E

Edit Distance Under Block Operations

2000; Cormode, Paterson, Sahinalp, Vishkin
2000; Muthukrishnan, Sahinalp

S. Cenk Sahinalp
Lab for Computational Biology, Simon Fraser University, Burnaby, BC, USA

Keywords and Synonyms

Block edit distance

Problem Definition

Given two strings $S = s_1 s_2 \ldots s_n$ and $R = r_1 r_2 \ldots r_m$ (wlog let $n \geq m$) over an alphabet $\sigma = \{\sigma_1, \sigma_2, \ldots \sigma_\ell\}$, the *standard edit distance* between S and R, denoted $ED(S, R)$ is the minimum number of *single character edits*, specifically *insertions*, *deletions* and *replacements*, to transform S into R (equivalently R into S).

If the input strings S and R are permutations of the alphabet σ (so that $|S| = |R| = |\sigma|$) then an analogous *permutation edit distance* between S and R, denoted $PED(S, R)$ can be defined as the minimum number of single character *moves*, to transform S into R (or vice versa).

A generalization of the standard edit distance is *edit distance with moves*, which, for input strings S and R is denoted $EDM(S, R)$, and is defined as the minimum number of character edits and *substring (block) moves* to transform one of the strings into the other. A move of block $s[j, k]$ to position h transforms $S = s_1 s_2 \ldots s_n$ into $S' = s_1 \ldots s_{j-1} s_{k+1} s_{k+2} \ldots s_{h-1} s_j \ldots s_k s_h \ldots s_n$ [4].

If the input strings S and R are permutations of the alphabet σ (so that $|S| = |R| = |\sigma|$) then $EDM(S, R)$ is also called as the transposition distance and is denoted $TED(S, R)$ [1].

Perhaps the most general form of the standard edit distance that involves edit operations on blocks/substrings is the *block edit distance*, denoted $BED(S, R)$. It is defined as the minimum number of single character edits, block moves, as well as *block copies and block uncopies* to transform one of the strings into the other. Copying of a block $s[j, k]$ to position h transforms $S = s_1 s_2 \ldots s_n$ into $S' = s_1 \ldots s_j s_{j+1} \ldots s_k \ldots s_{h-1} s_j \ldots s_k s_h \ldots s_n$. A block uncopy is the inverse of a block copy: it deletes a block $s[j, k]$ provided there exists $s[j', k'] = s[j, k]$ which does not overlap with $s[j, k]$ and transforms S into $S' = s_1 \ldots s_{j-1} s_{k+1} \ldots s_n$.

Throughout this discussion all edit operations have unit cost and they may overlap; i. e. a character can be edited on multiple times.

Key Results

There are exact and approximate solutions to computing the edit distances described above with varying performance guarantees. As can be expected, the best available running times as well as the approximation factors for computing these edit distances vary considerably with the edit operations allowed.

Exact Computation of the Standard and Permutation Edit Distance

The fastest algorithms for exactly computing the standard edit distance have been available for more than 25 years.

Theorem 1 (Levenshtein [9]) *The standard edit distance ED(S, R) can be computed exactly in time $O(n \cdot m)$ via dynamic programming.*

Theorem 2 (Masek-Paterson [11]) *The standard edit distance ED(S, R) can be computed exactly in time $O(n + n \cdot m/\log^2_{|\sigma|} n)$ via the "four-Russians trick".*

Theorem 3 (Landau-Vishkin [8]) *It is possible to compute ED(S, R) in time $O(n \cdot ED(S, R))$.*

Finally, note that if S and R are permutations of the alphabet σ, $PED(S, R)$ can be computed much faster than the standard edit distance for general strings: Observe

that $PED(S, R) = n - LCS(S, R)$ where $LCS(S, R)$ represents the longest common subsequence of S and R. For permutations S, R, $LCS(S, R)$ can be computed in time $O(n \cdot \log\log n)$ [3].

Approximate Computation of the Standard Edit Distance

If some approximation can be tolerated, it is possible to considerably improve the $\tilde{O}(n \cdot m)$ time ($\tilde{O}$ notation hides polylogarithmic factors) available by the techniques above. The fastest algorithm that *approximately* computes the standard edit distance works by *embedding* strings S and R from alphabet σ into shorter strings S' and R' from a larger alphabet σ' [2]. The embedding is achieved by applying a general version of the *Locally Consistent Parsing* [13,14] to partition the strings R and S into *consistent blocks* of size c to $2c - 1$; the partitioning is consistent in the sense that identical (long) substrings are partitioned identically. Each block is then replaced with a label such that identical blocks are identically labeled. The resulting strings S' and R' preserve the edit distance between S and R approximately as stated below.

Theorem 4(Batu-Ergun-Sahinalp [2]) *$ED(S, R)$ can be computed in time $\tilde{O}(n^{1+\epsilon})$ within an approximation factor of $\min\{n^{\frac{1-\epsilon}{3}+o(1)}, (ED(S, R)/n^{\epsilon})^{\frac{1}{2}+o(1)}\}$.*

For the case of $\epsilon = 0$, the above result provides an $\tilde{O}(n)$ time algorithm for approximating $ED(S, R)$ within a factor of $\min\{n^{\frac{1}{3}+o(1)}, ED(S, R)^{\frac{1}{2}+o(1)}\}$.

Approximate Computation of Edit Distances Involving Block Edits

For all edit distance variants described above which involve blocks, there are no known polynomial time algorithms; in fact it is NP-hard to compute $TED(S, R)$ [1], $EDM(S, R)$ and $BED(S, R)$ [10]. However, in case S and R are permutations of σ, there are polynomial time algorithms that approximate transposition distance within a constant factor:

Theorem 5 (Bafna-Pevzner [1]) *$TED(S, R)$ can be approximated within a factor of 1.5 in $O(n^2)$ time.*

Furthermore, even if S and R are arbitrary strings from σ, it is possible to approximately compute both $BED(S, R)$ and $EDM(S, R)$ in near linear time. More specifically obtain an embedding of S and R to binary vectors $f(S)$ and $f(R)$ such that:

Theorem 6 (Muthukrishnan-Sahinalp [12])
$\frac{||f(S)-f(R)||_1}{\log^* n} \leq BED(S, R) \leq ||f(S) - f(R)||_1 \cdot \log n.$

In other words, the Hamming distance between $f(S)$ and $f(R)$ approximates $BED(S, R)$ within a factor of $\log n \cdot \log^* n$. Similarly for $EDM(S, R)$, it is possible to embed S and R to integer valued vectors $F(S)$ and $F(R)$ such that:

Theorem 7 (Cormode-Muthukrishnan [4])
$\frac{||F(S)-F(R)||_1}{\log^* n} \leq EDM(S, R) \leq ||F(S) - F(R)||_1 \cdot \log n.$

In other words, the L_1 distance between $F(S)$ and $F(R)$ approximates $EDM(S, R)$ within a factor of $\log n \cdot \log^* n$.

The embedding of strings S and R into binary vectors $f(S)$ and $f(R)$ is introduced in [5] and is based on the Locally Consistent Parsing described above. To obtain the embedding, one needs to hierarchically partition S and R into growing size *core* blocks. Given an alphabet σ, Locally Consistent Parsing can identify only a limited number of substrings as core blocks. Consider the lexicographic ordering of these core blocks. Each dimension i of the embedding $f(S)$ simply indicates (by setting $f(S)[i] = 1$) whether S includes the ith core block corresponding to the alphabet σ as a substring. Note that if a core block exists in S as a substring, Locally Consistent Parsing will identify it.

Although the embedding above is exponential in size, the resulting binary vector $f(S)$ is very sparse. A simple representation of $f(S)$ and $f(R)$, exploiting their sparseness can be computed in time $O(n \log^* n)$ and the Hamming distance between $f(S)$ and $f(R)$ can be computed in linear time by the use of this representation [12].

The embedding of S and R into integer valued vectors $F(S)$ and $F(R)$ are based on similar techniques. Again, the total time needed to approximate $EDM(S, R)$ within a factor of $\log n \cdot \log^* n$ is $O(n \log^* n)$.

Applications

Edit distances have important uses in computational evolutionary biology, in estimating the evolutionary distance between pairs of genome sequences under various edit operations. There are also several applications to the *document exchange problem* or *document reconciliation problem* where two copies of a text string S have been subject to edit operations (both single character and block edits) by two parties resulting in two versions S_1 and S_2, and the parties communicate to reconcile the differences between the two versions. An information theoretic lower bound on the number of bits to communicate between the two parties is then $\Omega(BED(S, R)) \cdot \log n$. The embedding of S and R to binary strings $f(S)$ and $f(R)$ provides a simple protocol [5] which gives a near-optimal tradeoff between the number of rounds of communication and the total number of bits exchanged and works with high probability.

Another important application is to the Sequence Nearest Neighbors (SNN) problem, which asks to preprocess a set of strings $S_1, \ldots, S_k$ so that given an on-line query string R, the string S_i which has the lowest distance of choice to R can be computed in time polynomial with $|R|$ and polylogarithmic with $\sum_{j=1}^{k} |S_j|$. There are no known exact solutions for the SNN problem under any edit distance considered here. However, in [12], the embedding of strings S_i into binary vectors $f(S_i)$, combined with the Approximate Nearest Neighbors results given in [6] for Hamming Distance, provides an approximate solution to the SNN problem under block edit distance as follows.

Theorem 8 (Muthukrishnan-Sahinalp [12]) *It is possible to preprocess a set of strings $S_1, \ldots, S_k$ from a given alphabet σ in $O(poly(\sum_{j=1}^{k} |S_j|))$ time such that for any on-line query string R from σ one can compute a string S_i in time $O(polylog(\sum_{j=1}^{k} |S_j|) \cdot poly(|R|))$ which guarantees that for all $h \in [1, k]$, $BED(S_i, R) \leq BED(S_h, R) \cdot \log(\max_j |S_j|) \cdot \log^*(\max_j |S_j|)$.*

Open Problems

It is interesting to note that when dealing with permutations of the alphabet σ the problem of computing both character edit distances and block edit distances become much easier; one can compute $PED(S, R)$ exactly and $TED(S, R)$ within an approximation factor of 1.5 in $\tilde{O}(n)$ time. For arbitrary strings, it is an open question whether one can approximate $TED(S, R)$ or $BED(S, R)$ within a factor of $o(\log n)$ in polynomial time. One recent result in this direction shows that it is not possible to obtain a polylogarithmic approximation to $TED(S, R)$ via a greedy strategy [7]. Furthermore, although there is a lower bound of $\Omega(n^{\frac{1}{3}})$ on the approximation factor that can be achieved for computing the standard edit distance in $\tilde{O}(n)$ time by the use of string embeddings, there is no general lower bound on how closely one can approximate $ED(S, R)$ in near linear time.

Cross References

► Sequential Approximate String Matching

Recommended Reading

1. Bafna, V., Pevzner, P.A.: Sorting by Transpositions. SIAM J. Discret. Math. **11**(2), 224–240 (1998)
2. Batu, T., Ergün, F., Sahinalp, S.C.: Oblivious string embeddings and edit distance approximations. Proc. ACM-SIAM SODA 792–801 (2006)
3. Besmaphyatnikh, S., Segal, M.: Enumerating longest increasing subsequences and patience sorting. Inform. Proc. Lett. **76**(1–2), 7–11 (2000)
4. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. Proc. ACM-SIAM SODA 667–676 (2002)
5. Cormode, G., Paterson, M., Sahinalp, S.C., Vishkin, U.: Communication complexity of document exchange. Proc. ACM-SIAM SODA 197–206 (2000)
6. Indyk, P., Motwani, R.: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. Proc. ACM STOC 604–613 (1998)
7. Kaplan, H., Shafrir, N.: The greedy algorithm for shortest superstrings. Inform. Proc. Lett. **93**(1), 13–17 (2005)
8. Landau, G., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms **10**, 157–169 (1989)
9. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR **163**(4):845–848 (1965) (Russian). Soviet Physics Doklady **10**(8), 707–710 (1966) (English translation)
10. Lopresti, D.P., Tomkins, A.: Block Edit Models for Approximate String Matching. Theoretical. Comput. Sci. **181**(1), 159–179 (1997)
11. Masek, W., Paterson, M.: A faster algorithm for computing string edit distances. J. Comput. Syst. Sci. **20**, 18–31 (1980)
12. Muthukrishnan, S., Sahinalp, S.C.: Approximate nearest neighbors and sequence comparison with block operations. Proc. ACM STOC 416–424 (2000)
13. Sahinalp, S.C., Vishkin, U.: Symmetry breaking for suffix tree construction. ACM STOC 300–309 (1994)
14. Sahinalp, S.C., Vishkin, U.: Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. Proc. IEEE FOCS 320–328 (1996)

Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds

1993; Gusfield

Francis Chin, S. M. Yiu
Department of Computer Science,
University of Hong Kong, Hong Kong, China

Keywords and Synonyms

Multiple string alignment; Multiple global alignment

Problem Definition

Multiple sequence alignment is an important problem in computational biology. Applications include finding highly conserved subregions in a given set of biological sequences and inferring the evolutionary history of a set of taxa from their associated biological sequences (e. g., see [6]). There are a number of measures proposed for evaluating the goodness of a multiple alignment, but prior to this work, no efficient methods are known for computing the optimal alignment for any of these measures. The

work of Gusfield [5] gives two computationally efficient multiple alignment approximation algorithms for two of the measures with approximation ratio of less than 2. For one of the measures, they also derived a randomized algorithm, which is much faster and with high probability, reports a multiple alignment with small error bounds. To the best knowledge of the entry authors, this work is the first to provide approximation algorithms (with guarantee error bounds) for this problem.

Notations and Definitions

Let X and Y be two strings of alphabet Σ. The pairwise alignment of X and Y maps X and Y into strings X' and Y' that may contain spaces, denoted by '_', where (1) $|X'| = |Y'| = \ell$; and (2) removing spaces from X' and Y' returns X and Y, respectively. The score of the alignment is defined as $d(X', Y') = \sum_{i=1}^{\ell} s(X'(i), Y'(i))$ where $X'(i)$ (and $Y'(i)$) denotes the ith character in X' (and Y') and $s(a, b)$ with $a, b \in \Sigma \cup$ '_' is the distance-based scoring scheme that satisfies the following assumptions.

1. $s($'_', '_'$) = 0$;
2. triangular inequality: for any three characters, x, y, z, $s(x, z) \leq s(x, y) + s(y, z)$).

Let $\chi = X_1, X_2, \ldots, X_k$ be a set of $k > 2$ strings of alphabet Σ. A multiple alignment A of these k strings maps $X_1, X_2, \ldots, X_k$ to $X_1', X_2', \ldots, X_k$' that may contain spaces such that (1) $|X_1'| = |X_2'| = \cdots = |X_k'| = \ell$; and (2) removing spaces from X_i' returns X_i for all $1 \leq i \leq k$. The multiple alignment A can be represented as a $k \times \ell$ matrix.

The Sum of Pairs (SP) Measure

The score of a multiple alignment A, denoted by SP(A), is defined as the sum of the scores of pairwise alignments induced by A, that is, $\sum_{i<j} d(X_i', X_j') = \sum_{i<j} \sum_{p=1}^{\ell} s(X_i'[p], X_j'[p])$ where $1 \leq i < j \leq k$.

Problem 1 *Multiple Sequence Alignment with Minimum SP score*
INPUT: *A set of k strings, a scoring scheme s.*
OUTPUT: *A multiple alignment A of these k strings with minimum SP(A).*

The Tree Alignment (TA) Measure

In this measure, the multiple alignment is derived from an evolutionary tree. For a given set χ of k strings, let $\chi' \supseteq \chi$. An evolutionary tree $T_{\chi'}$ for χ is a tree with at least k nodes, where there is a one-to-one correspondence between the nodes and the strings in χ'. Let $X_u' \in \chi$' be the string for node u. The score of $T_{\chi'}$, denoted by $TA(T_{\chi'})$, is defined as $\sum_{e=(u,v)} D(X_u', X_v')$ where e is an edge in $T_{\chi'}$ and $D(X_u', X_v')$ denotes the score of the optimal pairwise alignment for X_u' and X_v'. Analogously, the multiple alignment of χ under the TA measure can also be represented by a $|\chi'| \times \ell$ matrix, where $|\chi'| \geq k$, with a score defined as $\sum_{e=(u,v)} d(X_u', X_v')$($e$ is an edge in $T_{\chi'}$), similar to the multiple alignment under the SP measure in which the score is the summation of the alignment scores of all pairs of strings. Under the TA measure, since it is always possible to construct the $|\chi'| \times \ell$ matrix such that $d(X_u', X_v') = D(X_u', X_v')$ for all $e = (u, v)$ in $T_{\chi'}$ and we are usually interested in finding the multiple alignment with the minimum TA value, so $D(X_u', X_v')$ is used instead of $d(X_u', X_v')$ in the definition of $TA(T_{\chi'})$.

Problem 2 *Multiple Sequence Alignment with Minimum TA score*
INPUT: *A set of k strings, a scoring scheme s.*
OUTPUT: *An evolutionary tree T for these k strings with minimum TA(T).*

Key Results

Theorem 1 *Let A^* be the optimal multiple alignment of the given k strings with minimum SP score. They provide an approximation algorithm (the center star method) that gives a multiple alignment A such that $\frac{SP(A)}{SP(A*)} \leq \frac{2(k-1)}{k} = 2 - \frac{2}{k}$.*

The center star method is to derive a multiple alignment which is consistent with the optimal pairwise alignments of a center string with all the other strings. The bound is derived based on the triangular inequality of the score function. The time complexity of this method is $O(k^2\ell^2)$, where ℓ^2 is the time to solve the pairwise alignment by dynamic programming and k^2 is needed to find the center string, X_c, which gives the minimum value of $\sum_{i \neq c} D(X_c, X_i)$.

Theorem 2 *Let A^* be the optimal multiple alignment of the given k strings with minimum SP score. They provide a randomized algorithm that gives a multiple alignment A such that $\frac{SP(A)}{SP(A*)} \leq 2 + \frac{1}{r-1}$ with probability at least $1 - \left(\frac{r-1}{r}\right)^p$ for any $r > 1$ and $p \geq 1$.*

Instead of computing $\binom{k}{2}$ optimal pairwise alignments to find the best center string, the randomized algorithm only considers p randomly selected strings to be candidates for the best center string, thus this method needs to x compute only $(k-1)p$ optimal pairwise alignments in $O(kp\ell^2)$ time where $1 \leq p \leq k$.

Theorem 3 *Let T^* be the optimal evolutionary tree of the given k strings with minimum TA score. They provide an*

approximation algorithm that gives an evolutionary tree T such that $\frac{TA(T)}{TA(T*)} \leq \frac{2(k-1)}{k} = 2 - \frac{2}{k}$.

In the algorithm, they first compute all the $\binom{k}{2}$ optimal pairwise alignments to construct a graph with every node representing a distinct string X_i and the weight of each edge (X_i, X_j) as $D(X_i, X_j)$. This step determines the overall time complexity $O(k^2\ell^2)$. Then, they find a minimum spanning tree from the graph. The multiple alignment has to be consistent with the optimal pairwise alignments represented by the edges of this minimum spanning tree.

Applications

Multiple sequence alignment is a fundamental problem in computational biology. In particular, multiple sequence alignment is useful in identifying those common structures, which may only be weakly reflected in the sequence and not easily revealed by pairwise alignment. These common structures may carry important information for their evolutionary history, critical conserved motifs, common 3D molecular structure, as well as biological functions.

More recently, multiple sequence alignment is also used in revealing non-coding RNAs (ncRNAs) [3]. In this type of multiple alignment, we are not only align the underlying sequences, but also the secondary structures (refer to chap. 16 of [10] for a brief introduction of secondary structure of a RNA) of the RNAs. Researchers believe that ncRNAs that belong to the same family should have common components giving a similar secondary structure. The multiple alignment can help to locate and identify these common components.

Open Problems

A number of open problems related to the work of Gusfield remain open. For the SP measure, the center star method can be extended to the q-star method ($q > 2$) with approximation ratio of $2 - q/k$ ([1,7], sect. 7.5 of [8]). Whether there exists an approximation algorithm with better approximation ratio or with better time complexity is still unknown. For the TA measure, to be the best knowledge of the entry authors, the approximation ratio in Theorem 3 is currently the best result.

Another interesting direction related to this problem is the constrained multiple sequence alignment problem [9] which requires the multiple alignment to contain certain aligned characters with respect to a given constrained sequence. The best known result [2] is an approximation algorithm (also follows the idea of center star method) which gives an alignment with approximation ratio of $2 - 2/k$ for k strings.

For the complexity of the problem, Wang and Jiang [11] were the first to prove the NP-hardness of the problem with SP score under a *non-metric* distance measure over a 4 symbol alphabet. More recently, in [4], the multiple alignment problem with SP score, star alignment, and TA score have been proved to be NP-hard for all binary or larger alphabets under *any metric*. Developing efficient approximation algorithms with good bounds for any of these measures is desirable.

Experimental Results

Two experiments have been reported in the paper showing that the worst case error bounds in Theorems 1 and 2 (for the SP measure) are pessimistic compared to the typical situation arising in practice.

The scoring scheme used in the experiments is: $s(a, b) = 0$ if $a = b$; $s(a, b) = 1$ if either a or b is a space; otherwise $s(a, b) = 2$. Since computing the optimal multiple alignment with minimum SP score has been shown to be NP-hard, they evaluate the performance of their algorithms using the lower bound of $\sum_{i<j} D(X_i, X_j)$ (recall that $D(X_i, X_j)$ is the score of the optimal pairwise alignment of X_i and X_j). They have aligned 19 similar amino acid sequences with average length of 60 of homeoboxs from different species. The ratio of the scores of reported alignment by the center star method to the lower bound is only 1.018 which is far from the worst case error bound given in Theorem 1. They also aligned 10 not-so-similar sequences near the homeoboxs, the ratio of the reported alignment to the lower bound is 1.162. Results also show that the alignment obtained by the randomized algorithm is usually not far away from the lower bound.

Data Sets

The exact sequences used in the experiments are not provided.

Cross References

► Statistical Multiple Alignment

Recommended Reading

1. Bafna, V., Lawler, E.L., Pevzner, P.A.: Approximation algorithms for multiple sequence alignment. Theor. Comput. Sci. **182**, 233–244 (1997)
2. Francis, Y.L., Chin, N.L.H., Lam, T.W., Prudence, W.H.W.: Efficient constrained multiple sequence alignment with performance guarantee. J. Bioinform. Comput. Biol. **3**(1), 1–18 (2005)
3. Dalli, D., Wilm, A., Mainz, I., Stegar, G.: STRAL: progressive alignment of non-coding RNA using base pairing probability vectors in quadratic time. Bioinformatics **22**(13), 1593–1599 (2006)

4. Elias, I.: Setting the intractability of multiple alignment. In: Proc. of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003), 2003, pp. 352–363
5. Gusfield, D.: Efficient methods for multiple sequence alignment with guaranteed error bounds. Bull. Math. Biol. **55**(1), 141–154 (1993)
6. Pevsner, J.: Bioinformatics and functional genomics. Wiley, New York (2003)
7. Pevzner, P.A.: Multiple alignment, communication cost, and graph matching. SIAM J. Appl. Math. **52**, 1763–1779 (1992)
8. Pevzner, P.A.: Computational molecular biology: an algorithmic approach. MIT Press, Cambridge, MA (2000)
9. Tang, C.Y., Lu, C.L., Chang, M.D.T., Tsai, Y.T., Sun, Y.J., Chao, K.M., Chang, J.M., Chiou, Y.H., Wu, C.M., Chang, H.T., Chou, W.I.: Constrained multiple sequence alignment tool development and its application to RNase family alignment. In: Proc. of the First IEEE Computer Society Bioinformatics Conference (CSB 2002), 2002, pp. 127–137
10. Tompa, M.: Lecture notes. Department of Computer Science & Engineering, University of Washington. http://www.cs.washington.edu/education/courses/527/00wi/. (2000)
11. Wang, L. Jiang, T.: On the complexity of multiple sequence alignment. J. Comp. Biol. **1**, 337–48 (1994)

Engineering Algorithms for Computational Biology

2002; Bader, Moret, Warnow

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

Keywords and Synonyms

High-performance computational biology

Problem Definition

In the 50 years since the discovery of the structure of DNA, and with new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Many of the newly faced challenges require high-performance computing, either due to the massive-parallelism required by the problem, or the difficult optimization problems that are often combinatoric and NP-hard. Unlike the traditional uses of supercomputers for regular, numerical computing, many problems in biology are irregular in structure, significantly more challenging to parallelize, and integer-based using abstract data structures.

Biologists are in search of biomolecular sequence data, for its comparison with other genomes, and because its structure determines function and leads to the understanding of biochemical pathways, disease prevention and cure, and the mechanisms of life itself. Computational biology has been aided by recent advances in both technology and algorithms; for instance, the ability to sequence short contiguous strings of DNA and from these reconstruct the whole genome and the proliferation of high-speed microarray, gene, and protein chips for the study of gene expression and function determination. These high-throughput techniques have led to an exponential growth of available genomic data.

Algorithms for solving problems from computational biology often require parallel processing techniques due to the data- and compute-intensive nature of the computations. Many problems use polynomial time algorithms (e. g., all-to-all comparisons) but have long running times due to the large number of items in the input; for example, the assembly of an entire genome or the all-to-all comparison of gene sequence data. Other problems are compute-intensive due to their inherent algorithmic complexity, such as protein folding and reconstructing evolutionary histories from molecular data, that are known to be NP-hard (or harder) and often require approximations that are also complex.

Key Results

None

Applications

Phylogeny Reconstruction: A phylogeny is a representation of the evolutionary history of a collection of organisms or genes (known as taxa). The basic assumption of process necessary to phylogenetic reconstruction is repeated divergence within species or genes. A phylogenetic reconstruction is usually depicted as a tree, in which modern taxa are depicted at the leaves and ancestral taxa occupy internal nodes, with the edges of the tree denoting evolutionary relationships among the taxa. Reconstructing phylogenies is a major component of modern research programs in biology and medicine (as well as linguistics). Naturally, scientists are interested in phylogenies for the sake of knowledge, but such analyses also have many uses in applied research and in the commercial arena. Existing phylogenetic reconstruction techniques suffer from serious problems of running time (or, when fast, of accuracy). The problem is particularly serious for large data sets: even though data sets comprised of sequence from a single gene continue to pose challenges (e. g., some analyses are still running after two years of computation on medium-sized clusters), using whole-genome data (such as gene content and gene order) gives rise to even more formidable computational problems, particularly in data sets with large numbers of genes and highly-rearranged genomes.

To date, almost every model of speciation and genomic evolution used in phylogenetic reconstruction has given rise to NP-hard optimization problems. Three major classes of methods are in common use. Heuristics (a natural consequence of the NP-hardness of the problems) run quickly, but may offer no quality guarantees and may not even have a well-defined optimization criterion, such as the popular *neighbor-joining* heuristic [9]. Optimization based on the criterion of *maximum parsimony* (MP) [4] seeks the phylogeny with the least total amount of change needed to explain modern data. Finally, optimization based on the criterion of *maximum likelihood* (ML) [5] seeks the phylogeny that is the most likely to have given rise to the modern data.

Heuristics are fast and often rival the optimization methods in terms of accuracy, at least on datasets of moderate size. Parsimony-based methods may take exponential time, but, at least for DNA and amino acid data, can often be run to completion on datasets of moderate size. Methods based on maximum likelihood are very slow (the point estimation problem alone appears intractable) and thus restricted to very small instances, and also require many more assumptions than parsimony-based methods, but appear capable of outperforming the others in terms of the quality of solutions when these assumptions are met. Both MP- and ML-based analyses are often run with various heuristics to ensure timely termination of the computation, with mostly unquantified effects on the quality of the answers returned.

Thus there is ample scope for the application of high-performance algorithm engineering in the area. As in all scientific computing areas, biologists want to study a particular dataset and are willing to spend months and even years in the process: accurate branch prediction is the main goal. However, since all exact algorithms scale exponentially (or worse, in the case of ML approaches) with the number of taxa, speed remains a crucial parameter – otherwise few datasets of more than a few dozen taxa could ever be analyzed.

Experimental Results

As an illustration, this entry briefly describes a high-performance software suite, GRAPPA (Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms) developed by Bader et al. *GRAPPA* extends Sankoff and Blanchette's breakpoint phylogeny algorithm [10] into the more biologically-meaningful inversion phylogeny and provides a highly-optimized code that can make use of distributed- and shared-memory parallel systems (see [1,2,6,7,8,11] for details). In [3], Bader et al. gives the first linear-time algorithm and fast implementation for computing inversion distance between two signed permutations. *GRAPPA* was run on a 512-processor IBM Linux cluster with Myrinet and obtained a 512-fold speedup (linear speedup with respect to the number of processors): a complete breakpoint analysis (with the more demanding inversion distance used in lieu of breakpoint distance) for the 13 genomes in the Campanulaceae data set ran in less than 1.5 hours in an October 2000 run, for a *million-fold* speedup over the original implementation. The latest version features significantly improved bounds and new distance correction methods and, on the same dataset, exhibits a speedup factor of *over one billion*. GRAPPA achieves this speedup through a combination of parallelism and high-performance algorithm engineering. Although such spectacular speedups will not always be realized, many algorithmic approaches now in use in the biological, pharmaceutical, and medical communities may benefit tremendously from such an application of high-performance techniques and platforms.

This example indicates the potential of applying high-performance algorithm engineering techniques to applications in computational biology, especially in areas that involve complex optimizations: Bader's reimplementation did not require new algorithms or entirely new techniques, yet achieved gains that turned an impractical approach into a usable one.

Cross References

- ▶ Distance-Based Phylogeny Reconstruction (Fast-Converging)
- ▶ Distance-Based Phylogeny Reconstruction (Optimal Radius)
- ▶ Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds
- ▶ High Performance Algorithm Engineering for Large-scale Problems
- ▶ Local Alignment (with Affine Gap Weights)
- ▶ Local Alignment (with Concave Gap Weights)
- ▶ Multiplex PCR for Gap Closing (Whole-genome Assembly)
- ▶ Peptide De Novo Sequencing with MS/MS
- ▶ Perfect Phylogeny Haplotyping
- ▶ Phylogenetic Tree Construction from a Distance Matrix
- ▶ Phylogeny Reconstruction
- ▶ Sorting Signed Permutations by Reversal (Reversal Distance)
- ▶ Sorting Signed Permutations by Reversal (Reversal Sequence)

- ▶ Sorting by Transpositions and Reversals (Approx Ratio 1.5)
- ▶ Substring Parsimony

Recommended Reading

1. Bader, D.A., Moret, B.M.E., Warnow, T., Wyman, S.K., Yan, M.: High-performance algorithm engineering for gene-order phylogenies. In: DIMACS Workshop on Whole Genome Comparison, Rutgers University, Piscataway, NJ (2001)
2. Bader, D.A., Moret, B.M.E., Vawter, L.: Industrial applications of high-performance computing for phylogeny reconstruction. In: Siegel, H.J. (ed.) Proc. SPIE Commercial Applications for High-Performance Computing, vol. 4528, pp. 159–168, Denver, CO (2001)
3. 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. Comp. Biol. **8**(5), 483–491 (2001)
4. Farris, J.S.: The logical basis of phylogenetic analysis. In: Platnick, N.I., Funk, V.A. (eds.) Advances in Cladistics, pp. 1–36. Columbia Univ. Press, New York (1983)
5. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17**, 368–376 (1981)
6. Moret, B.M.E., Bader, D.A., Warnow, T., Wyman, S.K., Yan, M.: GRAPPA: a highperformance computational tool for phylogeny reconstruction from gene-order data. In: Proc. Botany, Albuquerque, August 2001
7. Moret, B.M.E., Bader, D.A., Warnow, T.: High-performance algorithm engineering for computational phylogenetics. J. Supercomp. **22**, 99–111 (2002) Special issue on the best papers from ICCS'01
8. Moret, B.M.E., Wyman, S., Bader, D.A., Warnow, T., Yan, M.: A new implementation and detailed study of breakpoint analysis. In: Proc. 6th Pacific Symp. Biocomputing (PSB 2001), pp. 583–594, Hawaii, January 2001
9. Saitou, N., Nei, M.: The neighbor-joining method: A new method for reconstruction of phylogenetic trees. Mol. Biol. Evol. **4**, 406–425 (1987)
10. Sankoff, D., Blanchette, M.: Multiple genome rearrangement and breakpoint phylogeny. J. Comp. Biol. **5**, 555–570 (1998)
11. Yan, M.: High Performance Algorithms for Phylogeny Reconstruction with Maximum Parsimony. Ph. D. thesis, Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, January 2004

Engineering Algorithms for Large Network Applications

2002; Schulz, Wagner, Zaroliagis

Christos Zaroliagis
Department of Computer Engineering & Informatics, University of Patras, Patras, Greece

Problem Definition

Dealing effectively with applications in large networks, it typically requires the efficient solution of one ore more underlying algorithmic problems. Due to the size of the network, a considerable effort is inevitable in order to achieve the desired efficiency in the algorithm.

One of the primary tasks in large network applications is to answer queries for finding best routes or paths as efficiently as possible. Quite often, the challenge is to process a vast number of such queries on-line: a typical situation encountered in several real-time applications (e. g., traffic information systems, public transportation systems) concerns a query-intensive scenario, where a central server has to answer a huge number of on-line customer queries asking for their best routes (or optimal itineraries). The main goal in such an application is to reduce the (average) response time for a query.

Answering a best route (or optimal itinerary) query translates in computing a minimum cost (shortest) path on a suitably defined directed graph (digraph) with nonnegative edge costs. This in turn implies that the core algorithmic problem underlying the efficient answering of queries is the single-source single-target shortest path problem.

Although the straightforward approach of pre-computing and storing shortest paths for all pairs of vertices would enabling the optimal answering of shortest path queries, the quadratic space requirements for digraphs with more than 10^5 vertices makes such an approach prohibitive for large and very large networks. For this reason, the main goal of almost all known approaches is to keep the space requirements as small as possible. This in turn implies that one can afford a heavy (in time) preprocessing, which does not blow up space, in order to speed-up the query time.

The most commonly used approach for answering shortest path queries employs Dijkstra's algorithm and/or variants of it. Consequently, the main challenge is how to reduce the algorithm's *search-space* (number of vertices visited), as this would immediately yield a better query time.

Key Results

All results discussed concern answering of *optimal* (or *exact* or *distance-preserving*) shortest paths under the aforementioned query-intensive scenario, and are all based on the following generic approach. A preprocessing of the input network $G = (V, E)$ takes place that results in a data structure of size $O(|V| + |E|)$ (i. e., linear to the size of G). The data structure contains additional information regarding certain shortest paths that can be used later during querying.

Depending on the pre-computed additional information as well as on the way a shortest path query is answered, two approaches can be distinguished. In the first approach, *graph annotation*, the additional information is attached to vertices or edges of the graph. Then, speed-up techniques to Dijkstra's algorithm are employed that, based on this information, decide quickly which part of the graph does not need to be searched. In the second approach, an *auxiliary graph* G' is constructed hierarchically. A shortest path query is then answered by searching only a small part of G', using Dijkstra's algorithm enhanced with heuristics to further speed-up the query time.

In the following, the key results of the first [3,4,9,11] and the second approach [1,2,5,7,8,10] are discussed, as well as results concerning modeling issues.

First Approach – Graph Annotation

The first work under this approach concerns the study in [9] on large railway networks. In that paper, two new heuristics are introduced: the *angle-restriction* (that tries to reduce the search space by taking advantage of the geometric layout of the vertices) and the *selection of stations* (a subset of vertices is selected among which all pairs shortest paths are pre-computed). These two heuristics along with a combination of the classical *goal-directed* or A^* *search* turned out to be rather efficient. Moreover, they motivated two important generalizations [10,11] that gave further improvements to shortest path query times.

The full exploitation of geometry-based heuristics was investigated in [11], where both street and railway networks are considered. In that paper, it is shown that the search space of Dijkstra's algorithm can be significantly reduced (to 5%–10% of the initial graph size) by extracting geometric information from a given layout of the graph and by encapsulating pre-computed shortest path information in resulted geometric objects, called *containers*. Moreover, the dynamic case of the problem was investigated, where edge costs are subject to change and the geometric containers have to be updated.

A powerful modification to the classical Dijkstra's algorithm, called *reach-based routing*, was presented in [4]. Every vertex is assigned a so-called *reach value* that determines whether a particular vertex will be considered during Dijkstra's algorithm. A vertex is excluded from consideration if its reach value is small; that is, if it does not contribute to any path long enough to be of use for the current query.

A considerable enhancement of the classical A^* *search* algorithm using landmarks (selected vertices like in [9,10]) and the triangle inequality with respect to the shortest path distances was shown in [3]. Landmarks and triangle inequality help to provide better lower bounds and hence boost A^* search.

Second Approach – Auxiliary Graph

The first work under this approach concerns the study in [10], where a new hierarchical decomposition technique is introduced called *multi-level graph*. A multi-level graph $\mathcal{M}$ is a digraph which is determined by a sequence of subsets of V and which extends E by adding multiple levels of edges. This allows to efficiently construct, during querying, a subgraph of $\mathcal{M}$ which is substantially smaller than G and in which the shortest path distance between any of its vertices is equal to the shortest path distance between the same vertices in G. Further improvements of this approach have been presented recently in [1]. A refinement of the above idea was introduced in [5], where the multi-level overlay graphs are introduced. In such a graph, the decomposition hierarchy is not determined by application-specific information as it happens in [9,10].

An alternative hierarchical decomposition technique, called *highway hierarchies*, was presented in [7]. The approach takes advantage of the inherent hierarchy possessed by real-world road networks and computes a hierarchy of coarser views of the input graph. Then, the shortest path query algorithm considers mainly the (much smaller in size) coarser views, thus achieving dramatic speed-ups in query time. A revision and improvement of this method was given in [8]. A powerful combination of the highway hierarchies with the ideas in [3] was reported in [2].

Modeling Issues

The modeling of the original best route (or optimal itinerary) problem on a large network to a shortest path problem in a suitably defined directed graph with appropriate edge costs also plays a significant role in reducing the query time. Modeling issues are thoroughly investigated in [6]. In that paper, the first experimental comparison of two important approaches (time-expanded versus time-dependent) is carried out, along with new extensions of them towards realistic modeling. In addition, several new heuristics are introduced to speed-up query time.

Applications

Answering shortest path queries in large graphs has a multitude of applications, especially in traffic information systems under the aforementioned scenario; that is, a central server has to answer, as fast as possible, a huge number of on-line customer queries asking for their best routes or itineraries. Other applications of the above scenario

involve route planning systems for cars, bikes and hikers, public transport systems for itinerary information of scheduled vehicles (like trains or buses), answering queries in spatial databases, and web searching. All the above applications concern real-time systems in which users continuously enter their requests for finding their best connections or routes. Hence, the main goal is to reduce the (average) response time for answering a query.

Open Problems

Real-world networks increase constantly in size either as a result of accumulation of more and more information on them, or as a result of the digital convergence of media services, communication networks, and devices. This scaling-up of networks makes the scalability of the underlying algorithms questionable. As the networks continue to grow, there will be a constant need for designing faster algorithms to support core algorithmic problems.

Experimental Results

All papers discussed in Sect. "Key Results" contain important experimental studies on the various techniques they investigate.

Data Sets

The data sets used in [6,11] are available from http://lso-compendium.cti.gr/ under problems 26 and 20, respectively.

The data sets used in [1,2] are available from http://www.dis.uniroma1.it/~challenge9/.

URL to Code

The code used in [9] is available from http://doi.acm.org/10.1145/351827.384254.

The code used in [6,11] is available from http://lso-compendium.cti.gr/ under problems 26 and 20, respectively.

The code used in [3] is available from http://www.avglab.com/andrew/soft.html.

Cross References

- ► Implementation Challenge for Shortest Paths
- ► Shortest Paths Approaches for Timetable Information

Recommended Reading

1. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Graphs. In: 9th DIMACS Challenge on Shortest Paths, Nov 2006. Rutgers University, USA (2006)
2. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Challenge on Shortest Paths, Nov 2006 Rutgers University, USA (2006)
3. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A^* Search Meets Graph Theory. In: Proc. 16th ACM-SIAM Symposium on Discrete Algorithms – SODA, pp. 156–165. ACM, New York and SIAM, Philadelphia (2005)
4. Gutman, R.: Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: Algorithm Engineering and Experiments – ALENEX (SIAM, 2004), pp. 100–111. SIAM, Philadelphia (2004)
5. Holzer, M., Schulz, F., Wagner, D.: Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In: Algorithm Engineering and Experiments – ALENEX (SIAM, 2006), pp. 156–170. SIAM, Philadelphia (2006)
6. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. ACM J. Exp. Algorithmic **12**(2.4), 1–39 (2007)
7. Sanders, P., Schultes, D.: Highway Hierarchies Hasten Exact Shortest Path Queries. In: Algorithms – ESA 2005. Lect. Note Comp. Sci. **3669**, 568–579 (2005)
8. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Algorithms – ESA 2006. Lect. Note Comp. Sci. **4168**, 804–816 (2006)
9. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. ACM J. Exp. Algorithmics **5**(12), 1–23 (2000)
10. Schulz, F., Wagner, D., Zaroliagis, C.: Using Multi-Level Graphs for Timetable Information in Railway Systems. In: Algorithm Engineering and Experiments – ALENEX 2002. Lect. Note Comp. Sci. **2409**, 43–59 (2002)
11. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric Containers for Efficient Shortest Path Computation. ACM J. Exp. Algorithmics **10**(1.3), 1–30 (2005)

Engineering Geometric Algorithms

2004; Halperin

Dan Halperin
School of Computer Science,
Tel-Aviv University, Tel Aviv, Israel

Keywords and Synonyms

Certified and efficient implementation of geometric algorithms; Geometric computing with certified numerics and topology

Problem Definition

Transforming a theoretical geometric algorithm into an effective computer program abounds with hurdles. Overcoming these difficulties is the concern of *engineering geometric algorithms*, which deals, more generally, with the design and implementation of certified and efficient solutions to algorithmic problems of geometric nature. Typ-

ical problems in this family include the construction of Voronoi diagrams, triangulations, arrangements of curves and surfaces (namely, space subdivisions), two- or higher-dimensional search structures, convex hulls and more.

Geometric algorithms strongly couple topological/combinatorial structures (e. g., a graph describing the triangulation of a set of points) on the one hand, with numerical information (e. g., the coordinates of the vertices of the triangulation) on the other. Slight errors in the numerical calculations, which in many areas of science and engineering can be tolerated, may lead to detrimental mistakes in the topological structure, causing the computer program to crash, to loop infinitely, or plainly to give wrong results.

Straightforward implementation of geometric algorithms as they appear in a textbook, using standard machine arithmetic, is most likely to fail. This entry is concerned only with *certified* solutions, namely, solutions that are guaranteed to construct the exact desired structure or a good approximation of it; such solutions are often referred to as *robust*.

The goal of engineering geometric algorithms can be restated as follows: *Design and implement geometric algorithms that are at once robust and efficient in practice.*

Much of the difficulty in adapting in practice the existing vast algorithmic literature in computational geometry comes from the assumptions that are typically made in the theoretical study of geometric algorithms that (1) the input is in general position, namely, degenerate input is precluded, (2) computation is performed on an ideal computer that can carry out real arithmetic to infinite precision (so-called real RAM), and (3) the cost of operating on a small number of simple geometric objects is "unit" time (e. g., equal cost is assigned to intersecting three spheres and to comparing two integer numbers).

Now, in real life, geometric input is quite often degenerate, machine precision is limited, and operations on a small number of simple geometric objects within the same algorithm may differ hundredfold and more in the time they take to execute (when aiming for certified results). Just implementing an algorithm carefully may not suffice and often redesign is called for.

Key Results

Tremendous efforts have been invested in the design and implementation of robust computational-geometry software in recent years. Two notable large-scale efforts are the CGAL library [1] and the geometric part of the LEDA library [14]. These are jointly reviewed in the survey by Kettner and Näher [13]. Numerous other relevant projects, which for space constraints are not reviewed here, are surveyed by Joswig [12] with extensive references to papers and Web sites.

A fundamental engineering decision to take when coming to implement a geometric algorithm is what will the underlying arithmetic be, that is, whether to opt for exact computation or use the machine floating-point arithmetic. (Other less commonly used options exist as well.) To date, the CGAL and LEDA libraries are almost exclusively based on exact computation. One of the reasons for this exclusivity is that exact computation emulates the ideal computer (for restricted problems) and makes the adaptation of algorithms from theory to software easier. This is facilitated by major headway in developing tools for efficient computation with rational or algebraic numbers (GMP [3], LEDA [14], CORE [2] and more). On top of these tools, clever techniques for reducing the amount of exact computation were developed, such as floating-point filters and the higher- level geometric filtering.

The alternative is to use the machine floating-point arithmetic, having the advantage of being very fast. However, it is nowhere near the ideal infinite precision arithmetic assumed in the theoretical study of geometric algorithms and algorithms have to be carefully redesigned. See, for example, the discussion about imprecision in the manual of QHULL, the convex hull program by Barber et al. [5]. Over the years a variety of specially tailored floating-point variants of algorithms have been proposed, for example, the carefully crafted VRONI package by Held [11], which computes the Voronoi diagram of points and line segments using standard floating-point arithmetic, based on the topology-oriented approach of Sugihara and Iri. While VRONI works very well in practice, it is not theoretically certified. *Controlled perturbation* [9] emerges as a systematic method to produce certified approximations of complex geometric constructs while using floating-point arithmetic: the input is perturbed such that all predicates are computed accurately even with the limited-precision machine arithmetic, and a method is given to bound the necessary magnitude of perturbation that will guarantee the successful completion of the computation.

Another decision to take is how to represent the output of the algorithm, where the major issue is typically how to represent the coordinates of vertices of the output structure(s). Interestingly, this question is crucial when using exact computation since there the output coordinates can be prohibitively large or simply impossible to finitely enumerate. (One should note though that many geometric algorithms are *selective* only, namely, they do not produce new geometric entities but just select and order subsets of the input coordinates. For example, the output of an al-

gorithm for computing the convex hull of a set of points in the plane is an ordering of a subset of the input points. No new point is computed. The discussion in this paragraph mostly applies to algorithms that output new geometric constructs, such as the intersection point of two lines.) But even when using floating-point arithmetic, one may prefer to have a more compact bit-size representation than, say, machine doubles. In this direction there is an effective, well-studied solution for the case of polygonal objects in the plane, called *snap rounding*, where vertices and intersection points are snapped to grid vertices while retaining certain topological properties of the exact desired structure. Rounding with guarantees is in general a very difficult problem, and already for polyhedral objects in 3-space the current attempts at generalizing snap rounding are very costly (increasing the complexity of the rounded objects to the third, or even higher, power).

Then there are a variety of engineering issues depending on the problem at hand. Following are two examples of engineering studies where the experience in practice is different from what the asymptotic resource measures imply. The examples relate to fundamental steps in many geometric algorithms: decomposition and point location.

Decomposition

A basic step in many geometric algorithms is to decompose a (possibly complex) geometric object into simpler subobjects, where each subobject typically has constant descriptive complexity. A well-known example is the triangulation of a polygon. The choice of decomposition may have a significant effect on the efficiency in practice of various algorithms that rely on decomposition. Such is the case when constructing Minkowski sums of polygons in the plane. The Minkowski sum of two sets A and B in $\mathbb{R}^d$ is the vector sum of the two sets $A \oplus B = \{a + b | a \in A, b \in B\}$. The simplest approach to computing Minkowski sums of two polygons in the plane proceeds in three steps: triangulate each polygon, then compute the sum of each triangle of one polygon with each triangle of the other, and finally take the union of all the subsums. In asymptotic measures, the choice of triangulation (over alternative decompositions) has no effect. In practice though, triangulation is probably the worst choice compared with other convex decompositions, even fairly simple heuristic ones (not necessarily optimal), as shown by experiments on a dozen different decomposition methods [4]. The explanation is that triangulation increases the overall complexity of the subsums and in turn makes the union stage more complex--indeed by a constant factor, but a noticeable factor in practice. Similar phenomena were observed in other situations as well. For example, when using the prevalent vertical decomposition of arrangements--often it is too costly compared with sparser decompositions (i. e., decompositions that add fewer extra features).

Point Location

A recurring problem in geometric computing is to process given planar subdivision (planar map), so as to efficiently answer *point-location* queries: Given a point q in the plane, which face of the map contains q? Over the years a variety of point-location algorithms for planar maps were implemented in CGAL, in particular, a hierarchical search structure that guarantees logarithmic query time after expected $O(n \log n)$ preprocessing time of a map with n edges. This algorithm is referred to in CGAL as the *RIC* point-location algorithm after the preprocessing method which uses randomized incremental construction. Several simpler, easier-to-program algorithms for point location were also implemented. None of the latter beats the RIC algorithm in query time. However, the RIC is by far the slowest of all the implemented algorithms in terms of preprocessing, which in many scenarios renders it less effective. One of the simpler methods devised is a variant of the well-known *jump-and-walk* approach to point location. The algorithm scatters points (so-called *landmarks*) in the map and maintains the landmarks (together with their containing faces) in a nearest-neighbor search structure. Once a query q is issued it finds the nearest landmark ℓ to q, and "walks" in the map from ℓ toward q along the straight line segment connecting them. This landmark approach offers query time that is only slightly more expensive than the RIC method while being very efficient in preprocessing. The full details can be found in [10]. This is yet another consideration when designing (geometric) algorithms: the cost of preprocessing (and storage) versus the cost of a query. Quite often the effective (practical) tradeoff between these costs needs to be deduced experimentally.

Applications

Geometric algorithms are useful in many areas. Triangulations and arrangements are examples of basic constructs that have been intensively studied in computational geometry, carefully implemented and experimented with, as well as used in diverse applications.

Triangulations

Triangulations in two and three dimensions are implemented in CGAL [7]. In fact, CGAL offers many variants of triangulations useful for different applications. Among the applications where CGAL triangulations are employed are

meshing, molecular modeling, meteorology, photogrammetry, and geographic information systems (GIS). For other available triangulation packages, see the survey by Joswig [12].

Arrangements

Arrangements of curves in the plane are supported by CGAL [15], as well as envelopes of surfaces in three-dimensional space. Forthcoming is support also for arrangements of curves on surfaces. CGAL arrangements have been used in motion planning algorithms, computer-aided design and manufacturing, GIS, computer graphics, and more (see Chap. 1 in [6]).

Open Problems

In spite of the significant progress in certified implementation of effective geometric algorithms, the existing theoretical algorithmic solutions for many problems still need adaptation or redesign to be useful in practice. One example where progress can have wide repercussions is devising effective decompositions for curved geometric objects (e. g., arrangements) in the plane and for higher-dimensional objects. As mentioned earlier, suitable decompositions can have a significant effect on the performance of geometric algorithms in practice.

Certified fixed-precision geometric computing lags behind the exact computing paradigm in terms of available robust software, and moving forward in this direction is a major challenge. For example, creating a certified floating-point counterpart to CGAL is a desirable (and highly intricate) task.

Another important tool that is largely missing is consistent and efficient rounding of geometric objects. As mentioned earlier, a fairly satisfactory solution exists for polygonal objects in the plane. Good techniques are missing for curved objects in the plane and for higher-dimensional objects (both linear and curved).

URL to Code

http://www.cgal.org

Cross References

► LEDA: a Library of Efficient Algorithms
► Robust Geometric Computation

Recommended Reading

Conferences publishing papers on the topic include the ACM Symposium on Computational Geometry (SoCG), the Workshop on Algorithm Engineering and Experiments (ALENEX), the Engineering and Applications Track of the European Symposium on Algorithms (ESA), its predecessor and the Workshop on Experimental Algorithms (WEA). Relevant journals include the *ACM Journal on Experimental Algorithmics*, *Computational Geometry: Theory and Applications* and the *International Journal of Computational Geometry and Applications*. A wide range of relevant aspects are discussed in the recent book edited by Boissonnat and Teillaud [6], titled *Effective Computational Geometry for Curves and Surfaces*.

1. The CGAL project homepage. http://www.cgal.org/. Accessed 6 Apr 2008
2. The CORE library homepage. http://www.cs.nyu.edu/exact/core/. Accessed 6 Apr 2008
3. The GMP webpage. http://gmplib.org/. Accessed 6 Apr 2008
4. Agarwal, P.K., Flato, E., Halperin, D.: Polygon decomposition for efficient construction of Minkowski sums. Comput. Geom. Theor. Appl. **21**(1–2), 39–61 (2002)
5. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.T.: Imprecision in QHULL. http://www.qhull.org/html/qh-impre.htm. Accessed 6 Apr 2008
6. Boissonnat, J.-D., Teillaud, M. (eds.) Effective Computational Geometry for Curves and Surfaces. Springer, Berlin (2006)
7. Boissonat, J.-D., Devillers, O., Pion, S., Teillaud, M., Yvinec, M.: Triangulations in CGAL. Comput. Geom. Theor. Appl. **22**(1–3), 5-19 (2002)
8. Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., Schönherr, S.: On the design of CGAL a computational geometry algorithms library. Softw. Pract. Experience **30**(11), 1167–1202 (2000)
9. Halperin, D., Leiserowitz, E.: Controlled perturbation for arrangements of circles. Int. J. Comput. Geom. Appl. **14**(4–5), 277–310 (2004)
10. Haran, I., Halperin, D.: An experimental study of point location in general planar arrangements. In: Proceedings of 8th Workshop on Algorithm Engineering and Experiments, pp. 16–25 (2006)
11. Held, M.: VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. Comput. Geom. Theor. Appl. **18**(2), 95–123 (2001)
12. Joswig, M.: Software. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn., chap. 64, pp. 1415–1433. Chapman & Hall/CRC, Boca Raton (2004)
13. Kettner, L., Näher, S.: Two computational geometry libraries: LEDA and CGAL. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, Chapter 65, pp. 1435–1463, 2nd edn. Chapman & Hall/CRC, Boca Raton (2004)
14. Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (2000)
15. Wein, R., Fogel, E., Zukerman, B., Halperin, D.: Advanced programming techniques applied to CGAL's arrangement package. Comput. Geom. Theor. Appl. **36**(1–2), 37–63 (2007)

Equivalence Between Priority Queues and Sorting

2002; Thorup

Rezaul A. Chowdhury
Department of Computer Sciences,
University of Texas at Austin,
Austin, TX, USA

Keywords and Synonyms

Heap

Problem Definition

A *priority queue* is an abstract data structure that maintains a set Q of elements, each with an associated value called a *key*, under the following set of operations [4,7].

insert(Q, x, k)**:** Inserts element x with key k into Q.
find-min(Q)**:** Returns an element of Q with the minimum key, but does not change Q.
delete(Q, x, k)**:** Deletes element x with key k from Q.

Additionally, the following operations are often supported.

delete-min(Q)**:** Deletes an element with the minimum key value from Q, and returns it.
decrease-key(Q, x, k)**:** Decreases the current key k' of x to k assuming $k < k'$.
meld(Q_1, Q_2)**:** Given priority queues Q_1 and Q_2, returns the priority queue $Q_1 \cup Q_2$.

Observe that a *delete-min* can be implemented as a *find-min* followed by a *delete*, a *decrease-key* as a *delete* followed by an *insert*, and a *meld* as a series of *find-min*, *delete* and *insert*. However, more efficient implementations of *decrease-key* and *meld* often exist [4,7].

Priority queues have many practical applications including event-driven simulation, job scheduling on a shared computer, and computation of shortest paths, minimum spanning forests, minimum cost matching, optimum branching, etc. [4,7].

A priority queue can trivially be used for sorting by first inserting all keys to be sorted into the priority queue and then by repeatedly extracting the current minimum. The major contribution in [15] is a reduction showing that the converse is also true. The results in [15] imply that priority queues are computationally equivalent to sorting, that is, asymptotically, the per key cost of sorting is the update time of a priority queue.

A result similar to those in [15] was presented in [14] which resulted in monotone priority queues (i. e., meaning that the extracted minimums are non-decreasing) with amortized time bounds only. In contrast, general priority queues with worst-case bounds are constructed in [15].

In addition to establishing the equivalence between priority queues and sorting, the reductions in [15] are also used to translate several known sorting results into new results on priority queues.

Background

Some relevant background information is summarized below which will be useful in understanding the key results in Sect. "Key Results".

- A standard *word RAM* models what one programs in a standard programming language such as C. In addition to direct and indirect addressing and conditional jumps, there are functions, such as addition and multiplication, operating on a constant number of words. The memory is divided into words, addressed linearly starting from 0. The running time of a program is the number of instructions executed and the space is the maximal address used. The word-length is a machine dependent parameter which is big enough to hold a key, and at least logarithmic in the number of input keys so that they can be addressed.
- A pointer machine is like the word RAM except that addresses cannot be manipulated.
- The AC^0 complexity class consists of constant-depth circuits with unlimited fan-in [18]. Standard AC^0 operations refer to the operations available via C, but where the functions on words are in AC^0. For example, this includes addition but not multiplication.
- Integer keys will refer to non-negative integers. However, if the input keys are signed integers, the correct ordering of the keys is obtained by flipping their sign bits and interpreting them as unsigned integers. Similar tricks work for floating point numbers and integer fractions [14].
- The atomic heaps of Fredman and Willard [6] are used in one of the reductions in [15]. These heaps can support updates and searches in sets of $O(\log^2 n)$ keys in $O(1)$ worst-case time [19]. However, atomic heaps use multiplication operations which are not in AC^0.

Key Results

The main results in this paper are two reductions from priority queues to sorting. The stronger of the two, stated

in Theorem 1, is for integer priority queues running on a standard word RAM.

Theorem 1 *If for some non-decreasing function S, up to n integer keys can be sorted in S(n) time per key, an integer priority queue can be implemented supporting find-min in constant time, and updates, i. e., insert and delete, in $S(n) + O(1)$ time. Here n is the current number of keys in the queue. The reduction uses linear space. The reduction runs on a standard word RAM assuming that each integer key is contained in a single word.*

The reduction above provides the following new bounds for linear space integer priority queues improving previous bounds in [8,14] and [5], respectively.

1. **(Deterministic)** $O(\log\log n)$ update time using a sorting algorithm in [9].
2. **(Randomized)** $O\left(\sqrt{\log\log n}\right)$ expected update time using the sorting algorithm in [10].
3. **(Randomized with $O(1)$ *decrease-key*)** $O\left((\log n)^{\frac{1}{(2-\epsilon)}}\right)$ expected update time for word length $\geq \log n$ and any constant $\epsilon > 0$, using the sorting algorithm in [3].

The reduction in Theorem 1 employs atomic heaps [6] which in addition to being very complicated, use non-AC^0 operations. The following slightly weaker recursive reduction which does not restrict the domain of the keys is completely combinatorial.

Theorem 2 *Given a sorter that sorts up to n keys in S(n) time per key, a priority queue can be implemented supporting find-min in constant time, and updates in T(n) time where n is the current number of keys in the queue and T(n) satisfies the recurrence:*

$$T(n) = O\left(S(n) + T(\log^2 n)\right) .$$

The reduction runs on a pointer machine in linear space using only standard AC^0 operations. Key values are only accessed by the sorter.

This reduction implies the following new priority queue bounds not implied by Theorem 1, where the first two bounds improve previous bounds in [13] and [16], respectively.

1. **(Deterministic in Standard AC^0)** $O\left((\log\log n)^{1+\epsilon}\right)$ update time for any constant $\epsilon > 0$ using a standard AC^0 integer sorting algorithm in [10].
2. **(Randomized in Standard AC^0)** $O\left(\log\log n\right)$ expected update time using a standard AC^0 integer sorting algorithm in [16].
3. **(String of *l* Words)** $O(l + \log\log n)$ deterministic and $O\left(l + \sqrt{\log\log n}\right)$ randomized expected update time using the string sorting results in [10].

The Reduction in Theorem 1

Given a sorting routine that can sort up to n keys in $S(n)$ time per key, the priority queue is constructed as follows.

The data structure has two major components: a sorted list of keys and a set of update buffers. The key list is partitioned into small segments, each of which is maintained in an atomic heap allowing constant time update and search operations on that segment. Each update buffer has a different capacity and accumulates updates (*insert*/*delete*) with key values in a different range. Smaller update buffers accept updates with smaller keys. An atomic heap is used to determine in constant time which update buffer collects a new update. When an update buffer accumulates enough updates, they first enter a sorting phase and then a merging phase. In the merging phase each update is applied on the proper segment in the key list, and invariants on segment size and ranges of update buffers are fixed. These phases are not executed immediately, instead they are executed in fixed time increments over a period of time. An update buffer continues to accept new updates while some updates accepted by it earlier are still in the sorting phase, and some even older updates are in the merging phase. Every time it accepts a new update, $S(n)$ time is spent on the sorting phase associated with it, and $O(1)$ time on its merging phase. This strategy allows the sorting and merging phases to complete execution by the time the update buffer becomes full again, and thus keeping the movement of updates through different phases smooth while maintaining an $S(n) + O(1)$ worst-case time bound per update. Moreover, the size and capacity constraints ensure that the smallest key in the data structure is available in $O(1)$ time. More details are given below.

The Sorted Key List: The sorted key list contains most of the keys in the data structure including the minimum key, and is known as the *base list*. This list is partitioned into *base segments* containing $\Theta((\log n)^2)$ keys each. Keys in each segment are maintained in an atomic heap so that if a new update is known to apply to a particular segment it can be applied in $O(1)$ time. If a base segment becomes too large or too small, it is split or joined with an adjacent segment.

Update Buffers: The base segments are separated by *base splitters*, and $O(\log n)$ of them are chosen to become *top splitters* so that the number of keys in the base list be-

low the ith ($i > 0$) top splitter s_i is $\Theta\left(4^i(\log n)^2\right)$. These splitters are placed in an atomic heap. As the base list changes the top splitters are moved, as needed, in order to maintain their exponential distribution.

Associated with each top splitter s_i, $i > 1$, are three buffers: an *entrance*, a *sorter*, and a *merger*, each with capacity for 4^i keys. Top splitter s_i works in a cycle of 4^i steps. The cycle starts with an empty entrance, at most 4^i updates in the sorter, and a sorted list of at most 4^i updates in the merger. In each step one may accept an update for the entrance, spend $O(4^i) = S(n)$ time in the sorter, and $O(1)$ time in merging the sorted list in the merger with the $O(4^i)$ base splitters in $[s_{i-2}, s_{i+1})$ (assuming $s_0 = 0$, $s_{-1} = -\infty$) and scanning for a new s_i among them. The implementation guarantees that all keys in the buffers of s_i lie in $[s_{i-2}, s_{i+1})$. Therefore, after 4^i such steps, the sorted list is correctly merged with the base list, a new s_i is found, and a new sorted list is produced. The sorter then takes the role of the merger, the entrance becomes the sorter, and the empty merger becomes the new entrance.

Handling Updates: When a new update key k (*insert*/*delete*) is received, the atomic heap of top splitters is used to find in $O(1)$ time the s_i such that $k \in [s_{i-1}, s_i)$. If $k \in [s_0, s_1)$, its position is identified among the $O(1)$ base splitters below s_1, and the corresponding base segment is updated in $O(1)$ time using the atomic heap over the keys of that segment. If $k \in [s_{i-1}, s_i)$ for some $i > 1$, the update is placed in the entrance of s_i, performing one step of the cycle of s_i in $S(n) + O(1)$ time. Additionally, during each update another splitter s_r is chosen in a round-robin fashion, and a fraction $1/\log n$ of a step of a cycle of s_r is executed in $O(1)$ time. The work on s_r ensures that after every $O((\log n)^2)$ updates some progress is made on moving each top splitter.

A *find-min* returns the minimum element of the base list which is available in $O(1)$ time.

The Reduction in Theorem 2

This reduction follows from the previous reduction by replacing all atomic heaps by the buffer systems developed for the top splitters.

Further Improvement

In [1] Alstrup et al. present a general reduction that transforms a priority queue to support *insert* in $O(1)$ time while keeping the other bounds unchanged. This reduction can be used to reduce the cost of insertion to a constant in Theorems 1 and 2.

Applications

The equivalence results in [15] can be used to translate known sorting results into new results on priority queues for integers and strings in different computational models (see Sect. "Key Results"). These results can also be viewed as a new means of proving lower bounds for sorting via priority queues.

A new RAM priority queue that matches the bounds in Theorem 1 and also supports *decrease-key* in $O(1)$ time is presented in [17]. This construction combines Andersson's exponential search trees [2] with the priority queues implied by Theorem 1. The reduction in Theorem 1 is also used in [12] in order to develop an adaptive integer sorting algorithm for the word RAM. Reductions from meldable priority queues to sorting presented in [11] use the reductions from non-meldable priority queues to sorting given in [15].

Open Problems

As noted before, the combinatorial reduction for pointer machines given in Theorem 2 is weaker than the word RAM reduction. For example, for a hypothetical linear time sorting algorithm, Theorem 1 implies a priority queue with an update time of $O(1)$ while Theorem 2 implies $2^{O(\log^* n)}$-time updates. Whether this gap can be reduced or removed is still an open question.

Cross References

- ► Cache-Oblivious Sorting
- ► External Sorting and Permuting
- ► String Sorting
- ► Suffix Tree Construction in RAM

Recommended Reading

1. Alstrup, S., Husfeldt, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues (note). ACM TALG **1**(1), 102–106 (2005)
2. Andersson, A.: Faster deterministic sorting and searching in linear space. In: Proc. 37th FOCS, 1998, pp. 135–141
3. Andersson, A., Hagerup, T., Nilsson, S., Raman, R.: Sorting in linear time? J. Comp. Syst. Sci. **57**, 74–93 (1998). Announced at STOC'95
4. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge, MA (2001)
5. Fredman, M., Willard, D.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**, 424–436 (1993). Announced at STOC'90
6. Fredman, M., Willard, D.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. **48**, 533–551 (1994)

7. Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
8. Han, Y.: Improved fast integer sorting in linear space. Inf. Comput. **170**(8), 81–94 (2001). Announced at STACS'00 and SODA'01
9. Han, Y.: Deterministic sorting in $O(n \log\log n)$ time and linear space. J. Algorithms **50**(1), 96–105 (2004). Announced at STOC'02
10. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proc. 43rd FOCS, 2002, pp. 135–144
11. Mendelson, R., Tarjan, R., Thorup, M., Zwick, U.: Melding priority queues. ACM TALG **2**(4), 535–556 (2006). Announced at SODA'04
12. Pagh, A., Pagh, R., Thorup, M.: On adaptive integer sorting. In: Proc. 12th ESA, 2004, pp. 556–579
13. Thorup, M.: Faster deterministic sorting and priority queues in linear space. In: Proc. 9th SODA, 1998, pp. 550–555
14. Thorup, M.: On RAM priority queues. SIAM J. Comput. **30**(1), 86–109 (2000). Announced at SODA'96
15. Thorup, M.: Equivalence between priority queues and sorting. In: Proc. 43rd FOCS, 2002, pp. 125–134
16. Thorup, M.: Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. J. Algorithms **42**(2), 205–230 (2002). Announced at SODA'97
17. Thorup, M.: Integer priority queues with decrease key in constant time and the single source shortest paths problem. J. Comput. Syst. Sci. (special issue on STOC'03) **69**(3), 330–353 (2004)
18. Vollmer, H.: Introduction to circuit complexity: a uniform approach. Springer, New York (1999)
19. Willard, D.: Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. SIAM J. Comput. **29**(3), 1030–1049 (2000). Announced at SODA'92

Error-Control Codes, Reed–Muller Code

▶ Learning Heavy Fourier Coefficients of Boolean Functions

Error Correction

▶ Decoding Reed–Solomon Codes
▶ List Decoding near Capacity: Folded RS Codes
▶ LP Decoding

Euclidean Graphs and Trees

▶ Minimum Geometric Spanning Trees
▶ Minimum k-Connected Geometric Networks

Euclidean Traveling Salesperson Problem

1998; Arora

Artur Czumaj
DIMAP and Computer Science, University of Warwick, Coventry, UK

Keywords and Synonyms

Euclidean TSP; Geometric TSP; Geometric traveling salesman problem

Problem Definition

This entry considers geometric optimization $\mathcal{NP}$-hard problems like the Euclidean Traveling Salesperson problem and the Euclidean Steiner Tree problem. These problems are geometric variants of standard graph optimization problems, and the restriction of the input instances to geometric or Euclidean case arise in numerous applications (see [1,2]). The main focus of this chapter is the Euclidean Traveling Salesperson problem.

Notation

The *Euclidean Traveling Salesperson Problem (TSP)*: For a given set S of n points in the Euclidean space $\mathbb{R}^d$, find a path of minimum length that visits each point exactly once.

The cost $\delta(x, y)$ of an edge connecting a pair of points $x, y \in \mathbb{R}^d$ is equal to the Euclidean distance between points x and y. That is, $\delta(x, y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$, where $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$. More generally, the distance can be defined using other norms, such as ℓ_p norms for any $p > 1$, $\delta(x, y) = \left(\sum_{i=1}^{d}(x_i - y_i)^p\right)^{1/p}$.

For a given set S of points in the Euclidean space $\mathbb{R}^d$, for an integer d, $d \geq 2$, an *Euclidean graph* (network) is a graph $G = (S, E)$, where E is the set of straight-line segments connecting pairs of points in S. If all pairs of points in S are connected by edges in E, then G is called a *complete Euclidean graph on S*. The cost of the graph is equal to the sum of the costs of the edges of the graph: $\text{cost}(G) = \sum_{(x,y)\in E} \delta(x, y)$.

A *polynomial-time approximation scheme (PTAS)* is a family of algorithms $\{\mathcal{A}_\varepsilon\}$ such that, for each fixed $\varepsilon > 0$, $\mathcal{A}_\varepsilon$ runs in a time which is a polynomial of the size of the input, and produces a $(1 + \varepsilon)$-approximation.

Related work

The classical book by Lawler et al. [12] provides extensive information about the TSP. Also, the survey exposition of Bern and Eppstein [7] presents state of the art research done on the geometric TSP up to 1995, and the survey of Arora [2] discusses the research done after 1995.

Key Results

In general graphs the TSP graph problem is well known to be $\mathcal{NP}$-hard, and the same claim holds for the Euclidean TSP problem [11], [14].

Theorem 1 *The Euclidean TSP problem is $\mathcal{NP}$-hard.*

Perhaps rather surprisingly, it is still not known if the decision version of the problem is $\mathcal{NP}$-complete [11]. (The decision version of the Euclidean TSP problem is for a given point set in the Euclidean space $\mathbb{R}^d$ and a number t, verify if there is a simple path of length smaller than t that visits each point exactly once.)

The approximability of TSP has been studied extensively over the last few decades. It is not hard to see that TSP is not approximable in polynomial-time (unless $\mathcal{P} = \mathcal{NP}$) for arbitrary graphs with arbitrary edge costs. When the weights satisfy the triangle inequality (the so called *metric TSP*), there is a polynomial-time 3/2-approximation algorithm due to Christofides [8], and it is known that no PTAS exists (unless $\mathcal{P} = \mathcal{NP}$). This result has been strengthened by Trevisan [17] to include Euclidean graphs in high-dimensions (the same result holds also for any ℓ_p metric).

Theorem 2 (Trevisan [17]) *If $d \geq \log n$, then there exists a constant $\epsilon > 0$ such that the Euclidean TSP problem in $\mathbb{R}^d$ is $\mathcal{NP}$-hard to approximate within a factor of $1 + \epsilon$.*

In particular, this result implies that if $d \geq \log n$, then the Euclidean TSP problem in $\mathbb{R}^d$ has no PTAS unless $\mathcal{P} = \mathcal{NP}$.

The same result also holds for any ℓ_p metric. Furthermore, Theorem 2 implies that the Euclidean TSP in $\mathbb{R}^{\log n}$ is APX PB-hard under E-reductions and APX-complete under AP-reductions.

It was believed that Theorem 2 might hold for smaller values of d, in particular even for $d = 2$, but this has been disproved independently by Arora [1] and Mitchell [13].

Theorem 3 (Arora [1], Mitchell [13]) *The Euclidean TSP on the plane has a PTAS.*

The main idea of the algorithms of Arora and Mitchell is rather simple, but the details of the analysis are quite complicated. Both algorithms follow the same approach. First, one proves a so-called *Structure Theorem*. This demonstrates that there is a $(1 + \epsilon)$-approximation that has some local properties. In the case of the Euclidean TSP problem, there is a quadtree partition of the space containing all the points, such that each cell of the quadtree is crossed by the tour at most a constant number of times, and only in some pre-specified locations. After proving the Structure Theorem, one uses dynamic programming to find an optimal (or almost optimal) solution that obeys the local properties specified in the Structure Theorem.

The original algorithms presented in the first conference version of [1] and in the early version of [13] have running times of the form $\mathcal{O}(n^{1/\epsilon})$ to obtain a $(1 + \epsilon)$-approximation, but this has been subsequently improved. In particular, Arora's randomized algorithm in [1] runs in time $\mathcal{O}(n(\log n)^{1/\epsilon})$, and it can be derandomized with a slow-down of $\mathcal{O}(n)$. The result from Theorem 3 can be also extended to higher dimensions. Arora shows the following result.

Theorem 4 (Arora [1]) *For every constant d, the Euclidean TSP in $\mathbb{R}^d$ has a PTAS.*

For every fixed $c > 1$ and given any n nodes in $\mathbb{R}^d$, there is a randomized algorithm that finds a $(1 + 1/c)$-approximation of the optimum traveling salesman tour in $\mathcal{O}(n\,(\log n)^{(\mathcal{O}(\sqrt{d}c))^{d-1}})$ time. In particular, for any constant d and c, the running time is $\mathcal{O}(n\,(\log n)^{\mathcal{O}(1)})$. The algorithm can be derandomized by increasing the running time by a factor of $\mathcal{O}(n^d)$.

This was later extended by Rao and Smith [15], who proved the following theorem.

Theorem 5 (Rao and Smith [15]) *There is a deterministic algorithm that computes a $(1 + 1/c)$-approximation of the optimum traveling salesman tour in $\mathcal{O}(2^{(cd)^{\mathcal{O}(d)}}\, n + (cd)^{\mathcal{O}(d)}\, n \log n)$ time.*

There is also a randomized Monte Carlo algorithm that succeeds with probability at least 1/2 and that computes a $(1 + 1/c)$-approximation of the optimum traveling salesman tour in the expected $(c\sqrt{d})^{\mathcal{O}(d(c\sqrt{d})^{d-1})}\, n + \mathcal{O}(d\, n \log n)$ time.

In the special and most interesting case, when $d = 2$, Rao and Smith show the following.

Theorem 6 (Rao and Smith [15]) *There is a deterministic algorithm that computes a $(1 + 1/c)$-approximation of the optimum traveling salesman tour in $\mathcal{O}(n\, 2^{c^{\mathcal{O}(1)}} + c^{\mathcal{O}(1)}\, n \log n)$ time.*

There is a randomized Monte Carlo algorithm (which fails with probability smaller than 1/2) that computes

a $(1 + 1/c)$*-approximation of the optimum traveling salesman tour in the expected* $O(n\, 2^{c^{O(1)}} + n \log n)$ *time.*

Applications

The techniques developed by Arora [1] and Mitchell [13] found numerous applications in the design of polynomial-time approximation schemes for geometric optimization problems.

Euclidean Minimum Steiner Tree Problem For a given set S of n points in the Euclidean space $\mathbb{R}^d$, find the minimum cost network connecting all the points in S (where the cost of a network is equal to the sum of the lengths of the edges defining it).

Theorem 7 ([1], [15]) *For every constant d, the Euclidean Minimum Steiner tree problem in* $\mathbb{R}^d$ *has a PTAS.*

Euclidean *k*-median Problem For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find k medians among the points in S so that the sum of the distances from each point in S to its closest median is minimized.

Theorem 8 ([5]) *For every constant d, the Euclidean k-median problem in* $\mathbb{R}^d$ *has a PTAS.*

Euclidean *k*-TSP Problem For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the shortest tour that visits at least k points in S.

Euclidean *k*-MST Problem For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the shortest tree that contains at least k points from S.

Theorem 9 ([1]) *For every constant d, the Euclidean k-TSP and the Euclidean k-MST problems in* $\mathbb{R}^d$ *have a PTAS.*

Euclidean Minimum-cost *k*-connected Subgraph Problem For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the minimum-cost subgraph (of the complete graph on S) that is k-connected

Theorem 10 ([9]) *For every constant d and constant k, the Euclidean minimum-cost k-connected subgraph problem in* $\mathbb{R}^d$ *has a PTAS.*

The technique developed by Arora [1] and Mitchell [13] also led to some quasi-polynomial-time approximation schemes, that is, the algorithms with the running time of $n^{O(\log n)}$. For example, Arora and Karakostas [4] gave a quasi-polynomial-time approximation scheme for the Euclidean minimum latency problem, and Remy and Steger [16] gave a quasi-polynomial-time approximation scheme for the minimum-weight triangulation problem.

For more discussion, see the survey by Arora [2] and [10].

Extensions to Planar Graphs

The dynamic programming approach used by Arora [1] and Mitchell [13] is also related to the recent advances for a number of optimization problems for planar graphs. For example, Arora et al. [3] designed a PTAS for the TSP problem in weighted planar graphs, and there is a PTAS for the problem of finding a minimum-cost spanning 2-connected subgraph of a planar graph [6].

Open Problems

One interesting open problem is if the quasi-polynomial-time approximation schemes mentioned above (for the minimum latency and the minimum-weight triangulation problems) can be extended to obtain polynomial-time approximation schemes. For more open problems, see Arora [2].

Cross References

- ▶ Metric TSP
- ▶ Minimum k-Connected Geometric Networks
- ▶ Minimum Weight Triangulation

Recommended Reading

1. Arora, S.: Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J. ACM **45**(5), 753–782 (1998)
2. Arora, S.: Approximation schemes for $\mathcal{NP}$-hard geometric optimization problems: A survey. Math. Program. Ser. B **97**, 43–69 (2003)
3. Arora, S., Grigni, M., Karger, D., Klein, P., Woloszyn, A..: A polynomial time approximation scheme for weighted planar graph TSP. In: Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 33–41
4. Arora, S., Karakostas, G.: Approximation schemes for minimum latency problems. In: Proc. 31st Annual ACM Symposium on Theory of Computing, 1999, pp. 688–693
5. Arora, S., Raghavan, P., Rao, S.: Approximation schemes for Euclidean *k*-medians and related problems. In: Proc. 30th Annual ACM Symposium on Theory of Computing, 1998, pp. 106–113
6. Berger, A., Czumaj, A., Grigni, M., Zhao, H.: Approximation schemes for minimum 2-connected spanning subgraphs in weighted planar graphs. In: Proc. 13th Annual European Symposium on Algorithms, 2005, pp. 472–483
7. Bern, M., Eppstein, D.: Approximation algorithms for geometric problems. In: Hochbaum, D. (ed.) Approximation Algorithms for NP-hard problems. PWS Publishing, Boston (1996)

8. Christofides, N.: Worst-case analysis of a new heuristic for the traveling salesman problem. In: Technical report, Graduate School of Industrial Administration. Carnegie-Mellon University, Pittsburgh (1976)
9. Czumaj, A., Lingas, A.: On approximability of the minimum-cost *k*-connected spanning subgraph problem. In: Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 281–290
10. Czumaj, A., Lingas, A.: Approximation schemes for minimum-cost *k*-connectivity problems in geometric graphs. In: Gonzalez, T.F. (eds.) Handbook of Approximation Algorithms and Metaheuristics. CRC Press, Boca Raton (2007)
11. Garey, M.R., Graham, R.L., Johnson, D.S.: Some NP-complete geometric problems. In: Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 10–22
12. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, Chichester (1985)
13. Mitchell, J.S.B.: Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, *k*-MST, and related problems. SIAM J. Comput. **28**(4), 1298–1309 (1999)
14. Papadimitriou, C.H.: Euclidean TSP is NP-complete. Theor. Comput. Sci. **4**, 237–244 (1977)
15. Rao, S.B., Smith, W.D.: Approximating geometrical graphs via "spanners" and "banyans". In: Proc. 30th Annual ACM Symposium on Theory of Computing, 1998, pp. 540–550
16. Remy, J., Steger, A.: A quasi-polynomial time approximation scheme for minimum weight triangulation. In: Proc. 38th Annual ACM Symposium on Theory of Computing, 2006
17. Trevisan, L.: When Hamming meets Euclid: the approximability of geometric TSP and Steiner Tree. SIAM J. Comput. **30**(2), 475–485 (2000)

Exact Algorithms for Dominating Set

2005; Fomin, Grandoni, Kratsch

DIETER KRATSCH
UFM MIM – LITA, Paul Verlaine University, Metz, France

Keywords and Synonyms

Connected dominating set

Problem Definition

The dominating set problem is a classical NP-hard optimization problem which fits into the broader class of covering problems. Hundreds of papers have been written on this problem that has a natural motivation in facility location.

Definition 1 For a given undirected, simple graph $G = (V, E)$ a subset of vertices $D \subseteq V$ is called a *dominating set* if every vertex $u \in V - D$ has a neighbor in D. The minimum dominating set (MDS) problem is to find a *minimum dominating set* of G, i. e. a dominating set of G of minimum cardinality.

Problem 1 (MDS)
Input: *Undirected simple graph* $G = (V, E)$.
Output: *A minimum dominating set* D *of* G.

Various modifications of the dominating set problem are of interest, some of them obtained by putting additional constraints on the dominating set such as, for example, requesting it to be an independent set or to be connected. In graph theory there is a huge literature on domination dealing with the problem and its many modifications (see e. g.[9]). In graph algorithms the MDS problem and some of its modifications like Independent Dominating Set and Connected Dominating Set have been studied as benchmark problems for attacking NP-hard problems under various algorithmic approaches.

Known Results

The algorithmic complexity of MDS and its modifications when restricted to inputs from a particular graph class has been studied extensively (see e. g. [10]). Among others, it is known that MDS remains NP-hard on bipartite graphs, split graphs, planar graphs and graphs of maximum degree three. Polynomial time algorithms to compute a minimum dominating set are known, for example, for permutation, interval and k-polygon graphs. There is also a $O(4^k n^{O(1)})$ time algorithm to solve MDS on graphs of treewidth at most k.

The dominating set problem is one of the basic problems in parameterized complexity [3]; it is W[2]-complete and thus it is unlikely that the problem is fixed parameter tractable. On the other hand, the problem is fixed parameter tractable on planar graphs. Concerning approximation, MDS is equivalent to MINIMUM SET COVER under L-reductions. There is an approximation algorithm solving MDS within a factor of $1 + \ln |V|$ and it cannot be approximated within a factor of $(1 - \epsilon) \ln |V|$ for any $\epsilon > 0$, unless NP $\subset$ DTIME($n^{\log \log n}$) [1].

Moderately Exponential Time Algorithms

If P $\neq$ NP then no polynomial time algorithm can solve MDS. Even worse, it has been observed in [7] that unless SNP$\subseteq$ SUBEXP (which is considered to be highly unlikely), there is not even a subexponential time algorithm solving the dominating set problem.

The trivial $O(2^n (n+m))$ algorithm, which simply checks all the 2^n vertex subsets as to whether they are dominating, clearly solves MDS. Three faster algorithms

were established in 2004. The algorithm of Fomin et al. [7] uses a deep graph-theoretic result due to B. Reed, stating that every graph on n vertices with minimum degree at least three has a dominating set of size at most $3n/8$, to establish an $O(2^{0.955n})$ time algorithm solving MDS. The $O(2^{0.919n})$ time algorithm of Randerath and Schiermeyer [11] uses very nice ideas including matching techniques to restrict the search space. Finally, Grandoni [8] established an $O(2^{0.850n})$ time algorithm to solve MDS.

The work of Fomin, Grandoni, and Kratsch [5] presents a simple and easy to implement recursive branch & reduce algorithm to solve MDS. The running time of the algorithm is significantly faster than the ones stated for previous algorithms. This is heavily based on the analysis of the running time by measure & conquer, which is a method to analyze the worst case running time of (simple) branch & reduce algorithms based on a sophisticated choice of the measure of a problem instance.

Key Results

Theorem 1 *There is a branch & reduce algorithm solving* MDS *in time* $O(2^{0.610n})$ *using polynomial space.*

Theorem 2 *There is an algorithm solving* MDS *in time* $O(2^{0.598n}$ *using exponential space.*

The algorithms of Theorem 1 and 2 are simple consequences of a transformation from MDS to Minimum Set Cover (MSC) combined with new moderately exponential time algorithms for MSC.

Problem 2 (MSC)
Input: *Finite set* $\mathcal{U}$ *and a collection* S *of subsets* $S_1, S_2, \ldots S_t$ *of* $\mathcal{U}$.
Output: *A minimum set cover* S', *where* $S' \subseteq S$ *is a set cover of* $(\mathcal{U}, S)$ *if* $\bigcup_{S_i \in S'} S_i = \mathcal{U}$.

Theorem 3 *There is a branch & reduce algorithm solving* MSC *in time* $O(2^{0.305(|\mathcal{U}|+|S|)})$ *using polynomial space.*

Applying memorization to the polynomial space algorithm of Theorem 3 the running time can be improved as follows.

Theorem 4 *There is an algorithm solving* MSC *in time* $O(2^{0.299(|S|+|\mathcal{U}|)})$ *using exponential space.*

The analysis of the worst case running time of the simple branch & reduce algorithm solving MSC (of Theorem 3) is done by a careful choice of the measure of a problem instance which allows one to obtain an upper bound that is significantly smaller than the one that could be obtained using the standard measure. The refined analysis leads to a collection of recurrences. Then random local search is used to compute the weights, used in the definition of the measure, aiming at the best achievable upper bound of the worst-case running time.

Since current tools to analyze the worst-case running time of branch & reduce algorithms do not seem to produce tight upper bounds, exponential lower bounds of the worst-case running time of the algorithm are of interest.

Theorem 5 *The worst-case running time of the branch & reduce algorithm solving* MDS *(see Theorem 1) is* $\Omega(2^{n/3})$.

Applications

There are various other NP-hard domination-type problems that can be solved by a moderately exponential time algorithm based on an algorithm solving Minimum Set Cover: any instance of the initial problem is transformed to an instance of MSC (preferably with $|\mathcal{U}| = |S|$), and then the algorithm of Theorem 3 or 4 is used to solve MSC and thus the initial problem. Examples of such problems are Total Dominating Set, k-Dominating Set, k-Center and MDS on split graphs.

Measure & Conquer and the strongly related quasi-convex analysis of Eppstein [4] have been used to design and analyze a variety of moderately exponential time algorithms for NP-hard problems: optimization, counting and enumeration problems. See for example [2,6].

Open Problems

A number of problems related to the work of Fomin, Grandoni, and Kratsch remain open. Although for various graph classes there are algorithms to solve MDS which are faster than the one for general graphs (of Theorem 1 and 2), no such algorithm is known for solving MDS on bipartite graphs.

The worst-case running times of simple branch & reduce algorithms like those solving MDS and MSC remain unknown. In the case of the polynomial space algorithm solving MDS there is a large gap between the $O(2^{0.610n})$ upper bound and the $\Omega(2^{n/3})$ lower bound. The situation is similar for other branch & reduce algorithms. Consequently, there is a strong need for new and better tools to analyze the worst-case running time of branch & reduce algorithms.

Cross References

- ► Connected Dominating Set
- ► Data Reduction for Domination in Graphs
- ► Vertex Cover Search Trees

Recommended Reading

1. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccalema, A., Protasi, M.: Complexity and Approximation. Springer, Berlin (1999)
2. Byskov, J.M.: Exact algorithms for graph colouring and exact satisfiability. Ph. D. thesis, University of Aarhus, Denmark (2004)
3. Downey, R.G., Fellows, M.R.: Parameterized complexity. Springer, New York (1999)
4. Eppstein, D.: Quasiconvex analysis of backtracking algorithms. In: Proceedings of SODA 2004, pp. 781–790
5. Fomin, F.V., Grandoni, F., Kratsch, D.: Measure and conquer: Domination – A case study. In: Proceedings of ICALP 2005. LNCS, vol. 3380, pp. 192–203. Springer, Berlin (2005)
6. Fomin, F.V., Grandoni, F., Kratsch, D.: Measure and Conquer: A simple $O(2^{0.288n})$ Independent Set Algorithm. In: Proceedings of SODA 2006, pp. 18–25
7. Fomin, F.V., Kratsch, D., Woeginger, G.J.: Exact (exponential) algorithms for the dominating set problem. In: Proceedings of WG 2004. LNCS, vol. 3353, pp. 245–256. Springer, Berlin (2004)
8. Grandoni, F.: Exact Algorithms for Hard Graph Problems. Ph. D. thesis, Università di Roma "Tor Vergata", Roma, Italy (2004)
9. Haynes, T.W., Hedetniemi, S.T., Slater, P.J.: Fundamentals of domination in graphs. Marcel Dekker, New York (1998)
10. Kratsch, D.: Algorithms. In: Haynes, T., Hedetniemi, S., Slater, P. (eds.) Domination in Graphs: Advanced Topics, pp. 191–231. Marcel Dekker, New York (1998)
11. Randerath, B., Schiermeyer, I.: Exact algorithms for MINIMUM DOMINATING SET. Technical Report, zaik-469, Zentrum für Angewandte Informatik Köln (2004)
12. Woeginger, G.J.: Exact algorithms for NP-hard problems: A survey. In: Combinatorial Optimization – Eureka, You Shrink. LNCS, vol. 2570, pp. 185–207. Springer, Berlin (2003)

Exact Algorithms for General CNF SAT

1998; Hirsch
2003; Schuler

EDWARD A. HIRSCH
Laboratory of Mathematical Logic, Steklov Institute of Mathematics at St. Petersburg, St. Petersburg, Russia

Keywords and Synonyms

SAT; Boolean satisfiability

Problem Definition

The satisfiability problem (*SAT*) for Boolean formulas in conjunctive normal form (*CNF*) is one of the first **NP**-complete problems [2,13]. Since its **NP**-completeness currently leaves no hope for polynomial-time algorithms, the progress goes by decreasing the exponent. There are several versions of this parametrized problem that differ in the parameter used for the estimation of the running time.

Problem 1 (SAT) INPUT: *Formula F in CNF containing n variables, m clauses, and l literals in total.*

OUTPUT: *"Yes" if F has a* satisfying assignment, *i. e., a substitution of Boolean values for the variables that makes F true. "No" otherwise.*

The bounds on the running time of SAT algorithms can be thus given in the form $|F|^{O(1)} \cdot \alpha^n$, $|F|^{O(1)} \cdot \beta^m$, or $|F|^{O(1)} \cdot \gamma^l$, where $|F|$ is the length of a reasonable bit representation of F (i. e., the formal input to the algorithm). In fact, for the present algorithms the bases β and γ are constants while α is a function $\alpha(n, m)$ of the formula parameters (because no better constant than $\alpha = 2$ is known).

Notation

A formula in conjunctive normal form is a set of clauses (understood as the conjunction of these clauses), a clause is a set of literals (understood as the disjunction of these literals), and a literal is either a Boolean variable or the negation of a Boolean variable. A truth assignment assigns Boolean values (`false` or `true`) to one or more variables. An assignment is abbreviated as the list of literals that are made true under this assignment (for example, assigning `false` to x and `true` to y is denoted by $\neg x, y$). The result of the application of an assignment A to a formula F (denoted $F[A]$) is the formula obtained by removing the clauses containing the true literals from F and removing the falsified literals from the remaining clauses. For example, if $F = (x \vee \neg y \vee z) \wedge (y \vee \neg z)$, then $F[\neg x, y] = (z)$. A *satisfying* assignment for F is an assignment A such that $F[A] =$ `true`. If such an assignment exists, F is called *satisfiable*.

Key Results

Bounds for β and γ

General Approach and a Bound for β The trivial brute-force algorithm enumerating all possible assignments to the n variables runs in 2^n polynomial-time steps. Thus $\alpha \leq 2$, and by trivial reasons also $\beta, \gamma \leq 2$. In the early 1980s Monien and Speckenmeyer noticed that β could be made smaller[1]. Then Kullmann and Luckhardt [12] set up a framework for divide-and-conquer[2] algorithms for SAT that split the original problem into several (yet usu-

[1] They and other researchers also noticed that α could be made smaller for a special case of the problem where the length of each clause is bounded by a constant; the reader is referred to another entry (*Local search algorithms for k-SAT*) of the *Encyclopedia* for relevant references and algorithms.

[2] Also called *DPLL* due to the papers of Davis and Putnam [7] and Davis, Logemann, and Loveland [6].

ally a constant number of) subproblems by substituting the values of some variables and simplifying the obtained formulas. This line of research resulted in the following upper bounds for β and γ:

Theorem 1 (Hirsch [8]) *SAT can be solved in time*
1. $|F|^{O(1)} \cdot 2^{0.30897m}$;
2. $|F|^{O(1)} \cdot 2^{0.10299l}$.

A typical divide-and-conquer algorithm for SAT consists of two phases: *splitting* of the original problem into several subproblems (for example, reducing *SAT(F)* to *SAT(F[x])* and *SAT*$(F[\neg x])$) and *simplification* of the obtained subproblems using polynomial-time transformation rules that do not affect the satisfiability of the subproblems (i. e., they replace a formula by an *equi-satisfiable* one). The subproblems $F_1, \ldots, F_k$ for splitting are chosen so that the corresponding recurrent inequality using the simplified problems $F'_1, \ldots, F'_k$,

$$T(F) \leq \sum_{i=1}^{k} T(F'_i) + \text{const}\,,$$

gives a desired upper bound on the number of leaves in the recurrence tree and, hence, on the running time of the algorithm. In particular, in order to obtain the bound $|F|^{O(1)} \cdot 2^{0.30897m}$ one takes either two subproblems $F[x], F[\neg x]$ with recurrent inequality

$$t_m \leq t_{m-3} + t_{m-4}$$

or four subproblems $F[x, y], F[x, \neg y], F[\neg x, y], F[\neg x, \neg y]$ with recurrent inequality

$$t_m \leq 2t_{m-6} + 2t_{m-7}$$

where $t_i = \max_{m(G) \leq i} T(G)$. The simplification rules used in the $|F|^{O(1)} \cdot 2^{0.30897m}$-time and the $|F|^{O(1)} \cdot 2^{0.10299l}$-time algorithms are as follows.

Simplification Rules

Elimination of 1-clauses If F contains a 1-clause (a), replace F by $F[a]$.

Subsumption If F contains two clauses C and D such that $C \subseteq D$, replace F by $F \setminus \{D\}$.

Resolution with Subsumption Suppose a literal a and clauses C and D are such that a is the only literal satisfying both conditions $a \in C$ and $\neg a \in D$. In this case, the clause $(C \cup D) \setminus \{a, \neg a\}$ is called the *resolvent by the literal a* of the clauses C and D and denoted by $R(C, D)$.

The rule is: if $R(C, D) \subseteq D$, replace F by $(F \setminus \{D\}) \cup \{R(C, D)\}$.

Elimination of a Variable by Resolution [7] Given a literal a, construct the formula $\mathrm{DP}_a(F)$ by
1. adding to F all resolvents by a;
2. removing from F all clauses containing a or $\neg a$.

The rule is: if $\mathrm{DP}_a(F)$ is not larger in m (resp., in l) than F, then replace F by $\mathrm{DP}_a(F)$.

Elimination of Blocked Clauses A clause C is *blocked* for a literal a w.r.t. F if C contains the literal a, and the literal $\neg a$ occurs only in the clauses of F that contain the negation of at least one of the literals occurring in $C \setminus \{a\}$. For a CNF-formula F and a literal a occurring in it, the assignment $I(a, F)$ is defined as

$$\{a\} \cup \{\text{literals } x \notin \{a, \neg a\} \mid \text{ the clause } \{\neg a, x\} \text{ is blocked for } \neg a \text{ w.r.t. } F\}\,.$$

Lemma 2 (Kullmann [11])

(1) *If a clause C is blocked for a literal a w.r.t. F, then F and* $F \setminus \{C\}$ *are equi-satisfiable.*
(2) *Given a literal a, the formula F is satisfiable iff at least one of the formulas* $F[\neg a]$ *and* $F[I(a, F)]$ *is satisfiable.*

The first claim of the lemma is employed as a simplification rule.

Application of the Black and White Literals Principle Let P be a binary relation between literals and formulas in CNF such that for a variable v and a formula F, at most one of $P(v, F)$ and $P(\neg v, F)$ holds.

Lemma 3 *Suppose that each clause of F that contains a literal w satisfying* $P(w, F)$ *contains also at least one literal b satisfying* $P(\neg b, F)$. *Then F and* $F[\{l | P(\neg l, F)\}]$ *are equi-satisfiable.*

A Bound for γ To obtain the bound $|F|^{O(1)} \cdot 2^{0.10299l}$, it is enough to use a pair $F[\neg a], F[I(a, F)]$ of subproblems (see Lemma 2(2)) achieving the desired recurrent inequality $t_l \leq t_{l-5} + t_{l-17}$ and to switch to the $|F|^{O(1)} \cdot 2^{0.30897m}$-time algorithm if there are none. A recent (much more technically involved) improvement to this algorithm [16] achieves the bound $|F|^{O(1)} \cdot 2^{0.0926l}$.

A Bound for α

Currently, no non-trivial constant upper bound for α is known. However, starting with [14] there was an interest to non-constant bounds. A series of randomized and deterministic algorithms showing successive improvements was developed, and at the moment the best possible bound

is achieved by a deterministic divide-and-conquer algorithm employing the following recursive procedure. The idea behind it is a dichotomy: either each clause of the input formula can be shortened to its first k literals (then a k-CNF algorithm can be applied), or all these literals in one of the clauses can be assumed false. (This clause-shortening approach can be attributed to Schuler [15] who used it in a randomized fashion. The following version of the deterministic algorithm achieving the best known bound both for deterministic and randomized algorithms appears in [5].)

`Procedure` S
`Input`: a CNF formula F and a positive integer k.

1. Assume F consists of clauses $C_1, \ldots, C_m$. Change each clause C_i to a clause D_i as follows: If $|C_i| > k$ then choose any k literals in C_i and drop the other literals; otherwise leave C_i as is, i. e., $D_i = C_i$. Let F' denote the resulting formula.
2. Test satisfiability of F' using the $m \cdot \mathrm{poly}(n) \cdot (2 - 2/(k+1))^n$-time k-CNF algorithm defined in [3].
3. If F' is satisfiable, output "satisfiable" and halt. Otherwise, for each i, do the following:
   (a) Convert F to F_i as follows:
      i. Replace C_j by D_j for all $j < i$;
      ii. Assign `false` to all literals in D_i.
   (b) Recursively invoke `Procedure` S on (F_i, k).
4. Return "unsatisfiable".

The algorithm just invokes `Procedure` S on the original formula and the integer parameter $k = k*(m,n)$. The most accurate analysis of this family of algorithms by Calabro, Impagliazzo, and Paturi [1] implies that, assuming that $m > n$, one can obtain the following bound by taking $k(m,n) = 2\log(m/n) + \text{const}$. (This explicit bound is not stated in [1] and is inferred in [4].)

Theorem 4 (Dantsin, Hirsch [4]) *Assuming $m > n$, SAT can be solved in time*

$$|F|^{O(1)} \cdot 2^{n\left(1-\frac{1}{O(\log(m/n))}\right)}.$$

Applications

While SAT has numerous applications, the presented algorithms have no direct effect on them.

Open Problems

Proving a constant upper bound on $\alpha < 2$ remains a major open problem in the field, as well as the hypothetic existence of $(1+\varepsilon)^l$-time algorithms for arbitrary small $\varepsilon > 0$.

It is possible to perform the analysis of a divide-and-conquer algorithm and even to generate simplification rules automatically [10]. However, this approach so far led to new bounds only for the (NP-complete) optimization version of 2-SAT [9].

Experimental Results

Jun Wang has implemented the algorithm yielding the bound on β and collected some statistics regarding the number of applications of the simplification rules [17].

Cross References

- ▶ Local Search Algorithms for kSAT
- ▶ Parameterized SAT

Recommended Reading

1. Calabro, C., Impagliazzo, R., Paturi, R.: A duality between clause width and clause density for SAT. In: Proceedings of the 21st Annual IEEE Conference on Computational Complexity (CCC 2006), pp. 252–260. IEEE Computer Society (2006)
2. Cook, S.A.: The Complexity of Theorem Proving Procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing, May 1971, pp. 151–158. ACM (2006)
3. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J., Papadimitriou, C., Raghavan, P., Schöning, U.: A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. Theor. Comput. Sci. **289**(1), 69–83 (2002)
4. Dantsin, E., Hirsch, E.A.: Worst-Case Upper Bounds. In: Biere, A., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. IOS Press (2008) To appear
5. Dantsin, E., Hirsch, E.A., Wolpert, A.: Clause shortening combined with pruning yields a new upper bound for deterministic SAT algorithms. In: Proceedings of CIAC-2006. Lecture Notes in Computer Science, vol. 3998, pp. 60–68. Springer, Berlin (2006)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**, 394–397 (1962)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**, 201–215 (1960)
8. Hirsch, E.A.: New worst-case upper bounds for SAT. J. Autom. Reason. **24**(4), 397–420 (2000)
9. Kojevnikov, A., Kulikov, A.: A New Approach to Proving Upper Bounds for MAX-2-SAT. Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), pp. 11–17. ACM, SIAM (2006)
10. Kulikov, A.: Automated Generation of Simplification Rules for SAT and MAXSAT. Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005). Lecture Notes in Computer Science, vol. 3569, pp. 430–436. Springer, Berlin (2005)
11. Kullmann, O.: New methods for 3-{SAT} decision and worst-case analysis. Theor. Comput. Sci. **223**(1–2):1–72 (1999)
12. Kullmann, O., Luckhardt, H.: Algorithms for SAT/TAUT decision based on various measures, preprint, 71 pages, http://cs-svr1.swan.ac.uk/csoliver/papers.html (1998)
13. Levin, L.A.: Universal Search Problems. Проблемы передачи информации **9**(3), 265–266, (1973). In Russian. English translation in: Trakhtenbrot, B.A.: A Survey of Russian Approaches to

Perebor (Brute-force Search) Algorithms. Annals of the History of Computing **6**(4), 384–400 (1984)
14. Pudlák, P.: Satisfiability – algorithms and logic. In: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS'98. Lecture Notes in Computer Science, vol. 1450, pp. 129–141. Springer, Berlin (1998)
15. Schuler, R.: An algorithm for the satisfiability problem of formulas in conjunctive normal form. J. Algorithms **54**(1), 40–44 (2005)
16. Wahlström, M.: An algorithm for the SAT problem for formulae of linear length. In: Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005. Lecture Notes in Computer Science, vol. 3669, pp. 107–118. Springer, Berlin (2005)
17. Wang, J.: Generating and solving 3-SAT, MSc Thesis. Rochester Institute of Technology, Rochester (2002)

Exact Graph Coloring Using Inclusion–Exclusion

2006; Björklund, Husfeldt

Andreas Björklund, Thore Husfeldt
Department of Computer Science, Lund University, Lund, Sweden

Keywords and Synonyms

Vertex coloring

Problem Definition

A *k-coloring* of a graph $G = (V, E)$ assigns one of k colors to each vertex such that neighboring vertices have different colors. This is sometimes called *vertex coloring*.

The smallest integer k for which the graph G admits a k-coloring is denoted $\chi(G)$ and called the *chromatic number*. The number of k-colorings of G is denoted $P(G;k)$ and called the *chromatic polynomial*.

Key Results

The central observation is that $\chi(G)$ and $P(G;k)$ can be expressed by an inclusion–exclusion formula whose terms are determined by the number of independent sets of induced subgraphs of G. For $X \subseteq V$, let $s(X)$ denote the number of nonempty independent vertex subsets disjoint from X, and let $s_r(X)$ denote the number of ways to choose r nonempty independent vertex subsets $S_1, \ldots, S_r$ (possibly overlapping and with repetitions), all disjoint from X, such that $|S_1| + \cdots + |S_r| = |V|$.

Theorem 1 *Let G be a graph on n vertices.*
1.

$$\chi(G) = \min_{k \in \{1,\ldots,n\}} \left\{ k \colon \sum_{X \subseteq V} (-1)^{|X|} s(X)^k > 0 \right\}.$$

2. For $k = 1, \ldots, k$,

$$P(G;k) = \sum_{r=1}^{k} \binom{k}{r} \Big(\sum_{X \subseteq V} (-1)^{|X|} s_r(X) \Big), \quad (k = 1, 2, \ldots, n).$$

The time needed to evaluate these expressions is dominated by the 2^n evaluations of $s(X)$ and $s_r(X)$, respectively. These values can be pre-computed in time and space within a polynomial factor of 2^n because they satisfy

$$s(X) = \begin{cases} 0, & \text{if } X = V, \\ s\Big(X \cup \{v\}\Big) + s\Big(X \cup \{v\} \cup N(v)\Big) + 1, & \text{for } v \notin X, \end{cases}$$

where $N(v)$ are the neighbors of v in G. Alternatively, the values can be computed using exponential-time, polynomial-space algorithms from the literature.

This leads to the following bounds:

Theorem 2 *For a graph G on n vertices, $\chi(G)$ and $P(G;k)$ can be computed in*
1. time and space $2^n n^{O(1)}$.
2. time $O(2.2461^n)$ and polynomial space

An optimal coloring that achieves $\chi(G)$ can be found within the same bounds.

The techniques generalize to arbitrary families of subsets over a universe of size n, provided membership in the family can be decided in polynomial time.

Applications

In addition to being a fundamental problem in combinatorial optimization, graph coloring also arises in many applications, including register allocation and scheduling.

Cross References

Recommended Reading

1. Björklund, A., Husfeldt, T.: Exact algorithms for exact satisfiability and number of perfect matchings. In: Proc. 33rd ICALP. LNCS, vol. 4051, pp. 548–1559. Springer (2006). Algorithmica, doi:10.1007/s00453-007-9149-8
2. Björklund, A., Husfeldt, T., Koivisto, M.: Set partitioning via inclusion–exclusion. SIAM J. Comput.
3. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbius: fast subset convolution. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), San Diego, CA, June 11–13, 2007. Association for Computing Machinery, pp. 67–74. New York (2007)

Experimental Methods for Algorithm Analysis

2001; McGeoch

Catherine C. McGeoch
Department of Mathematics and Computer Science, Amherst College, Amherst, MA, USA

Keywords and Synonyms

Experimental algorithmics; Empirical algorithmics; Empirical analysis of algorithms; Algorithm engineering

Problem Definition

Experimental analysis of algorithms describes not a specific algorithmic problem, but rather an approach to algorithm design and analysis. It complements, and forms a bridge between, traditional *theoretical analysis*, and the application-driven methodology used in *empirical analysis*.

The traditional theoretical approach to algorithm analysis defines algorithm efficiency in terms of counts of dominant operations, under some abstract model of computation such as a RAM; the input model is typically either worst-case or average-case. Theoretical results are usually expressed in terms of asymptotic bounds on the function relating input size to number of dominant operations performed.

This contrasts with the tradition of empirical analysis that has developed primarily in fields such as operations research, scientific computing, and artificial intelligence. In this tradition, the efficiency of implemented programs is typically evaluated according to CPU or wall-clock times; inputs are drawn from real-world applications or collections of benchmark test sets, and experimental results are usually expressed in comparative terms using tables and charts.

Experimental analysis of algorithms spans these two approaches by combining the sensibilities of the theoretician with the tools of the empiricist. Algorithm and program performance can be measured experimentally according to a wide variety of *performance indicators*, including the dominant cost traditional to theory, bottleneck operations that tend to dominate running time, data structure updates, instruction counts, and memory access costs. A researcher in experimental analysis selects performance indicators most appropriate to the scale and scope of the specific research question at hand. (Of course time is not the only metric of interest in algorithm studies; this approach can be used to analyze other properties such as solution quality or space use.)

Input instances for experimental algorithm analysis may be randomly generated or derived from application instances. In either case, they typically are described in terms of a small- to medium-sized collection of *controlled parameters*. A primary goal of experimentation is to investigate the cause-and-effect relationship between input parameters and algorithm/program performance indicators.

Research goals of experimental algorithmics may include discovering functions (not necessarily asymptotic) that describe the relationship between input and performance, assessing the strengths and weaknesses of different algorithm/data structures/programming strategies, and finding best algorithmic strategies for different input categories. Results are typically presented and illustrated with graphs showing comparisons and trends discovered in the data.

The two terms "empirical" and "experimental", are often used interchangeably in the literature. Sometimes the terms "old style" and "new style" are used to describe, respectively, the empirical and experimental approaches to this type of research. The related term "algorithm engineering" refers to a systematic design process that takes an abstract algorithm all the way to an implemented program, with an emphasis on program efficiency. Experimental and empirical analysis is often used to guide the algorithm engineering process. The general term *algorithmics* can refer to both design and analysis in algorithm research.

Key Results

None

Applications

Experimental analysis of algorithms has been used to investigate research problems originating in theoretical computer science. One example arises in the average-case analysis of algorithms for the One-Dimensional Bin Packing problem. Experimental analyses have led to new theorems about the performance of the optimal algorithm; new asymptotic bounds on average-case performance of approximation algorithms; extensions of theoretical results to new models of inputs; and to new algorithms with tighter approximation guarantees. Another example is the experimental discovery of a type of phase-transition behavior for random instances of the 3CNF-Satisfiabilty problem, which has led to new ways to characterize the difficulty of problem instances.

A second application of experimental algorithmics is to find more realistic models of computation, and to design new algorithms that perform better on these models. One example is found in the development of new memory-based models of computation that give more accurate time predictions than traditional unit-cost models. Using these models, researchers have found new cache-efficient and I/O-efficient algorithms that exploit properties of the memory hierarchy to achieve significant reductions in running time.

Experimental analysis is also used to design and select algorithms that work best in practice, algorithms that work best on specific categories of inputs, and algorithms that are most robust with respect to bad inputs.

Data Sets

Many repositories for data sets and instance generators to support experimental research are available on the Internet. They are usually organized according to specific combinatorial problems or classes of problems.

URL to Code

Many code repositories to support experimental research are available on the Internet. They are usually organized according to specific combinatorial problems or classes of problems. Skiena's *Stony Brook Algorithm Repository* (www.cs.sunysb.edu/~algorith/) provides a comprehensive collection of problem definitions and algorithm descriptions, with numerous links to implemented algorithms.

Recommended Reading

The algorithmic literature containing examples of experimental research is much too large to list here. Some articles containing advice and commentary on experimental methodology in the context of algorithm research appear in the list below.

The workshops and journals listed below are specifically intended to support research in experimental analysis of algorithms. Experimental work also appears in more general algorithm research venues such as SODA (ACM/IEEE Symposium on Data Structures and Algorithms), *Algorithmica*, and *ACM Transactions on Algorithms*.

1. *ACM Journal of Experimental Algorithmics*. Launched in 1996, this journal publishes contributed articles as well as special sections containing selected papers from ALENEX and WEA. Visit www.jea.acm.org, or visit portal.acm.org and click on ACM Digital Library/Journals/Journal of Experimental Algorithmics
2. ALENEX. Beginning in 1999, the annual workshop on Algorithm Engineering and Experimentation is sponsored by SIAM and ACM. It is co-located with SODA, the SIAM Symposium on Data Structures and Algorithms. Workshop proceedings are published in the Springer LNCS series. Visit www.siam.org/meetings/ for more information
3. Barr, R.S., Golden, B.L., Kelly, J.P., Resende, M.G.C., Stewart, W.R.: Designing and reporting on computational experiments with heuristic methods. J. Heuristic **1**(1), 9–32 (1995)
4. Cohen, P.R.: Empirical Methods for Artificial Intelligence. MIT Press, Cambridge (1995)
5. DIMACS Implementation Challenges. Each DIMACS Implementation Challenge is a year-long cooperative research event in which researchers cooperate to find the most efficient algorithms and strategies for selected algorithmic problems. The DIMACS Challenges since 1991 have targeted a variety of optimization problems on graphs; advanced data structures; and scientific application areas involving computational biology and parallel computation. The DIMACS Challenge proceedings are published by AMS as part of the DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Visit dimacs.rutgers.edu/Challenges for more information
6. Johnson, D.S.: A theoretician's guide to the experimental analysis of algorithms. In: Goodrich, M.H., Johnson, D.S., McGeoch, C.C. (eds.) Data Structures, Near Neighbors Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 59. American Mathematical Society, Providence (2002)
7. McGeoch, C.C.: Toward an experimental method for algorithm simulation. INFORMS J. Comp. **1**(1), 1–15 (1996)
8. WEA. Beginning in 2001, the annual Workshop on Experimental and Efficient Algorithms is sponsored by EATCS. Workshop proceedings are published in the Springer LNCS series

External Memory

► I/O-model
► R-Trees

External Sorting and Permuting

1988; Aggarwal, Vitter

Jeffrey Scott Vitter
Department of Computer Science, Purdue University, West Lafayette, IN, USA

Keywords and Synonyms

Out-of-core sorting

Problem Definition

Notations The main properties of magnetic disks and multiple disk systems can be captured by the commonly used *parallel disk model* (PDM), which is summarized

below in its current form as developed by Vitter and Shriver [16]:

N = problem size (in units of data items) ;
M = internal memory size (in units of data items) ;
B = block transfer size (in units of data items) ;
D = number of independent disk drives ;
P = number of CPUs ,

where $M < N$, and $1 \le DB \le M/2$. The data items are assumed to be of fixed length. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. (In the original 1988 article [2], the D blocks per I/O were allowed to come from the same disk, which is not realistic.) If $P \le D$, each of the P processors can drive about D/P disks; if $D < P$, each disk is shared by about P/D processors. The internal memory size is M/P per processor, and the P processors are connected by an interconnection network.

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the resulting formulas are often simplified. We define the lowercase notation

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad q = \frac{Q}{B}, \qquad z = \frac{Z}{B} \tag{1}$$

to be the problem input size, internal memory size, query specification size, and query output size, respectively, in units of disk blocks.

The primary measures of performance in PDM are
1. the number of I/O operations performed,
2. the amount of disk space used, and
3. the internal (sequential or parallel) computation time.

For reasons of brevity in this survey, focus is restricted to only the first two measures. Most of the algorithms run in optimal CPU time, at least for the single-processor case. Ideally algorithms and data structures should use linear space, which means $O(N/B) = O(n)$ disk blocks of storage.

Problem 1 (External sorting) INPUT: *The input data records R_0, R_1, R_2, ... are initially "striped" across the D disks, in units of blocks, so that record R_i is in block $\lfloor i/B \rfloor$, and block j is stored on disk j* mod *D.*

OUTPUT: *A striped representation of a permuted ordering $R_{\sigma(0)}$, $R_{\sigma(1)}$, $R_{\sigma(2)}$, ... of the input records with the property that* $\text{key}(R_{\sigma(i)}) \le \text{key}(R_{\sigma(i+1)})$ *for all* $i \ge 0$.

Permuting is the special case of sorting in which the permutation that describes the final position of the records is given explicitly and does not have to be discovered, for example, by comparing keys.

Problem 2 (Permuting) INPUT: *Same input assumptions as in external sorting. In addition, a permutation σ of the integers $\{0, 1, 2, \ldots, N-1\}$ is specified.*

OUTPUT: *A striped representation of a permuted ordering $R_{\sigma(0)}, R_{\sigma(1)}, R_{\sigma(2)}, \ldots$ of the input records.*

Key Results

Theorem 1 ([2,12]) *The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is*

$$\text{Sort}(N) = \Theta\left(\frac{n}{D}\log_m n\right). \tag{2}$$

Theorem 2 ([2]) *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, \text{Sort}(N)\right\}\right). \tag{3}$$

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

Theorem 3 ([2]) *With D disks, the number of I/Os required to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right), \tag{4}$$

where $N = pq$ and $n = N/B$.

Matrix transposition is a special case of a more general class of permutations called *bit-permute/complement* (BPC) permutations, which in turn is a subset of the class of *bit-matrix-multiply/complement* (BMMC) permutations. BMMC permutations are defined by a $\log N \times \log N$ nonsingular 0-1 matrix A and a $(\log N)$-length 0-1 vector c. An item with binary address x is mapped by the permutation to the binary address given by $Ax \oplus c$, where $\oplus$ denotes bitwise exclusive-or. BPC permutations are the special case of BMMC permutations in which A is a permutation matrix, that is, each row and each column of A contain a single 1. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen et al. [6] char-

acterize the optimal number of I/Os needed to perform any given BMMC permutation solely as a function of the associated matrix A, and they give an optimal algorithm for implementing it.

Theorem 4 ([6]) *With D disks, the number of I/Os required to perform the BMMC permutation defined by matrix A and vector c is*

$$\Theta\left(\frac{n}{D}\left(1+\frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \tag{5}$$

where γ is the lower-left $\log n \times \log B$ submatrix of A.

The two main paradigms for external sorting are *distribution* and *merging*, which are discussed in the following sections for the PDM model.

Sorting by Distribution

Distribution sort [9] is a recursive process that uses a set of $S-1$ partitioning elements to partition the items into S disjoint buckets. All the items in one bucket precede all the items in the next bucket. The sort is completed by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

One requirement is to choose the $S-1$ partitioning elements so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease from one level of recursion to the next by a relative factor of $\Theta(S)$, and thus there are $O(\log_S n)$ levels of recursion. During each level of recursion, the data are scanned. As the items stream through internal memory, they are partitioned into S buckets in an online manner. When a buffer of size B fills for one of the buckets, its block is written to the disks in the next I/O, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number of buckets (and partitioning elements) is $S = \Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$. How to perform each level of recursion in a linear number of I/Os is discussed in [2,11,16].

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [11]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. It maintains an invariant that guarantees good distribution across the disks for each bucket.

The distribution sort methods mentioned above for parallel disks perform write operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and thus the disks must be used independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. As a result, the write operations must use disks independently, since during each write, multiple buckets will be writing to multiple stripes. Error correction and recovery can still be handled efficiently by devoting to each bucket one block-sized buffer in internal memory. The buffer is continuously updated to contain the exclusive-or (parity) of the blocks written to the current stripe, and after $D-1$ blocks have been written, the parity information in the buffer can be written to the final (Dth) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to read one memoryload at a time and partition the items into S buckets. However, unlike before, the blocks for each individual bucket will reside on the disks in contiguous stripes. Each block therefore has a predefined place where it must be written. With the normal round-robin ordering for the stripes (namely, $\ldots, 1, 2, 3, \ldots, D, 1, 2, 3, \ldots, D, \ldots$), the blocks of different buckets may "collide," meaning that they need to be written to the same disk, and subsequent blocks in those same buckets will also tend to collide. Vitter and Hutchinson [15] solve this problem by the technique of *randomized cycling*. For each of the S buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1, 2, \ldots, D\}$. The S random permutations are chosen independently. If two blocks (from different buckets) happen to collide during a write to the same disk, one block is written to the disk and the other is kept on a write queue. With high probability, subsequent blocks in those two buckets will be written to different disks and thus will not collide. As long as there is a small pool of available buffer space to temporarily cache the blocks in the write queues, Vitter and Hutchinson [15] show that with high probability the writing proceeds optimally.

The randomized cycling method or the related merge sort methods discussed at the end of Section Sorting by Merging are the methods of choice for sorting with parallel disks. Distribution sort algorithms may have an advantage over the merge approaches presented in Section Sorting by Merging in that they typically make better use of lower levels of cache in the memory hierarchy of real systems, based upon analysis of distribution sort and merge sort algorithms on models of hierarchical memory.

Sorting by Merging

The *merge* paradigm is somewhat orthogonal to the distribution paradigm of the previous section. A typical merge sort algorithm works as follows [9]: In the "run formation" phase, the n blocks of data are scanned, one memoryload at a time; each memoryload is sorted into a single "run," which is then output onto a series of stripes on the disks. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, "replacement-selection" can be used to get runs of $2M$ data items, on the average, when $M \gg B$ [9].) After the initial runs are formed, the merging phase begins. In each pass of the merging phase, R runs are merged at a time. For each merge, the R runs are scanned and its items merged in an online manner as they stream through internal memory. Double buffering is used to overlap I/O and computation. At most $R = \Theta(m)$ runs can be merged at a time, and the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (2), each merging pass must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

The Greed Sort method of Nodine and Vitter [12] was the first optimal deterministic EM algorithm for sorting with multiple disks. It works by relaxing the merging process with a final pass to fix the merging. Aggarwal and Plaxton [1] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious. Like Greed Sort, the Sharesort algorithm is theoretically optimal (i. e., within a constant factor of optimal), but the constant factor is larger than the distribution sort methods.

One of the most practical methods for sorting is based upon the *simple randomized merge sort* (SRM) algorithm of Barve et al. [5], referred to as "randomized striping" by Knuth [9]. Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space.

Further improvements in merge sort are possible by a more careful prefetching schedule for the runs. Barve et al. [4], Kallahalla and Varman [8], Shah et al. [13], and others have developed competitive and optimal methods for prefetching blocks in parallel I/O systems. Hutchinson et al. [7] have demonstrated a powerful duality between parallel writing and parallel prefetching, which gives an easy way to compute optimal prefetching and caching schedules for multiple disks. More significantly, they show that the same duality exists between distribution and merging, which they exploit to get a provably optimal and very practical parallel disk merge sort. Rather than use random starting points and round-robin stripes as in SRM, Hutchinson et al. [7] order the stripes for each run independently, based upon the randomized cycling strategy discussed in Section Sorting by Distribution for distribution sort.

Handling Duplicates: Bundle Sorting

For the problem of *duplicate removal*, in which there are a total of K distinct items among the N items, Arge et al. [3] use a modification of merge sort to solve the problem in $O\left(n \max\left\{1, \log_m(K/B)\right\}\right)$ I/Os, which is optimal in the comparison model. When duplicates get grouped together during a merge, they are replaced by a single copy of the item and a count of the occurrences. The algorithm can be used to sort the file, assuming that a group of equal items can be represented by a single item and a count.

A harder instance of sorting called *bundle sorting* arises when there are K distinct key values among the N items, but all the items have different secondary information that must be maintained, and therefore items cannot be aggregated with a count. Matias et al. [10] develop optimal distribution sort algorithms for bundle sorting using

$$O\left(n \max\left\{1, \log_m \min\{K, n\}\right\}\right) \tag{6}$$

I/Os and prove the matching lower bound. They also show how to do bundle sorting (and sorting in general) *in place* (i. e., without extra disk space).

Permuting and Transposition

Permuting is the special case of sorting in which the key values of the N data items form a permutation of $\{1, 2, \ldots, N\}$. The I/O bound (3) for permuting can be realized by one of the optimal sorting algorithms except in the extreme case $B \log m = o(\log n)$, where it is faster to

move the data items one by one in a nonblocked way. The one-by-one method is trivial if $D = 1$, but with multiple disks there may be bottlenecks on individual disks; one solution for doing the permuting in $O(N/D)$ I/Os is to apply the randomized balancing strategies of [16].

Matrix transposition can be as hard as general permuting when B is relatively large (say, $1/2M$) and N is $O(M^2)$, but for smaller B, the special structure of the transposition permutation makes transposition easier. In particular, the matrix can be broken up into square submatrices of B^2 elements such that each submatrix contains B blocks of the matrix in row-major order and also B blocks of the matrix in column-major order. Thus, if $B^2 < M$, the transpositions can be done in a simple one-pass operation by transposing the submatrices one at a time in internal memory.

Fast Fourier Transform and Permutation Networks

Computing the fast Fourier transform (FFT) in external memory consists of a series of I/Os that permit each computation implied by the FFT directed graph (or butterfly) to be done while its arguments are in internal memory. A permutation network computation consists of an oblivious (fixed) pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs.

The algorithms for FFT are faster and simpler than for sorting because the computation is nonadaptive in nature, and thus the communication pattern is fixed in advance [16].

Lower Bounds on I/O

The following proof of the permutation lower bound (3) of Theorem 2 is due to Aggarwal and Vitter [2]. The idea of the proof is to calculate, for each $t \geq 0$, the number of distinct orderings that are realizable by sequences of t I/Os. The value of t for which the number of distinct orderings first exceeds $N!/2$ is a lower bound on the average number of I/Os (and hence the worst-case number of I/Os) needed for permuting.

Assuming for the moment that there is only one disk, $D = 1$, consider how the number of realizable orderings can change as a result of an I/O. In terms of increasing the number of realizable orderings, the effect of reading a disk block is considerably more than that of writing a disk block, so it suffices to consider only the effect of read operations. During a read operation, there are at most B data items in the read block, and they can be interspersed among the M items in internal memory in at most $\binom{M}{B}$ ways, so the number of realizable orderings increases by a factor of $\binom{M}{B}$. If the block has never before resided in internal memory, the number of realizable orderings increases by an extra $B!$ factor, since the items in the block can be permuted among themselves. (This extra contribution of $B!$ can only happen once for each of the N/B original blocks.) There are at most $n + t \leq N \log N$ ways to choose which disk block is involved in the tth I/O (allowing an arbitrary amount of disk space). Hence, the number of distinct orderings that can be realized by all possible sequences of t I/Os is at most

$$(B!)^{N/B} \left(N(\log N) \binom{M}{B} \right)^t . \tag{7}$$

Setting the expression in (7) to be at least $N!/2$, and simplifying by taking the logarithm, the result is

$$N \log B + t \left(\log N + B \log \frac{M}{B} \right) = \Omega(N \log N) . \tag{8}$$

Solving for t gives the matching lower bound $\Omega(n \log_m n)$ for permuting for the case $D = 1$. The general lower bound (3) of Theorem 2 follows by dividing by D.

A stronger lower bound follows from a more refined argument that counts input operations separately from output operations [7]. For the typical case in which $B \log m = \omega(\log N)$, the I/O lower bound, up to lower order terms, is $2n \log_m n$. For the pathological in which $B \log m = o(\log N)$, the I/O lower bound, up to lower order terms, is N/D.

Permuting is a special case of sorting, and hence, the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and in that case the comparison model must be used to get the full lower bound (2) of Theorem 1 [2]. In the typical case in which $B \log m = \Omega(\log n)$, the comparison model is not needed to prove the sorting lower bound; the difficulty of sorting in that case arises not from determining the order of the data but from permuting (or routing) the data.

The proof used above for permuting also works for permutation networks, in which the communication pattern is oblivious (fixed). Since the choice of disk block is fixed for each t, there is no $N \log N$ term as there is in (7), and correspondingly there is no additive $\log N$ term in the inner expression as there is in (8). Hence, solving for t gives the lower bound (2) rather than (3). The lower bound follows directly from the counting argument; unlike the sorting derivation, it does not require the com-

parison model for the case $B \log m = o(\log n)$. The lower bound also applies directly to FFT, since permutation networks can be formed from three FFTs in sequence. The transposition lower bound involves a potential argument based upon a togetherness relation [2].

For the problem of bundle sorting, in which the N items have a total of K distinct key values (but the secondary information of each item is different), Matias et al. [10] derive the matching lower bound.

The lower bounds mentioned above assume that the data items are in some sense "indivisible," in that they are not split up and reassembled in some magic way to get the desired output. It is conjectured that the sorting lower bound (2) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, removing the indivisibility assumption can lead to faster algorithms. Whether the conjecture is true is a challenging theoretical open problem.

Applications

Sorting and sorting-like operations account for a significant percentage of computer use [9], with numerous database applications. In addition, sorting is an important paradigm in the design of efficient EM algorithms, as shown in [14], where several applications can be found. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as permuting, list ranking, expression tree evaluation, and finding connected components in a sparse graph, require the same number of I/Os in PDM as does sorting.

Open Problems

Several interesting challenges remain. One difficult theoretical problem is to prove lower bounds for permuting and sorting without the indivisibility assumption. Another question is to determine the I/O cost for each individual permutation, as a function of some simple characterization of the permutation, such as number of inversions. A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice. Many interesting challenges and opportunities in algorithm design and analysis arise from new architectures being developed, such as networks of workstations, hierarchical storage devices, disk drives with processing capabilities, and storage devices based upon microelectromechanical systems (MEMS). Active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have recently been proposed to further reduce the I/O bottleneck, especially in large database applications. MEMS-based nonvolatile storage has the potential to serve as an intermediate level in the memory hierarchy between DRAM and disks. It could ultimately provide better latency and bandwidth than disks, at less cost per bit than DRAM.

URL to Code

Two systems for developing external memory algorithms are TPIE and STXXL, which can be downloaded from http://www.cs.duke.edu/TPIE/ and http://sttxl.sourceforge.net/, respectively. Both systems include subroutines for sorting and permuting and facilitate development of more advanced algorithms.

Cross References

► I/O-model

Recommended Reading

1. Aggarwal, A., Plaxton, C.G.: Optimal parallel sorting in multi-level storage. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, vol. 5, pp. 659–668. ACM Press, New York (1994)
2. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. In: Communications of the ACM, 31 (1988), pp. 1116–1127. ACM Press, New York (1988)
3. Arge, L., Knudsen, M., Larsen, K.: A general lower bound on the I/O-complexity of comparison-based algorithms. In: Proceedings of the Workshop on Algorithms and Data Structures. Lect. Notes Comput. Sci. **709**, 83–94 (1993)
4. Barve, R.D., Kallahalla, M., Varman, P.J., Vitter, J.S.: Competitive analysis of buffer management algorithms. J. Algorithms **36**, 152–181 (2000)
5. Barve, R.D., Vitter, J.S.: A simple and efficient parallel disk mergesort. ACM Trans. Comput. Syst. **35**, 189–215 (2002)
6. Cormen, T.H., Sundquist, T., Wisniewski, L.F.: Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. SIAM J. Comput. **28**, 105–136 (1999)
7. Hutchinson, D.A., Sanders, P., Vitter, J.S.: Duality between prefetching and queued writing with parallel disks. SIAM J. Comput. **34**, 1443–1463 (2005)
8. Kallahalla, M., Varman, P.J.: Optimal read-once parallel disk scheduling. Algorithmica **43**, 309–343 (2005)
9. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3, 2nd edn. Addison-Wesley, Reading (1998)
10. Matias, Y., Segal, E., Vitter, J.S.: Efficient bundle sorting. SIAM J. Comput. **36**(2), 394–410 (2006)
11. Nodine, M.H., Vitter, J.S.: Deterministic distribution sort in shared and distributed memory multiprocessors. In: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, June–July 1993, vol. 5, pp. 120–129, ACM Press, New York (1993)
12. Nodine, M.H., Vitter, J.S.: Greed Sort: An optimal sorting algorithm for multiple disks. J. ACM **42**, 919–933 (1995)

13. Shah, R., Varman, P.J., Vitter, J.S.: Online algorithms for prefetching and caching on parallel disks. In: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, pp. 255–264. ACM Press, New York (2004)
14. Vitter, J.S.: External memory algorithms and data structures: Dealing with Massive Data. ACM Comput. Surv. **33**(2), 209–271 (2001) Revised version available at http://www.cs.purdue.edu/homes/jsv/Papers/Vit.IO_survey.pdf
15. Vitter, J.S., Hutchinson, D.A.: Distribution sort with randomized cycling. J. ACM. **53** (2006)
16. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory I: Two-level memories. Algorithmica **12**, 110–147 (1994)

Extremal Problems

► Max Leaf Spanning Tree
► Online Interval Coloring