of the kidney exchange problem. Inf. Proc. Lett. **101**(5), 199–202 (2007)
6. Bogomolnaia, A., Jackson, M.O.: The Stability of Hedonic Coalition Structures. Games. Econ. Behav. **38**(2), 201–230 (2002)
7. Burani, N., Zwicker, W.S.: Coalition formation games with separable preferences. Math. Soc. Sci. **45**, 27–52 (2003)
8. Cechlárová, K., Fleiner, T., Manlove, D.: The kidney exchange game. In: Zadnik-Stirn, L., Drobne, S. (eds.) Proc. SOR '05, pp. 77–83. Nova Gorica, September 2005
9. Cechlárová, K., Hajduková, J.: Stability testing in coalition formation games. In: Rupnik, V., Zadnik-Stirn, L., Drobne, S. (eds.) Proceedings of SOR'99, pp. 111–116. Predvor, Slovenia (1999)
10. Cechlárová, K., Hajduková, J.: Computational complexity of stable partitions with $\mathcal{B}$-preferences. Int. J. Game. Theory **31**(3), 353–364 (2002)
11. Cechlárová, K., Hajduková, J.: Stable partitions with $\mathcal{W}$-preferences. Discret. Appl. Math. **138**(3), 333–347 (2004)
12. Cechlárová, K., Hajduková, J.: Stability of partitions under WB-preferences and BW-preferences. Int. J. Inform. Techn. Decis. Mak. Special Issue on Computational Finance and Economics. **3**(4), 605–614 (2004)
13. Cechlárová, K., Romero-Medina, A.: Stability in coalition formation games. Int. J. Game. Theor. **29**, 487–494 (2001)
14. Cechlárová, K., Dahm, M., Lacko, V.: Efficiency and stability in a discrete model of country formation. J. Glob. Opt. **20**(3–4), 239–256 (2001)
15. Cechlárová, K., Lacko, V.: The Kidney Exchange problem: How hard is it to find a donor? IM Preprint A4/2006, Institute of Mathematics, P.J. Šafárik University, Košice, Slovakia, (2006)
16. Dimitrov, D., Borm, P., Hendrickx, R., Sung, S. Ch.: Simple priorities and core stability in hedonic games. Soc. Choice. Welf. **26**(2), 421–433 (2006)
17. Gusfield, D., Irving, R.W.: The Stable Marriage Problem. Structure and Algorithms. MIT Press, Cambridge (1989)
18. Roth, A., Sönmez, T., Ünver, U.: Kidney Exchange. Quarter. J. Econ. **119**, 457–488 (2004)
19. Shapley, L., Scarf, H.: On cores and indivisibility. J. Math. Econ. **1**, 23–37 (1974)
20. Yuan, Y.: Residence exchange wanted: A stable residence exchange problem. Eur. J. Oper. Res. **90**, 536–546 (1996)

Stackelberg Games: The Price of Optimum

2006; Kaporis, Spirakis

Alexis Kaporis[1], Paul Spirakis[2]

[1] Department of Computer Engineering & Informatics, University of Patras, Patras, Greece

[2] Computer Engineering and Informatics, Research and Academic Computer Technology Institute, Patras University, Patras, Greece

Keywords and Synonyms

Cournot game; Coordination ratio

Problem Definition

Stackelberg games [15] may model the interplay amongst an authority and rational individuals that selfishly demand resources on a large scale network. In such a game, the authority *(Leader)* of the network is modeled by a distinguished player. The selfish users *(Followers)* are modeled by the remaining players.

It is well known that selfish behavior may yield a *Nash Equilibrium* with cost arbitrarily higher than the optimum one, yielding unbounded *Coordination Ratio* or *Price of Anarchy (PoA)* [7,13]. Leader plays his strategy first assigning a portion of the total demand to some resources of the network. Followers observe and react selfishly assigning their demand to the most appealing resources. Leader aims to drive the system to an a posteriori Nash equilibrium with cost close to the overall optimum one [4,6,8,10]. Leader may also eager for his own rather than system's performance [2,3].

A Stackelberg game can be seen as a special, and easy [6] to implement, case of *Mechanism Design*. It avoids the complexities of either computing taxes or assigning

prices, or even designing the network at hand [9]. However, a central authority capable to control the overall demand on the resources of a network may be unrealistic in networks which evolute and operate under the effect of many and diversing economic entities. A realistic way [4] to act centrally even in large nets could be via *Virtual Private Networks (VPNs)* [1]. Another flexible way is to combine such strategies with *Tolls* [5,14].

A dictator controlling the entire demand optimally on the resources surely yields $PoA = 1$. On the other hand, rational users do prefer a liberal world to live. Thus, it is important to compute the optimal Leader-strategy which controls the *minimum* of the resources *(Price of Optimum)* and yields $PoA = 1$. What is the complexity of computing the Price of Optimum? This is not trivial to answer, since the Price of Optimum depends crucially on computing an optimal Leader strategy. In particular, [6] proved that computing the optimal Leader strategy is hard.

The central result of this lemma is Theorem 5. It says that on nonatomic flows and arbitrary s-t networks & latencies, computing the minimum portion of flow and Leader's optimal strategy sufficient to induce $PoA = 1$ is easy [10].

Problem $(G(V, E), s, t \in V, r)$
INPUT: *Graph G, $\forall e \in E$ latency ℓ_e, flow r, a source-destination pair (s, t) of vertices in V.*
OUTPUT: *(i) The minimum portion α_G of the total flow r sufficient for an optimal Stackelberg strategy to induce the optimum on G. (ii) The optimal Stackelberg strategy.*

Models & Notations

Consider a graph $G(V, E)$ with parallel edges allowed. A number of rational and selfish users wish to route from a given source s to a destination node t an amount of flow r. Alternatively, consider a partition of users in k commodities, where user(s) in commodity i wish to route flow r_i through a source-destination pair (s_i, t_i), for each $i = 1, \ldots, k$. Each edge $e \in E$ is associated to a latency function $\ell_e()$, positive, differentiable and strictly increasing on the flow traversing it.

Nonatomic Flows There are infinitely many users, each routing his infinitesimally small amount of the total flow r_i from a given source s_i to a destination vertex t_i in graph $G(V, E)$. A flow f is an assignment of jobs f_e on each edge $e \in E$. The cost of the injected flow f_e (satisfying the standard constraints of the corresponding network-flow problem) that traverses edge $e \in E$ equals $c_e(f_e) = f_e \times \ell_e(f_e)$. It is assumed that on each edge e the cost is convex with respect the injected flow f_e. The overall system's cost is the sum $\sum_{e \in E} f_e \times \ell_e(f_e)$ of all edge-costs in G. Let $f_{\mathcal{P}}$ the amount of flow traversing the s_i-t_i path $\mathcal{P}$. The latency $\ell_{\mathcal{P}}(f)$ of s_i-t_i path $\mathcal{P}$ is the sum $\sum_{e \in \mathcal{P}} \ell_e(f_e)$ of latencies per edge $e \in \mathcal{P}$. The cost $C_{\mathcal{P}}(f)$ of s_i-t_i path $\mathcal{P}$ equals the flow $f_{\mathcal{P}}$ traversing it multiplied by path-latency $\ell_{\mathcal{P}}(f)$. That is, $C_{\mathcal{P}}(f) = f_{\mathcal{P}} \times \sum_{e \in \mathcal{P}} \ell_e(f_e)$.

In an Nash equilibrium, all s_i-t_i paths traversed by nonatomic users in part i have a common latency, which is at most the latency of any untraversed s_i-t_i path. More formally, for any part i and any pair $\mathcal{P}_1, \mathcal{P}_2$ of s_i-t_i paths, if $f_{\mathcal{P}_1} > 0$ then $\ell_{\mathcal{P}_1}(f) \leq \ell_{\mathcal{P}_2}(f)$. By the convexity of edge-costs the Nash equilibrium is unique and computable in polynomial time given a floating-point precision. Also computable is the unique *Optimum* assignment O of flow, assigning flow o_e on each $e \in E$ and minimizing the overall cost $\sum_{e \in E} o_e \ell_e(o_e)$. However, not all optimally traversed s_i-t_i paths experience the same latency. In particular, users traversing paths with high latency have incentive to reroute towards more speedy paths. Therefore the optimal assignment is unstable on selfish behavior.

A Leader dictates a *weak* Stackelberg strategy if on each commodity $i = 1, \ldots, k$ controls a fixed α portion of flow r_i, $\alpha \in [0, 1]$. A *strong* Stackelberg strategy is more flexible, since Leader may control $\alpha_i r_i$ flow in commodity i such that $\sum_{i=1}^{k} \alpha_i = \alpha$. Let a Leader dictating flow s_e on edge $e \in E$. The a posteriori latency $\widetilde{\ell}_e(n_e)$ of edge e, with respect to the induced flow n_e by the selfish users, equals $\widetilde{\ell}_e(n_e) = \ell_e(n_e + s_e)$. In the a posteriori Nash equilibrium, all s_i-t_i paths traversed by the free selfish users in commodity i have a common latency, which is at most the latency of any selfishly untraversed path, and its cost is $\sum_{e \in E}(n_e + s_e) \times \widetilde{\ell}_e(n_e)$.

Atomic Splittable Flows There is a finite number of atomic users $1, \ldots, k$. Each user i is responsible for routing a non-negligible flow-amount r_i from a given source s_i to a destination vertex t_i in graph G. In turn, each flow-amount r_i consists of infinitesimally small jobs.

Let flow f assigning jobs f_e on each edge $e \in E$. Each edge-flow f_e is the sum of partial flows $f_e^1, \ldots, f_e^k$ injected by the corresponding users $1, \ldots, k$. That is, $f_e = f_e^1 + \cdots + f_e^k$. As in the model above, the latency on a given s_i-t_i path $\mathcal{P}$ is the sum $\sum_{e \in \mathcal{P}} \ell_e(f_e)$ of latencies per edge $e \in \mathcal{P}$. Let $f_{\mathcal{P}}^i$ be the flow that user i ships through an s_i-t_i path $\mathcal{P}$. The cost of user i on a given s_i-t_i path $\mathcal{P}$ is analogous to her path-flow $f_{\mathcal{P}}^i$ routed via $\mathcal{P}$ times the total path-latency $\sum_{e \in \mathcal{P}} \ell_e(f_e)$. That is, the path-cost equals $f_{\mathcal{P}}^i \times \sum_{e \in \mathcal{P}} \ell_e(f_e)$. The overall cost $C_i(f)$ of user i is the sum of the corresponding path-costs of all s_i-t_i paths.

In a Nash equilibrium no user i can improve his cost $C_i(f)$ by rerouting, given that any user $j \neq i$ keeps his

routing fixed. Since each atomic user minimizes its cost, if the game consists of only one user then the cost of the Nash equilibrium coincides to the optimal one.

In a Stackelberg game, a distinguished atomic Leader-player controls flow r_0 and plays first assigning flow s_e on edge $e \in E$. The a posteriori latency $\widetilde{\ell}_e(x)$ of edge e on induced flow x equals $\widetilde{\ell}_e(x) = \ell_e(x + s_e)$. Intuitively, after Leader's move, the induced selfish play of the k atomic users is equivalent to atomic splittable flows on a graph where each initial edge-latency ℓ_e has been mapped to $\widetilde{\ell}_e$. In game-parlance, each atomic user $i \in \{1, \ldots, k\}$, having *fixed* Leader's strategy, computes his *best reply* against all others atomic users $\{1, \ldots, k\} \setminus \{i\}$. If n_e is the induced Nash flow on edge e this yields total cost $\sum_{e \in E}(n_e + s_e) \times \widetilde{\ell}_e(n_e)$.

Atomic Unsplittable Flows The users are finite $1, \ldots, k$ and user i is allowed to sent his non-negligible job r_i only on a *single* path. Despite this restriction, all definitions given in atomic splittable model remain the same.

Key Results

Let us see first the case of atomic splittable flows, on parallel M/M/1 links with different speeds connecting a given source-destination pair of vertices.

Theorem 1 (Korilis, Lazar, Orda [6]) *The Leader can enforce in polynomial time the network optimum if she controls flow r_0 exceeding a critical value $\underline{r}^0$.*

In the sequel, we focus on nonatomic flows on s-t graphs with parallel links. In [6] primarily were studied cases that Leader's flow cannot induce network's optimum and was shown that an optimal Stackelberg strategy is easy to compute. In this vain, if s-t parallel-links instances are restricted to ones with linear latencies of equal slope then an optimal strategy is easy [4].

Theorem 2 (Kaporis, Spirakis [4]) *The optimal Leader strategy can be computed in polynomial time on any instance (G, r, α) where G is an s-t graph with parallel-links and linear latencies of equal slope.*

Another positive result is that the optimal strategy can be approximated within $(1 + \epsilon)$ in polynomial time, given that link-latencies are polynomials with non-negative coefficients.

Theorem 3 (Kumar, Marathe [8]) *There is a* fully polynomial approximate *Stackelberg scheme that runs in $poly(m, \frac{1}{\epsilon})$ time and outputs a strategy with cost $(1 + \epsilon)$ within the optimum strategy.*

For parallel link s-t graphs with arbitrary latencies more can be achieved: in polynomial time a "threshold" value α_G is computed, sufficient for the Leader's portion to induce the optimum. The complexity of computing optimal strategies changes in a dramatic way around the critical value α_G from "hard" to "easy" (G, r, α) Stackelberg scheduling instances. Call α_G as the *Price of Optimum* for graph G.

Theorem 4 (Kaporis, Spirakis [4]) *On input an s-t parallel link graph G with arbitrary strictly increasing latencies the minimum portion α_G sufficient for a Leader to induce the optimum, as well as her optimal strategy, can be computed in polynomial time.*

As a conclusion, the Price of Optimum α_G essentially captures the hardness of instances (G, r, α). Since, for Stackelberg scheduling instances $(G, r, \alpha \geq \alpha_G)$ the optimal Leader strategy yields $PoA = 1$ and it is computed as hard as in P, while for $(G, r, \alpha < \alpha_G)$ the optimal strategy yields $PoA < 1$ and it is as easy as NP [10].

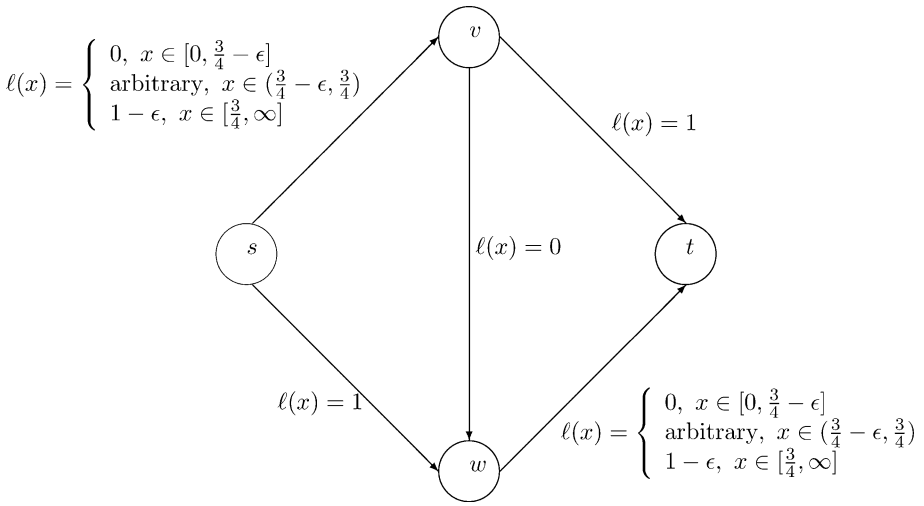
The results above are limited to parallel-links connecting a given s-t pair of vertices. Is it possible to efficiently compute the Price of Optimum for nonatomic flows on arbitrary graphs? This is not trivial to settle. Not only because it relies on computing an optimal Stackelberg strategy, which is hard to tackle [10], but also because Proposition B.3.1 in [11] ruled out previously known performance guarantees for Stackelberg strategies on general nets.

The central result of this lemma is presented below and completely resolves this question (extending Theorem 4).

Theorem 5 (Kaporis, Spirakis [4]) *On arbitrary s-t graphs G with arbitrary latencies the minimum portion α_G sufficient for a Leader to induce the optimum, as well as her optimal strategy, can be computed in polynomial time.*

Example

Consider the optimum assignment O of flow r that wishes to travel from source vertex s to sink t. O assigns flow o_e incurring latency $\ell_e(o_e)$ per edge $e \in G$. Let $\mathcal{P}_{s \to t}$ the set of all s-t paths. The *shortest paths* in $\mathcal{P}_{s \to t}$ with respect to costs $\ell_e(o_e)$ per edge $e \in G$ can be computed in polynomial time. That is, the paths that given flow assignment O attain latency: $\min_{P \in \mathcal{P}_{s \to t}} \left(\sum_{e \in P} \ell_e(o_e)\right)$ i. e., minimize their latency. It is crucial to observe that, if we want the *induced* Nash assignment by the Stackelberg strategy to attain the optimum cost, then these shortest paths are *the only choice* for selfish users that eager to travel from s to t. Furthermore, the uniqueness of the optimum assignment O determines the minimum part of flow which can be selfishly scheduled on these shortest paths. Observe that any



Stackelberg Games: The Price of Optimum, Figure 1
A bad example for Stackelberg routing

flow assigned by O on a non-shortest s-t path has incentive to opt for a shortest one. Then a Stackelberg strategy *must* frozen the flow on all non-shortest s-t paths.

In particular, the idea sketched above achieves coordination ratio 1 on the graph in Fig. 1. On this graph Roughgarden proved that $\frac{1}{\alpha}\times$ (optimum cost) guarantee is *not* possible for general (s,t)-networks, Appendix B.3 in [11]. The optimal edge-flows are ($r = 1$):

$$o_{s\to v} = \frac{3}{4} - \epsilon\,,\ o_{s\to w} = \frac{1}{4} + \epsilon\,,\ o_{v\to w} = \frac{1}{2} - 2\epsilon\,,$$
$$o_{v\to t} = \frac{1}{4} + \epsilon\,,\ o_{w\to t} = \frac{3}{4} - \epsilon$$

The shortest path $P_0 \in \mathcal{P}$ with respect to the optimum O is $P_0 = s \to v \to w \to t$ (see [11] pp. 143, 5th-3th lines before the end) and its flow is $f_{P_0} = \frac{1}{2} - 2\epsilon$. The non shortest paths are: $P_1 = s \to v \to t$ and $P_2 = s \to w \to t$ with corresponding optimal flows: $f_{P_1} = \frac{1}{4} + \epsilon$ and $f_{P_2} = \frac{1}{4} + \epsilon$. Thus the Price of Optimum is

$$f_{P_1} + f_{P_2} = \frac{1}{2} + 2\epsilon = r - f_{P_0}$$

Applications

Stackelberg strategies are widely applicable in networking [6], see also Section 6.7 in [12].

Open Problems

It is important to extend the above results on atomic unsplittable flows.

Cross References

- Algorithmic Mechanism Design
- Best Response Algorithms for Selfish Routing
- Facility Location
- Non-approximability of Bimatrix Nash Equilibria
- Price of Anarchy
- Selfish Unsplittable Flows: Algorithms for Pure Equilibria

Recommended Reading

1. Birman, K.: Building Secure and Reliable Network Applications. Manning, (1996)
2. Douligeris, C., Mazumdar, R.: Multilevel flow control of Queues. In: Johns Hopkins Conference on Information Sciences, Baltimore, 22–24 Mar 1989 (2006)
3. Economides, A., Silvester, J.: Priority load sharing: an approach using stackelberg games. In: 28th Annual Allerton Conference on Communications, Control and Computing (1990)
4. Kaporis, A., Spirakis, P.G.: Stackelberg games on arbitrary networks and latency functions. In: 18th ACM Symposium on Parallelism in Algorithms and Architectures (2006)
5. Karakostas, G., Kolliopoulos, G.: Stackelberg strategies for selfish routing in general multicommodity networks. Technical report, Advanced Optimization Laboratory, McMaster Univercity (2006) AdvOL2006/08, 2006-06-27
6. Korilis, Y.A., Lazar, A.A., Orda, A.: Achieving network optima using stackelberg routing strategies. IEEE/ACM Trans. Netw. **5**(1), 161–173 (1997)
7. Koutsoupias, E., Papadimitriou, C.: Worst-case equilibria. In: 16th Symposium on Theoretical Aspects in Computer Science, Trier, Germany. LNCS, vol. 1563, pp. 404–413. Springer (1999)
8. Kumar, V.S.A., Marathe, M.V.: Improved results for stackelberg scheduling strategies. In: 29th International Colloquium, Au-

tomata, Languages and Programming. LNCS, pp. 776–787. Springer (2002)
9. Roughgarden, T.: Designing networks for selfish users is hard. In: 42nd IEEE Annual Symposium of Foundations of Computer Science, pp. 472–481 (2001)
10. Roughgarden, T.: Stackelberg scheduling strategies. In: 33rd ACM Annual Symposium on Theory of Computing, pp. 104–113 (2001)
11. Roughgarden, T.: Selfish Routing. Dissertation, Cornell University, USA, May 2002, http://theory.stanford.edu/~tim/
12. Roughgarden, T.: Selfish Routing and the Price of Anarchy. The MIT Press, Cambridge (2005)
13. Roughgarden, T., Tardos, E.: How bad is selfish routing? In: 41st IEEE Annual Symposium of Foundations of Computer Science, pp. 93–102. J. ACM 49(2), pp 236–259, 2002, ACM, New York (2000)
14. Swamy, C.: The effectiveness of stackelberg strategies and tolls for network congestion games. In: ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA (2007)
15. von Stackelberg, H.: Marktform und Gleichgewicht. Springer, Vienna (1934)

Statistical Data Compression

► Arithmetic Coding for Data Compression

Statistical Multiple Alignment

2003; Hein, Jensen, Pedersen

István Miklós
Department of Plant Taxonomy and Ecology,
Eötvös Lóránd University, Budapest, Hungary

Keywords and Synonyms

Evolutionary hidden Markov models

Problem Definition

The three main types of mutations modifying biological sequences are insertions, deletions and substitutions. The simplest model involving these three types of mutations is the so-called Thorne–Kishino–Felsenstein model [13]. In this model, the characters of a sequence evolve independently. Each character in the sequence can be substituted with another character according to a prescribed reversible time-continuous Markov model on the possible characters. Insertion-deletions are modeled as a birth-death process, characters evolve independently and identically, with insertion and deletion rates λ and μ.

The multiple statistical alignment problem is to calculate the likelihood of a set of sequences, namely, what is the probability of observing a set of sequences, given all the necessary parameters that describe the evolution of sequences. Hein, Jensen and Pedersen were the first who gave an algorithm to calculate this probability [4]. Their algorithm has $O(5^n L^n)$ running time, where n is the number of sequences, and L is the geometric mean of the sequences. The running time has been improved to $O(2^n L^n)$ by Lunter et al. [10].

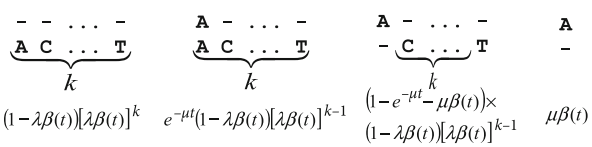
Notations

Insertions and Deletions In the Thorne–Kishino–Felsenstein model (TKF91 model) [13], both the birth and the death processes are Poisson processes with parameters λ and μ, respectively. Since each character evolves independently, the probability of an insertion-deletion pattern given by an alignment can be calculated as the product of the probabilities of patterns. Each pattern starts with an ancestral character, except the first that starts with the beginning of the alignment, end ends before the next ancestral character, except the last that ends at the end of the alignment. The probability of the possible patterns can be found on Fig. 1.

Evolutionary Trees An evolutionary tree is a leaf-labeled, edge weighted, rooted binary tree. Labels are the species related by the evolutionary tree, weights are evolutionary distances. It might happen that the evolutionary changes had different speed at different lineages, and hence the tree is not necessary ultrametric, namely, the root not necessary has the same distance to all leaves.

Given a set S of l-long sequences over alphabet Σ, a substitution model M on Σ and an evolutionary tree T labeled by the sequences. The likelihood of the tree is the probability of observing the sequences at the leaves of the tree, given that the substitution process starts at the root of the tree with the equilibrium distribution. This likelihood is denoted by $P(S|T, M)$. The substitution likelihood problem is to calculate the likelihood of the tree.

Let Σ be a finite alphabet and let $S_1 = s_{1,1}s_{1,2}\dots s_{1,L_1}$, $S_2 = s_{2,1}s_{2,2}\dots s_{2,L_2}$, $\dots$ $S_n = s_{n,1}s_{n,2}\dots s_{n,L_n}$ be se-



Statistical Multiple Alignment, Figure 1
The probabilities of alignment patterns. From *left* to *right*: k insertions at the beginning of the alignment, a match followed by $k-1$ insertions, a deletion followed by k insertions, a deletion not followed by insertions. $\beta = \frac{1-e^{(\lambda-\mu)t}}{\mu-\lambda e^{(\lambda-\mu)t}}$

quences over this alphabet. Let a TKF91 model $TKF91$ be given with its parameters: substitution model M, insertion rate λ and deletion rate μ. Let T be an evolutionary tree labeled by $S_1, S_2 \ldots S_n$. The multiple statistical alignment problem is to calculate the likelihood of the tree, $P(S_1, S_2, \ldots S_n | T, TKF91)$, given that the TKF91 process starts at the root with the equilibrium distribution.

Multiple Hidden Markov Models It will turn out that the TKF91 model can be transformed to a multiple Hidden Markov Model, therefore it is formally defined here. A multiple Hidden Markov Model (multiple-HMM) is a directed graph with a distinguished Start and End state, (the in-degree of the Start and the out-degree of the End state are both 0), together with the following described transition and emission distributions. Each vertex has a transition distribution over its out-edges. The vertexes can be divided for two classes, the emitting and silent states. Each emitting state emits one-one random character to a prescribed set of sequences, it is possible that a state emits only one character to one sequence. For each state, an emission distribution over the alphabet and the set of sequences gives the probabilities which characters will be emitted to which sequences. The Markov process is a random walk from the Start to the End, following the transition distribution on the out edges. When the walk is in an emitting state, characters are emitted according to the emission distribution of the state. The process is hidden since the observer sees only the emitted sequences, and the observer does not observe which character is emitted by which state, even the observer does not see which characters are co-emitted. The multiple-HMM problem is to calculate the emission probability of a set of sequences for a multiple-HMM. This probability can be calculated with the Forward algorithm that has $O(V^2 L^n)$ running time, where V is the number of emitting states in the multiple-HMM, L is the geometric mean of the sequences and n is the number of sequences [2].

Key Results

Substitutions have been modeled with time-continuous Markov models since the late sixties [7] and an efficient algorithm for likelihood calculations was published in 1981 [3]. The running time of this efficient algorithm grows linearly both with the number of sequences and with the length of the sequences being analyzed, and it grows squarely with the size of the alphabet. The algorithm belongs to the class of dynamic programming algorithms.

Thorne, Kishino and Felsenstein gave an $O(nm)$ running time algorithm for calculating the likelihood of an n-long and an m-long sequence under their model [13]. It was not clear for long time how to extend this algorithm to more than two sequences. In 2001, several researchers [6,11] realized that the TKF91 model for two sequences is equivalent with a pair Hidden Markov Model (pair-HMM) in the sense that the transition and emission probabilities of the pair-HMM can be parameterized with λ, μ, and the transition and equilibrium probabilities of the substitution model, moreover there is a bijection between the paths emitting the two sequences and alignments such that the probability of a path in the pair-HMM equals to the probability of the corresponding alignment of the two sequences. Hence the likelihood of two sequences can be calculated with the Forward algorithm of the pair-HMM.

After this discovery, it was relatively easy to develop an algorithm for multiple statistical alignment [4]. The key observation is that a multiple-HMM can be created as a composition of pair-HMMs along the evolutionary tree. This technique was already known in the speech recognition literature [12], and was also rediscovered by Ian Holmes [5], who named this technique as transducer composition. The number of states in the so-created multiple-HMM is $O(5^{\frac{n}{2}})$, where n is the number of leaves of the tree. The emission probabilities are the substitution likelihoods on the tree, which can be efficiently calculated using Felsenstein's algorithm [3]. The running time of the Forward algorithm is $5^n L^n$, where L is the geometric mean of the sequence lengths.

Lunter et al. [10] introduced an algorithm that does not need a multiple-HMM description of the TKF91 model to calculate the likelihood of a tree. Using a logical sieve algorithm, they were able to reduce the running time to $O(2^n L^n)$. They called their algorithm the "one-state recursion" since their dynamic programming algorithm does not need different state of a multiple-HMM to calculate the likelihood correctly.

Applications

Since the running time of the best known algorithm for multiple statistical alignment grows exponentially with the number of sequences, on its own it is not useful in practice. However, Lunter et al. also showed that there is a one-state recursion to calculate the likelihood of the tree given an alignment [8]. The running time of this algorithm grows only linearly with both the alignment length and the number of sequences. Since the number of states in a multiple-HMM that can emit the same multiple alignment column might grow exponentially, this version of the one-state recursion is a significant improvement. The one-state recur-

sion for multiple alignments is used in a Bayesian Markov chain Monte Carlo where the state space is the Descart product of the possible multiple alignments and evolutionary trees. The one-state recursion provides an efficient likelihood calculation for a point in the state space [9].

Csűrös and Miklós introduced a model for gene content evolution that is equivalent with the multiple statistical alignment problem for alphabet size 1 [1]. They gave a polynomial running time algorithm that calculates the likelihood of the tree. The running time is $O(n + hL^2)$, where n is the number of sequences, h is the height of the evolutionary tree, and L is the sum of the sequences lengths.

Open Problems

It is conjectured that the multiple statistical alignment problem cannot be solved in polynomial time for any nontrivial alphabet size. One also can ask what the most likely multiple alignment is or equivalently, what the most probable path in the multiple-HMM is that emits the given sequences. For a set of sequences, a TKF91 model and an evolutionary tree, the decision problem "Is there a multiple alignment that is more probable than p" is conjectured to be NP-complete.

Thorne, Kishino and Felsenstein also introduced a fragment model, also called the TKF92 model, in which multiple insertions and deletions are allowed. The birth process is still a Poisson process, but instead of single characters, fragments of characters are inserted with a geometrically distributed length. The fragments are unbreakable, and the death process is going on the fragments. The TKF92 model for a pair of sequences also can be described into a pair-HMM and the TKF92 model on a tree can be transformed to a multiple-HMM. It is conjectured that there is no one-state recursion for the TKF92 model.

Cross References

- ► Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds
- ► Local Alignment (with Affine Gap Weights)

Recommended Reading

1. Csűrös, M., Miklós, I.: A probabilistic model for gene content evolution with duplication, loss, and horizontal transfer. In: Lecture Notes in Bioinformatics, Proceedings of RECOMB2006, vol. 3909, pp. 206–220 (2006)
2. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological sequence analysis. Cambridge University Press, Cambridge, UK (1998)
3. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17**, 368–376 (1981)
4. Hein, J., Jensen, J., Pedersen, C.: Recursions for statistical multiple alignment. PNAS **100**, 14,960–14,965 (2003)
5. Holmes, I.: Using guide trees to construct multiple-sequence evolutionary hmms. Bioinform. **19**, i147–i157 (2003)
6. Holmes, I., Bruno, W.J.: Evolutionary HMMs: a Bayesian approach to multiple alignment. Bioinform. **17**(9), 803–820 (2001)
7. Jukes, T.H., Cantor, C.R.: Evolution of protein molecules. In: Munro (ed.) Mammalian Protein Metabolism, pp. 21–132. Acad. Press (1969)
8. Lunter, G., Miklós, I., Drummond, A., Jensen, J., Hein, J.: Bayesian phylogenetic inference under a statistical indel model. In: Lecture Notes in Bioinformatics, Proceedings of WABI2003, vol. 2812, pp. 228–244 (2003)
9. Lunter, G., Miklós, I., Drummond, A., Jensen, J., Hein, J.: Bayesian coestimation of phylogeny and sequence alignment. BMC Bioinformatics (2005)
10. Lunter, G.A., Miklós, I., Song, Y.S., Hein, J.: An efficient algorithm for statistical multiple alignment on arbitrary phylogenetic trees. J. Comp. Biol. **10**(6), 869–889 (2003)
11. Metzler, D., Fleißner, R., Wakolbringer, A., von Haeseler, A.: Assessing variability by joint sampling of alignments and mutation rates. J. Mol. Evol. **53**, 660–669 (2001)
12. Pereira, F., Riley, M.: Speech recognition by composition of weighted finite automata. In: Finite-State Language Processing, pp. 149–173. MIT Press, Cambridge (1997)
13. Thorne, J.L., Kishino, H., Felsenstein, J.: An evolutionary model for maximum likelihood alignment of DNA sequences. J. Mol. Evol. **33**, 114–124 (1991)

Statistical Query Learning

1998; Kearns

Vitaly Feldman
Department of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA

Problem Definition

The problem deals with learning $\{-1, +1\}$-valued functions from random labeled examples in the presence of random noise in the labels. In the *random classification noise* model of Angluin and Laird [1] the label of each example given to the learning algorithm is flipped randomly and independently with some fixed probability η called the *noise rate*. The model is the extension of Valiant's PAC model [14] that formalizes the simplest type of white label noise.

Robustness to this relatively benign noise is an important goal in the design of learning algorithms. Kearns defined a powerful and convenient framework for constructing noise-tolerant algorithms based on *statistical queries*. Statistical query (SQ) learning is a natural restriction of PAC learning that models algorithms that use statistical properties of a data set rather than individual examples.

Kearns demonstrated that any learning algorithm that is based on statistical queries can be automatically converted to a learning algorithm in the presence of random classification noise of arbitrary rate smaller than the information-theoretic barrier of 1/2. This result was used to give the first noise-tolerant algorithm for a number of important learning problems. In fact, virtually all known noise-tolerant PAC algorithms were either obtained from SQ algorithms or can be easily cast into the SQ model.

Definitions and Notation

Let C be a class of $\{-1, +1\}$-valued functions (also called *concepts*) over an input space X. In the basic PAC model a learning algorithm is given examples of an unknown function f from C on points randomly chosen from some unknown distribution $\mathcal{D}$ over X and should produce a hypothesis h that approximates f. More formally, an *example oracle* $\mathrm{EX}(f, \mathcal{D})$ is an oracle that upon being invoked returns an example $\langle x, f(x)\rangle$, where x is chosen randomly with respect to $\mathcal{D}$, independently of any previous examples. A learning algorithm for C is an algorithm that for every $\varepsilon > 0$, $\delta > 0$, $f \in C$, and distribution $\mathcal{D}$ over X, given ε, δ, and access to $\mathrm{EX}(f, \mathcal{D})$ outputs, with probability at least $1 - \delta$, a hypothesis h that ε-approximates f with respect to $\mathcal{D}$ (i. e. $\mathbf{Pr}_{\mathcal{D}}[f(x) \neq h(x)] \leq \varepsilon$). Efficient learning algorithms are algorithms that run in time polynomial in $1/\varepsilon$, $1/\delta$, and the size of the learning problem s. The size of a learning problem is determined by the description length of f under some fixed representation scheme for functions in C and the description length of an element in X (often proportional to the dimension n of the input space).

A number of variants of this basic framework are commonly considered. The basic PAC model is also referred to as *distribution-independent* learning to distinguish it from *distribution-specific* PAC learning in which the learning algorithm is required to learn with respect to a single distribution $\mathcal{D}$ known in advance. A *weak* learning algorithm is a learning algorithm that can produce a hypothesis whose error on the target concept is noticeably less than 1/2 (and not necessarily any $\varepsilon > 0$). More precisely, a weak learning algorithm produces a hypothesis h such that $\mathbf{Pr}_{\mathcal{D}}[f(x) \neq h(x)] \leq 1/2 - 1/p(s)$ for some fixed polynomial p. The basic PAC model is often referred to as *strong* learning in this context.

In the random classification noise model $\mathrm{EX}(f, \mathcal{D})$ is replaced by a faulty oracle $\mathrm{EX}^{\eta}(f, \mathcal{D})$, where η is the noise rate. When queried, this oracle returns a noisy example $\langle x, b\rangle$ where $b = f(x)$ with probability $1 - \eta$ and $\neg f(x)$ with probability η independently of previous examples. When η approaches 1/2 the label of the corrupted example approaches the result of a random coin flip, and therefore the running time of learning algorithms in this model is allowed to depend on $\frac{1}{1-2\eta}$ (the dependence must be polynomial for the algorithm to be considered efficient). For simplicity one usually assumes that η is known to the learning algorithm. This assumption can be removed using a simple technique due to Laird [12].

To formalize the idea of learning from statistical properties of a large number of examples, Kearns introduced a new oracle $\mathrm{STAT}(f, \mathcal{D})$ that replaces $\mathrm{EX}(f, \mathcal{D})$. The oracle $\mathrm{STAT}(f, \mathcal{D})$ takes as input a *statistical query* (SQ) of the form (χ, τ) where χ is a $\{-1, +1\}$-valued function on labeled examples and $\tau \in [0, 1]$ is the *tolerance* parameter. Given such a query the oracle responds with an estimate of $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ that is accurate to within an additive $\pm\tau$. Chernoff bounds easily imply that $\mathrm{STAT}(f, \mathcal{D})$ can, with high probability, be simulated using $\mathrm{EX}(f, \mathcal{D})$ by estimating $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ on $O(\tau^{-2})$ examples. Therefore the SQ model is a restriction of the PAC model. Efficient SQ algorithms allow only efficiently evaluable χ's and impose an inverse polynomial lower bound on the tolerance parameter over all oracle calls.

Key Results

Statistical Queries and Noise-Tolerance

The main result given by Kearns is a way to simulate statistical queries using noisy examples.

Lemma 1 ([10]) *Let (χ, τ) be a statistical query such that χ can be evaluated on any input in time T and let $EX^{\eta}(f, \mathcal{D})$ be a noisy oracle. The value $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ can, with probability at least $1 - \delta$, be estimated within τ using $O(\tau^{-2}(1-2\eta)^{-2}\log(1/\delta))$ examples from $EX^{\eta}(f, \mathcal{D})$ and time $O(\tau^{-2}(1-2\eta)^{-2}\log(1/\delta)\cdot T)$.*

This simulation is based on estimating several probabilities using examples from the noisy oracle and then offsetting the effect of noise. The lemma implies that any efficient SQ algorithm for a concept class C can be converted to an efficient learning algorithm for C tolerating random classification noise of any rate $\eta < 1/2$.

Theorem 2 ([10]) *Let C be a concept class efficiently PAC learnable from statistical queries. Then C is efficiently PAC learnable in the presence of random classification noise of rate η for any $\eta < 1/2$.*

Kearns also shows that in order to simulate all the statistical queries used by an algorithm one does not necessarily need new examples for each estimation. Instead, assuming that the set of possible queries of the algorithm has Vapnik–Chervonenkis dimension d, all its statistical queries

can be simulated using $\tilde{O}(d\tau^{-2}(1-2\eta)^{-2}\log(1/\delta)+\varepsilon^{-2})$ examples [10].

One of the most significant results on learning in the distribution-independent PAC learning model is the equivalence of weak and strong learnability demonstrated by Schapire's celebrated *boosting* method [13]. Aslam and Decatur showed that this equivalence holds in the SQ model as well [2].

A natural way to extend the SQ model is to allow query functions that depend on a t-tuple of examples instead of just one example. Blum et al. proved that this extension does not increase the power of the model as long as $t = O(\log s)$ [5].

Statistical Query Dimension

The restricted way in which SQ algorithms use examples makes it simpler to understand the limitations of efficient learning in this model. A long-standing open problem in learning theory is learning of the concept class of all parity functions over $\{0,1\}^n$ with noise (a parity function is a XOR of some subset of n Boolean inputs). Kearns has demonstrated that parities cannot be efficiently learned using statistical queries even under the uniform distribution over $\{0,1\}^n$ [10]. This hardness result is unconditional in the sense that it does not rely on any unproven complexity assumptions.

The technique of Kearns was generalized by Blum et al. who proved that efficient SQ learnability of a concept class C is characterized by a relatively simple combinatorial parameter of C called the *statistical query dimension* [4]. The quantity they defined measures the maximum number of "nearly uncorrelated" functions in a concept class. More formally,

Definition 3 For a concept class C and distribution $\mathcal{D}$, the *statistical query dimension* of C with respect to $\mathcal{D}$, denoted SQ-DIM$(C,\mathcal{D})$, is the largest number d such that C contains d functions $f_1, f_2, \ldots, f_d$ such that for all $i \neq j$, $|\mathbf{E}_{\mathcal{D}}[f_i f_j]| \leq \frac{1}{d^3}$.

Blum et al. relate the SQ dimension to learning in the SQ model as follows.

Theorem 4 ([4]) *Let C be a concept class and $\mathcal{D}$ be a distribution such that SQ-DIM$(C,\mathcal{D}) = d$.*

- *If all queries are made with tolerance of at least $1/d^{1/3}$, then at least $d^{1/3}$ queries are required to learn C with error $1/2 - 1/d^3$ in the SQ model.*
- *There exists an algorithm for learning C with respect to $\mathcal{D}$ that makes d fixed queries, each of tolerance $1/3d^3$, and finds a hypothesis with error at most $1/2 - 1/3d^3$.*

Thus SQ-DIM characterizes weak learnability in the SQ model up to a polynomial factor. Parity functions are uncorrelated with respect to the uniform distribution and therefore any concept class that contains a superpolynomial number of parity functions cannot be learned by statistical queries with respect to the uniform distribution. This for example includes such important concept classes as *k-juntas* over $\{0,1\}^n$ (or functions that depend on at most k input variables) for $k = \omega(1)$ and *decision trees* of superconstant size.

The following important result is due to Blum et al. [5]:

Theorem 5 ([5]) *For any constant $\eta < 1/2$, parities that depend on the first $\log n \log\log n$ input variables are efficiently PAC learnable in the presence of random classification noise of rate η.*

Since there are $n^{\log\log n}$ parity functions that depend on the first $\log n \log\log n$ input variables, this shows that there exist concept classes that are efficiently learnable in the presence of noise (at constant rate $\eta < 1/2$) but are not efficiently learnable in the SQ model.

Applications

Learning by statistical queries was used to obtain noise-tolerant algorithms for a number of important concept classes. One of the ways this can be done is by showing that a PAC learning algorithm can be modified to use statistical queries instead of random examples. Examples of learning problems for which the first noise-tolerant algorithm was obtained using this approach include [10]:

- Learning decision trees of constant rank.
- *Attribute-efficient* algorithms for learning conjunctions.
- Learning axis-aligned rectangles over $\mathbb{R}^n$.
- Learning AC^0 (constant-depth unbounded fan-in) Boolean circuits over $\{0,1\}^n$ with respect to the uniform distribution in quasipolynomial time.

Blum et al. also use the SQ model to show that their algorithm for learning linear threshold functions is noise-tolerant [3], resolving an important open problem.

The ideas behind the use of statistical queries to produce noise tolerant algorithms were adapted to learning using *membership queries* (or ability to ask for the value of the unknown function at any point). There the noise model has to be modified slightly to prevent the learner from asking for independently corrupted labels on the same point. An appropriate modification is the introduction of *persistent classification noise* by Goldman et al. [7]. In this model, as before, the answer to a query at each point x is flipped with probability $1-\eta$. However, if the mem-

bership oracle was already queried about the value of f at some specific point x or x was already generated as a random example, the returned label has the same value as in the first occurrence.

Extensions of the SQ model suggested by Jackson et al. [9] and Bshouty and Feldman [6] allow any algorithm based on these extended statistical queries to be converted to a noise-tolerant PAC algorithm with membership queries. In particular, they used this approach to convert Jackson's algorithm for learning DNF with respect to the uniform distribution to a noise-tolerant one. Bshouty and Feldman also show that learnability in their extension can be characterized using a dimension similar to the SQ dimension of Blum et al. [4].

Open Problems

The main questions related to learning with random classification noise are still open. Is every concept class efficiently learnable in the PAC model also learnable in the presence of random classification noise? Is every concept class efficiently learnable in the presence of random classification noise of arbitrarily high rate (less than 1/2) also efficiently learnable using statistical queries? Note that the algorithm of Blum et al. assumes that the noise rate is a constant and therefore does not provide a complete answer to this question [5]. For both questions a central issue seems to be obtaining a better understanding of the complexity of learning parities with noise.

Another important direction of research is learning with weaker assumptions on the nature of noise. A natural model that places no assumptions on the way in which the labels are corrupted is the *agnostic* learning model defined by Haussler [8] and Kearns et al. [11]. Efficient learning algorithms that can cope with this, possibly adversarial, noise is a very desirable if hard to achieve goal. For example, learning conjunctions of input variables in this model is an open problem known to be at least as hard as learning DNF expressions in the PAC model [11]. It is therefore important to identify and investigate useful and general models of noise based on less pessimistic assumptions.

Cross References

- ▶ Attribute-Efficient Learning
- ▶ Learning Constant-Depth Circuits
- ▶ Learning DNF Formulas
- ▶ Learning Heavy Fourier Coefficients of Boolean Functions
- ▶ Learning with Malicious Noise
- ▶ PAC Learning

Recommended Reading

1. Angluin, D., Laird, P.: Learning from noisy examples. Mach. Learn. **2**, 343–370 (1988)
2. Aslam, J., Decatur, S.: Specification and simulation of statistical query algorithms for efficiency and noise tolerance. J. Comput. Syst. Sci. **56**, 191–208 (1998)
3. Blum, A., Frieze, A., Kannan, R., Vempala, S.: A polynomial time algorithm for learning noisy linear threshold functions. Algorithmica **22**(1/2), 35–52 (1997)
4. Blum, A., Furst, M., Jackson, J., Kearns, M., Mansour, Y., Rudich, S.: Weakly learning DNF and characterizing statistical query learning using Fourier analysis. In: Proceedings of STOC, pp. 253–262 (1994)
5. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM **50**(4), 506–519 (2003)
6. Bshouty, N., Feldman, V.: On using extended statistical queries to avoid membership queries. J. Mach. Learn. Res. **2**, 359–395 (2002)
7. Goldman, S., Kearns, M., Schapire, R.: Exact identification of read-once formulas using fixed points of amplification functions. SIAM J. Comput. **22**(4), 705–726 (1993)
8. Haussler, D.: Decision theoretic generalizations of the PAC model for neural net and other learning applications. Inf. Comput. **100**(1), 78–150 (1992)
9. Jackson, J., Shamir, E., Shwartzman, C.: Learning with queries corrupted by classification noise. In: Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems, pp. 45–53 (1997)
10. Kearns, M.: Efficient noise-tolerant learning from statistical queries. J. ACM **45**(6), 983–1006 (1998)
11. Kearns, M., Schapire, R., Sellie, L.: Toward efficient agnostic learning. Mach. Learn. **17**(2-3), 115–141 (1994)
12. Laird, P.: Learning from good and bad data. Kluwer Academic Publishers (1988)
13. Schapire, R.: The strength of weak learnability. Mach. Learn. **5**(2), 197–227 (1990)
14. Valiant, L.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)

Steiner Forest

1995; Agrawal, Klein, Ravi

Guido Schäfer
Institute for Mathematics and Computer Science, Technical University of Berlin, Berlin, Germany

Keywords and Synonyms

Requirement join; *R*-join, Requirement Join

Problem Definition

The *Steiner forest problem* is a fundamental problem in network design. Informally, the goal is to establish connections between pairs of vertices in a given network

at minimum cost. The problem generalizes the well-known *Steiner tree problem*. As an example, assume that a telecommunication company receives communication requests from their customers. Each customer asks for a connection between two vertices in a given network. The company's goal is to build a minimum cost network infrastructure such that all communication requests are satisfied.

Formal Definition and Notation

More formally, an instance $I = (G, c, R)$ of the Steiner forest problem is given by an undirected graph $G = (V, E)$ with vertex set V and edge set E, a non-negative cost function $c\colon E \to \mathbb{Q}^+$, and a set of vertex pairs $R = \{(s_1, t_1), \ldots, (s_k, t_k)\} \subseteq V \times V$. The pairs in R are called *terminal pairs*. A feasible solution is a subset $F \subseteq E$ of the edges of G such that for every terminal pair $(s_i, t_i) \in R$ there is a path between s_i and t_i in the subgraph $G[F]$ induced by F. Let the cost $c(F)$ of F be defined as the total cost of all edges in F, i. e., $c(F) = \sum_{e \in F} c(e)$. The goal is to find a feasible solution F of minimum cost $c(F)$. It is easy to see that there exists an optimum solution which is a forest.

The Steiner forest problem may alternatively be defined by a set of *terminal groups* $R = \{g_1, \ldots, g_k\}$ with $g_i \subseteq V$ instead of terminal pairs. The objective is to compute a minimum cost subgraph such that all terminals belonging to the same group are connected. This definition is equivalent to the one given above.

Related Problems

A special case of the Steiner forest problem is the *Steiner tree problem* (see also the entry ► Steiner Tree). Here, all terminal pairs share a common root vertex $r \in V$, i. e., $r \in \{s_i, t_i\}$ for all terminal pairs $(s_i, t_i) \in R$. In other words, the problem consists of a set of terminal vertices $R \subseteq V$ and a root vertex $r \in V$ and the goal is to connect the terminals in R to r in the cheapest possible way. A minimum cost solution is a tree.

The *generalized Steiner network problem* (see the entry ► Generalized Steiner Network), also known as the *survivable network design problem*, is a generalization of the Steiner forest problem. Here, a *connectivity requirement* function $r\colon V \times V \to \mathbb{N}$ specifies the number of edge disjoint paths that need to be established between every pair of vertices. That is, the goal is to find a minimum cost multi-subset H of the edges of G (H may contain the same edge several times) such that for every pair of vertices $(x, y) \in V$ there are $r(x, y)$ edge disjoint paths from x to y in $G[H]$. The goal is to find a set H of minimum cost. Clearly, if $r(x, y) \in \{0, 1\}$ for all $(x, y) \in V \times V$, this problem reduces to the Steiner forest problem.

Key Results

Agrawal, Klein and Ravi [1,2] give an approximation algorithm for the Steiner forest problem that achieves an approximation ratio of 2. More precisely, the authors prove the following theorem.

Theorem 1 *There exists an approximation algorithm that for every instance $I = (G, c, R)$ of the Steiner forest problem, computes a feasible forest F such that*

$$c(F) \leq \left(2 - \frac{1}{k}\right) \cdot \mathit{OPT}(I)\,,$$

where k is the number of terminal pairs in R and $\mathit{OPT}(I)$ is the cost of an optimal Steiner forest for I.

Related Work

The Steiner tree problem is NP-hard [10] and APX-complete [4,8]. The current best lower bound on the achievable approximation ratio for the Steiner tree problem is 1.0074 [21]. Goemans and Williamson [11] generalized the results obtained by Agrawal, Klein and Ravi to a larger class of connectivity problems, which they term *constrained forest problems*. For the Steiner forest problem, their algorithm achieves the same approximation ratio of $(2 - 1/k)$. The algorithms of Agrawal, Klein and Ravi [2] and Goemans and Williamson [11] are both based on the classical *undirected cut formulation* for the Steiner forest problem [3]. The integrality gap of this relaxation is known to be $(2 - 1/k)$ and the results in [2,11] are therefore tight. Jain [15] presents a 2-approximation algorithm for the generalized Steiner network problem.

Primal-Dual Algorithm

The main ideas of the algorithm by Agrawal, Klein and Ravi [2] are sketched below; subsequently, AKR is used to refer to this algorithm. The description given here differs from the one in [2]; the interested reader is referred to [2] for more details.

The algorithm is based on the following integer programming formulation for the Steiner forest problem. Let $I = (G, c, R)$ be an instance of the Steiner forest problem. Associate an indicator variable $x_e \in \{0, 1\}$ with every edge $e \in E$. The value of x_e is 1 if e is part of the forest F and 0 otherwise. A subset $S \subseteq V$ of the vertices is called a *Steiner cut* if there exists at least one terminal pair $(s_i, t_i) \in R$ such that $|\{s_i, t_i\} \cap S| = 1$; S is said to *separate* terminal

pair (s_i, t_i). Let S be the set of all Steiner cuts. For a subset $S \subseteq V$, define $\delta(S)$ as the the set of all edges in E that have exactly one endpoint in S. Given a Steiner cut $S \in \mathcal{S}$, any feasible solution F of I must contain at least one edge that *crosses* the cut S, i. e., $\sum_{e \in \delta(S)} x_e \geq 1$. This gives rise to the following *undirected cut formulation*:

$$\text{minimize} \quad \sum_{e \in E} c(e) x_e \tag{IP}$$

$$\text{subject to} \quad \sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{S} \tag{1}$$

$$x_e \in \{0, 1\} \quad \forall e \in E\,. \tag{2}$$

The dual of the linear programming relaxation of (IP) has a variable y_S for every Steiner cut $S \in \mathcal{S}$. There is a constraint for every edge $e \in E$ that requires that the total dual assigned to sets $S \in \mathcal{S}$ that contain exactly one endpoint of e is at most the cost $c(e)$ of the edge:

$$\text{maximize} \quad \sum_{S \in \mathcal{S}} y_S \tag{D}$$

$$\text{subject to} \quad \sum_{S \in \mathcal{S}:\ e \in \delta(S)} y_S \leq c(e) \quad \forall e \in E \tag{3}$$

$$y_S \geq 0 \quad \forall S \in \mathcal{S}\,. \tag{4}$$

Algorithm AKR is based on the *primal-dual schema* (see, e. g., [22]). That is, the algorithm constructs both a feasible primal solution for (IP) and a feasible dual solution for (D). The algorithm starts with an infeasible primal solution and reduces its degree of infeasibility as it progresses. At the same time, it creates a feasible dual packing of subsets of large total value by raising dual variables of Steiner cuts.

One can think of an execution of AKR as a process over time. Let x^τ and y^τ, respectively, be the primal incidence vector and feasible dual solution at time τ. Initially, let $x_e^0 = 0$ for all $e \in E$ and $y_S^0 = 0$ for all $S \in \mathcal{S}$. Let F^τ denote the forest corresponding to the set of edges with $x_e^\tau = 1$. A tree T in F^τ is called *active* at time τ if it contains a terminal that is separated from its mate; otherwise it is *inactive*. Intuitively, AKR grows trees in F^τ that are active. At the same time, the algorithm raises dual values of Steiner cuts that correspond to active trees. If two active trees collide, they are merged. The process terminates if all trees are inactive and thus there are no unconnected terminal pairs. The interplay of the primal (growing trees) and the dual process (raising duals) is somewhat subtle and outlined next.

An edge $e \in E$ is *tight* if the corresponding constraint (3) holds with equality; a path is tight if all its edges are tight. Let H^τ be the subgraph of G that is induced by the tight edges for dual y^τ. The connected components of H^τ induce a partition C^τ on the vertex set V. Let S^τ be the set of all Steiner cuts contained in C^τ, i. e., $S^\tau = C^\tau \cap \mathcal{S}$. AKR raises the dual values y_S for all sets $S \in S^\tau$ uniformly at all times $\tau \geq 0$. Note that y^τ is dual feasible. The algorithm maintains the invariant that F^τ is a subgraph of H^τ at all times. Consider the event that a path P between two trees T_1 and T_2 of F^τ becomes tight. The missing edges of P are then added to F^τ and the process continues. Eventually, all trees in F^τ are inactive and the process halts.

Applications

The computation of (approximate) solutions for the Steiner forest problem has various applications both in theory and practice; only a few recent developments are mentioned here.

Algorithms for more complex network design problems often rely on good approximation algorithms for the Steiner forest problem. For example, the recent approximation algorithms [6,9,12] for the *multi-commodity rent-or-buy problem* (MRoB) are based on the random sampling framework by Gupta et al. [12,13]. The framework uses a Steiner forest approximation algorithm that satisfies a certain *strictness* property as a subroutine. Fleischer et al. [9] show that AKR meets this strictness requirement, which leads to the current best 5-approximation algorithm for MRoB. The strictness property also plays a crucial role in the boosted sampling framework by Gupta et al. [14] for two-stage stochastic optimization problems with recourse.

Online versions of Steiner tree and forest problems have been studied by by Awerbuch et al. [5] and Berman and Coulston [7]. In the area of algorithmic game theory, the development of *group-strategyproof cost sharing mechanisms* for network design problems such as the Steiner tree problem has recently received a lot of attention; see e. g., [16,17,19,20]. An adaptation of AKR yields such a cost sharing mechanism for the Steiner forest problem [18].

Cross References

- ▶ Generalized Steiner Network
- ▶ Steiner Trees

Recommended Reading

The interested reader is referred in particular to the articles [2,11] for a more detailed description of primal-dual approximation algorithms for general network design problems.

1. Agrawal, A., Klein, P., Ravi, R.: When trees collide: an approximation algorithm for the generalized Steiner problem on networks. In: Proc. of the 23rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 134–144 (1991)
2. Agrawal, A., Klein, P., Ravi, R.: When trees collide: An approximation algorithm for the generalized Steiner problem in networks. SIAM J. Comput. **24**(3), 445–456 (1995)
3. Aneja, Y.P.: An integer linear programming approach to the Steiner problem in graphs. Networks **10**(2), 167–178 (1980)
4. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. J. ACM **45**(3), 501–555 (1998)
5. Awerbuch, B., Azar, Y., Bartal, Y.: On-line generalized Steiner problem. In: Proc. of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 2005, pp. 68–74 (1996)
6. Becchetti, L., Könemann, J., Leonardi, S., Pál, M.: Sharing the cost more efficiently: improved approximation for multicommodity rent-or-buy. In: Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, pp. 375–384 (2005)
7. Berman, P., Coulston, C.: On-line algorithms for Steiner tree problems. In: Proc. of the 29th Annual ACM Symposium on Theory of Computing, pp. 344–353. Association for Computing Machinery, New York (1997)
8. Bern, M., Plassmann, P.: The Steiner problem with edge lengths 1 and 2. Inf. Process. Lett. **32**(4), 171–176 (1989)
9. Fleischer, L., Könemann, J., Leonardi, S., Schäfer, G.: Simple cost sharing schemes for multicommodity rent-or-buy and stochastic Steiner tree. In: Proc. of the 38th Annual ACM Symposium on Theory of Computing, pp. 663–670. Association for Computing Machinery, New York (2006)
10. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco (1979)
11. Goemans, M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. SIAM J. Comput. **24**(2), 296–317 (1995)
12. Gupta, A., Kumar, A., Pál, M., Roughgarden, T.: Approximation via cost-sharing: a simple approximation algorithm for the multicommodity rent-or-buy problem. In: Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 606–617., IEEE Computer Society, Washington (2003)
13. Gupta, A., Kumar, A., Pál, M., Roughgarden, T.: Approximation via cost-sharing: simpler and better approximation algorithms for network design. J. ACM 54(3), Article 11 (2007)
14. Gupta, A., Pál, M., Ravi, R., Sinha, A.: Boosted sampling: approximation algorithms for stochastic optimization. In: Proc. of the 36th Annual ACM Symposium on Theory of Computing, pp. 417–426. Association for Computing Machinery, New York (2004)
15. Jain, K.: A factor 2 approximation for the generalized Steiner network problem. Combinatorica **21**(1), 39–60 (2001)
16. Jain, K., Vazirani, V.V.: Applications of approximation algorithms to cooperative games. In: Proc. of the 33rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 364–372 (2001)
17. Kent, K., Skorin-Kapov, D.: Population monotonic cost allocation on mst's. In: Proc. of the 6th International Conference on Operational Research, Croatian Operational Research Society, Zagreb, pp. 43–48 (1996)
18. Könemann, J., Leonardi, S., Schäfer, G.: A group-strategyproof mechanism for Steiner forests. In: Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 612–619. Society for Industrial and Applied Mathematics, Philadelphia (2005)
19. Megiddo, N.: Cost allocation for Steiner trees. Networks **8**(1), 1–6 (1978)
20. Moulin, H., Shenker, S.: Strategyproof sharing of submodular costs: budget balance versus efficiency. Econ. Theor. **18**(3), 511–533 (2001)
21. Thimm, M.: On the approximability of the Steiner tree problem. Theor. Comput. Sci. **295**(1–3), 387–402 (2003)
22. Vazirani, V.V.: Approximation algorithms. Springer, Berlin (2001)

Steiner Trees

2006; Du, Graham, Pardalos, Wan, Wu, Zhao

YAOCUN HUANG, WEILI WU
Department of Computer Science,
University of Texas at Dallas, Richardson, TX, USA

Keywords and Synonyms

Approximation algorithm design

Definition

Given a set of points, called *terminals*, in a metric space, the problem is to find the shortest tree interconnecting all terminals. There are three important metric spaces for Steiner trees, the Euclidean plane, the rectilinear plane, and the edge-weighted network. The Steiner tree problems in those metric spaces are called the *Euclidean Steiner Tree (EST)*, the *Rectilinear Steiner Tree (RST)*, and the *Network Steiner Tree (NST)*, respectively. EST and RST has been found to have polynomial-time approximation schemes (PTAS) by using adaptive partition. However, for NST, there exists a positive number r such that computing r-approximation is NP-hard. So far, the best performance ratio of polynomial-time approximation for NST is achieved by k-restricted Steiner trees. However, in practice, the iterated 1-Steiner tree is used very often. Actually, the iterated 1-Steiner was proposed as a candidate of good approximation for Steiner minimum trees a long time ago. It has a very good record in computer experiments, but no correct analysis was given showing the iterated 1-Steiner tree having a performance ratio better than that of the minimum spanning tree until the recent work by Du et al. [9]. There is minimal difference in construction of the 3-restricted Steiner tree and the iterated 1-Steiner

tree, which makes a big difference in analysis of those two types of trees. Why does the difficulty of analysis make so much difference? This will be explained in this article.

History and Background

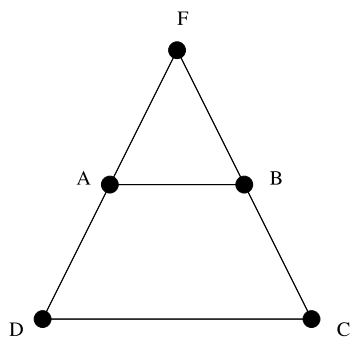
The Steiner tree problem was proposed by Gauss in 1835 as a generalization of the Fermat problem. Given three points A, B, and C in the Euclidean plane, Fermat studied the problem of finding a point S to minimize $|SA| + |SB| + |SC|$. He determined that when all three inner angles of triangle ABC are less than 120°, the optimal S should be at the position that $\angle ASB = \angle BSC = \angle CSA = 120°$.

The generalization of the Fermat problem has two directions:

1. Given n points in the Euclidean plane, find a point S to minimize the total distance from S to n given points. This is still called the Fermat problem.
2. Given n points in the Euclidean plane, find the shortest network interconnecting all given points.

Gauss found the second generalization through communication with Schumacher. On March 19, 1836, Schumacher wrote a letter to Gauss and mentioned a paradox about Fermat's problem: Consider a convex quadrilateral $ABCD$. It is known that the solution of Fermat's problem for four points A, B, C, and D is the intersection E of diagonals AC and BD. Suppose extending DA and CB can obtain an intersection F. Now, move A and B to F. Then E will also be moved to F. However, when the angle at F is less than 120°, the point F cannot be the solution of Fermat's problem for three given points F, D, and C. What happens? (Fig. 1.)

On March 21, 1836, Gauss wrote a letter replying to Schumacher in which he explained that the mistake of Schumacher's paradox occurs at the place where Fermat's problem for four points A, B, C, and D is changed to Fermat's problem for three points F, C, and D. When A and B are identical to F, the total distance from E to four points A, B, C, and D equals $2|EF| + |EC| + |ED|$, not $|EF| + |EC| + |ED|$. Thus, the point E may not be the solution of Fermat's problem for F, C, and D. More importantly, Gauss proposed a new problem. He said that it is more interesting to find the shortest network rather than a point. Gauss also presented several possible connections of the shortest network for four given points.



Steiner Trees, Figure 1

It was unfortunate that Gauss' letter was not seen by researchers of Steiner trees at an earlier stage. Especially, R. Courant and H. Robbins who in their popular book *What is mathematics?* (published in 1941) [6] called Gauss' problem the Steiner tree so that "Steiner tree" became a popular name for the problem.

The Steiner tree became an important research topic in mathematics and computer science due to its applications in telecommunication and computer networks. Starting with Gilbert and Pollak's work published in 1968, many publications on Steiner trees have been generated to solve various problems concerning it.

One well-known problem is the *Gilbert–Pollak conjecture* on the Steiner ratio, which is the least ratio of lengths between the Steiner minimum tree and the minimum spanning tree on the same set of given points. Gilbert and Pollak in 1968 conjectured that the Steiner ratio in the Euclidean plane is $\sqrt{3}/2$ which is achieved by three vertices of an equilateral triangle. A great deal of research effort has been put into the conjecture and it was finally proved by Du and Hwang [7].

Another important problem is called the *better approximation*. For a long time no approximation could be proved to have a performance ratio smaller than the inverse of the Steiner ratio. Zelikovsky [14] made the first breakthrough. He found a polynomial-time 11/6-approximation for NST which beats 1/2, the inverse of the Steiner ratio in the edge-weighted network. Later, Berman and Ramaiye [2] gave a polynomial-time 92/72-approximation for RST and Du, Zhang, and Feng [8] closed the story by showing that in any metric space, there exists a polynomial-time approximation with a performance ratio better than the inverse of the Steiner ratio provided that for any set of a fixed number of points, the Steiner minimum tree is polynomial-time computable.

All the above better approximations came from the family of k-restricted Steiner trees. By improving some detail of construction, the constant performance ratio was decreasing, but the improvements were also becoming smaller. In 1996, Arora [1] made significant progress for EST and RST. He showed the existence of PTAS for EST and RST. Therefore, the theoretical researchers now pay

more attention to NST. Bern and [3] showed that NST is MAX SNP-complete. This means that there exists a positive number r, computing the r-approximation for NST is NP-hard. The best-known performance for NST was given by Robin and Zelikovsky [12]. They also gave a very simple analysis to a well-known heuristic, the iterated 1-Steiner tree for pseudo-bipartite graphs.

Analysis of the iterated 1-Steiner tree is another long-standing open problem. Since Chang [4,5] proposed that the iterated 1-Steiner tree approximates the Steiner minimum tree in 1972, its performance has been claimed to be very good through computer experiments[10,13], but no theoretical analysis supported this claim. Actually, both the k-restricted Steiner tree and the iterated 1-Steiner tree are obtained by greedy algorithms, but with different types of potential functions. For the iterated 1-Steiner tree, the potential function is non-submodular, but for the k-restricted Steiner tree, it is submodular; a property that holds for k-restricted Steiner trees may not hold for iterated 1-Steiner trees. Actually, the submodularity of potential function is very important in analysis of greedy approximations [11]. Du et al. [9] gave a correct analysis for the iterated 1-Steiner tree with a general technique to deal with non-submodular potential function.

Key Results

Consider input edge-weighted graph $G = (V, E)$ of NST. Assume that G is a complete graph and the edge-weight satisfies the triangular inequality, otherwise, consider the complete graph on V with each edge (u, v) having a weight equal to the length of the shortest path between u and v in G. Given a set P of terminals, a *Steiner tree* is a tree interconnecting all given terminals such that every leaf is a terminal.

In a Steiner tree, a terminal may have degree more than one. The Steiner tree can be decomposed, at those terminals with degree more than one, into smaller trees in which every terminal is a leaf. In such a decomposition, each resulting small tree is called a *full component*. The *size* of a full component is the number of terminals in it. A Steiner tree is *k-restricted* if every full component of it has a size at most k. The shortest k-restricted Steiner tree is also called the *k-restricted Steiner minimum tree*. Its length is denoted by $smt_k(P)$. Clearly, $smt_2(P)$ is the length of the minimum spanning tree on P, which is also denoted by $mst(P)$. Let $smt(P)$ denote the length of the Steiner minimum tree on P. If $smt_3(P)$ can be computed in polynomial-time, then it is better than $mst(P)$ for an approximation of $smt(P)$. However, so far no polynomial-time approximation has been found for $smt_3(P)$. Therefore, Zelikovsky [14] used a greedy approximation of $smt_3(P)$ to approximate $smt(P)$. Actually, Chang [4,5] used a similar greedy algorithm to compute an iterated 1-Steiner tree. Let $\mathcal{F}$ be a family of subgraphs of input edge-weighted graph G. For any connected subgraph H, denote by $mst(H)$ the length of the minimum spanning tree of H and for any subgraph H, denote by $mst(H)$ the sum of $mst(H')$ for H' over all connected components of H. Define

$$gain(H) = mst(P) - mst(P : H) - mst(H) ,$$

where $mst(P : H)$ is the length of the minimum spanning tree interconnecting all unconnected terminals in P after every edge of H shrinks into a point.

Greedy Algorithm $H \leftarrow \emptyset$;
  while P has not been interconnected by H **do**
    choose $F \in \mathcal{F}$ to maximize $gain(H \cup F)$;
  output $mst(H)$.

When $\mathcal{F}$ consists of all full components of size at most three, this greedy algorithm gives the 3-restricted Steiner tree of Zelikovsky [14]. When $\mathcal{F}$ consists of all 3-stars and all edges where a 3-star is a tree with three leaves and a central vertex, this greedy algorithm produces the iterated 1-Steiner tree. An interesting fact pointed out by Du et al. [9] is that the function $gain(\cdot)$ is submodular over all full components of size at most three, but not submodular over all 3-stars and edges.

Let us consider a base set E and a function f from all subsets of E to real numbers. f is *submodular* if for any two subsets A, B of E,

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B) .$$

For $x \in E$ and $A \subseteq E$, denote $\Delta_x f(A) = f(A \cup \{x\}) - f(A)$.

Lemma 1 *f is submodular if and only if for any $A \subset E$ and distinct $x, y \in E - A$,*

$$\Delta_x \Delta_y f(A) \leq 0 . \tag{1}$$

Proof Suppose f is submodular. Set $B = A \cup \{x\}$ and $C = A \cup \{y\}$. Then $B \cup C = A \cup A \cup \{x, y\}$ and $B \cap C = A$. Therefore, one has

$$f(A \cup \{x, y\}) - f(A \cup \{x\}) - f(A \cup \{y\}) + f(A) \leq 0 ,$$

that is, (1) holds.

Conversely, suppose (1) holds for any $A \subset E$ and distinct $x, y \in E - A$. Consider two subsets A, B of E. If $A \subseteq B$ or $B \subseteq A$, it is trivial to have

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B) .$$

Therefore, one may assume that $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$. Write $A \setminus B = \{x_1, \ldots, x_k\}$ and $B \setminus A = \{y_1, \ldots, y_h\}$. Then

$$
\begin{aligned}
& f(A \cup B) - f(A) - f(B) + f(A \cap B) \\
&= \sum_{i=1}^{k} \sum_{j=1}^{h} \Delta_{x_i} \Delta_{y_j} f(A \cup \{x_1, \ldots, x_{i-1}\} \cup \{y_1, \ldots, y_{j-1}\}) \\
&\leq 0 ,
\end{aligned}
$$

where $\{x_1, \ldots x_{i-1}\} = \emptyset$ for $i = 1$ and $\{y_1, \ldots, y_{j-1}\} = \emptyset$ for $j = 1$. □

Lemma 2 *Define $f(H) = -mst(P : H)$. Then f is submodular over edge set E.*

Proof Note that for any two distinct edges x and y not in subgraph H,

$$
\begin{aligned}
& \Delta_x \Delta f(H) \\
&= -mst(P : H \cup x \cup y) + mst(P : H \cup x) \\
&\quad + mst(P : H \cup y) - mst(P : H) \\
&= (mst(P : H) - mst(P : H \cup x \cup y)) \\
&\quad - (mst(P : H) - mst(P : H \cup x)) + (mst(P : H) \\
&\quad - mst(P : H \cup y)) .
\end{aligned}
$$

Let T be a minimum spanning tree for unconnected terminals after every edge of H shrinks into a point. T contains a path P_x connecting two endpoints of x and also a path P_y connecting two endpoints of y. Let e_x (e_y) be a longest edge in P_x (P_y). Then

$$
\begin{aligned}
& mst(P : H) - mst(P : H \cup x) = length(e_x) , \\
& mst(P : H) - mst(P : H \cup y) = length(e_y) .
\end{aligned}
$$

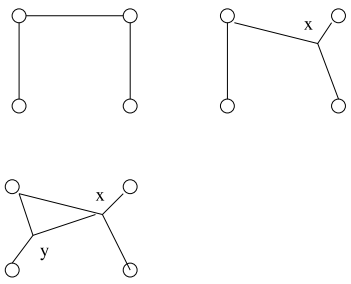
$mst(P : H) - mst(P : H \cup x \cup y)$ can be computed as follows: Choose a longest edge e' from $P_x \cup P_y$. Note that $T \cup x \cup y - e'$ contains a unique cycle Q. Choose a longest edge e'' from $(P_x \cup P_y) \cap Q$. Then

$$
mst(P : H) - mst(P : H \cup x \cup y) = length(e') + length(e'').
$$

Now, to show the submodularity of f, it suffices to prove

$$
length(e_x) + length(e_y) \geq length(e') + length(e''). \quad (2)
$$

Case 1. $e_x \notin P_x \cap P_y$ and $e_y \notin P_x \cap P_y$. Without loss of generality, assume $length(e_x) \geq length(e_y)$. Then one may choose $e' = e_x$ so that $(P_x \cup P_y) \cap Q = P_y$. Hence one can choose $e'' = e_y$. Therefore, the equality holds for (2).



Steiner Trees, Figure 2

Case 2. $e_x \notin P_x \cap P_y$ and $e_y \in P_x \cap P_y$. Clearly, $length(e_x) \geq length(e_y)$. Hence, one may choose $e' = e_x$ so that $(P_x \cup P_y) \cap Q = P_y$. Hence one can choose $e'' = e_y$. Therefore, the equality holds for (2).

Case 3. $e_x \in P_x \cap P_y$ and $e_y \notin P_x \cap P_y$. Similar to Case 2.

Case 4. $e_x \in P_x \cap P_y$ and $e_y \in P_x \cap P_y$. In this case, $length(e_x) = length(e_y) = length(e')$. Hence, (2) holds. □

The following explains that the submodularity of $gain(\cdot)$ holds for a k-restricted Steiner tree.

Proposition *Let $\mathcal{E}$ be the set of all full components of a Steiner tree. Then $gain(\cdot)$ as a function on the power set of $\mathcal{E}$ is submodular.*

Proof Note that for any $\mathcal{H} \subset \mathcal{E}$ and $x, y \in \mathcal{E} - \mathcal{H}$,

$$
\Delta_x \Delta_y mst(H) = 0 ,
$$

where $H = \cup_{z \in \mathcal{H}} z$. Thus, this proposition follows from Lemma 2. □

Let $\mathcal{F}$ be the set of 3-stars and edges chosen in the greedy algorithm to produce an iterated 1-Steiner tree. Then $gain(\cdot)$ may not be submodular on $\mathcal{F}$. To see this fact, consider two 3-stars x and y in Fig. 2. Note that $gain(x \cup y) > gain(x), gain(y) \leq 0$ and $gain(\emptyset) = 0$. One has

$$
gain(x \cup y) - gain(x) - gain(y) + gain(\emptyset) > 0 .
$$

Applications

The Steiner tree problem is a classic NP-hard problem with many applications in the design of computer circuits, long-distance telephone lines, multicast routing in communication networks, etc. There exist many heuristics of the greedy-type for Steiner trees in the literature. Most of them have a good performance in computer experiments,

without support from theoretical analysis. The approach given in this work may apply to them.

Open Problems

It is still open whether computing the 3-restricted Steiner minimum tree is NP-hard or not. For $k \geq 4$, it is known that computing the k-restricted Steiner minimum tree is NP-hard.

Cross References

- ▶ Greedy Approximation Algorithms
- ▶ Minimum Spanning Trees
- ▶ Rectilinear Steiner Tree

Recommended Reading

1. Arora, S.: Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. Proc. 37th IEEE Symp. on Foundations of Computer Science, pp. 2–12 (1996)
2. Berman, P., Ramaiyer, V.: Improved approximations for the Steiner tree problem. J. Algorithms **17**, 381–408 (1994)
3. Bern, M., Plassmann, P.: The Steiner problem with edge lengths 1 and 2. Inf. Proc. Lett. **32**, 171–176 (1989)
4. Chang, S.K.: The generation of minimal trees with a Steiner topology. J. ACM **19**, 699–711 (1972)
5. Chang, S.K.: The design of network configurations with linear or piecewise linear cost functions. In: Symp. on Computer-Communications, Networks, and Teletraffic, pp. 363–369 IEEE Computer Society Press, California (1972)
6. Crourant, R., Robbins, H.: What Is Mathematics? Oxford University Press, New York (1941)
7. Du, D.Z., Hwang, F.K.: The Steiner ratio conjecture of Gilbert-Pollak is true. Proc. Natl. Acad. Sci. USA **87**, 9464–9466 (1990)
8. Du, D.Z., Zhang, Y., Feng, Q.: On better heuristic for euclidean Steiner minimum trees. In: Proceedings 32nd FOCS, IEEE Computer Society Press, California (1991)
9. Du, D.Z., Graham, R.L., Pardalos, P.M., Wan, P.J., Wu, W., Zhao, W.: Analysis of greedy approximations with nonsubmodular potential functions. In: Proceedings of 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 167–175. ACM, New York (2008)
10. Kahng, A., Robins, G.: A new family of Steiner tree heuristics with good performance: the iterated 1-Steiner approach. In: Proceedings of IEEE Int. Conf. on Computer-Aided Design, Santa Clara, pp.428–431 (1990)
11. Wolsey, L.A.: An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica **2**, 385–393 (1982)
12. Robin, G., Zelikovsky, A.: Improved Steiner trees approximation in graphs. In: SIAM-ACM Symposium on Discrete Algorithms (SODA), San Francisco, CA, pp. 770–779. January (2000)
13. Smith, J.M., Lee, D.T., Liebman, J.S.: An $O(N \log N)$ heuristic for Steiner minimal tree problems in the Euclidean metric. Networks **11**, 23–39 (1981)
14. Zelikovsky, A.Z.: The 11/6-approximation algorithm for the Steiner problem on networks. Algorithmica **9**, 463–470 (1993)

Stochastic Scheduling

2001; Glazebrook, Nino-Mora

JAY SETHURAMAN
Industrial Engineering and Operations Research, Columbia University, New York, NY, USA

Keywords and Synonyms

Sequencing; Queueing

Problem Definition

Scheduling is concerned with the allocation of scarce resources (such as machines or servers) to competing activities (such as jobs or customers) over time. The distinguishing feature of a *stochastic* scheduling problem is that some of the relevant data are modeled as *random variables*, whose distributions are known, but whose actual realizations are not. Stochastic scheduling problems inherit several characteristics of their deterministic counterparts. In particular, there are virtually an unlimited number of problem types depending on the machine environment (single machine, parallel machines, job shops, flow shops), processing characteristics (preemptive versus non-preemptive; batch scheduling versus allowing jobs to arrive "over time"; due-dates; deadlines) and objectives (makespan, weighted completion time, weighted flow time, weighted tardiness). Furthermore, stochastic scheduling models have some new, interesting features (or difficulties!):

- The scheduler may be able to make inferences about the remaining processing time of a job by using information about its elapsed processing time; whether the scheduler is allowed to make use of this information or not is a question for the modeler.
- Many scheduling algorithms make decisions by comparing the processing times of jobs. If jobs have deterministic processing times, this poses no problems as there is only one way to compare them. If the processing times are random variables, comparing processing times is a subtle issue. There are many ways to compare pairs of random variables, and some are only *partial* orders. Thus any algorithm that operates by comparing processing times must now specify the particular ordering used to compare random variables (and to determine what to do if two random variables are not comparable under the specified ordering).

These considerations lead to the notion of a scheduling *policy*, which specifies how the scarce resources have to be allocated to the competing activities as a function of

the *state* of the system at any point in time. The state of the system includes information such as prior job completions, the elapsed time of jobs currently in service, the realizations of the random release dates and due-dates (if any), and any other information that can be inferred based on the history observed so far. A policy that is allowed to make use of all this information is said to be *dynamic*, whereas a policy that is not allowed to use any state information is *static*.

Given any policy, the objective function for a stochastic scheduling model operating under that policy is typically a random variable. Thus comparison of two policies entails the comparison of the associated random variables, so the *sense* in which these random variables are compared must be specified. A common approach is to find a solution that optimizes the *expected value* of the objective function (which has the advantage that it is a total ordering); less commonly, other orderings such as the stochastic ordering or the likelihood ratio ordering are used.

Key Results

Consider a single machine that processes n jobs, with the (random) processing time of job i given by a distribution $F_i(\cdot)$ whose mean is p_i. The Weighted Shortest Expected Processing Time first (WSEPT) rule sequences the jobs in decreasing order of w_i/p_i. Smith [13] proved that the WSEPT rule minimizes the sum of weighted completion times when the processing times are deterministic. Rothkopf [11] generalized this result and proved the following:

Theorem 1 *The WSEPT rule minimizes the expected sum of the weighted completion times in the class of all nonpreemptive dynamic policies (and hence also in the class of all nonpreemptive static policies).*

If preemption is allowed, the WSEPT rule is not optimal. Nevertheless, Sevcik [12] showed how to assign an "index" to each job at each point in time such that scheduling a job with the largest index at each point in time is optimal. Such policies are called index policies and have been investigated extensively because they are (relatively) simple to implement and analyze. Often the optimality of index policies can be proved under some assumptions on the processing time distributions. For instance, Weber, Varaiya, and Walrand [14] proved the following result for scheduling n jobs on m identical parallel machines:

Theorem 2 *The SEPT rule minimizes the expected sum of completion times in the class of all nonpreemptive dynamic polices, if the processing time distributions of the jobs are stochastically ordered.*

For the same problem but with the makespan objective, Bruno, Downey, and Frederickson [3] proved the optimality of the Longest Expected Processing Time first rule provided all the jobs have exponentially distributed processing times.

One of the most significant achievements in stochastic scheduling is the proof of optimality of index policies for the multi-armed bandit problem and its many variants, due originally to Gittins and Jones [5,6]. In an instance of the bandit problem there are N projects, each of which is in any one of a possibly finite number of states. At each (discrete) time, any one of the projects can be attempted, resulting in a random reward; the attempted project undergoes a (Markovian) state-transition, whereas the other projects remain frozen and do not change state. The goal of the decision maker is to determine an optimal way to attempt the projects so as to maximize the total discounted reward. Of course one can solve this problem as a large, stochastic dynamic program, but such an approach does not reveal any structure, and is moreover computationally impractical except for very small problems. (Also, if the state space of any project is countable or infinite, it is not clear how one can solve the resulting DP exactly!) The remarkable result of Gittins and Jones [5] is the optimality of index policies: to each state of each project, one can associate an index so that attempting a project with the largest index at any point in time is optimal. The original proof of Gittins and Jones [5] has subsequently been simplified by many authors; moreover, several alternative proofs based on different techniques have appeared, leading to a much better understanding of the class of problems for which index policies are optimal.[2,4,6,10,17]

While index policies are easy to implement and analyze, they are often not optimal in many problems. It is therefore natural to investigate the gap between an optimal index policy (or a natural heuristic) and an optimal policy. For example, the WSEPT rule is a natural heuristic for the problem of scheduling jobs on identical parallel machines to minimize the expected sum of the weighted completion times. However, the WSEPT rule is not necessarily optimal. Weiss [16] showed that, under mild and reasonable assumptions, the expected number of times that the WSEPT rule differs from the optimal decision is bounded above by a *constant*, independent of the number of jobs. Thus, the WSEPT rule is asymptotically optimal. As another example of a similar result, Whittle [18] generalized the multi-armed bandit model to allow for state-transitions in projects that are *not* activated, giving rise to the "restless bandit" model. For this model, Whittle [18] proposed an index policy whose asymptotic optimality was established by Weber and Weiss [15].

A number of stochastic scheduling models allow for jobs to arrive over time according to a stochastic process. A commonly used model in this setting is that of a multiclass queueing network. Multiclass queueing networks serve as useful models for problems in which several types of *activities* compete for a limited number of shared *resources*. They generalize deterministic job-shop problems in two ways: jobs arrive *over time*, and each job has a *random* processing time at each stage. The optimal control problem in a multiclass queueing network is to find an optimal allocation of the available resources to activities over time. Not surprisingly, index policies are optimal only for restricted versions of this general model. An important example is scheduling a multiclass single-server system with feedback: there are N types of jobs, type i jobs arrive according to a Poisson process with rate λ_i, require service according to a service-time distribution $F_i(\cdot)$ with mean processing time s_i, and incur holding costs at rate c_i per unit time. A type i job after undergoing processing becomes a type j job with probability p_{ij}, or exits the system with probability $1 - \sum_j p_{ij}$. The objective is to find a scheduling policy that minimizes the expected holding cost rate in steady-state. Klimov [9] proved the optimality of index policies for this model, as well as for the objective in which the total discounted holding cost is to be minimized. While the optimality result does not hold when there are many parallel machines, Glazebrook and Niño-Mora [7] showed that this rule is asymptotically optimal. For more general models, the prevailing approach is to use approximations such as fluid approximations [1] or diffusion approximations [8].

Applications

Stochastic scheduling models are applicable in many settings, most prominently in computer and communication networks, call centers, logistics and transportation, and manufacturing systems [4,10].

Cross References

- ► List Scheduling
- ► Minimum Weighted Completion Time

Recommended Reading

1. Avram, F., Bertsimas, D., Ricard, M.: Fluid models of sequencing problems in open queueing networks: an optimal control approach. In: Kelly, F.P., Williams, R.J. (eds.) Stochastic Networks. Proceedings of the International Mathematics Association, vol. 71, pp. 199–234. Springer, New York (1995)
2. Bertsimas, D., Niño-Mora, J.: Conservation laws, extended polymatroids and multiarmed bandit problems: polyhedral approaches to indexable systems. Math. Oper. Res. **21**(2), 257–306 (1996)
3. Bruno, J., Downey, P., Frederickson, G.N.: Sequencing tasks with exponential service times to minimize the expected flow time or makespan. J. ACM **28**, 100–113 (1981)
4. Dacre, M., Glazebrook, K., Nino-Mora, J.: The achievable region approach to the optimal control of stochastic systems. J. R. Stat. Soc. Series B **61**(4), 747–791 (1999)
5. Gittins, J.C., Jones, D.M.: A dynamic allocation index for the sequential design experiments. In: Gani, J., Sarkadu, K., Vince, I. (eds.) Progress in Statistics. European Meeting of Statisticians I, pp. 241–266. North Holland, Amsterdam (1974)
6. Gittins, J.C.: Bandit processes and dynamic allocation indices. J. R. Stat. Soc. Series B, **41**(2), 148–177 (1979)
7. Glazebrook, K., Niño-Mora, J.: Parallel scheduling of multiclass M/M/m queues: approximate and heavy-traffic optimization of achievable performance. Oper. Res. **49**(4), 609–623 (2001)
8. Harrison, J.M.: Brownian models of queueing networks with heterogenous customer populations. In: Fleming, W., Lions, P.L. (eds.) Stochastic Differential Systems, Stochastic Control Theory and Applications. Proceedings of the International Mathematics Association, pp. 147–186. Springer, New York (1988)
9. Klimov, G.P.: Time-sharing service systems I. Theory Probab. Appl. **19**, 532–551 (1974)
10. Pinedo, M.: Scheduling: Theory, Algorithms and Systems, 2nd ed. Prentice Hall, Englewood Cliffs (2002)
11. Rothkopf, M.: Scheduling with Random Service Times. Manag. Sci. **12**, 707–713 (1966)
12. Sevcik, K.C.: Scheduling for minimum total loss using service time distributions. J. ACM **21**, 66–75 (1974)
13. Smith, W.E.: Various optimizers for single-stage production. Nav. Res. Logist. Quart. **3**, 59–66 (1956)
14. Weber, R.R., Varaiya, P., Walrand, J.: Scheduling jobs with stochastically ordered processing times on parallel machines to minimize expected flow time. J. Appl. Probab. **23**, 841–847 (1986)
15. Weber, R.R., Weiss, G.: On an index policy for restless bandits. J. Appl. Probab. **27**, 637–648 (1990)
16. Weiss, G.: Turnpike optimality of Smith's rule in parallel machine stochastic scheduling. Math. Oper. Res. **17**, 255–270 (1992)
17. Whittle, P.: Multiarmed bandit and the Gittins index. J. R. Stat. Soc. Series B **42**, 143–149 (1980)
18. Whittle, P.: Restless bandits: Activity allocation in a changing world. In: Gani, J. (ed.) A Celebration of Applied Probability. J Appl. Probab. **25A**, 287–298 (1988)

Strategyproof

- ► Nash Equilibria and Dominant Strategies in Routing
- ► Truthful Multicast

Stretch Factor

- ► Applications of Geometric Spanner Networks
- ► Dilation of Geometric Networks
- ► Geometric Dilation of Geometric Networks

String

- ▶ Compressed Pattern Matching
- ▶ Sequential Approximate String Matching
- ▶ Suffix Tree Construction in Hierarchical Memory
- ▶ Text Indexing

String Sorting

1997; Bentley, Sedgewick

ROLF FAGERBERG
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Keywords and Synonyms

Sorting of multi-dimensional keys; Vector sorting

Problem Definition

The problem is to sort a set of strings into lexicographical order. More formally: A *string* over an *alphabet* Σ is a finite sequence $x_1x_2x_3\ldots x_k$ where $x_i \in \Sigma$ for $i = 1, \ldots, k$. The x_i's are called the *characters* of the string, and k is the *length* of the string. If the alphabet Σ is ordered, the *lexicographical order* on the set of strings over Σ is defined by declaring a string $x = x_1x_2x_3\ldots x_k$ smaller than a string $y = y_1y_2y_3\ldots y_l$ if either there exists a $j \geq 1$ such that $x_i = y_i$ for $1 \leq i < j$ and $x_j < y_j$, or if $k < l$ and $x_i = y_i$ for $1 \leq i \leq k$. Given a set S of strings over some ordered alphabet, the problem is to sort S according to lexicographical order.

The input to the string sorting problem consists of an array of pointers to the strings to be sorted. The output is a permutation of the array of pointers, such that traversing the array will point to the strings in non-decreasing lexicographical order.

The complexity of string sorting depends on the alphabet as well as the machine model. The main solution [15] described in this entry works for alphabets of unbounded size (i. e., comparisons are the only operations on characters of Σ), and can be implemented on a pointer machine. See below for more information on the asymptotic complexity of string sorting in various settings.

Key Results

This section is structured as follows: first the key result appearing in title of this entry [15] is described, then an overview of other relevant results in the area of string sorting is given.

The string sorting algorithm proposed by Bentley and Sedgewick in 1997 [15] is called Three-Way Radix Quicksort [5]. It works for unbounded alphabets, for which it achieves optimal performance.

Theorem 1 *The algorithm Three-Way Radix Quicksort sorts K strings of total length N in time $O(K \log K + N)$.*

That this time complexity is optimal follows by considering strings of the form `bbb...bx`, where all x's are different: Sorting the strings can be no faster than sorting the x's, and all `b`'s must be read (else an adversary could change one unread `b` to `a` or `c`, making the returned order incorrect). A more precise version of the bounds above (upper as well as lower) is $K \log K + D$, where D is the sum of the lengths of the *distinguishing prefixes* of the strings. The distinguishing prefix d_s of a string s in a set S is the shortest prefix of s which is not a prefix of another string in S (or is s itself, if s is a prefix of another string). Clearly, $K \leq D \leq N$.

The Three-Way Radix Quicksort of Bentley and Sedgewick is not the first algorithm to achieve this complexity—however, it is a very simple and elegant way of doing it. As demonstrated in [3,15], it is also very fast in practice. Although various elements of the algorithm had been noted earlier, their practical usefulness for string sorting was overlooked until the work in [15].

Three-Way Radix Quicksort is shown in pseudo-code in Fig. 1 (adapted from [5]), where S is a list of strings to be sorted and d is an integer. To sort S, an initial call SORT$(S, 1)$ is made. The value s_d denotes the dth character of the string s, and + denotes concatenation. The presentation in Fig. 1 assumes that all strings end in a special

```
SORT(S, d)
    IF |S| ≤ 1:
        RETURN
    Choose a partitioning character
    v ∈ {s_d | s ∈ S}
    S_< = {s ∈ S | s_d < v}
    S_= = {s ∈ S | s_d = v}
    S_> = {s ∈ S | s_d > v}
    SORT(S_<, d)
    IF v ≠ EOS:
        SORT(S_=, d + 1)
    SORT(S_>, d)
    S = S_< + S_= + S_>
```

String Sorting, Figure 1
Three-Way Radix Quicksort (assuming each string ends in a special EOS character)

End-Of-String (EOS) character (such as the null character in C). In an actual implementation, S will be an array of pointers to strings, and the sort will in-place (using an in-place method from standard Quicksort for three-way partitioning of the array into segments holding $S_<$, $S_=$, and $S_>$), rendering concatenation superfluous.

Correctness follows from the following invariant being maintained by the algorithm: At the start of a call SORT(S, d), all strings in S agree on the first $d-1$ characters.

Time complexity depends on how the partitioning character v is chosen. One particular choice is the median of all the dth characters (including doublets) of the strings in S. Partitioning and median finding can be done in time $O(|S|)$, which is $O(1)$ time per string partitioned. Hence, the total running time of the algorithm is the sum over all strings of the number of partitionings they take part in. For each string, let a partitioning be of type I if the string ends up in $S_<$ or $S_>$, and of type II if it ends up in $S_=$. For a string s, type II can only occur $|d_s|$ times and type I can only occur $\log K$ times. Hence, the running time is $O(K \log K + D)$.

Like for standard Quicksort, median finding impairs the constant factors of the algorithm, and more practical choices of partitioning character include selecting a random element among all the dth characters of the strings in S, and selecting the median of three elements in this set. The worst-case bound is lost, but the result is a fast, randomized algorithm.

Note that the ternary recursion tree of Three-Way Radix Quicksort is equivalent to a trie over the strings sorted, with trie nodes implemented by binary trees (where the elements stored in a binary tree are the characters of the trie edges leaving the trie node). The equivalence is as follows: an edge representing a recursive call on $S_<$ or $S_>$ corresponds to an edge of a binary tree (implementing a trie node), and an edge representing a recursive call on $S_=$ corresponds to a trie edge leading to a child node in the trie. This trie implementation is named Ternary Search Trees in [15]. Hence, Three-Way Radix Quicksort may additionally be viewed as a construction algorithm for an efficient dictionary structure for strings.

For the version of the algorithm where the partitioning character v is chosen as the median of all the dth characters, it is not hard to see that the binary trees representing the trie nodes become weighted trees, i. e., binary trees in which each element x has an associated weight w_x, and searches for x takes $O(\log W/w_x)$, where $W = \Sigma_x w_x$ is the sum of all weights in the binary tree. The weight of a binary tree node storing character x is the number of strings in the trie which reside below the trie edge labeled with character x and leaving the trie node represented by the binary tree. As shown in [13], in such a trie implementation searching for a string P among K stored strings takes time $O(\log K + |P|)$, which is optimal for unbounded (i. e., comparison-based) alphabets.

Other key results in the area of string sorting are now described. The classic string sorting algorithm is Radixsort, which assumes a constant sized alphabet. The Least-Significant-Digit-first variant is easy to implement, and runs in $O(N + l|\Sigma|)$ time, where l is the length of the longest string. The Most-Significant-Digit-first variant is more complicated to implement, but has a better running time of $O(D + d|\Sigma|)$, where D is the sum of the lengths of the distinguishing prefixes, and d is the longest distinguishing prefix. [12] discusses in depth efficient implementations of Radixsort.

If the alphabet consists of integers, then on a word-RAM the complexity of string sorting is essentially determined by the complexity of integer sorting. More precisely, the time (when allowing randomization) for sorting strings is $\Theta(\text{Sort}_{\text{Int}}(K) + N)$, where $\text{Sort}_{\text{Int}}(K)$ is the time to sort K integers [2], which currently is known to be $O(K\sqrt{\log\log K})$ [11].

Returning to comparison-based model, the papers [8,10] give generic methods for turning any data structure over one-dimensional keys into a data structure over strings. Using finger search trees, this gives an adaptive sorting method for strings which uses $O(N + K \log(F/K))$ time, where F is the number of inversions among the strings to be sorted.

Concerning space complexity, it has been shown [9] that string sorting can still be done in $O(K \log K + N)$ time using only $O(1)$ space besides the strings themselves. However, this assumes that all strings have equal lengths.

All algorithms so far are designed to work in internal memory, where CPU time is assumed to be the dominating factor. For external memory computation, a more relevant cost measure is the number of I/Os performed, as captured by the I/O-model [1], which models a two-level memory hierarchy with an infinite outer memory, an inner memory of size M, and transfer (I/Os) between the two levels taking place in blocks of size B. In external memory, upper bounds were first given in [4], along with matching lower bounds in restricted I/O-models. For a comparison based model where strings may only be moved in blocks of size B (hence, characters may not be moved individually), it is shown that string sorting takes $\Theta(N_1/B \log_{M/B}(N_1/B) + K_2 \log_{M/B} K_2 + N/B)$ I/Os, where N_1 is the total length of strings shorter than B characters, K_2 is the number of strings of at least B characters, and N is the total number of characters. This

bound is equal to the sum of the I/O costs of sorting the characters of the short strings, sorting B characters from each of the long strings, and scanning all strings. In the same paper, slightly better bounds in a model where characters may be moved individually in internal memory are given, as well as some upper bounds for non-comparison based string sorting. Further bounds (using randomization) for non-comparison based string sorting have been given, with I/O bounds of $O(K/B \log_{M/B}(K/M) \log\log_{M/B}(K/M) + N/B)$ [7] and[1] $O(K/B(\log_{M/B}(N/M))^2 \log_2 K + N/B)$.

Returning to internal memory, it may also there be the case that memory hierarchy effects are the determining factor for the running time of algorithms, but now due to cache faults rather than disk I/Os. Heuristic algorithms (i. e., algorithms without good worst case bounds), aiming at minimizing cache faults for internal memory string sorting, have been developed. Of these, the Burstsort line of algorithms [16] have particularly promising experimental results reported.

Applications

Data sets consisting partly or entirely of string data are very common: Most database applications have strings as one of the data types used, and in some areas, such as bioinformatics, web retrieval, and word processing, string data is predominant. Additionally, strings form a general and fundamental data model, containing e. g. integers and multi-dimensional data as special cases. Since sorting is arguably among the most important data processing tasks in any domain, string sorting is a general and important problem with wide practical applications.

Open Problems

As appears from the bounds discussed above, the asymptotic complexity of the string sorting problem is known for comparison based alphabets. For integer alphabets on the word-RAM, the problem is almost closed in the sense that it is equivalent to integer sorting, for which the gap left between the known bounds and the trivial linear lower bound is small.

In external memory, the situation is less settled. As noted in [4], a natural upper bound to hope for in a comparison based setting is to meet the lower bound of $\Theta(K/B \log_{M/B} K/M + N/B)$ I/Os, which is the sorting bound for K single characters plus the complexity of scanning the input. The currently known upper bounds only gets close to this if leaving the comparison based setting and allowing randomization.

Further open problems include adaptive sorting algorithms for other measures of presortedness than that used in [8,10], and algorithms for sorting general strings (not necessarily of equal lengths) using only $O(1)$ additional space [9].

Experimental Results

In [15], experimental comparison of two implementations (one simple and one tuned) of Three-Way Radix Quicksort with a tuned Quicksort [6] and a tuned Radixsort [12] showed the simple implementation to always outperform the Quicksort implementation, and the tuned implementation to be competitive with the Radixsort implementation.

In [3], experimental comparison among existing and new Radixsort implementations (including the one used in [15]), as well as tuned Quicksort and tuned Three-Way Radix Quicksort was performed. This study confirms the picture of Three-Way Radix Quicksort as very competitive, always being one of the fastest algorithms, and arguably the most robust across various input distributions.

Data Sets

The data sets used in [15]: http://www.cs.princeton.edu/~rs/strings/. The data sets used in [3]: http://www.jea.acm.org/1998/AnderssonRadixsort/.

URL to Code

Code in C from [15]:
http://www.cs.princeton.edu/~rs/strings/.
Code in C from [3]:
http://www.jea.acm.org/1998/AnderssonRadixsort/.
Code in Java from [14]:
http://www.cs.princeton.edu/~rs/Algs3.java1-4/code.txt.

Cross References

- ► Suffix Array Construction
- ► Suffix Tree Construction in Hierarchical Memory
- ► Suffix Tree Construction in RAM

Recommended Reading

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**, 1116–1127 (1988)
2. Andersson, A., Nilsson, S.: A new efficient radix sort. In: Proceedings of the 35th Annual Symposium on Foundations of

[1] Ferragina, personal communication.

Computer Science (FOCS '94), IEEE Comput. Soc. Press, pp. 714–721 (1994)
3. Andersson, A., Nilsson, S.: Implementing radixsort. ACM J. Exp. Algorithmics **3**, 7 (1998)
4. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory (extended abstract). In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97), ACM, ed., pp. 540–548. ACM Press, El Paso (1997),
5. Bentley, J., Sedgewick, R.: Algorithm alley: Sorting strings with three-way radix quicksort. Dr. Dobb's J. Softw. Tools **23**, 133–134, 136–138 (1998)
6. Bentley, J.L., McIlroy, M.D.: Engineering a sort function. Softw. Pract. Exp. **23**, 1249–1265 (1993)
7. Fagerberg, R., Pagh, A., Pagh, R.: External string sorting: Faster and cache-oblivious. In: Proceedings of STACS '06. LNCS, vol. 3884, pp. 68–79. Springer, Marseille (2006)
8. Franceschini, G., Grossi, R.: A general technique for managing strings in comparison-driven data structures. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP '04). LNCS, vol. 3142, pp. 606–617. Springer, Turku (2004)
9. Franceschini, G., Grossi, R.: Optimal in-place sorting of vectors and records. In: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05). LNCS, vol. 3580, pp. 90–102. Springer, Lisbon (2005)
10. Grossi, R., Italiano, G.F.: Efficient techniques for maintaining multidimensional keys in linked data structures. In: Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99). LNCS, vol. 1644, pp. 372–381. Springer, Prague (1999)
11. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proceedings of the 43rd Annual Symposium on Foundations of Computer Science (FOCS '02), pp. 135–144. IEEE Computer Society Press, Vancouver (2002)
12. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Comput. Syst. **6**, 5–27 (1993)
13. Mehlhorn, K.: Dynamic binary search. SIAM J. Comput. **8**, 175–198 (1979)
14. Sedgewick, R.: Algorithms in Java, Parts 1–4, 3rd edn. Addison-Wesley, (2003)
15. Sedgewick, R., Bentley, J.: Fast algorithms for sorting and searching strings. In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97), ACM, ed., pp. 360–369. ACM Press, New Orleans (1997)
16. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. ACM J. Exp. Algorithmics. **11** (2006)

Substring Parsimony

2001; Blanchette, Schwikowski, Tompa

Mathieu Blanchette
Department of Computer Science, McGill University, Montreal, QC, Canada

Problem Definition

The Substring Parsimony Problem, introduced by Blanchette et al. [1] in the context of motif discovery in biological sequences, can be described in a more general framework:

Input:

- A discrete space S on which an integral distance d is defined (i. e. $d(x, y) \in \mathbb{N}\ \forall x, y \in S$).
- A rooted binary tree $T = (V, E)$ with n leaves. Vertices are labeled $\{1, 2, \ldots, n, \ldots, |V|\}$, where the leaves are vertices $\{1, 2, \ldots, n\}$.
- Finite sets $S_1, S_2, \ldots, S_n$, where set $S_i \subseteq S$ is assigned to leaf i, for all $i = 1 \ldots n$.
- A non-negative integer t

Output: All solutions of the form $(x_1, x_2, \ldots, x_n, \ldots, x_{|V|})$ such that:

- $x_i \in S$ for all $i = 1 \ldots |V|$
- $x_i \in S_i$ for all $i = 1 \ldots n$
- $\sum_{(u,v)\in E} d(x_u, x_v) \leq t$

The problem thus consists of choosing one element x_i from each set S_i such that the Steiner distance of the set of points is at most t. This is done on a Steiner tree T of fixed topology. The case where $|S_i| = 1$ for all $i = 1 \ldots n$ is a standard Steiner tree problem on a fixed tree topology (see [11]). It is known as the Maximum Parsimony Problem and its complexity depends on the space S.

Key Results

The substring parsimony problem can be solved using a dynamic programming algorithm. Let $u \in V$ and $s \in S$. Let $W_u[s]$ be the score of the best solution that can be obtained for the subtree rooted at node u, under the constraint that node u is labeled with s, i. e.

$$W_u[s] = \min_{\substack{x_1,\ldots,x_{|V|}\in S \\ x_u = s}} \sum_{\substack{(i,j)\in E \\ i,j\in \text{subtree}(u)}} d(x_i, x_j)\,.$$

Let v be a child of u, and let $X_{(u,v)}[s]$ be the score of the best solution that can be obtained for the subtree consisting of node u together with the subtree rooted at its child v, under the constraint that node u is labeled with s:

$$X_{(u,v)}[s] = \min_{\substack{x_1,\ldots,x_{|V|}\in S \\ x_u = s}} \sum_{\substack{(i,j)\in E \\ i,j\in \text{subtree}(v)\cup\{(u,v)\}}} d(x_i, x_j)\,.$$

Then, we have:

$$W_u[s] = \begin{cases} 0 & \text{if } u \text{ is a leaf and } s \in S_u \\ +\infty & \text{if } u \text{ is a leaf and } s \notin S_u \\ \sum\limits_{v\in \text{children}(u)} X_{(u,v)}[s] & \text{if } u \text{ is not a leaf} \end{cases}$$

and

$$X_{(u,v)}[s] = \min_{s'\in S} W_u[s'] + d(s, s')\,.$$

Tables W and X can thus be computed using a dynamic programming algorithm, proceeding in a post-order traversal of the tree. Solutions can then be recovered by tracing the computation back for all s such that $W_{\text{root}}[s] \leq t$. Note that the same solution may be recovered more than once in this process.

A straight-forward implementation of this dynamic programming algorithm would run in time $O(n \cdot |S|^2 \cdot \gamma(S))$, where $\gamma(S)$ is the time needed to compute the distance between any two points in S. Let $N_a(S)$ be the maximum number of a-neighbors a point in S can have, i. e. $N_a(S) = \max_{x \in S} |\{y \in S : d(x, y) = a\}|$. Blanchette et al. [3] showed how to use a modified breadth-first search of the space S to compute each table $X_{(u,v)}$ in time $O(|S| \cdot N_1(S))$, thus reducing the total time complexity to $O(n \cdot |S| \cdot N_1(S))$. Since only solutions with a score of at most t are of interest, the complexity can be further reduced by only computing those table entries which will yield a score of at most t. This results in an algorithm whose running time is $O(n \cdot M \cdot N_{\lfloor t/2 \rfloor}(S) \cdot N_1(S))$ where $M = \max_{i=1\ldots n} |S_i|$.

The problem has been mostly studied in the context of biological sequence analysis, where $S = \{A, C, G, T\}^k$, for some small k ($k = 5, \ldots, 20$ are typical values). The distance d is the Hamming distance, and a phylogenetic tree T is given. The case where $|S_i| = 1$ for all $i = 1 \ldots n$ is known as the Maximum Parsimony Problem and can be solved in time $O(n \cdot k)$ using Fitch's algorithm [9] or Sankoff's algorithm [12]. In the more general version, a long DNA sequence P_u of length L is assigned to each leaf u. The set S_u is defined as the set of all k-substrings of P_u. In this case, $M = L - k + 1 \in O(L)$, and $N_a \in O(\min(4^k, (3k)^a))$, resulting in a complexity of $O(n \cdot L \cdot 3k \cdot \min(4^k, (3k)^{\lfloor d/2 \rfloor}))$. Notice that for a fixed k and d, the algorithm is linear over the whole sequence. The problem was independently shown to be NP-hard by Blanchette et al. [3] and by Elias [7].

Applications

Most applications are found in computational biology, although the algorithm can be applied to a wide variety of domains. The algorithm for the substring parsimony problem has been implemented in a software package called FootPrinter [5] and applied to the detection of transcription factor binding sites in orthologous DNA regulatory sequences through a method called phylogenetic footprinting [4]. Other applications include the search for conserved RNA secondary structure motifs in orthologous RNA sequences [2]. Variants of the problem have been defined to identify motifs regulating alternative splicing [13]. Blanchette et al. [3] study a relaxation of the problem where one does not require that a substring be chosen from each of the input sequences, but instead asks that substrings be chosen from a sufficiently large subset of the input sequence. Fang and Blanchette [8] formulate another variant of the problem where substring choices are constrained to respect a partial order relation defined by a set of local multiple sequence alignments.

Open Problems

Optimizations taking advantage of the specific structure of the space S may yield more efficient algorithms in certain cases. Many important variations could be considered. First, the case where the tree topology is not given needs to be considered, although the resulting problems would usually be NP-hard even when $|S_i| = 1$. Another important variation is one where the phylogenetic relationships between trees is not given by a tree but rather by a phylogenetic network [10]. Finally, randomized algorithms similar to those proposed by Buhler et al. [6] may yield important and practical improvements.

URL to Code

http://bio.cs.washington.edu/software.html

Cross References

- ▶ Closest Substring
- ▶ Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds
- ▶ Local Alignment (with Affine Gap Weights)
- ▶ Local Alignment (with Concave Gap Weights)
- ▶ Statistical Multiple Alignment
- ▶ Steiner Trees

Recommended Reading

1. Blanchette, M.: Algorithms for phylogenetic footprinting. In: RECOMB01: Proceedings of the Fifth Annual International Conference on Computational Molecular Biology, pp. 49–58. ACM Press, Montreal (2001)
2. Blanchette, M.: Algorithms for phylogenetic footprinting. Ph. D. thesis, University of Washington (2002)
3. Blanchette, M., Schwikowski, B., Tompa, M.: Algorithms for phylogenetic footprinting. J. Comput. Biol. **9**(2), 211–223 (2002)
4. Blanchette, M., Tompa, M.: Discovery of regulatory elements by a computational method for phylogenetic footprinting. Genome Res. **12**, 739–748 (2002)
5. Blanchette, M., Tompa, M.: Footprinter: A program designed for phylogenetic footprinting. Nucleic Acids Res. **31**(13), 3840–3842 (2003)
6. Buhler, J., Tompa, M.: Finding motifs using random projections. In: RECOMB01: Proceedings of the Fifth Annual Interna-

tional Conference on Computational Molecular Biology, 2001, pp. 69–76
7. Elias, I.: Settling the intractability of multiple alignment. J. Comput. Biol. **13**, 1323–1339 (2006)
8. Fang, F., Blanchette, M.: Footprinter3: phylogenetic footprinting in partially alignable sequences. Nucleic Acids Res. **34**(2), 617–620 (2006)
9. Fitch, W.M.: Toward defining the course of evolution: Minimum change for a specified tree topology. Syst. Zool. **20**, 406–416 (1971)
10. Huson, D.H., Bryant, D.: Application of phylogenetic networks in evolutionary studies. Mol. Biol. Evol. **23**(2), 254–267 (2006)
11. Sankoff, D., Rousseau, P.: Locating the vertices of a Steiner tree in arbitrary metric space. Math. Program. **9**, 240–246 (1975)
12. Sankoff, D.D.: Minimal mutation trees of sequences. SIAM J. Appl. Math. **28**, 35–42 (1975)
13. Shigemizu, D., Maruyama, O.: Searching for regulatory elements of alternative splicing events using phylogenetic footprinting. In: Proceedings of the Fourth Workshop on Algorithms for Bioinformatics. Lecture Notes in Computer Science, pp. 147–158. Springer, Berlin (2004)

Succinct Data Structures for Parentheses Matching

2001; Munro, Raman

Meng He
School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Keywords and Synonyms

Succinct balanced parentheses

Problem Definition

This problem is to design succinct representation of balanced parentheses in a manner in which a number of "natural" queries can be supported quickly, and use it to represent trees and graphs succinctly. The problem of succinctly representing balanced parentheses was initially proposed by Jacobson [6] in 1989, when he proposed *succinct data structures*, i. e. data structures that occupy space close to the information-theoretic lower bound to represent them, while supporting efficient navigational operations. Succinct data structures provide solutions to manipulate large data in modern applications. The work of Munro and Raman [8] provides an optimal solution to the problem of balanced parentheses representation under the word RAM model, based on which they design succinct trees and graphs.

Balanced Parentheses

Given a balanced parenthesis sequence of length $2n$, where there are n opening parentheses and n closing parentheses, consider the following operations:

- `findclose`(i) (`findopen`(i)), the matching closing (opening) parenthesis for the opening (closing) parenthesis at position i;
- `excess`(i), the number of opening parentheses minus the number of closing parentheses in the sequence up to (and including) position i;
- `enclose`(i), the closest enclosing (matching parenthesis) pair of a given matching parenthesis pair whose opening parenthesis is at position i.

Trees

There are essentially two forms of trees. An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their ranks, while in a *cardinal tree* of degree k, each child of a node is identified by a unique number from the set $\{1, 2, \cdots, k\}$. An *binary tree* is a cardinal tree of degree 2. The information-theoretic lower bound of representing an ordinal tree or binary tree of n nodes is $2n - o(n)$ bits, as there are $\binom{2n}{n}/(n+1)$ different ordinal trees or binary trees.

Consider the following operations on ordinal trees (a node is referred to by its preorder number):

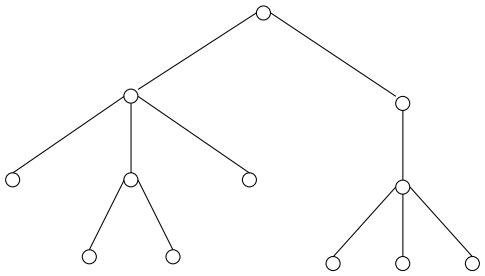
- `child`(x,i), the ith child of node x for $i \geq 1$;
- `child_rank`(x), the number of left siblings of node x;
- `depth`(x), the depth of x, i. e. the number of edges in the rooted path to node x;
- `parent`(x), the parent of node x;
- `nbdesc`(x), the number of descendants of node x;
- `height`(x), the height of the subtree rooted at node x;
- `LCA`(x,y), the lowest common ancestor of node x and node y.

On binary trees, the operations `parent`, `nbdesc` and the following operations are considered:

- `leftchild`(x) (`rightchild`(x)), the left (right) child of node x.

Graphs

Consider an undirected graph G of n vertices and m edges. Bernhart and Kainen [1] introduced the concept of *page book embedding*. A *k-book embedding* of a graph is a topological embedding of it in a book of k pages that specifies the ordering of the vertices along the spine, and carries each edge into the interior of one page, such that the edges on a given page do not intersect. Thus, a graph with



Balanced parentheses: ((( ) (( ) ( )) ( )) ((( ) ( ) ( ))))

Succinct Data Structures for Parentheses Matching, Figure 1
An example of the balanced parenthesis sequence of a given ordinal tree

one page is an *outerplanar graph*. The *pagenumber* or *book thickness* [1] of a graph is the minimum number of pages that the graph can be embedded in. A very common type of graphs are planar graphs, and any planar graph can be embedded in at most 4 pages [15]. Consider the following operations on graphs:

- `adjacency`(x,y), whether vertices x and y are adjacent;
- `degree`(x), the degree of vertex x;
- `neighbors`(x), the neighbors of vertex x.

Key Results

All the results cited are under the word RAM model with word size $\Theta(\lg n)$ bits[1], where n is the size of the problem considered.

Theorem 1 ([8]) *A sequence of balanced parentheses of length $2n$ can be represented using $2n + o(n)$ bits to support the operations* `findclose`, `findopen`, `excess` *and* `enclose` *in constant time.*

There is a polymorphism between a balanced parenthesis sequence and an ordinal tree: when performing a depth-first traversal of the tree, output an opening parenthesis each time a node is visited, and a closing parenthesis immediately after all the descendants of a node are visited (see Fig. 1 for an example). The work of Munro and Raman proposes a succinct representation of ordinal trees using $2n + o(n)$ bits to support `depth`, `parent` and `nbdesc` in constant time, and `child`(x,i) in $O(i)$ time. Lu and Yeh have further extended this representation to support `child`, `child_rank`, `height` and `LCA` in constant time.

[1] $\lg n$ denotes $\lceil \log_2 n \rceil$.

Theorem 2 ([8,7]) *An ordinal tree of n nodes can be represented using $2n + o(n)$ bits to support the operations* `child`, `child_rank`, `parent`, `depth`, `nbdesc`, `height` *and* `LCA` *in constant time.*

A similar approach can be used to represent binary trees:

Theorem 3 ([8]) *A binary tree of n nodes can be represented using $2n + o(n)$ bits to support the operations* `leftchild`, `rightchild`, `parent` *and* `nbdesc` *in constant time.*

Finally, balanced parentheses can be used to represent graphs. To represent a one-page graph, the work of Munro and Raman proposes to list the vertices from left to right along the spine, and each node is represented by a pair of parentheses, followed by zero or more closing parentheses and then zero or more opening parentheses, where the number of closing (or opening) parentheses is equal to the number of adjacent vertices to its left (or right) along the spine (see Fig. 2 for an example). This representation can be applied to each page to represent a graph with pagenumber k.

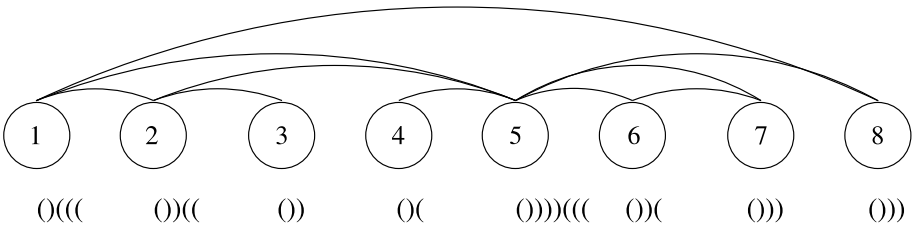
Theorem 4 ([8]) *An outerplanar graph of n vertices and m edges can be represented using $2n + 2m + o(n + m)$ bits to support operations* `adjacency` *and* `degree` *in constant time, and* `neighbors`*(x) in time proportional to the degree of x.*

Theorem 5 ([8]) *A graph of n vertices and m edges with pagenumber k can be represented using $2kn+2m+o(nk+m)$ bits to support operations* `adjacency` *and* `degree` *in $O(k)$ time, and* `neighbors`*(x) in $O(d(x) + k)$ time where $d(x)$ is the degree of x. In particular, a planar graph of n vertices and m nodes can be represented using $8n + 2m + o(n)$ bits to support operations* `adjacency` *and* `degree` *in constant time, and* `neighbors`*(x) in $O(d(x))$ time where $d(x)$ is the degree of x.*

Applications

Succinct Representation of Suffix Trees

As a result of the growth of the textual data in databases and on the World Wide Web, and also applications in bioinformatics, various indexing techniques have been developed to facilitate pattern searching. Suffix trees [14] are a popular type of text indexes. A suffix tree is constructed over the suffixes of the text as a tree-based data structure, so that queries can be performed by searching the suffixes of the text. It takes $O(m)$ time to use a suffix tree to check whether an arbitrary pattern P of length m is a substring of a given text T of length n, and to count the number of the occurrences, occ, of P in T. $O(occ)$ additional time



Succinct Data Structures for Parentheses Matching, Figure 2
An example of the balanced parenthesis sequence of a graph with one page

is required to list all the occurrences of P in T. However, a standard representation of a suffix tree requires somewhere between $4n \lg n$ and $6n \lg n$ bits, which is impractical for many applications.

By reducing the space cost of representing the tree structure of a suffix tree (using the work of Munro and Raman), Munro, Raman and Rao [9] have designed space-efficient suffix trees. Given a string of n characters over a fixed alphabet, they can represent a suffix tree using $n \lg n + O(n)$ bits to support the search of a pattern in $O(m + occ)$ time. To achieve this result, they have also extended the work of Munro and Raman to support various operations to retrieve the leaves of a given subtree in an ordinal tree. Based on similar ideas and by applying compressed suffix arrays [5], Sadakane [13] has proposed a different trade-off; his compressed suffix tree occupies $O(n \lg \sigma)$ bits, where σ is the size of the alphabet, and can support any algorithm on a suffix tree with a slight slowdown of a factor of polylog(n).

Succinct Representation of Functions

Munro and Rao [11] have considered the problem of succinctly representing a given function, $f : [n] \to [n]$, to support the computation of $f^k(i)$ for an arbitrary integer k. The straightforward representation of a function is to store the sequence $f(i)$, for $i = 0, 1, \ldots, n-1$. This takes $n \lg n$ bits, which is optimal. However, the computation of $f^k(i)$ takes $\Theta(k)$ time even in the easier case when k is positive. To address this problem, Munro and Rao [11] first extends the representation of balanced parenthesis to support the `next_excess`(i,k) operator, which returns the minimum j such that $j > i$ and `excess`(j) $= k$. They further use this operator to support the `level_anc`(x,i) operator on succinct ordinal trees, which returns the ith ancestor of node x for $i \geq 0$ (given a node x at depth d, its ith ancestor is the ancestor of x at depth $d - i$). Then, using succinct ordinal trees with the support for `level_anc`, they propose a succinct representation of functions using $(1 + \epsilon)n \lg n + O(1)$ bits for any fixed positive constant ϵ, to support $f^k(i)$ in constant time when $k > 0$, and $f^k(i)$ in $O(1 + |f^k(i)|)$ time when $k < 0$.

Multiple Parentheses and Graphs

Chuang et al. [3] have proposed to succinctly represent *multiple parentheses*, which is a string of $O(1)$ types of parentheses that may be unbalanced. They have extended the operations on balanced parentheses to multiple parentheses and designed a succinct representation. Based on the properties of canonical orderings for planar graphs, they have used multiple parentheses and the succinct ordinal trees to represent planar graphs. One of their main results is a succinct representation of planar graphs of n vertices and m edges in $2m + (5 + \epsilon)n + o(m + n)$ bits, for any constant $\epsilon > 0$, to support the operations supported on planar graphs in Theorem 5 in asymptotically the same amount of time. Chiang et al. [2] have further reduced the space cost to $2m + 3n + o(m + n)$ bits. In their paper, they have also shown how to support the operation `wrapped`(i), which returns the number of matching parenthesis pairs whose closest enclosing (matching parenthesis) pair is the pair whose opening parenthesis is at position i, in constant time on balanced parentheses. They have used it to show how to support the operation `degree`(x), which returns the degree of node x (i. e. the number of its children), in constant time on succinct ordinal trees.

Open Problems

One open research area is to support more operations on succinct trees. For example, it is not known how to support the operation to convert a given node's rank in a preorder traversal into its rank in a level-order traversal.

Another open research area is to further reduce the space cost of succinct planar graphs. It is not known whether it is possible to further improve the encoding of Chiang et al. [2].

A third direction for future work is to design succinct representations of dynamic trees and graphs. There have been some preliminary results by Munro et al. [10] on succinctly representing dynamic binary trees, which have been further improved by Raman and Rao [12]. It may be possible to further improve these results, and there are other related dynamic data structures that do not have succinct representations.

Experimental Results

Geary et al. [4] have engineered the implementation of succinct ordinal trees based on balanced parentheses. They have performed experiments on large XML trees. Their implementation uses orders of magnitude less space than the standard pointed-based representation, while supporting tree traversal operations with only a slight slowdown.

Cross References

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

Recommended Reading

1. Bernhart, F., Kainen P.C.: The book thickness of a graph. J. Comb. Theory B **27**(3), 320–331 (1979)
2. Chiang, Y.-T., Lin, C.-C., Lu, H.-I.: Orderly spanning trees with applications. SIAM J. Comput. **34**(4), 924–945 (2005)
3. Chuang, R.C.-N., Garg, A., He, X., Kao, M.-Y., Lu, H.-I.: Compact encodings of planar graphs via canonical orderings and multiple parentheses. Comput. Res. Repos. cs.DS/0102005 (2001)
4. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. **368**(3), 231–246 (2006)
5. Grossi, R., Gupta, A., Vitter J.S.: High-order entropy-compressed text indexes. In: Farach-Colton, M. (ed) Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 841–850, Philadelphia (2003)
6. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE, pp. 549–554, New York (1989)
7. Lu, H.-I., Yeh, C.-C.: Balanced parentheses strike back. Accepted to ACM Trans. Algorithms (2007)
8. Munro, J.I., Raman V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
9. Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. J. Algorithms **39**(2), 205–222 (2001)
10. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: Rao Kosaraju, S. (ed.) Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 529–536, Philadelphia (2001)
11. Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.): Proceedings of the 31st International Colloquium on Automata, Languages and Programming, pp. 1006–1015. Springer, Heidelberg (2004)
12. Raman, R., Rao, S. S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow J., Woeginger, G.J. (eds.) Proceedings of the 30th International Colloquium on Automata, Languages and Programming, pp. 357–368. Springer, Heidelberg (2003)
13. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. (2007) Online first. http://dx.doi.org/10.1007/s00224-006-1198-x
14. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11. IEEE, New York (1973)
15. Yannakakis, M.: Four pages are necessary and sufficient for planar graphs. In: Hartmanis, J. (ed.) Proceedings of the 18th Annual ACM-SIAM Symposium on Theory of Computing, pp. 104–108. ACM, New York (1986)

Succinct Encoding of Permutations: Applications to Text Indexing

2003; Munro, Raman, Raman, Rao

Jérémy Barbay[1], J. Ian Munro[2]

[1] Department of Computer Science, University of Chile, Santiago, Chile

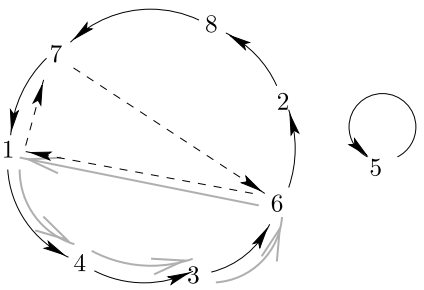
[2] Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Problem Definition

A succinct data structure for a given data type is a representation of the underlying combinatorial object that uses an amount of space "close" to the information theoretic lower bound together with algorithms that support operations of the data type "quickly." A natural example is the representation of a binary tree [5]: an arbitrary binary tree on n nodes can be represented in $2n + o(n)$ bits while supporting a variety of operations on any node, which include finding its parent, its left or right child, and returning the size of its subtree, each in $O(1)$ time. As there are $\binom{2n}{n}/(n+1)$ binary trees on n nodes and the logarithm of this term[1] is $2n - o(n)$, the space used by this representation is optimal to within a lower-order term.

In the applications considered in this entry, the principle concern is with indexes supporting search in strings and in XML-like documents (i. e., tree-structured objects with labels and "free text" at various nodes). As it happens, not only labeled trees but also arbitrary binary relations

[1] All logarithms are taken to the base 2. By convention, the iterated logarithm is denoted by $\lg^{(i)} n$; hence, $\lg \lg \lg x$ is $\lg^{(3)} x$.



i		1	2	3	4	5	6	7	8
$\pi(i)$	=	4	8	6	3	5	2	1	7
B	=	1	0	0	0	0	1	1	0
S	=	7					1	6	

Succinct Encoding of Permutations: Applications to Text Indexing, Figure 1
A permutation on {1, . . . , 8}, with two cycles and three back pointers. The *full black lines* correspond to the permutation, the *dashed lines* to the back pointers and the *gray lines* to the edges traversed to compute $\pi^{-1}(3)$

over finite domains are key building blocks for this. Preprocessing such data structures so as to be able to perform searches is a complex process requiring a variety of subordinate structures.

A basic building block for this work is the representation of a permutation of the integers $\{0, \ldots, n-1\}$, denoted by $[n]$. A permutation π is trivially representable in $n\lceil \lg n \rceil$ bits which is within $O(n)$ bits of the information theoretic bound of $\lg(n!)$. The interesting problem is to support both the permutation and its inverse: namely, how to represent an arbitrary permutation π on $[n]$ in a succinct manner so that $\pi^k(i)$ (π iteratively applied k times starting at i, where k can be any integer so that π^{-1} is the inverse of π) can be evaluated quickly.

Key Results

Munro et al. [7] studied the problem of succinctly representing a permutation to support computing $\pi^k(i)$ quickly. They give two solutions: one supports the operations arbitrarily quickly, at the cost of extra space; the other uses essentially optimal space at the cost of slower evaluation.

Given an integer parameter t, the permutations π and π^{-1} can be supported by simply writing down π in an array of n words of $\lceil \lg n \rceil$ bits each, plus an auxiliary array S of at most n/t shortcuts or back pointers. In each cycle of length at least t, every tth element has a pointer t steps back. $\pi(i)$ is simply the ith value in the primary structure, and $\pi^{-1}(i)$ is found by moving forward until a back pointer is found and then continuing to follow the cycle to the location that contains the value i. The trick is in the encoding of the locations of the back pointers: this is done with a simple bit vector B of length n, in which a 1 indicates that a back pointer is associated with a given location. B is augmented using $o(n)$ additional bits so that the number of 1's up to a given position and the position of the rth 1 can be found in constant time (i. e., using the rank and select operations on binary strings [8]). This gives the location of the appropriate back pointer in the auxiliary array S.

For example, the permutation $\pi = (4, 8, 6, 3, 5, 2, 1, 7)$ consists of two cycles, $(\mathbf{1}, 4, 3, \mathbf{6}, 2, 8, \mathbf{7})$ and (5) (Fig. 1). For $t = 3$, the back pointers are cycling backward between 1, 6 and 7 in the largest cycle (there are none in the other because it is smaller than t). To find $\pi^{-1}(3)$, follow π from 3 to 6, observe that 6 is a back pointer because it is marked by the second 1 in B, and follow the second value of S to 1, then follow π from 1 to 4 and then to 3: the predecessor of 3 has been found. As there are back pointers every t elements in the cycle, finding the predecessor requires $O(t)$ memory accesses.

For arbitrary i and k, $\pi^k(i)$ is supported by writing the cycles of π together with a bit vector B marking the beginning of each cycle. Observe that the cycle representation itself is a permutation in "standard form," call it σ. For example, the permutation $\pi = (6, 4, 3, 5, 2, 1)$ has three cycles $\{(1, 6), (3), (2, 5, 4)\}$ and is encoded by the permutation $\sigma = (1, 6, 3, 2, 5, 4)$ and the bit vector $B = (1, 0, 1, 1, 0, 0)$. The first task is to find i in the representation: it is in position $\sigma^{-1}(i)$. The segment of the representation containing i is found through the rank and select operations on B. From this $\pi^k(i)$ is easily determined by taking k modulo the cycle length and moving that number of steps around the cycle starting at the position of i.

Other than the support of the inverse of σ, all operations are performed in constant time; hence, the total time depends on the value chosen for t.

Theorem 1 (Munro et al. [7]) *There is a representation of an arbitrary permutation π on $[n]$ using at most $(1+\varepsilon)n \lg n + O(n)$ bits that can support the operation $\pi^k()$ in time $O(1/\varepsilon)$, for any constant ε less than 1 and for any arbitrary value of k.*

It is not difficult to prove that this technique is optimal under a restricted model of a pointer machine. So, for example, using $O(n)$ extra bits (i. e., $O(n/\lg n)$ extra words), $\Omega(\lg n)$ time is necessary to compute both π and π^{-1}. However, using another approach Munro et al. [7] demonstrated that the lower bound suggested does not hold in the RAM model. The approach is based on the Benes network, a communication network composed of switches that can be used to implement permutations.

Theorem 2 (Munro et al. [7]) *There is a representation of an arbitrary permutation π on $[n]$ using at most $\lceil \lg(n!) \rceil + O(n)$ bits that can support the operation $\pi^k()$ in time $O(\lg n/\lg^{(2)} n)$.*

While this data structure uses less space than the other, it requires more time for each operation. It is not known whether this time bound can be improved using only $O(n)$ "extra space." As a consequence, the first data structure is used in all applications. Obviously, any other solution can be used, potentially with a better time/space trade-off.

Applications

The results on permutations are particularly useful for two lines of research: first in the extension of the results on permutation to arbitrary integer functions; and second, and probably more importantly, in encoding and indexing text strings, which themselves are used to encode sparse binary relations and labeled trees. This section summarizes some of these results.

Functions

Munro and Rao [9] extended the results on permutations to arbitrary functions from $[n]$ to $[n]$. Again $f^k(i)$ indicates the function iterated k times starting at i. If k is nonnegative, this is straightforward. The case in which k is negative is more interesting as the image is a (possibly empty) multiset over $[n]$ (see Fig. 2 for an example). Whereas π is a set of cycles, f can be viewed as a set of cycles in which each node is the root of a tree. Starting at any node (element of $[n]$), the evaluation moves one step toward the root of the tree or one step along a cycle (e. g., $f(8) = 7$, $f(10) = 11$). Moving k steps in a positive direction is straightforward; one moves up a tree and perhaps around a cycle (e. g. $f^5(9) = f^3(9) = 3$) When k is negative one must determine all nodes of distance k from the starting location, i, in the direction towards the leaves of the trees (e. g., $f^{-1}(13) = \{1, 11, 12\}$, $f^{-1}(3) = \{4, 5\}$). The key technical issue is to run across succinct tree representations picking off all nodes at the appropriate levels.

Theorem 3 (Munro and Rao [9]) *For any fixed ε, there is a representation of a function $f : [n] \to [n]$ that takes $(1+\varepsilon)n \lg n + O(1)$ bits of space, and supports $f^k(i)$ in $O(1 + |f^k(i)|)$ time, for any integer k and for any $i \in [n]$.*

Text Strings

Indexing text strings to support the search for patterns is an important general issue. Barbay et al. [2] considered "negative" searches, along the following lines.

Definition 1 Consider a string $S[1, n]$ over the alphabet $[l]$. A position $x \in [n]$ *matches a literal* $\alpha \in [l]$ if $S[x] = \alpha$. A position $x \in [n]$ *matches a literal* $\bar{\alpha}$ if $S[x] \neq \alpha$. The set $\{\bar{1}, \ldots, \bar{l}\}$ is denoted by $[\bar{l}]$.

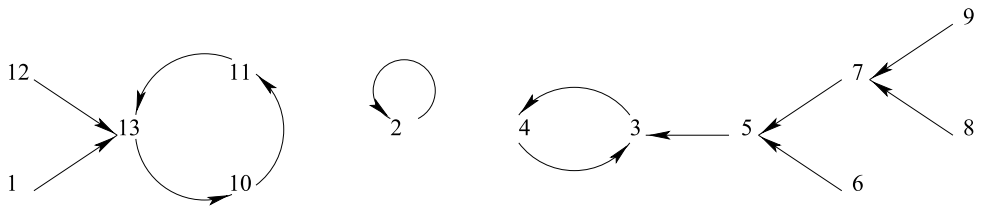
Given a string S of length n over an alphabet of size l, for any position x in the string, any literal $\alpha \in [l] \cup [\bar{l}]$ and any integer r, consider the following operators:

- $\texttt{string\_rank}_S(\alpha, x)$: the number of occurrences of α in $S[1..x]$;
- $\texttt{string\_select}_S(\alpha, r)$: the position of the rth occurrence of α in S, or ∞ if none exists;
- $\texttt{string\_access}_S(x)$: the label $S[x]$;
- $\texttt{string\_pred}_S(\alpha, x)$: the last occurrence of α in $S[1 \ldots x]$, or ∞ if none exists;
- $\texttt{string\_succ}_S(\alpha, r)$: the first occurrence of α in $S[x \ldots]$, or ∞ if none exists.

Golynski et al. [4] observed that a string of length l on alphabet $[l]$ can be encoded and indexed by a permutation on $[l]$ (which for each label lists the positions of all its occurrences) together with a bit vector of length $2l$ (which signals the end of each sublist of occurrences corresponding to a label). For instance, the string *ACCA* on alphabet $\{A, B, C, D\}$ is encoded by the permutation $(1, 4, 2, 3)$ and the bit vector $(0, 0, 1, 1, 1, 0, 0, 1)$. Golynski et al. were then able to support the operators rank, select and access in time $O(\lg^{(2)} n)$, by using a value of $t = \lg^{(2)} n$ in the encoding of permutation of Theorem 1.

This encoding achieves fast support for the search operators defined above restricted to labels (not literals), with a small overhead in space, by integrating the encodings of the text and the indexing information. Barbay et al. [2] extended those operators to literals, and showed how to separate the *succinct encoding* of the string S, in a manner that assumes we can access a word of S in a fixed time bound, and a *succinct index* containing auxiliary information useful to support the search operators defined above.

Theorem 4 (Barbay et al. [2]) *Given access to a label in the raw encoding of a string $S \in [l]^n$ in time $f(n, l)$, there is a succinct index using $n(1 + o(\lg l))$ bits that supports the operators* $\texttt{string\_rank}_S$, $\texttt{string\_pred}_S$



i		1	2	3	4	5	6	7	8	9	10	11	12	13
$f(i)$	=	13	2	4	3	3	5	5	7	7	11	13	13	10
$f^{-1}(i)$	=	{}	{2}	{4,5}	{3}	{6,7}	{}	{8,9}	{}	{}	{13}	{10}	{}	{1,11,12}

Succinct Encoding of Permutations: Applications to Text Indexing, Figure 2
A function on {1, . . . , 13}, with three cycles and two nontrivial tree structures

and $\texttt{string\_succ}_S$ *for any literal* $\alpha \in [l] \cup [\bar{l}]$ *in* $O(\lg^{(2)} l \cdot \lg^{(3)} l \cdot (f(n,t) + \lg^{(2)} l))$ *time, and the operator* $\texttt{string\_select}_S$ *for any label* $\alpha \in [l]$ *in* $O(\lg^{(3)} l \cdot (f(n,t) + \lg^{(2)} l))$ *time.*

The separation between the encoding of the string or of an XML-like document and its index has two main advantages:

1. The string can now be compressed and searched at the same time, provided that the compressed encoding of the string supports the access in reasonable time, as does the one described by Ferragina and Venturini [3].
2. The operators can be supported for several orderings of the string, for instance, induced by distinct traversals of a labeled tree, with only a small cost in space. It is important, for instance, when those orders correspond to various traversals of a labeled structure, such as the depth-first and Depth First Uniary Degree Sequence (DFUDS) traversals of a labeled tree [2].

Binary Relations

Given two ordered sets of sizes l and n, denoted by $[l]$ and $[n]$, a binary relation R between these sets is a subset of their Cartesian product, i. e., $R \subset [l] \times [n]$. It is used, for instance, to represent the relation between a set of labels $[l]$ and a set of objects $[n]$.

Although a string can be seen as a particular case of a binary relation, where the objects are positions and exactly one label is associated with each position, the search operations on binary relations are diverse, including operators on both the labels and the objects. For any literal α, object x and integer r, consider the following operators:

- $\texttt{label\_rank}_R(\alpha, x)$: the number of objects labeled α preceding or equal to x;
- $\texttt{label\_select}_R(\alpha, r)$: the position of the rth object labeled α if any, or ∞ otherwise;
- $\texttt{label\_nb}_R(\alpha)$, the number of objects with label α;
- $\texttt{object\_rank}_R(x, \alpha)$: the number of labels associated with object x preceding or equal to label α;
- $\texttt{object\_select}_R(x, r)$: the rth label associated with object x, if any, or ∞ otherwise;
- $\texttt{object\_nb}_R(x)$: the number of labels associated with object x;
- $\texttt{table\_access}_R(x, \alpha)$: checks whether object x is associated with label α.

Barbay et al. [1] observed that such a binary relation, consisting of t pairs from $[n] \times [l]$, can be encoded as a text string S listing the t labels, and a binary string B indicating how many labels are associated with each object. So search operations on the objects associated with a fixed label are reduced to a combination of operators on text and binary strings. Using a more direct reduction to the encoding of permutations, the index of the binary relation can be separated from its encoding, and even more operators can be supported [2].

Theorem 5 (Barbay et al. [2]) *Given support for* $\texttt{object\_access}_R$ *in* $f(n, l, t)$ *time on a binary relation formed by* t *pairs from an object set* $[n]$ *and a label set* $[l]$, *there is a succinct index using* $t(1 + o(\lg l))$ *bits that supports* $\texttt{label\_rank}_R$ *for any literal* $\alpha \in [l] \cup [\bar{l}]$ *and* $\texttt{label\_access}_R$ *for any label* $\alpha \in [l]$ *in* $O(\lg^{(2)} l \cdot \lg^{(3)} l \cdot (f(n, l, t) + \lg^{(2)} l))$ *time, and* $\texttt{label\_select}_R$ *for any label* $\alpha \in [l]$ *in* $O(\lg^{(3)} l \cdot (f(n, l, t) + \lg^{(2)} l))$ *time.*

To conclude this entry, note that a labeled tree T can be represented by an ordinal tree coding its structure [6] and a string S listing the labels of the nodes. If the labels are listed in preorder (respectively in DFUDS order) the operator $\texttt{string\_succ}_S$ enumerates all the descendants (respectively children) of a node matching some literal α. Using succinct indexes, a single encoding of the labels and the support of a permutation between orders is sufficient to implement both enumerations, and other search operators on the labels. These issues, along with strings and la-

beled trees compression techniques which achieve the entropy of the indexed data, are covered in more detail in the entries cited in ► Tree Compression and Indexing.

Cross References

► Compressed Suffix Array
► Compressed Text Indexing
► Rank and Select Operations on Binary Strings
► 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: Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM). Lecture Notes in Computer Science (LNCS), vol. 4009, pp. 24–35. Springer, Berlin (2006)
2. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 680–689. ACM, SIAM (2007)
3. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. In: Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 690–695. ACM, SIAM (2007)
4. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 368–373. ACM, SIAM (2006)
5. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 549–554 (1989)
6. Jansson, J., Sadakane, K., Sung, W.-K.: Ultra-succinct representation of ordered trees. In: Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 575–584. ACM, SIAM (2007)
7. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Computer Science (LNCS), vol. 2719, pp. 345–356. Springer, Berlin (2003)
8. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**, 762–776 (2001)
9. Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Computer Science (LNCS), vol. 3142, pp. 1006–1015. Springer, Berlin (2004)

Suffix Array Construction

2006; Kärkkäinen, Sanders, Burkhardt

Juha Kärkkäinen
Department of Computer Science, University of Helsinki, Helsinki, Finland

Keywords and Synonyms

Suffix sorting; Full-text index construction

Problem Definition

The *suffix array* [5,14] is the lexicographically sorted array of all the suffixes of a string. It is a popular text index structure with many applications. The subject of this entry are algorithms that construct the suffix array.

More precisely, the input to a suffix array construction algorithm is a *text string* $T = T[0, n) = t_0 t_1 \cdots t_{n-1}$, i. e., a sequence of n *characters* from an *alphabet* Σ. For $i \in [0, n]$, let S_i denote the *suffix* $T[i, n) = t_i t_{i+1} \cdots t_{n-1}$. The output is the *suffix array* $SA[0, n]$ of T, a permutation of $[0, n]$ satisfying $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n]}$, where $<$ denotes the *lexicographic order* of strings.

Two specific models for the alphabet Σ are considered. An *ordered alphabet* is an arbitrary ordered set with constant time character comparisons. An *integer alphabet* is the integer range $[1, n]$. There is also a result that holds for any alphabet.

Many applications require that the suffix array is augmented with additional information, most commonly with the *longest common prefix* array $LCP[0, n)$. An entry $LCP[i]$ of the LCP array is the length of the longest common prefix of the suffixes $S_{SA[i]}$ and $S_{SA[i+1]}$. The *enhanced suffix array* [1] adds two more arrays to obtain a full range of text index functionalities.

Another related array, the *Burrows–Wheeler transform* $BWT[0, n)$ is often computed by suffix array construction using the equations $BWT[i] = T[SA[i] - 1]$ when $SA[i] \neq 0$ and $BWT[i] = T[n - 1]$ when $SA[i] = 0$.

There are other important text indexes, most notably suffix trees and compressed text indexes, covered in separate entries. Each of these indexes have their own construction algorithms, but they can also be constructed efficiently from each other. However, in this entry, the focus is on direct suffix array construction algorithms that do not rely on other text indexes.

Key Results

The naive approach to suffix array construction is to use a general sorting algorithm or an algorithm for sorting strings. However, any such algorithm has a worst-case time complexity $\Omega(n^2)$ because the total length of the suffixes is $\Omega(n^2)$.

The first efficient algorithms were based on the *doubling technique* of Karp, Miller, and Rosenberg [8]. The idea is to assign a *rank* to all substrings whose length is a power of two. The rank tells the lexicographic order of

the substring among substrings of the same length. Given the ranks for substrings of length h, the ranks for substrings of length $2h$ can be computed using a radixsort step in linear time (doubling). The technique was first applied to suffix array construction by Manber and Myers [14]. The best practical algorithm based on the technique is by Larsson and Sadakane [13].

Theorem 1 (Manber and Myers [14]; Larsson and Sadakane [13]) *The suffix array can be constructed in $\mathcal{O}(n \log n)$ worst-case time, which is optimal for the ordered alphabet.*

Faster algorithms for the integer alphabet are based on a different technique, recursion. The basic procedure is as follows.

1. Sort a subset of the suffixes. This is done by constructing a shorter string, whose suffix array gives the order of the desired subset. The suffix array of the shorter string is constructed by recursion.
2. Extend the subset order to full order.

The technique first appeared in suffix tree construction [4], but 2003 saw the independent and simultaneous publication of three linear time suffix array construction algorithms based on the approach but not using suffix trees. Each of the three algorithms uses a different subset of suffixes requiring a different implementation of the second step.

Theorem 2 (Kärkkäinen, Sanders and Burkhardt [7]; Kim el al. [10]; Ko and Aluru [11]) *The suffix array can be constructed in the optimal linear time for the integer alphabet.*

The algorithm of Kärkkäinen, Sanders, and Burkhardt [7] has generalizations for several parallel and hierarchical memory models of computation including an optimal algorithm for external memory and a linear work algorithm for the BSP model.

The above algorithms and many other suffix array construction algorithms are surveyed in [18].

The $\Omega(n \log n)$ lower bound for the ordered alphabet mentioned in Theorem 1 comes from the sorting complexity of characters, since the initial characters of the sorted suffixes are the text characters in sorted order. Theorem 2 allows a generalization of this result. For any alphabet, one can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1, n]$ by replacing the characters of T with their ranks. The suffix array of T' is exactly the same as the suffix array of T. Optimal algorithms for the integer alphabet then give the following result.

Theorem 3 *For any alphabet, the complexity of suffix array construction is the same as the complexity of sorting the characters of the string.*

The result extends to the related arrays.

Theorem 4 (Kasai et al. [9]; Abouelhoda, Kurtz and Ohlebusch [1]) *The LCP array, the enhanced suffix array, and the BWT can be computed in linear time given the suffix array.*

One of the main advantages of suffix arrays over suffix trees is their smaller space requirement (by a constant factor), and a significant effort has been spent making construction algorithms space efficient, too. A technique based on the notion of *difference covers* gives the following results.

Theorem 5 (Burkhardt and Kärkkäinen [2]; Kärkkäinen, Sanders and Burkhardt [7]) *For any $v = \mathcal{O}(n^{2/3})$, the suffix array can be constructed in $\mathcal{O}(n(v + \log n))$ time for the ordered alphabet and in $\mathcal{O}(nv)$ time for the integer alphabet using $\mathcal{O}(n/\sqrt{v})$ space in addition to the input (the string T) and the output (the suffix array).*

Kärkkäinen [6] uses the difference cover technique to construct the suffix array in blocks without ever storing the full suffix array obtaining the following result for computing the BWT.

Theorem 6 (Kärkkäinen [6]) *For any $v = \mathcal{O}(n^{2/3})$, the BWT can be constructed in $\mathcal{O}(n(v + \log n))$ time for the ordered alphabet using $\mathcal{O}(n/\sqrt{v})$ space in addition to the input (the string T) and the output (the BWT).*

Compressed text index construction algorithms are alternatives to space-efficient BWT computation.

Applications

The suffix array is a simple and powerful text index structure with numerous applications detailed in the entry *Text Indexing*. In addition, due to the existence of efficient and practical construction algorithms, the suffix array is often used as an intermediate data structure in computing something else. The BWT is usually computed from the suffix array and has applications in text compression and compressed index construction. The suffix tree is also easy to construct given the suffix array and the LCP array.

Open Problems

Theoretically, the suffix array construction problem is essentially solved. The development of ever more efficient

practical algorithms is still going on with several different nontrivial heuristics available [18] including very recent ones [15].

Experimental Results

An experimental comparison of a large number of suffix array construction algorithms is presented in [18]. The best algorithms in the comparison are the algorithm by Maniscalco and Puglisi [15], which is the fastest but has an $\Omega(n^2)$ worst-case complexity, and a variant of the algorithm by Burkhardt and Kärkkäinen [2], which is the fastest among algorithms with good worst-case complexity. Both algorithms are also space efficient. The algorithm of Manzini and Ferragina [17] is still slightly more space efficient and also very fast in practice.

There are also experiments with parallel [12] and external memory algorithms [3]. Variants of the algorithm by Kärkkäinen, Sanders and Burkhardt [7] show high performance and scalability in both cases.

Algorithms for computing the LCP array from the suffix array are compared in [16].

Data Sets

The input to a suffix array construction algorithm is simply a text, so an abundance of data exists. Commonly used text collections include the Canterbury Corpus at http://corpus.canterbury.ac.nz/, the corpus compiled by Manzini and Ferragina at http://www.mfn.unipmn.it/~manzini/lightweight/corpus/, and the Pizza&Chili Corpus at http://pizzachili.dcc.uchile.cl/texts.html.

URL to Code

The implementations of many of the algorithms mentioned here are publicly available, for example: http://www.larsson.dogma.net/research.html [13], http://www.mpi-sb.mpg.de/~sanders/programs/suffix/ [7], and http://www.cs.helsinki.fi/juha.karkkainen/publications/cpm03.tar.gz [2]. Manzini provides a package that computes the LCP array and the BWT, too, at http://www.mfn.unipmn.it/~manzini/lightweight/index.html. The `bzip2` compression program (http://www.bzip.org/) computes the BWT through suffix array construction.

Cross References

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

Recommended Reading

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discret. Algorithms **2**, 53–86 (2004)
2. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching. LNCS, vol. 2676, pp. 55–69. Springer, Berlin/Heidelberg (2003)
3. Dementiev, R., Mehnert, J., Kärkkäinen, J., Sanders, P.: Better external memory suffix array construction. ACM J. Exp. Algorithmics (2008) in press
4. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. Assoc. Comput. Mach. **47**, 987–1011 (2000)
5. Gonnet, G., Baeza-Yates, R., Snider, T.: New indices for text: PAT trees and PAT arrays. In: Frakes, W.B., Baeza-Yates, R. (eds.) Information Retrieval: Data Structures & Algorithms. pp. 66–82 Prentice-Hall, Englewood Cliffs (1992)
6. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theor. Comput. Sci. **387**, 249–257 (2007)
7. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. Assoc. Comput. Mach. **53**, 918–936 (2006)
8. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proc. 4th Annual ACM Symposium on Theory of Computing, pp. 125–136. ACM Press, New York (1972)
9. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. 12th Annual Symposium on Combinatorial Pattern Matching, vol. (2089) of LNCS. pp. 181–192. Springer, Berlin/Heidelberg (2001)
10. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. J. Discret. Algorithms **3**, 126–142 (2005)
11. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discret. Algorithms **3**, 143–156 (2005)
12. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. In: Proc. 13th European PVM/MPI User's Group Meeting. LNCS, vol. 4192, pp. 22–29. Springer, Berlin/Heidelberg (2006)
13. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theor. Comput. Sci. **387**, 258–272 (2006)
14. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**, 935–948 (1993)
15. Maniscalco, M.A., Puglisi, S.J.: Faster lightweight suffix array construction. In: Proc. 17th Australasian Workshop on Combinatorial Algorithms, pp. 16–29. Univ. Ballavat, Ballavat (2006)
16. Manzini, G.: Two space saving tricks for linear time LCP array computation. In: Proc. 9th Scandinavian Workshop on Algorithm Theory. LNCS, vol. 3111, pp. 372–383. Springer, Berlin/Heidelberg (2004)
17. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**, 33–50 (2004)
18. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2), Article 4, 31 pages (2007)

Suffix Tree Construction in Hierarchical Memory

2000; Farach-Colton, Ferragina, Muthukrishnan

Paolo Ferragina
Department of Computer Science, University of Pisa, Pisa, Italy

Keywords and Synonyms

Suffix array construction; String B-tree construction; Full-text index construction

Problem Definition

The suffix tree is the ubiquitous data structure of combinatorial pattern matching because of its elegant uses in a myriad of situations--just to cite a few, searching, data compression and mining, bioinformatics [6]. In these applications, the large data sets now available involve the use of numerous memory levels which constitute the storage medium of modern PCs: L1 and L2 caches, internal memory, multiple disks and remote hosts over a network. The power of this memory organization is that it may be able to offer the expected access time of the fastest level (i. e. cache) while keeping the average cost per memory cell near the one of the cheapest level (i. e. disk), provided that data are properly *cached* and *delivered* to the requiring algorithms. Neglecting questions pertaining to the cost of memory references may even prevent the use of algorithms on large sets of input data. Engineering research is presently trying to improve the input/output subsystem to reduce the impact of these issues, but it is very well known [16] that the improvements achievable by means of a *proper arrangement of data* and a properly *structured algorithmic computation* abundantly surpass the best-expected technology advancements.

The Model of Computation

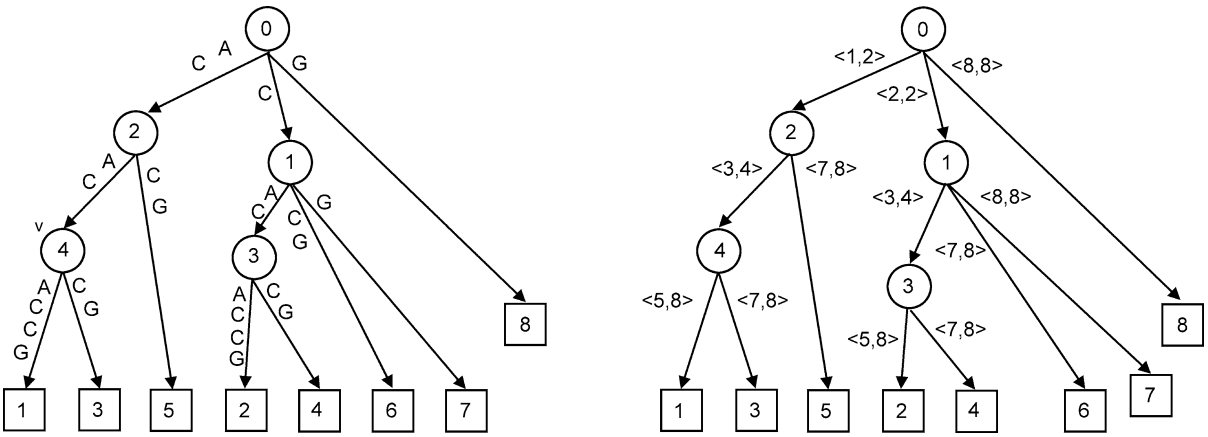
In order to reason about algorithms and data structures operating on hierarchical memories, it is necessary to introduce a *model of computation* that grasps the essence of real situations so that algorithms that are good in the model are also good in practice. The model considered here is the *external memory model* [16], which received much attention because of its simplicity and reasonable accuracy. A computer is abstracted to consist of *two memory levels*: the internal memory of size M, and the (unbounded) disk memory which operates by reading/writing data in blocks of size B (called *disk pages*). The performance of algorithms is then evaluated by counting: (a) the number of disk accesses (I/Os), (b) the internal running time (CPU time), and (c) the number of disk pages occupied by the data structure or used by the algorithm as its working space. This simple model suggests, correctly, that a good external-memory algorithm should exploit both *spatial locality* and *temporal locality*. Of course, "I/O" and "two-level view" refer to any two levels of the memory hierarchy with their parameters M and B properly set.

Notation

Let $S[1, n]$ be a string drawn from alphabet Σ, and consider the notation: S_i for the ith *suffix* of string S, $\mathtt{lcp}(\alpha, \beta)$ for the *longest common prefix* between the two strings α and β, and $\mathtt{lca}(u, v)$ for the *lowest common ancestor* between two nodes u and v in a tree.

The suffix tree of $S[1, n]$, denoted hereafter by $\mathcal{T}_S$, is a tree that stores all suffixes of $S\#$ in a compact form, where $\# \notin \Sigma$ is a special character (see Fig. 1). $\mathcal{T}_S$ consists of n leaves, numbered from 1 to n, and any root-to-leaf path spells out a suffix of $S\#$. The endmarker # guarantees that no suffix is the prefix of another suffix in $S\#$. Each internal node has at least two children and each edge is labeled with a non empty substring of S. No two edges out of a node can begin with the same character, and sibling edges are ordered lexicographically according to that character. Edge labels are encoded with pairs of integers – say $S[x, y]$ is represented by the pair $\langle x, y\rangle$. As a result, all $\Theta(n^2)$ substrings of S can be represented in $O(n)$ optimal space by $\mathcal{T}_S$'s structure and edge encoding. Furthermore, the rightward scan of the suffix tree leaves gives the ordered set of S's suffixes, also known as the *suffix array* of S [12]. Notice that the case of a large string collection $\Delta = \{S^1, S^2, \ldots, S^k\}$ *reduces to* the case of one long string $S = S^1 \#_1 S^2 \#_2 \cdots S^k \#_k$, where $\#_i \notin \Sigma$ are special symbols.

Numerous algorithms are known that build the suffix tree optimally in the RAM model (see [3] and references therein). However, most of them exhibit a marked absence of *locality of references* and thus elicit many I/Os when the size of the indexed string is too large to be fit into the internal memory of the computer. This is a serious problem because the slow performance of these algorithms can prevent the suffix tree being used even in medium-scale applications. This encyclopedia's entry surveys algorithmic solutions that deal efficiently with the *construction of suffix trees over large string collections* by executing an optimal number of I/Os. Since it is assumed that the edges leaving a node in $\mathcal{T}_S$ are lexicographically sorted, sorting is an obvious *lower bound* for building suffix trees (consider the suffix tree of a permutation!). The presented algorithms



Suffix Tree Construction in Hierarchical Memory, Figure 1
The suffix tree of S = ACACACCG on the left, and its compact edge-encoding on the right. The endmarker # is not shown. Node v spells out the string ACAC. Each internal node stores the length of its associated string, and each leaf stores the starting position of its corresponding suffix

DIVIDE-AND-CONQUER ALGORITHM
(1) Construct the string $S'[j]$ = rank of $\langle S[2j], S[2j+1]\rangle$, and recursively compute $\mathcal{T}_{S'}$.
(2) Derive from $\mathcal{T}_{S'}$ the compacted trie $\mathcal{T}_o$ of all suffixes of S beginning at odd positions.
(3) Derive from $\mathcal{T}_o$ the compacted trie $\mathcal{T}_e$ of all suffixes of S beginning at even positions.
(4) Merge $\mathcal{T}_o$ and $\mathcal{T}_e$ into the whole suffix tree $\mathcal{T}_S$, as follows:
  (4.1) Overmerge $\mathcal{T}_o$ and $\mathcal{T}_e$ into the tree $\mathcal{T}_M$.
  (4.2) Partially unmerge $\mathcal{T}_M$ to get $\mathcal{T}_S$.

Suffix Tree Construction in Hierarchical Memory, Figure 2
The algorithm that builds the suffix tree directly

have sorting as their bottleneck, thus establishing that *the complexity of sorting and suffix tree construction match*.

Key Results

Designing a disk-efficient approach to suffix-tree construction has found efficient solutions only in the last few years [4]. The present section surveys two theoretical approaches which achieve the best (optimal!) I/O-bounds in the worst case, the next section will discuss some practical solutions.

The first algorithm is based on a *Divide-and-Conquer* approach that allows us to reduce the construction process to external-memory sorting and few low-I/O primitives. It builds the suffix tree $\mathcal{T}_S$ by executing four (macro)steps, detailed in Fig. 2. It is not difficult to implement the first three steps in $\mathtt{Sort}(n) = O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os [16]. The last (merging) step is the most difficult one and its I/O-complexity bounds the cost of the overall approach. [3] proposes an elegant merge for $\mathcal{T}_o$ and $\mathcal{T}_e$: substep (4.1) temporarily relaxes the requirement of getting $\mathcal{T}_S$ in one shot, and thus it blindly (over)merges the paths of $\mathcal{T}_o$ and $\mathcal{T}_e$ by comparing edges only via their first characters; then substep (4.2) re-fixes $\mathcal{T}_M$ by detecting and undoing in an I/O-efficient manner the (over)merged paths. Note that the time and I/O-complexity of this algorithm follow a nice recursive relation: $T(n) = T(n/2) + O(\mathtt{Sort}(n))$.

Theorem 1 (Farach-Colton et al. 1999) *Given an arbitrary string $S[1, n]$, its suffix tree can be constructed in $O(\mathtt{Sort}(n))$ I/Os, $O(n \log n)$ time and using $O(n/B)$ disk pages.*

The second algorithm is deceptively simple, elegant and I/O-optimal, and applies successfully to the construction of other indexing data structures, like the String B-tree [5]. The key idea is to derive $\mathcal{T}_S$ from the suffix array $\mathcal{A}_S$ and from the lcp array, which stores the longest-common-prefix length of adjacent suffixes in $\mathcal{A}_S$. Its pseudocode is given in Fig. 3. Note that Step (1) may deploy any external-memory algorithm for suffix array construc-

SUFFIXARRAY-BASED ALGORITHM
(1) Construct the suffix array $\mathcal{A}_S$ and the array $\texttt{lcp}_S$ of the string S.
(2) Initially set $\mathcal{T}_S$ as a single edge connecting the root to a leaf pointing to suffix $\mathcal{A}_S[1]$.
(2) For $i = 2, \ldots, n$:
  (2.1) Create a new leaf ℓ_i that points to the suffix $\mathcal{A}_S[i]$.
  (2.2) Walk up from ℓ_{i-1} until a node u_i is met whose string-length x_i is $\leq \texttt{lcp}_S[i]$.
  (2.3) If $x_i = \texttt{lcp}_S[i]$, leaf ℓ_i is attached to u_i.
  (2.4) If $x_i < \texttt{lcp}_S[i]$, create node u'_i with string-length x_i, attach it to u_i and leaf ℓ_i to u'_i.

Suffix Tree Construction in Hierarchical Memory, Figure 3
The algorithm that builds the suffix tree passing through the suffix array

tion: Used here is the elegant and optimal *Skew* algorithm of [9] which takes $O(\texttt{Sort}(n))$ I/Os. Step (2) takes a total of $O(n/B)$ I/Os by using a stack that stores the nodes on the current rightmost path of $\mathcal{T}_S$ in reversed order, i. e. leaf ℓ_i is on top. Walking upward, splitting edges or attaching nodes in $\mathcal{T}_S$ boils down to popping/pushing nodes from this stack. As a result, the time and I/O-complexity of this algorithm follow the recursive relation: $T(n) = T(2n/3) + O(\texttt{Sort}(n))$.

Theorem 2 (Kärkkäinen and Sanders 2003) *Given an arbitrary string $S[1, n]$, its suffix tree can be constructed in $O(\texttt{Sort}(n))$ I/Os, $O(n \log n)$ time and using $O(n/B)$ disk pages.*

It is not evident which one of these two algorithms is better in practice. The first one exploits a recursion with parameter 1/2 but incurs a large space overhead because of the management of the tree topology; the second one is more space efficient and easier to implement, but exploits a recursion with parameter 2/3.

Applications

The reader is referred to [4] and [6] for a long list of applications of large suffix trees.

Open Problems

The recent theoretical and practical achievements mean the idea that "suffix trees are not practical except when the text size to handle is so small that the suffix tree fits in internal memory" is no longer the case [13]. Given a suffix tree, it is known now (see e. g. [4,10]) how to map it onto a disk-memory system in order to allow I/O-efficient traversals for subsequent pattern searches. A fortiori, suffix-tree storage and construction are challenging problems that need further investigation.

Space optimization is closely related to time optimization in a disk-memory system, so the design of *succinct* suffix-tree implementations is a key issue in order to scale to Gigabytes of data in reasonable time. This topic is an active area of theoretical research with many fascinating solutions (see e. g. [14]), which have not yet been fully explored in the practical setting.

It is theoretically challenging to design a suffix-tree construction algorithm that takes optimal I/Os and space proportional to the *entropy* of the indexed string. The more compressible is the string, the lighter should be the space requirement of this algorithm. Some results are known [7,10,11], but both issues of compression and I/Os have not yet been tackled jointly.

Experimental Results

The interest in building large suffix trees arose in the last few years because of the recent advances in sequencing technology, which have allowed the rapid accumulation of DNA and protein data. Some recent papers [1,2,8,15] proposed new practical algorithms that allow us to scale to Gbps/hours. Surprisingly enough, these algorithms are based on *disk-inefficient* schemes, but they properly select the insertion order of the suffixes and exploit carefully the internal memory as a buffer, so that their performance does not suffers significantly from the theoretical I/O-bottleneck.

In [8] the authors propose an *incremental* algorithm, called `PrePar`, which performs multiple passes over the string S and constructs the suffix tree for a *subrange* of suffixes at each pass. For a user-defined a parameter q, a suffix subrange is defined as the set of suffixes prefixed by the same q-long string. Suffix subranges induce subtrees of $\mathcal{T}_S$ which can thus be built independently, and evicted from internal memory as they are completed. The experiments reported in [8] successfully index 286 Mbps using 2 Gb internal memory.

In [2] the authors propose an improved version of `PrePar`, called `DynaCluster`, that deploys a *dynamic*

technique to identify suffix subranges. Unlike `Prepar`, `DynaCluster` does not scan over and over the string S, but it starts from the q-based subranges and then splits them recursively in a DFS-manner if their size is larger than a fixed threshold τ. Splitting is implemented by looking at the next q characters of the suffixes in the subrange. This clustering and lazy-DFS visit of $\mathcal{T}_S$ significantly reduce the number of I/Os incurred by the frequent edge-splitting operations that occur during the suffix tree construction process; and allow it to cope efficiently with skew data. As a result, `DynaCluster` constructs suffix trees for 200Mbps with only 16 Mb internal memory.

More recently, [15] improved the space requirement and the buffering efficiency, thus being able to construct a suffix tree of 3 Gbps in 30 hours; whereas [1] improved the I/O behavior of RAM-algorithms for online suffix-tree construction, by devising a novel low-overhead buffering policy.

Cross References

- ► Cache-Oblivious Sorting
- ► String Sorting
- ► Suffix Array Construction
- ► Suffix Tree Construction in RAM
- ► Text Indexing

Recommended Reading

1. Bedathur, S.J., Haritsa, J.R.: Engineering a fast online persistent suffix tree construction., In: Proc. 20th International Conference on Data Engineering, pp. 720–731, Boston, USA (2004)
2. Cheung, C., Yu, J., Lu, H.: Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans. Knowl. Data Eng. **17**, 90–105 (2005)
3. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47** 987–1011 (2000)
4. Ferragina, P.: Handbook of Computational Molecular Biology. In: Computer and Information Science Series, ch. 35 on "String search in external memory: algorithms and data structures". Chapman & Hall/CRC, Florida (2005)
5. Ferragina, P., Grossi, R.: The string B-tree: A new data structure for string search in external memory and its applications. J. ACM **46**, 236–280 (1999)
6. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
7. Hon, W., Sadakane, K., Sung, W.: Breaking a time-and-space barrier in constructing full-text indices. In: IEEE Symposium on Foundations of Computer Science (FOCS), 2003, pp. 251–260
8. Hunt, E., Atkinson, M., Irving, R.: Database indexing for large DNA and protein sequence collections. Int. J. Very Large Data Bases **11**, 256–271 (2002)
9. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**, 918–936 (2006)
10. Ko, P., Aluru, S.: Optimal self-adjusting trees for dynamic string data in secondary storage. In: Symposium on String Processing and Information Retrieval (SPIRE). LNCS, vol. 4726, pp. 184-194. Springer, Berlin (2007)
11. Mäkinen, V., Navarro, G.: Dynamic Entropy-Compressed Sequences and Full-Text Indexes. In: Proc. 17th Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 4009, pp. 307–318. Springer, Berlin (2006)
12. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**, 935–948 (1993)
13. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. J. Discret. Algorithms **1**, 21–49 (2000)
14. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Comput. Surv. **39**(1) (2007)
15. Tata, S., Hankins, R.A., Patel, J.M.: Practical suffix tree construction. In: Proc. 13th International Conference on Very Large Data Bases (VLDB), pp. 36–47, Toronto, Canada (2004)
16. Vitter, J.: External memory algorithms and data structures: Dealing with MASSIVE DATA. ACM Comput. Surv. **33**, 209–271 (2002)

Suffix Tree Construction in RAM

1997; Farach-Colton

Jens Stoye
Department of Technology,
University of Bielefeld, Bielefeld, Germany

Keywords and Synonyms

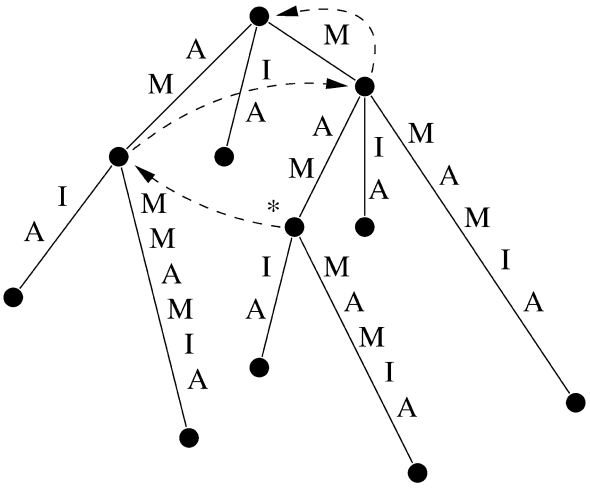
Full-text index construction

Problem Definition

The suffix tree is perhaps the best-known and most-studied data structure for string indexing with applications in many fields of sequence analysis. After its invention in the early 1970s, several approaches for the efficient construction of the suffix tree of a string have been developed for various models of computation. The most prominent of those that construct the suffix tree in main memory are summarized in this entry.

Notations

Given an alphabet Σ, a *trie* over Σ is a rooted tree whose edges are labeled with strings over Σ such that no two labels of edges leaving the same vertex start with the same symbol. A trie is *compacted* if all its internal vertices, except possibly the root, are branching. Given a finite string $S \in \Sigma^n$, the *suffix tree* of S, $T(S)$, is the compacted trie over Σ such that the concatenations of the edge labels along the paths from the root to the leaves are the suffixes of S. An example is given in Fig. 1.



Suffix Tree Construction in RAM, Figure 1
The suffix tree for the string *S* = *MAMMAMIA*. Dashed arrows denote suffix links that are employed by all efficient suffix tree construction algorithms

The concatenation of the edge labels from the root to a vertex v of $T(S)$ is called the *path-label* of v, $P(v)$. For example, the path-label of the vertex indicated by the asterisk in Fig. 1 is $P(*) = MAM$.

Constraints

The time complexity of constructing the suffix tree of a string S of length n depends on the size of the underlying alphabet Σ. It may be constant, it may be the alphabet of integers $\Sigma = \{1, 2, \ldots, n\}$, or it may be an arbitrary finite set whose elements can be compared in constant time. Note that the latter case reduces to the previous one if one maps the symbols of the alphabet to the set $\{1, \ldots, n\}$, though at the additional cost of sorting Σ.

Problem 1 (suffix tree construction)
INPUT: *A finite string S of length n over an alphabet* Σ.
OUTPUT: *The suffix tree T(S).*

If one assumes that the outgoing edges at each vertex are lexicographically sorted, which is usually the case, the suffix tree allows to retrieve the sorted order of S's characters in linear time. Therefore, suffix tree construction inherits the lower bounds from the problem complexity of sorting: $\Omega(n \log n)$ in the general alphabet case, and $\Omega(n)$ for integer alphabets.

Key Results

Theorem 1 *The suffix tree of a string of length n requires* $\Theta(n \log n)$ *bits of space.*

This is easy to see since the number of leaves of $T(S)$ is at most n, and so is the number of internal vertices that, by definition, are all branching, as well as the number of edges. In order to see that each edge label can be stored in $O(\log n)$ bits of space, note that an edge label is always a substring of S. Hence it can be represented by a pair (ℓ, r) consisting of *left pointer* ℓ and *right pointer* r, if the label is $S[\ell, r]$.

Note that this space bound is not optimal since there are $|\Sigma|^n$ different strings and hence suffix trees, while $n \log n$ bits would allow to represent $n!$ different entities.

Theorem 2 *Suffix trees can be constructed in optimal time, in particular:*

1. *For constant-size alphabet, the suffix tree T(S) of a string S of length n can be constructed in O(n) time* [11,12,13]. *For general alphabet, these algorithms require* $O(n \log n)$ *time.*
2. *For integer alphabet, the suffix tree of S can be constructed in O(n) time* [4,9].

Generally, there is a natural strategy to construct a suffix tree: Iteratively all suffixes are inserted into an initially empty structure. Such a strategy will immediately lead to a linear-time construction algorithm if each suffix can be inserted in constant time. Finding the correct position where to insert a suffix, however, is the main difficulty of suffix tree construction.

The first solution for this problem was given by Weiner in his seminal 1973 paper [13]. His algorithm inserts the suffixes from shortest to longest, and the insertion point is found in amortized constant time for constant-size alphabet, using rather a complicated amount of additional data structures. A simplified version of the algorithm was presented by Chen and Seiferas [3]. They give a cleaner presentation of the three types of links that are required in order to find the insertion points of suffixes efficiently, and their complexity proof is easier to follow. Since the suffix tree is constructed while reading the text from right to left, these two algorithms are sometimes called *anti-online* constructions.

A different algorithm was given 1976 by McCreight [11]. In this algorithm the suffixes are inserted into the growing tree from longest to shortest. This simplifies the update procedure, and the additional data structure is limited to just one type of link: an internal vertex v with path label $P(v) = aw$ for some symbol $a \in \Sigma$ and string $w \in \Sigma^*$ has a *suffix link* to the vertex u with path label $P(u) = w$. In Fig. 1, suffix links are shown as dashed arrows. They often connect vertices above the insertion points of consecutively inserted suffixes, like the vertex with path-label "M" and the root, when inserting suffixes

"MAMIA" and "AMIA" in the example of Fig. 1. This property allows to reach the next insertion point without having to search for it from the root of the tree, thus ensuring amortized constant time per suffix insertion. Note that since McCreight's algorithm treats the suffixes from longest to shortest and the intermediate structures are not suffix trees, the algorithm is not an online algorithm.

Another linear-time algorithm for constant size alphabet is the online construction by Ukkonen [12]. It reads the text from left to right and updates the suffix tree in amortized constant time per added symbol. Again, the algorithm uses suffix links in order to quickly find the insertion points for the suffixes to be inserted. Moreover, since during a single update the edge labels of all leaf-edges need to be extended by the new symbol, it requires a trick to extend all these labels in constant time: all the right pointers of the leaf edges refer to the same *end of string* value, which is just incremented.

An even stronger concept than online construction is *real-time* construction, where the worst-case (instead of amortized) time per symbol is considered. Amir et al. [1] present for general alphabet a suffix tree construction algorithm that requires $O(\log n)$ worst-case update time per every single input symbol when the text is read from right to left, and thus requires overall $O(n \log n)$ time, like the other algorithms for general alphabet mentioned so far. They achieve this goal using a binary search tree on the suffixes of the text, enhanced by additional pointers representing the lexicographic and the textual order of the suffixes, called *Balanced Indexing Structure*. This tree can be constructed in $O(\log n)$ worst-case time per added symbol and allows to maintain the suffix tree in the same time bound.

The first linear-time suffix tree construction algorithm for integer alphabets was given by Farach–Colton [4]. It uses the so-called *odd-even technique* that proceeds in three steps:

1. Recursively compute the compacted trie of all suffixes of S beginning at odd positions, called the *odd tree* T_o.
2. From T_o compute the *even tree* T_e, the compacted trie of the suffixes beginning at even positions in S.
3. Merge T_o and T_e into the whole suffix tree $T(S)$.

The basic idea of the first step is to encode pairs of characters as single characters. Since at most $n/2$ different such characters can occur, these can be radix-sorted and range-reduced to an alphabet of size $n/2$. Thus, the string S of length n over the integer alphabet $\Sigma = \{1, \ldots, n\}$ is translated in $O(n)$ time into a string S' of length $n/2$ over the integer alphabet $\Sigma' = \{1, \ldots, n/2\}$. Applying the algorithm recursively to this string yields the suffix tree of S'. After translating the edge labels from substrings of S' back to substrings of S, some vertices may exist with outgoing edges whose labels start with the same symbol, because two distinct symbols from Σ' may be pairs with the same first symbol from Σ. In such cases, by local modifications of edge labels or adding additional vertices the trie property can be regained and the desired tree T_o is obtained.

In the second step, the odd tree T_o from the first step is used to generate the lexicographically sorted list (*lex-ordering* for short) of the suffixes starting at odd positions. Radix-sorting these with the characters at the preceding even positions as keys yields a lex-ordering of the even suffixes in linear time. Together with the longest common prefixes of consecutive positions that can be computed in linear time from T_o using constant-time lowest common ancestor queries and the identity

$$lcp(l_{2i}, l_{2j}) = \begin{cases} lcp(l_{2i+1}, l_{2j+1}) + 1 & \text{if } S[2i] = S[2j] \\ 0 & \text{otherwise} \end{cases}$$

this ordering allows to reconstruct the even tree T_e in linear time.

In the third step, the two tries T_o and T_e are merged into the suffix tree $T(S)$. Conceptually, this is a straightforward procedure: the two tries are traversed in parallel, and every part that is present in one or both of the two trees, is inserted in the common structure. However, this procedure is simple only if edges are traversed character by character such that common and differing parts can be observed directly. Such a traversal would, however, require $O(n^2)$ time in the worst case, impeding the desired overall linear running time. Therefore, Farach-Colton suggests to use an oracle that tells, for an edge of T_o and an edge of T_e the length of their common prefix. However, the suggested oracle may overestimate this length, and that is why sometimes the tree generated must be corrected, called *unmerging*. The full details of the oracle and the unmerging procedure can be found in [4].

Overall, if $T(n)$ is the time it takes to build the suffix tree of a string $S \in \{1, \ldots, n\}^n$, the first step takes $T(n/2) + O(n)$ time and the second and third step take $O(n)$ time, thus the whole procedure takes $O(n)$ overall time on the RAM model.

Another linear-time construction of suffix trees for integer alphabets can be achieved via linear-time construction of suffix arrays together with longest common prefix tabulation, as described by Kärkkäinen and Sanders in [9].

In some applications the so-called *generalized* suffix tree of several strings is used, a dictionary obtained by constructing the suffix tree of the concatenation of the contained strings. An important question that arises in this context is that of dynamically updating the tree upon insertion and deletion of strings from the dictionary. More

specifically, since edge-labels are stored as pairs of pointers into the original string, when deleting a string from the dictionary the corresponding pointers may become invalid and need to be updated. An algorithm to solve this problem in amortized linear time was given by Fiala and Greene [6], a linear worst-case (and hence real-time) algorithm was given by Ferragina et al. [5].

Applications

The suffix tree supports many applications, most of them in optimal time and space, including exact string matching, set matching, longest common substring of two or more sequences, all-pairs suffix-prefix matching, repeat finding, and text compression. These and several other applications, many of them from bioinformatics, are given in [2] and [8].

Open Problems

Some theoretical questions regarding the expected size and branching structure of suffix trees under more complicated than i. i. d. sequence models are still open. Currently most of the research has moved towards more space-efficient data structures like suffix arrays and compressed string indices.

Experimental Results

Suffix trees are infamous for their high memory requirements. The practical space consumption is between 9 and 11 times the size of the string to be indexed, even in the most space-efficient implementations known [7,10]. Moreover, [7] also shows that suboptimal algorithms like the very simple quadratic-time *write-only top-down* (WOTD) algorithm can outperform optimal algorithms on many real-world instances in practice, if carefully engineered.

URL to Code

Several sequence analysis libraries contain code for suffix tree construction. For example, Strmat (http://www.cs.ucdavis.edu/~gusfield/strmat.html) by Gusfield et al. contains implementations of Weiner's and Ukkonen's algorithm. An implementation of the WOTD algorithm by Kurtz can be found at (http://bibiserv.techfak.uni-bielefeld.de/wotd).

Cross References

- ▶ Compressed Text Indexing
- ▶ String Sorting
- ▶ Suffix Array Construction
- ▶ Suffix Tree Construction in Hierarchical Memory
- ▶ Text Indexing

Recommended Reading

1. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards real-time suffix tree construction. In: Proceedings of the 12th International Symposium on String Processing and Information Retrieval, SPIRE 2005. LNCS, vol. 3772, pp. 67–78. Springer, Berlin (2005)
2. Apostolico, A.: The myriad virtues of subword trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO ASI Series, vol. F12, pp. 85–96. Springer, Berlin (1985)
3. Chen, M.T., Seiferas, J.: Efficient and elegant subword tree construction. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. Springer, New York (1985)
4. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. 38th Annu. Symp. Found. Comput. Sci., FOCS 1997, pp. 137–143. IEEE Press, New York (1997)
5. Ferragina, P., Grossi, R., Montangero, M.: A note on updating suffix tree labels. Theor. Comput. Sci. **201**, 249–262 (1998)
6. Fiala, E.R., Greene, D.H.: Data compression with finite windows. Commun. ACM **32**, 490–505 (1989)
7. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Softw. Pract. Exp. **33**, 1035–1049 (2003)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
9. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proceedings of the 30th International Colloquium on Automata, Languages, and Programming, ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Berlin (2003)
10. Kurtz, S.: Reducing the space requirements of suffix trees. Softw. Pract. Exp. **29**, 1149–1171 (1999)
11. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**, 262–272 (1976)
12. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**, 249–260 (1995)
13. Weiner, P.: Linear pattern matching algorithms. In: Proc. of the 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11. IEEE Press, New York (1973)

Support Vector Machines

1992; Boser, Guyon, Vapnik

Nello Cristianini[1], Elisa Ricci[2]

[1] Department of Engineering Mathematics, and Computer Science, University of Bristol, Bristol, UK

[2] Department of Engineering Mathematics, University of Perugia, Perugia, Italy

Problem Definition

In 1992 Vapnik and coworkers [1] proposed a supervised algorithm for classification that has since evolved into what are now known as Support Vector Machines

(SVMs) [2]: a class of algorithms for classification, regression and other applications that represent the current state of the art in the field. Among the key innovations of this method were the explicit use of convex optimization, statistical learning theory, and kernel functions.

Classification

Given a *training set* $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\}$ of data points $\mathbf{x}_i$ from $X \subseteq \mathbb{R}^n$ with corresponding labels y_i from $Y = \{-1, +1\}$, generated from an unknown distribution, the task of classification is to learn a function $g : X \to Y$ that correctly classifies new examples $(\mathbf{x}, y)$ (i. e. such that $g(\mathbf{x}) = y$) generated from the same underlying distribution as the training data.

A good classifier should guarantee the best possible generalization performance (e. g. the smallest error on unseen examples). Statistical learning theory [3], from which SVMs originated, provides a link between the expected generalization error for a given training set and a property of the classifier known as its capacity. The SV algorithm effectively regulates the capacity by considering the function corresponding to the hyperplane that separates, according to the labels, the given training data and it is maximally distant from them (*maximal margin hyperplane*). When no linear separation is possible a non-linear mapping into a higher dimensional *feature space* is realized. The hyperplane found in the feature space corresponds to a non-linear decision boundary in the input space.

Let $\phi : I \subseteq \mathbb{R}^n \to F \subseteq \mathbb{R}^n$ a mapping from the input space I to the feature space F (Fig. 1a). In the learning phase, the algorithm finds a hyperplane defined by the equation $\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle = b$ such that the *margin*

$$\gamma = \min_{1\le i\le \ell} y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) = \min_{1\le i\le \ell} y_i g(\mathbf{x}_i) \quad (1)$$

is maximized, where $\langle , \rangle$ denotes the inner product, $\mathbf{w}$ is a ℓ dimensional vector of weights, b is a threshold.

The quantity $(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b)/\|\mathbf{w}\|$ is the distance of the sample $\mathbf{x}_i$ from the hyperplane. When multiplied by the label y_i it gives a positive value for correct classification and a negative value for an uncorrect one. Given a new data point $\mathbf{x}$ a label is assigned evaluating the decision function:

$$g(\mathbf{x}) = sign(\langle \mathbf{w}, \phi(\mathbf{x})\rangle - b) \quad (2)$$

Maximizing the Margin

For linearly separable classes, there exists a hyperplane $(\mathbf{w}, b)$ such that:

$$y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) \ge \gamma \quad i = 1, \ldots, \ell \quad (3)$$

Imposing $\|\mathbf{w}\|^2 = 1$, the choice of the hyperplane such that the margin is maximized is equivalent to the following optimization problem:

$$\begin{aligned} &\max_{\mathbf{w},b,\gamma} \gamma \\ &\text{subject to } y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) \ge \gamma \quad i = 1, \ldots, \ell \\ &\qquad \text{and } \|\mathbf{w}\|^2 = 1. \end{aligned} \quad (4)$$

An efficient solution can be found in the dual space by introducing the Lagrange multipliers α_i, $i = 1, \ldots \ell$. The problem (4) can be recast in the following dual form:

$$\begin{aligned} &\max_{\boldsymbol{\alpha}} \sum_{i=1}^{\ell} \alpha_i - \sum_{i=1}^{\ell}\sum_{j=1}^{\ell} \alpha_i \alpha_j y_i y_j \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle \\ &\text{subject to } \sum_{i=1}^{\ell} \alpha_i y_i = 0, \quad \alpha_i \ge 0 \end{aligned} \quad (5)$$

This formulation shows how the problem reduces to a *convex* (quadratic) optimization task. A key property of solutions $\boldsymbol{\alpha}^*$ of this kind of problems is that they must satisfy the Karush–Kuhn–Tucker (KKT) conditions, that ensure that only a subset of training examples needs to be associated to a non-zero α_i. This property is called *sparseness* of the SVM solution, and is crucial in practical applications.

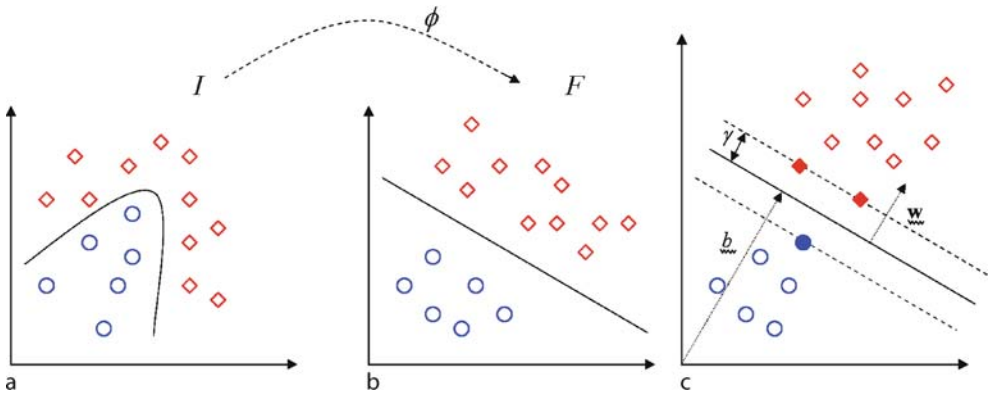
In the solution $\boldsymbol{\alpha}^*$, often only a subset of training examples is associated to non-zero α_i. These are called *support vectors* and correspond to the points that lie closest to the separating hyperplane (Fig. 1b). For the maximal margin hyperplane the weights vector $\mathbf{w}^*$ is given by linear function of the training points:

$$\mathbf{w}^* = \sum_{i=1}^{\ell} \alpha_i^* y_i \phi(\mathbf{x}_i) \quad (6)$$

Then the decision function (2) can equivalently be expressed as:

$$g(\mathbf{x}) = sign(\sum_{i=1}^{\ell} \alpha_i^* y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x})\rangle - b) \quad (7)$$

For a support vector $\mathbf{x}_i$, it is $\langle \mathbf{w}^*, \phi(\mathbf{x}_i)\rangle - b = y_i$ from which the optimum bias b^* can be computed. However, it is better to average the values obtained by considering all the support vectors [2]. Both the quadratic programming (QP) problem (5) and the decision function (7) depend only on the dot product between data points. The matrix of dot products with elements $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle$ is called the *kernel matrix*. In the case of linear separation $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j\rangle$,



Support Vector Machines, Figure 1
a The feature map simplifies the classification task. b A maximal margin hyperplane with its support vectors highlighted

but in general, one can use functions that provide nonlinear decision boundaries. Widely used kernels are the polynomial $K(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + 1)^d$ or the Gaussian $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma^2}}$ where d and σ are user-defined parameters.

Key Results

In the framework of learning from examples, SVMs have shown several advantages compared to traditional neural network models (which represented the state of the art in many classification tasks up to 1992). The statistical motivation for seeking the maximal margin solution is to minimize an upper bound on the test error that is independent of the number of dimensions and inversely proportional to the separation margin (and the sample size). This directly suggests embedding of the data in a high-dimensional space where a large separation margin can be achieved; that this can be done efficiently with kernels, and in a convex fashion, are two crucial computational considerations. The sparseness of the solution, implied by the KKT conditions, adds to the efficiency of the result.

The initial formulation of SVMs by Vapnik and coworkers [1] has been extended by many other researchers. Here we summarize some key contributions.

Soft Margin

In the presence of noise the SV algorithm can be subjected to overfitting. In this case one needs to tolerate some training errors in order to obtain a better generalization power. This has led to the development of the *soft margin* classifiers [4]. Introducing the slack variables $\xi_i \geq 0$, optimal class separation can be obtained by:

$$\begin{aligned} &\min_{\mathbf{w},b,\gamma,\boldsymbol{\xi}} \; -\gamma + C\sum_{i=1}^{\ell}\xi_i \\ &\text{subject to } y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) \geq \gamma - \xi_i,\ \xi_i \geq 0 \\ &\qquad i = 1,\ldots,\ell \ \text{ and } \ \|\mathbf{w}\|^2 = 1. \end{aligned} \tag{8}$$

The constant C is user-defined and controls the trade-off between the maximization of the margin and the number of classification errors. The dual formulation is the same as (5) with the only difference in the bound constraints ($0 \leq \alpha_i \leq C, \quad i = 1,\ldots,\ell$). The choice of soft margin parameter is one of the two main design choices (together with the kernel function) in applications. It is an elegant result [5] that the entire set of solutions for all possible values of C can be found with essentially the same computational cost over finding a single solution: this set is often called the *regularization path*.

Regression

A SV algorithm for regression, called support vector regression (SVR), was proposed in 1996 [6]. A linear algorithm is used in the kernel-induced feature space to construct a function such that the training points are inside a tube of given radius ε. As for classification the regression function only depends on a subset of the training data.

Speeding up the Quadratic Program

Since the emergence of SVMs, many researchers have developed techniques to effectively solve the problem (5): a quite time-consuming task, especially for large training sets. Most methods decompose large-scale problems into a series of smaller ones. The most widely used method is

that of Platt [7] and it is known as Sequential Minimal Optimization.

Kernel Methods

In SVMs, both the learning problem and the decision function can be formulated only in terms of dot products between data points. Other popular methods (i. e. Principal Component Analysis, Canonical Correlation Analysis, Fisher Discriminant) have the same property. This fact has led to a huge number of algorithms that effectively use kernels to deal with non-linear functions keeping the same complexity of the linear case. They are referred to as *kernel methods* [8,9].

Choosing the Kernel

The main design choice when using SVMs is the selection of an appropriate kernel function, a problem of model selection that roughly relates to the choice of a topology for a neural network. It is a non-trivial result [10] that also this key task can be translated into a convex optimization problem (a semi-definite program) under general conditions. A kernel can be optimally selected from a kernel space resulting from all linear combinations of a basic set of kernels.

Kernels for General Data

Kernels are not just useful tools to allow us to deploy methods of linear statistics in a non-linear setting. They also allow us to apply them to non-vectorial data: kernels have been designed to operate on sequences, graphs, text, images, and many other kinds of data [8].

Applications

Since their emergence, SVMs have been widely used in a huge variety of applications. To give some examples good results have been obtained in text categorization, handwritten character recognition, and biosequence analysis.

Text Categorization

Automatic text categorization is where text documents are classified into a fixed number of predefined categories based on their content. In the works performed by Joachims [11] and by Dumais et al. [12], documents are represented by vectors with the so-called bag-of-words approach used in the information retrieval field. The distance between two documents is given by the inner product between the corresponding vectors. Experiments on the collection of Reuters news stories showed good results of SVMs compared to other classification methods.

Hand-Written Character Recognition

This is the first real-world task on which SVMs were tested. In particular two publicly available data sets (USPS and NIST) have been considered since they are usually used for benchmarking classifiers. A lot of experiments, mainly summarized in [13], were performed which showed that SVMs can perform as well as other complex systems without incorporating any detailed prior knowledge about the task.

Bioinformatics

SVMs have been widely used also in bioinformatics. For example, Jaakkola and Haussler [14] applied SVMs to the problem of protein homology detection, i. e. the task of relating new protein sequences to proteins whose properties are already known. Brown et al. [15] describe a successful use of SVMs for the automatic categorization of gene expression data from DNA microarrays.

URL to Code

Many free software implementations of SVMs are available at the website

- www.support-vector.net/software.html

Two in particular deserve a special mention for their efficiency:

- *SVMlight*: Joachims T. Making large-scale SVM learning practical. In: Schölkopf B, Burges CJC, and Smola AJ (eds) Advances in Kernel Methods Support Vector Learning, MIT Press, 1999. Software available at http://svmlight.joachims.org
- *LIBSVM*: Chang CC, and Lin CJ, LIBSVM: a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

Cross References

► PAC Learning
► Perceptron Algorithm

Recommended Reading

1. Boser, B., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, Pittsburgh (1992)
2. Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and other kernel-based learning methods. Cambridge University Press, Cambrigde, Book website: www.support-vector.net (2000)

3. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, New York (1995)
4. Cortes, C., Vapnik, V.: Support-vector network. Mach. Learn. **20**, 273–297 (1995)
5. Hastie, T., Rosset, S., Tibshirani, R., Zhu, J.: The entire regularization path for the support vector machine. J. Mach. Learn. Res. **5**, 1391–1415 (2004)
6. Drucker, H., Burges, C.J.C., Kaufman, L., Smola, A., Vapnik, V.: Support Vector Regression Machines. Adv. Neural. Inf. Process. Syst. (NIPS) **9**, 155–161 MIT Press (1997)
7. Platt, J.: Fast training of support vector machines using sequential minimal optimization. In: Schölkopf, B., Burges, C.J.C., Smola, A.J. (eds.) Advances in Kernel Methods Support Vector Learning. pp 185–208. MIT Press, Cambridge (1999)
8. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge. Book website: www.kernel-methods.net (2004)
9. Scholkopf, B., Smola, A.J.: Learning with Kernels. MIT Press, Cambridge (2002)
10. Lanckriet, G.R.G., Cristianini, N., Bartlett, P., El Ghaoui, L., Jordan, M.I.: Learning the Kernel Matrix with Semidefinite Programming. J. Mach. Learn. Res. **5**, 27–72 (2004)
11. Joachims, T.: Text categorization with support vector machines. In: Proceedings of European Conference on Machine Learning (ECML) Chemnitz (1998)
12. Dumais, S., Platt, J., Heckerman, D., Sahami, M.: Inductive learning algorithms and representations for text categorization. In: 7th International Conference on Information and Knowledge Management (1998)
13. LeCun, Y., Jackel, L.D., Bottou, L., Brunot, A., Cortes, C., Denker, J.S., Drucker, H., Guyon, I., Muller, U.A., Sackinger, E., Simard, P., Vapnik, V.: Comparison of learning algorithms for handwritten digit recognition. In: Fogelman-Soulie F., Gallinari P. (eds.), Proceedings International Conference on Artificial Neural Networks (ICANN) **2**, 5360. EC2 (1995)
14. Jaakkola, T.S., Haussler, D.: Probabilistic kernel regression models. In: Proceedings of the 1999 Conference on AI and Statistics Fort Lauderdale (1999)
15. Brown, M., Grundy, W., Lin, D., Cristianini, N., Sugnet, C., Furey, T., Ares Jr., M., Haussler, D.: Knowledge-based analysis of mircoarray gene expression data using support vector machines. In: Proceedings of the National Academy of Sciences **97**(1), 262–267 (2000)

Symbolic Model Checking

1990; Burch, Clarke, McMillan, Dill

Amit Prakash[1], Adnan Aziz[2]
[1] Microsoft, MSN, Redmond, WA, USA
[2] Department of Electrical and Computer Engineering, University of Texas, Austin, TX, USA

Keywords and Synonyms

Formal hardware verification

Problem Definition

Design verification is the process of taking a design and checking that it works correctly. More specifically, every design verification paradigm has three components [6]—(1.) a language for specifying the design in an unambiguous way, (2.) a language for specifying properties that are to be checked of the design, and (3.) a checking procedure, which determines whether the properties hold of the design.

The verification problem is very general: it arises in low-level designs, e. g., checking that a combinational circuit correctly implements arithmetic, as well as high-level designs, e. g., checking that a library written in high-level language correctly implements an abstract data type.

Hardware Verification

The verification of hardware designs is particularly challenging. Verification is difficult in part because the large number of concurrent operations, make it very difficult to conceive of and construct all possible corner-cases, e. g., one unit initiating a transaction at the same cycle as another receiving an exception. In addition, software models used for simulation run orders of several magnitude slower than the final chip operates at. Faulty hardware is usually impossible to correct after fabrication, which means that the cost of a defect is very high, since it takes several months to go through the process of designing and fabricating new hardware. Wile et al. [15] provide a comprehensive account of hardware verification.

State Explosion

Since the number of state holding elements in digital hardware is bounded, the number of possible states that the design can be in is infinite, so complete automated verification is, in principle, possible. However, the number of states that a hardware design can reach from the initial state can be exponential in the size of the design; this phenomenon is referred to as "state explosion." In particular, algorithms for verifying hardware that explicitly record visited states, e. g., in a hash table, have very high time complexity, making them infeasible for all but the smallest designs. The problem of complete hardware verification is known to be PSPACE-hard, which means that any approach must be based on heuristics.

Hardware Model

A hardware design is formally described using *circuits* [4,8]. A *combinational circuit* consists of *Boolean combinational elements* connected by *wires*. The Boolean

combinational elements are *gates* and *primary inputs*. Gates come in three types: *NOT*, *AND*, and *OR*. The NOT gate functions as follows: it takes a single Boolean-valued *input*, and produces a single Boolean-valued *output* which takes value 0 if the input is 1, and 1 if the input is 0. The AND gate takes two Boolean-valued inputs and produce a single output; the output is 1 if both inputs are 1, and 0 otherwise. The OR gate is similar to AND, except that its output is 1 if one or both inputs are 1. A circuit can be represented as a directed graph where the nodes represent the gates and wires represent edges in the direction of signal flow.

A circuit can be represented by a directed graph where the nodes represent the gates and primary inputs, and edges represent wires in the direction of signal flow. Circuits are required to be acyclic, that is there is no cycle of gates. The absence of cycles implies that a Boolean-assignment to the primary inputs can be propagated through the gates in topological order.

A *sequential circuit* extends the notion of circuit described above by adding *stateful elements*. Specifically, a sequential circuit includes *registers*. Each register has a single input, which is referred to as its *next-state input*.

A *valuation* on a set V is a function whose domain is V. A *state* in a sequential circuit is a Boolean-valued valuation on the set of registers. An *input* to a sequential circuit is a Boolean-valued valuation on the set of primary inputs. Given a state s and an input i, the logic gates in the circuit uniquely define a Boolean-valued valuation t to the set of register inputs—this is referred to as the next state of the circuit at state s under input i, and say s *transitions* to t on input i. It is convenient to denote such a transition by $s \xrightarrow{i} t$.

A sequential circuit can naturally be identified with a *finite state machine* (FSM), which is a graph defined over the set of all states; an edge (s, t) exists in the FSM graph if there exists an input i, state s transitions to t on input i.

Invariant Checking

An *invariant* is a set of states; informally, the term is used to refer to a set of states that are "good" in some sense. One common way to specify an invariant is to write a Boolean formula on the register variables—the states which satisfy the formula are precisely the states in the invariant.

Given states r and s, define r to be *reachable* from s if there is a sequence of inputs $\langle i_0, i_1, \ldots, i_{n-1}\rangle$ such that $s = s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \cdots s_n = t$. A fundamental problem in hardware verification is the following—given an invariant A, and a state s, does there exists a state r reachable from s which is not in A?

Key Results

Symbolic model checking (SMC) is a heuristic approach to hardware verification. It is based on the idea that rather than representing and manipulating states one-at-a-time, it is more efficient to use symbolic expressions to represent and manipulate sets of states.

A key idea in SMC is that given a set $A \subset \{0,1\}^n$, a Boolean function A can be constructed such that $f_A\colon \{0,1\}^n \mapsto \{0,1\}$ given by $f(\alpha_1, \ldots, \alpha_n) = 1$ iff $(\alpha_1, \ldots, \alpha_n) \in A$. Note that given a characteristic function f_A, A can be obtained and vice versa.

There are many ways in which a Boolean function can be represented—formulas in DNF, general Boolean formulas, combinational circuits, etc. In addition to an efficient representation for state sets, the ability to perform fast computations with sets of states is also important—for example, in order to determine if an invariant holds, it is required to compute the set of states reachable from a given state. BDDs [2] are particularly well-suited to representing Boolean functions, as they combine succinct representation with efficient manipulation; they are the data structure underlying SMC.

Image Computation

A key computation that arises in verification is determining the *image* of a set of states A in a design D—the image of A is the set of all states t for which there exists a state in A and an input i such that state s transitions to t under input i. The image of A is denoted by $\mathrm{Img}(A)$.

The *transition relation* of a design is the set of (s, i, t) triples such that s transitions to t under input i. Let the design have n registers, and m primary inputs; then the transition relation is subset of $\{0,1\}^n \times \{0,1\}^m \times \{0,1\}^n$.

Conceptually, the transition relation completely captures the dynamics of the design—given an initial state, and input sequence, the evolution of the design is completely determined by the transition relation.

Since the transition relation is a subset of $\{0,1\}^{n+m+n}$, it has a characteristic function $f_T\colon \{0,1\}^{n+m+n} \mapsto \{0,1\}$. View f_T as being defined over the variables $x_0, \ldots, x_{n-1}$, $i_0, \ldots, i_{m-1}, y_0, \ldots, y_{n-1}$. Let the set of states A be represented by the function f_A defined over variables $x_0, \ldots, x_{n-1}$. Then the following identity holds

$$\mathrm{Img}(A) = (\exists x_0 \cdot \exists x_{n-1} \exists i_0 \cdots \exists i_{m-1})(f_A \cdot f_T)\,.$$

The identity hold because $(\beta_0, \ldots, \beta_{n-1})$ satisfies the right-hand side expression exactly when there are values $\alpha_0, \ldots, \alpha_{n-1}$ and $\iota_0, \ldots, \iota_{m-1}$ such that $(\alpha_0, \ldots, \alpha_{n-1}) \in A$ and the state $(\alpha_0, \ldots, \alpha_{n-1})$ transitions to $(\beta_0, \ldots, \beta_{n-1})$ on input $(\iota_0, \ldots, \iota_{m-1})$.

Invariant Checking

The set of all states reachable from a given set A is the limit as n tends to infinity of the sequence of states $\langle R_0, R_1, \dots \rangle$ defined below:

$$R_0 = A$$
$$R_{i+1} = R_i \cup \mathrm{Img}(R_i) .$$

Since for all i, $R_i \subseteq R_{i+1}$ and the number of distinct state sets is finite, the limit is reached in some finite number of steps, i. e., for some n, it must be that $R_{n+1} = R_n$. It is straightforward to show that the limit is exactly equal to the set of states reachable from A—the basic idea is to inductively construct input sequences that lead from states in A to R_i, and to show that state t is reachable from a state in A under an input sequence of length l, then t must be in R_l.

Given BDDs F and G representing functions f and g respectively, there is an algorithm based on dynamic programming for performing conjunction, i. e., for computing the BDD for $f \cdot g$. The algorithm has polynomial complexity, specifically $O(|F| \cdot |G|)$, where $|B|$ denotes the number of nodes in the BDD B. There are similar algorithms for performing disjunction ($f + g$), and computing cofactors (f_x and $f_{x'}$). Together these yield an algorithm for the operation of existential quantification, since $(\exists x) f = f_x + f_{x'}$.

It is straightforward to build BDDs for f_A and f_T: A is typically given using a propositional formula, and the BDD for f_A can be built up using functions for conjunction, disjunction, and negation. The BDD for f_T is built using from the BDDs for the next-state nodes, over the register and primary input variables. Since the only gate types are AND, OR, and NOT, the BDD can be built using the standard BDD operators for conjunction, disjunction, and negation. Let the next state functions be $f_0, \dots, f_{n-1}$; then f_T is $(y_0 = f_0) \cdot (y_1 = f_1) \cdot \dots \cdot (y_{n-1} = f_{n-1})$, and so the BDD for f_T can be constructed using the usual BDD operators.

Since the image computation operation can be expressed in terms of f_A and F_T, and conjunction and existential quantification operations, it can be performed using BDDs. The computation of R_i involves an image operation, and a disjunction, and since BDDs are canonical, the test for fixed-point is trivial.

Applications

The primary application of the technique described above is for checking properties of hardware designs. These properties can be invariants described using propositional formulae over the register variables, in which case the approach above is directly applicable. More generally, properties can be expressed in a *temporal logic* [5], specifically through formulae which express acceptable sequences of outputs and transitions.

CTL is one common temporal logic. A CTL formula is given by the following grammar: if x is a variable corresponding to a register, then $\mathbf{x}$ is a CTL formula; otherwise, if φ and ψ are CTL formulas, then so as $(\neg\phi)$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $(\phi \rightarrow \psi)$, and $EX\phi$, $E\phi U\psi$, and $EG\phi$.

A CTL formula is interpreted as being true at a state; a formula $\mathbf{x}$ is true at a state if that register is 1 in that state. Propositional connectives are handled in the standard way, e. g., a state satisfies a formula $(\phi \wedge \psi)$ if it satisfies both φ and ψ. A state s satisfies $EX\phi$ if there exists a state t such that s transitions to, and t satisfies φ. A state s satisfies $E\phi U\psi$ if there exists a sequence of inputs $\langle i_0, \dots, i_n \rangle$ leading through state $\langle s_0 = s, s_1, s_2, \dots, s_{n+1} \rangle$ such that s_{n+1} satisfies ψ, and all states $s_i, i \leq n+1$ satisfy φ. A state s satisfies $EG\phi$ if there exists an infinite sequence of inputs $\langle i_0, i_1, \dots \rangle$ leading through state $\langle s_0 = s, s_1, s_2, \dots \rangle$ such that all states s_i satisfy φ.

CTL formulas can be checked by a straightforward extension of the technique described above for invariant checking. One approach is to compute the set of states in the design satisfying subformulas of φ, starting from the subformulas at the bottom of the parse tree for φ. A minor difference between invariant checking and this approach, is that the latter relies on *pre-image* computation; the pre-image of A is the set of all states t for which there exists an input i such that t transitions under i to a state in A.

Symbolic analysis can also be used to check the equivalence of two designs by forming a new design which operates the two initial designs in parallel, and has a single output that is set to 1 if the two initial designs differ [14]. In practice this approach is too inefficient to be useful, and techniques which rely more on identifying common substructures across designs are more successful.

The complement of the set of reachable states can be used to identify parts of the design which are redundant, and to propagate don't care conditions from the input of the design to internal nodes [12].

Many of the ideas in SMC can be applied to software verification—the basic idea is to "finitize" the problem, e. g., by considering integers to lie in a restricted range, or setting an a priori bound on the size of arrays [7].

Experimental Results

Many enhancements have been made to the basic approach described above. For example, the BDD for the entire transition relation can grow large, so *partitioned tran-*

sition relations [11] are used instead; these are based on the observation that $\exists x.(f \cdot g) = f \cdot \exists x.g$, in the special case that f is independent of x. Another optimization is the use of *don't cares*; for example when computing the image of A, the BDD for f_T can be simplified with respect to transitions originating at A' [13]. Techniques based on SAT have enjoyed great success recently. These approach case the verification problem in terms of satisfiability of a CNF formula. They tend to be used for bounded checks, i. e., determining that a given invariant holds on all input sequences of length k [1]. Approaches based on *transformation-based verification*, complement symbolic model checking by simplifying the design prior to verification. These simplifications typically remove complexity that was added for performance rather than functionality, e. g., pipeline registers.

The original paper by Clarke et al. [3] reported results on a toy example, which could be described in a few dozen lines of a high-level language. Currently, the most sophisticated model checking tool for which published results are ready is SixthSense, developed at IBM [10].

A large number of papers have been published on applying SMC to academic and industrial designs. Many report success on designs with an astronomical number of states—these results become less impressive when taking into consideration the fact that a design with n registers has 2^n states.

It is very difficult to define the complexity of a design. One measure is the number of registers in the design. Realistically, a hundred registers is at the limit of design complexity that can be handles using symbolic model checking. There are cases of designs with many more registers that have been successfully verified with symbolic model checking, but these registers are invariably part of a very regular structure, such as a memory array.

Data Sets

The SMV system described in [9] has been updated, and its latest incarnation nuSMV (http://nusmv.irst.itc.it/) include a number of examples.

The VIS (http://embedded.eecs.berkeley.edu/pubs/downloads/vis) system from UC Berkeley and UC Boulder also includes a large collection of verification problems, ranging from simple hardware circuits, to complex multiprocessor cache systems.

The SIS (http://embedded.eecs.berkeley.edu/pubs/downloads/sis/) system from UC Berkeley is used for logic synthesis. It comes with a number of sequential circuits that have been used for benchmarking symbolic reachability analysis.

Cross References

► Binary Decision Graph

Recommended Reading

1. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic Model Checking Using Sat Procedures Instead of BDDs. In: ACM Design Automation Conference. (1999)
2. Bryant, R.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Trans. Comp. **C-35**, 677–691 (1986)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Symbolic Model Checking: 10^{20} States and Beyond. Inf. Comp. **98**(2), 142–170 (1992)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.H., Stein, C.: Introduction to Algorithms. MIT Press (2001)
5. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Formal Models and Semantics, vol. B of Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier Science (1990)
6. Gupta, A.: Formal Hardware Verification Methods: A Survey. Formal Method Syst. Des. **1**, 151–238 (1993)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
8. Katz, R.: Contemporary logic design. Benjamin/Cummings Pub. Co. (1993)
9. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
10. Mony, H., Baumgartner, J., Paruthi, V., Kanzelman, R., Kuehlmann, A.: Scalable Automated Verification via Expert-System Guided Transformations. In: Formal Methods in CAD. (2004)
11. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient BDD Algorithms for FSM Synthesis and Verification. In: Proceedings of the International Workshop on Logic Synthesis, May 1995
12. Savoj, H.: Don't Cares in Multi-Level Network Optimization. Ph. D. thesis, University of California, Berkeley, Electronics Research Laboratory, College of Engineering. University of California, Berkeley, CA (1992)
13. Shiple, T.R., Hojati, R., Sangiovanni-Vincentelli, A.L., Brayton, R.K.: Heuristic Minimization of BDDs Using Don't Cares. In: ACM Design Automation Conference, San Diego, CA, June (1994)
14. Touati, H., Savoj, H., Lin, B., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Implicit State Enumeration of Finite State Machines using BDDs. In: IEEE International Conference on Computer-Aided Design, pp. 130–133, November (1990)
15. Wile, B., Goss, J., Roesner, W.: Comprehensive Functional Verification. Morgan-Kaufmann (2005)

Synchronizers, Spanners

1985; Awerbuch

Michael Elkin
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel

Keywords and Synonyms

Network synchronization; Low-stretch spanning subgraphs

Problem Definition

Consider a communication network, modeled by an n-vertex undirected unweighted graph $G = (V, E)$, for some positive integer n. Each vertex of G hosts a processor of unlimited computational power; the vertices have unique identity numbers, and they communicate via the edges of G by sending messages of size $O(\log n)$ each.

In the *synchronous* setting the communication occurs in discrete *rounds*, and a message sent in the beginning of a round R arrives at its destination before the round R ends. In the *asynchronous* setting each vertex maintains its own clock, and clocks of distinct vertices may disagree. It is assumed that each message sent (in the asynchronous detting) arrives at its destination within a certain time τ after it was sent, but the value of τ is not known to the processors.

It is generally much easier to devise algorithms that apply to the synchronous setting (henceforth, synchronous algorithms) rather than to the asynchronous one (henceforth, asynchronous algorithms). In [1] Awerbuch initiated the study of simulation techniques that translate synchronous algorithms to asynchronous ones. These simulation techniques are called *synchronizers*.

To devise the first synchronizers Awerbuch [1] constructed a certain graph partition which is of its own interest. In particular, Peleg and Schäffer noticed [8] that this graph partition induces a subgraph with certain interesting properties. They called this subgraph a *graph spanner*. Formally, for an integer positive parameter k, a k-spanner of a graph $G = (V, E)$ is a subgraph $G' = (V, H)$, $H \subseteq E$, such that for every edge $e = (v, u) \in E$, the distance between the vertices v and u in H, $\text{dist}_{G'}(v, u)$, is at most k.

Key Results

Awerbuch devised three basic synchronizers, called α, β, and γ. The synchronizer α is the simplest one; using it results in only a constant overhead in time, but in a very significant overhead in communication. Specifically, the latter overhead is linear in the number of edges of the underlying network. Unlike the synchronizer α, the synchronizer β requires a somewhat costly initialization stage. In addition, using it results in a significant time overhead (linear in the number of vertices n), but it is more communication-efficient than α. Specifically, its communication overhead is linear in n.

Finally, the synchronizer γ represents a tradeoff between the synchronizers α and β. Specifically, this synchronizer is parametrized by a positive integer parameter k. When k is small then the synchronizer behaves similarly to the synchronizer α, and when k is large it behaves similarly to the synchronizer β. A particularly important choice of k is $k = \log n$. At this point on the tradeoff curve the synchronizer γ has a logarithmic in n time overhead, and a linear in n communication overhead. The synchronizer γ has, however, a quite costly initialization stage.

The main result of [1] concerning spanners is that for every $k = 1, 2, \ldots$, and every n-vertex unweighted undirected graph $G = (V, E)$, there exists an $O(k)$-spanner with $O(n^{1+1/k})$ edges. (This result was explicated by Peleg and Schäffer [8].)

Applications

Synchronizers are extensively used for constructing asynchronous algorithms. The first applications of synchronizers are constructing the *breadth-first-search tree* and computing the *maximum flow*. These applications were presented and analyzed by Awerbuch in [1]. Later synchronizers were used for maximum matching [10], for computing shortest paths [7], and for other problems.

Graph spanners were found useful for a variety of applications in distributed computing. In particular, some constructions of synchronizers employ graph spanners [1,9]. In addition, spanners were used for routing [4], and for computing almost shortest paths in graphs [5].

Open Problems

Synchronizers with improved properties were devised by Awerbuch and Peleg [3], and Awerbuch et al. [2]. Both these synchronizers have polylogarithmic time and communication overheads. However, the synchronizers of Awerbuch and Peleg [3] require a large initialization time. (The latter is at least linear in n.) On the other hand, the synchronizers of [2] are randomized. A major open problem is to obtain *deterministic* synchronizers with polylogarithmic time and communication overheads, and sublinear in n initialization time. In addition, the degrees of the logarithm in the polylogarithmic time and communication overheads in synchronizers of [2,3] are quite large. Another important open problem is to construct synchronizers with improved parameters.

In the area of spanners, spanners that distort large distances to a significantly smaller extent than they distort small distances were constructed by Elkin and Peleg in [6]. These spanners fall short from achieving a *purely additive*

distortion. Constructing spanners with a purely additive distortion is a major open problem.

Cross References

► Sparse Graph Spanners

Recommended Reading

1. Awerbuch, B.: Complexity of network synchronization. J. ACM **4**, 804–823 (1985)
2. Awerbuch, B., Patt-Shamir, B., Peleg, D., Saks, M.E.: Adapting to asynchronous dynamic networks. In: Proc. of the 24th Annual ACM Symp. on Theory of Computing, Victoria, 4–6 May 1992, pp. 557–570
3. Awerbuch, B., Peleg, D.: Network synchronization with polylogarithmic overhead. In: Proc. 31st IEEE Symp. on Foundations of Computer Science, Sankt Louis, 22–24 Oct. 1990, pp. 514–522
4. Awerbuch, B., Peleg, D.: Routing with polynomial communication-space tradeoff. SIAM J. Discret. Math. **5**, 151–162 (1992)
5. Elkin, M.: Computing Almost Shortest Paths. In: Proc. 20th ACM Symp. on Principles of Distributed Computing, Newport, RI, USA, 26–29 Aug. 2001, pp. 53–62
6. Elkin, M., Peleg, D.: Spanner constructions for general graphs. In: Proc. of the 33th ACM Symp. on Theory of Computing, Heraklion, 6–8 Jul. 2001, pp. 173–182
7. Lakshmanan, K.B., Thulasiraman, K., Comeau, M.A.: An efficient distributed protocol for finding shortest paths in networks with negative cycles. IEEE Trans. Softw. Eng. **15**, 639–644 (1989)
8. Peleg, D., Schäffer, A.: Graph spanners. J. Graph Theory **13**, 99–116 (1989)
9. Peleg, D., Ullman, J.D.: An optimal synchronizer for the hypercube. SIAM J. Comput. **18**, 740–747 (1989)
10. Schieber, B., Moran, S.: Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: Maximum matchings. In: Proc. of 5th ACM Symp. on Principles of Distributed Computing, Calgary, 11–13 Aug. 1986, pp. 282–292