Vector Bin Packing

David S. Johnson
Department of Computer Science, Columbia University, New York, NY, USA
AT&T Laboratories, Algorithms and Optimization Research Department, Florham Park, NJ, USA

Keywords

Approximation algorithms; Bin packing; Resource constraints; Shared hosting platforms; Worst-case analysis

Years and Authors of Summarized Original Work

1976; Garey, Graham, Johnson, Yao
1977; Kou, Markowski
1977; Maruyama, Chang, Tang
1981; de la Vega, Lueker
1987; Yao
1990; Csirik, Frenk, Labbé, Zhang
1997; Woeginger
2001; Caprara, Toth
2004; Chekuri, Khanna
2009; Bansal, Caprara, Sviridenko
2010; Stillwell, Schanzenback, Vivien, Casanova
2011; Panigrahy, Talwar, Uyeda, Wieder

Problem Definition

In the *vector bin packing problem*, we are given an integral dimension $d \geq 1$ and a list $L = (x_1, x_2, \ldots, x_n)$ of items, where each item is a d-dimensional tuple $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,d})$ with rational entries $x_{i,j} \in [0, 1]$. The goal is to assign the items to a minimum number of multidimensional *bins*, where if X is the set of items assigned to a bin, we must have, for each j, $1 \leq j \leq d$,

$$\sum_{x_i \in X} x_{i,j} \leq 1.$$

Note that when $d = 1$, the vector bin packing problem reduces to the classic (one-dimensional) *bin packing* problem.

One potential application of the vector bin packing problem is that of assigning jobs to servers in a shared hosting platform, where each job may require a specific number of cycles per second and specific amounts of memory, bandwidth, and other resources [12]. Here the servers

M.-Y. Kao (ed.), *Encyclopedia of Algorithms*,
DOI 10.1007/978-1-4939-2864-4

would correspond to the bins, the dimension d is the number of resources, the items are the jobs, and $x_{i,j}$ is the fraction of the total amount of a server's jth resource that job x_i requires.

In the early literature, this problem was often called the *multidimensional bin packing* problem. That term, however, is now more typically reserved for the related problem where the items are d-dimensional rectangular parallelepipeds (rectangles, when $d = 2$), the bins are d-dimensional unit cubes, and the items assigned must not only be assigned to bins but also to specific positions in the bins, in such a way that no point in any bin is in the interior of more than one item. With vector bin packing, in contrast, the dimensions are all independent and there is no geometric interpretation of the items.

Key Results

As a generalization of bin packing, vector bin packing is clearly NP-hard in the strong sense, and so most of the research on this problem has been directed toward the study of approximation algorithms for it. This will be the primary topic in this entry. Most of the theoretical results concerning these algorithms can be expressed in terms of *asymptotic worst-case ratios*. For a given algorithm A and a list of items L, let $A(L)$ denote the number of bins used by A for L. Let *OPT(L)* denote the optimal number of bins for list L. We define the asymptotic worst-case ratio $R_A^\infty(d)$ for algorithm A on d-dimensional instances as follows.

$$R_A^N(d) = \max\left\{ \frac{A(L)}{OPT(L)} : L \text{ is a list of } d\text{-dimensional items with } OPT(L) = N \right\}$$

$$R_A^\infty(d) = \limsup_{N\to\infty} R_A^N(d)$$

Generalizations of Classical Bin Packing Algorithms

Generalizing First Fit and First Fit Decreasing

Several classic one-dimensional bin packing algorithms have been generalized to vector bin packing. Imagine we have a potentially infinite sequence of empty bins $B_1, B_2, \ldots$, and let $X_{h,j}$ denote the total amount of resource j used by the items currently assigned to B_h. In the generalized "First Fit" algorithm, the first item goes in bin B_1, and thereafter each item goes into the lowest-index bin into which it can be legally placed, subject to the resource constraints. In generalized "Best Fit," each item is assigned to a bin with the maximum value of $\sum_{j=1}^{d} X_{h,j}$ among those to which it can legally be added, ties broken in favor of the smallest index h.

As in the one-dimensional case, a plausible way to improve the above two online algorithms is to first reorder the list in decreasing order, and then apply the packing algorithm. Now, however, there are a variety of ways to define "decreasing order," each leading to different algorithms. For example, in FFDmax items are ordered by nonincreasing value of $\max_{j=1}^{d} x_{i,j}$ and then FF is applied. Similarly, in FFDsum, the items are ordered by nonincreasing value of $\sum_{j=1}^{d} x_{i,j}$ and, in FFDprod, they are ordered by nonincreasing value of $\prod_{j=1}^{d} x_{i,j}$. In FFDlex, they are ordered so that x_i precedes $x_{i'}$ only if either $x_{i,j} = x_{i',j}$, $1 \le j \le d$, or there is a $j^* \le d$ such that $x_{i,j} = x_{i',j}$, $1 \le j < j^*$ and $x_{i,j^*} < x_{i',j^*}$. The algorithms BFDmax, BFDsum, BFCprod, and BFDlex are defined analogously, with Best Fit being used to pack the reordered list instead of First Fit.

Call an algorithm "reasonable" if it produces packings in which no two bins can be combined, that is, are such that all the items contained in the two would collectively fit together in a single bin [9]. All of the above algorithms are easily seen to be reasonable, and, indeed, any vector bin packing algorithm has a "reasonable" counterpart that uses no more bins and spends at most $O(n^2 d)$ additional time (in a final pass that combines legally combinable pairs of bins as long as such pairs exist). A general upper bound on asymptotic worst-case behavior is the following.

Theorem 1 (**[9]**) *If A is a reasonable vector bin packing algorithm, then for all $d \geq 1$,*

$$R_A^{\infty}(d) \leq d + 1.$$

Unfortunately, none of the above algorithms are much better.

Theorem 2 (**[9, 11]**) *For each of the 10 algorithms defined above and all $d \geq 1$,*

$$R_A^{\infty}(d) \geq d.$$

Tighter bounds have been proved for two of the algorithms.

Theorem 3 (**[6]**) *For all $d \geq 1$, $R_{\mathrm{FF}}^{\infty}(d) = d + \frac{7}{10}$.*

Theorem 4 (**[6]**) *For all $d \geq 1$, $d + \frac{d-1}{d(d+1)} \leq R_{\mathrm{FFDmax}}^{\infty}(d) \leq d + \frac{1}{3}$.*

Note that the classic one-dimensional bin packing results of [8] yield $R_{\mathrm{FF}}^{\infty}(1) = 17/10$ (the precise specialization of Theorem 3) and $R_{\mathrm{FFD}}^{\infty}(1) = 11/9$ (a tighter result than the specialization of Theorem 4). Matching upper and lower bounds are not known for $R_{\mathrm{FFDmax}}^{\infty}(d)$ for any $d > 1$. In special cases, however, the lower bounds can be improved. It was observed in [6] that the lower bounds for $d \in \{2, 3\}$ could be increased to $d + 11/60$ using ideas from [8]. And Csirik et al. [4] showed that for odd $d \geq 5$, the lower bound of Theorem 2 could be increased by $1/(d(d+1)(d+2))$.

Generalizing the de la Vega and Lueker Asymptotic Approximation Scheme

In [5], de la Vega and Lueker devised an "asymptotic polynomial-time approximation scheme" (APTAS) for one-dimensional bin packing, that is, a collection of polynomial-time algorithms A_ϵ with $R_{A_\epsilon}^{\infty}(1) \leq 1 + \epsilon$ for all $\epsilon > 0$. In that same paper, they also showed how to generalize the algorithms to provide a collection of vector bin packing algorithms B_ϵ such that for each ϵ and each integer $d \geq 1$, $R_{B_\epsilon}^{\infty}(d) \leq d + \epsilon$.

The algorithm B_ϵ is quite simple. Divide L into d sublists, $L_1, L_2, \ldots, L_d$, where L_j consists of all those items x for which j is the index of the dimension with the largest entry in the corresponding tuple, ties broken arbitrarily. Then apply $A_{\epsilon/d}$ to each list L_j separately, viewed as an instance of one-dimensional bin packing with the size of item x_i being $x_{i,j}$, and output the union of the d packings. Unfortunately, although the running times for the B_ϵ's are linear in dn, they contain additive constants that are potentially exponential in $(d/\epsilon)^2$, and so they may not be practical for small ϵ.

In contrast, FF, FFDmax, and all their variants mentioned in the previous section have straightforward $O(dn^2)$ implementations, and, although the data structures that allow them to be sped up to $O(n \log n)$ when $d = 1$ do not extend to higher dimensions, speedups should be possible by using d-dimensional dynamic range searching procedures to identify the set of bins that can contain the next item to be packed [11].

Hardness of Approximation Results

In [14], Yao observed that, under a standard decision tree model of computation, any vector bin packing algorithm A that has $R_A^{\infty}(d) < d$ for all d cannot have $o(n \log n)$ running time. This is not much of a constraint, however, since almost all the algorithms that have been proposed for Vector Bin Packing are slower than this. For those algorithms, a weaker bound applies. Assuming P $\neq$ NP, no polynomial-time vector bin packing algorithm A can have $R_A^{\infty}(d) < \sqrt{d} - \epsilon$ for all d and any $\epsilon > 0$. This follows from a straightforward reduction of graph coloring to vector bin packing and a result of Zuckerman [15] for the former [3]. Under the same assumption, there can be no APTAS for any fixed $d \geq 2$ [13].

Algorithms with $R_A^{\infty}(d) < d$

Chekuri and Khanna [3] devised the first polynomial-time algorithms to guarantee $R_A^{\infty}(d) < d$ for all sufficiently large d.

Theorem 5 (**[3]**) *For any $\epsilon > 0$ there is a polynomial-time algorithm C_ϵ such that*

$$R^{\infty}_{C_\epsilon}(d) < \epsilon \cdot d + \ln(\epsilon^{-1}) + 2.$$

The algorithm works in three phases. The first considers the following linear program (LP). Let $z_{i,k}$ be a decision variable with value 1 if item x_1 is packed in bin j. The LP's constraints are

$$\sum_{k=1}^{m} z_{i,k} = 1, \quad 1 \le i \le n \tag{1}$$

$$\sum_{i=1}^{n} x_{i,j} \cdot z_{i,k} \le 1, \quad 1 \le k \le m, 1 \le j \le d \tag{2}$$

$$z_{i,k} \ge 0, \quad 1 \le i \le n, 1 \le k \le m \tag{3}$$

This is the LP relaxation of an integer program (with $z_{i,k} \in \{0,1\}$) which has a feasible solution if and only if our list can be packed into m bins. The first set of constraints insures that each item is packed into exactly one bin. The second set insures that all resource constraints are satisfied by the packing.

Let M be the least value of m such that this LP is feasible. Then we clearly must have $OPT(L) \ge M$. Moreover we can in polynomial time determine M and a basic feasible solution for the corresponding LP, by using binary search and a polynomial time LP-solver. In this basic feasible solution, there will be at most $n + dM$ positive variables (the number of nontrivial constraints). Since each of the n items x_i by (1) must be assigned to at least one bin, at least one of the variables $z_{i,k}$ must be positive for each i, meaning that at most dM of the items can be assigned to more than one bin. That leaves $n - dM$ items assigned to exactly one bin, and consequently our LP solution yields a feasible packing of these items into $M \le OPT(L)$ bins, which is the output of our first phase. The remaining dM or fewer items will be packed into additional bins in two additional phases as follows.

Let $k = \lceil 1/\epsilon \rceil$. While there are at least k unpacked items that will fit in a single bin, find such a set and pack them all in a new bin (Phase 2). Otherwise, find a maximum size set of unpacked items that will fit in a bin, assign them to a new bin, and repeat until all items are packed (Phase 3). Note that in both of these phases the next set of items to be packed can be found in time $O(n^k kd)$, which is polynomial for fixed ϵ. Thus the overall time for the algorithm is itself polynomial for fixed ϵ. Phase 2 creates at most $dM/k < \epsilon d \cdot OPT(L)$ bins. Phase 3 can be interpreted as implementing the Greedy algorithm for Set Covering, as applied to the instance in which the elements to be covered are the items left to be packed after Phase 2, the sets are the collections of those items which will fit in a bin, and no set has size exceeding $k - 1$. Thus, by standard results about Greedy Set Covering (see [7] for example), the number of bins added in this phase is less than $(\ln(k-1)+1) \cdot OPT(L) < (\ln(1/\epsilon) + 1) \cdot OPT(L)$. Adding up the above three terms yields the claimed theorem.

Note that if we set $\epsilon = 1/d$ in the above, we get a series of algorithms $C_{1/d}$ with $R^{\infty}_{C_{1/d}}(d) \le \ln(d) + 3$, where the running time of each is polynomial in n, although exponential in d. A slight improvement to this has recently been obtained by Bansal et al. [1]. They devise algorithms $D_{d,\epsilon}$ that run in polynomial time for fixed d and ϵ (although exponential in both) that have $R^{\infty}_{D_{d,\epsilon}} \le \ln(d + \epsilon) + 1 + \epsilon$, which, for $d \ge 2$, already beats $\ln(d) + 3$ when $\epsilon = 1$.

Experimental Results

There have been several experimental studies of approximation algorithms for vector bin packing [2, 10–12]. These studies were for the most part limited to $d \le 10$ and $n \le 500$, which may well make sense in the context of the proposed applications, and used distinct sets of randomly generated test instances. In two cases ([10] and [12]), the algorithms were compared using objective functions other than the number of bins packed. Nevertheless, certain common conclusions emerge. The FFD algorithms in particular yielded substantially better packings than worst-case analysis suggests. Both [11] and [12] suggest, however, that a different class of algorithms, ones that attempt to keep the bins as "balanced" as possible, may perform even better. An example of such an algorithm is the "norm-based greedy" algorithm of [11], which packs the bins one-by-one, at each step adding to the current bin B_h that item x_i that fits and yields the smallest weighted L^2 norm for the resulting "gap vector" $(X_{h,1} - x_{i,1}, X_{h,2} - x_{i,2}, \ldots, X_{h,d} - x_{i,d})$. For more details, see [11]. As for the algorithms described above with $R_A^\infty(d) < d$ for large d, the only ones with hopes of feasible running times are the algorithms $C_{1/d}$ from [3] when n^d is of manageable size. Limited experiments from [12] indicate that the packings these algorithms produce are not competitive.

Cross-References

▸ Approximation Schemes for Bin Packing
▸ Bin Packing
▸ Graph Coloring
▸ Greedy Set-Cover Algorithms

Recommended Reading

1. Bansal N, Caprara A, Sviridenko M (2009) A new approximation method for set covering problems, with applications to multidimensional bin packing. SIAM J Comput 39:1256–1278
2. Caprara A, Toth P (2001) Lower bounds and algorithms for the 2-dimensional vector packing problem. Discret Appl Math 111:231–262
3. Chekuri C, Khanna S (2004) On multidimensional packing problems. SIAM J Comput 33:837–851
4. Csirik WF, Frenk JBG, Labbé M, Zhang S (1990) On the multidimensional vector packing. Acta Cybern 9:361–369
5. de la Vega J, Lueker GS (1981) Bin packing can be solved within $1 + \epsilon$ in linear time. Combinatorica 1:349–355
6. Garey MR, Graham RL, Johnson DS, Yao AC-C (1976) Resource constrained scheduling as generalized bin packing. J Comb Theory (A) 21:257–298
7. Johnson DS (1974) Approximation algorithms for combinatorial problems. J Comput Syst Sci 9:256–278
8. Johnson DS, Demers A, Ullman JD, Garey MR, Graham RL (1974) Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM J Comput 3:299–325
9. Kou LT, Markowski G (1977) Multidimensional bin packing algorithms. IBM J Res Dev 21:443–448
10. Maruyama K, Chang SK, Tang DT (1977) A general packing algorithm for multidimensional resource requirements. Int J Comput Inf Sci 6:131–149
11. Panigrahy R, Talwar K, Uyeda L, Wieder U (2011) Heuristics for vector bin packing. Unpublished manuscript. (Available on the web)
12. Stillwell M, Schanzenbach D, Vivien F, Casanova H (2010) Resource allocation algorithms for virtualized service hosting platforms. J Parallel Distrib Comput 70:962–974
13. Woeginger GJ (1997) There is no asymptotic PTAS for two-dimensional vector packing. Inf Proc Lett 64:293–297
14. Yao AC-C (1980) New algorithms for bin packing. J ACM 27:207–227
15. Zuckerman D (2007) Linear degree extractors and the inapproximability of max clique and chromatic number. Theory Comput 3:103–128

Vector Scheduling Problems

Tjark Vredeveld
Department of Quantitative Economics, Maastricht University, Maastricht, The Netherlands

Keywords

Approximation schemes; Vector scheduling

Years and Authors of Summarized Original Work

2004; Chekuri, Khanna

Problem Definition

Vector scheduling is a multidimensional extension of traditional machine scheduling problems. Whereas in traditional machine scheduling a job only uses a single resource, normally time, in vector scheduling a job uses several resources. In traditional scheduling, the load of a machine is the total resource consumption by the jobs that it serves. In vector scheduling, we define the load of a machine as the maximum resource usage over all resources of the jobs that are served by this machine. In the setting that we consider here, the makespan, which is normally defined to be the time by which all jobs are completed, is equal to the maximum machine load.

To define the vector scheduling problem that we consider more formally, we let $\|\mathbf{x}\|_\infty$ denote the standard ℓ_∞-norm of the vector $\mathbf{x}$. In the vector scheduling problem, the input consists of a set J of n jobs, where each job j is associated with a d-dimensional vector $\mathbf{p_j} \in [0,1]^d$, and m identical machines. The goal is to find an assignment of the jobs to the m machines such that $\max_{1 \le i \le m} \| \sum_{j \in M_i} \mathbf{p_j} \|_\infty$ is minimized, where M_i denotes the set of jobs that are assigned to machine i.

The traditional machine scheduling problem corresponds to the case $d = 1$, and this is known to be strongly NP-hard [8]. For $d = 1$, Graham's well-known list scheduling algorithm has a performance guarantee of 2 [9] and Hochbaum and Shmoys developed at PTAS [10]. For general vector scheduling, Graham's list scheduling algorithm can be extended to the d-dimensional case, having a performance guarantee of $d + 1$. In this entry we focus on the work of Chekuri and Khann [5], who developed a PTAS for fixed d and gave a polylogarithmic approximation factor for the case of general d.

Key Results

Constant Dimension d

Chekuri and Khann [5] designed an approximation scheme that runs in polynomial time whenever the dimension d of the job vectors is constant.

Theorem 1 *For any $\epsilon > 0$, there exists an $(1 + \epsilon)$-approximation algorithm that has a running time of $O((nd/\epsilon)^{O(s)})$, where s is in $O\left((\frac{\log(d/\epsilon)}{\epsilon})^d\right)$.*

The proof of this theorem is a nontrivial generalization of the ideas that Hochbaum and Shmoys used for the 1-dimensional case [10]. In this primal-dual approach, the main idea is to view the scheduling problem as a bin-packing problem in which the jobs need to be packed into a number of bins of a certain capacity B. If all jobs fit into m bins, then the makespan is bounded by B. Hochbaum and Shmoys gave an algorithm that determines whether the jobs fit into m bins of capacity $(1 + \epsilon)B$ or the jobs need to be packed into at least $m + 1$ bins of capacity B. Chekuri and Khanna extended this idea by using d-dimensional bins, where the jobs assigned to one bin should have a total resource usage of at most B in any of the d dimensions. By standard scaling techniques, we assume w.l.o.g. that $B = 1$.

Like in the 1-dimensional case, Chekuri and Khanna divide the jobs in small and large jobs, where the size of a job is based on the ℓ_∞ norm. They first do a preprocessing step in which each coordinate of the vectors is set to 0 whenever it is too small compared to the maximum value of the coordinates in the same vector. To find an $(1 + \epsilon)$-approximation, the algorithm performs two stages. In the first stage all large jobs will be assigned to the machines, and in the second stage all small jobs will be assigned to the machines. Whereas in the 1-dimensional case the assignment of the small jobs can be done greedily on top of the large jobs, for $d \ge 2$ the interaction between the two stages needs to be taken into account.

To accommodate this interaction, Chekuri and Khanna define a *capacity configuration* as a d-tuple $(c_1, \ldots, c_d)$ such that c_k is an integer between 0 and $\lceil 1/\epsilon \rceil$. A set of jobs S can be feasible, scheduled on one machine according to a capacity configuration $(c_1, \ldots, c_d)$ when for any dimension k, it holds that $\left(\sum_{j \in S} \mathbf{p_j}\right)_k \le a_k \cdot \epsilon$, i.e., in each dimension k the resource usage is not more than $c_k \cdot \epsilon$. The number of distinct capacity configurations is given by $t = (1 + \lceil 1/\epsilon \rceil)^d$.

A capacity configuration describes approximately how a machine is filled. As there are m machines available to process the jobs, a *machine configuration* can be described by a t-tuple $(m_1, \ldots, m_t)$, satisfying $m_i \ge 0$ and $\sum_i m_i = m$, where m_i denotes the number of machines of the ith capacity configuration. The number of distinct machine configurations is certainly bounded from above by m^t.

After the preprocessing of the vectors and the splitting of the jobs in small and large, it needs to be determined whether all large jobs can be scheduled according to a machine configuration M. As a first step, all nonzero elements of the large vectors are rounded (down) to the begin points of geometrically increasing intervals. Moreover, as the vectors are in some sense large, not too many vectors can be scheduled on one machine. Therefore, using a dynamic programming approach, one can approximately determine whether the set of large vectors can be scheduled according to machine configuration M.

When the set of large jobs are scheduled such that a certain machine i is scheduled according to a capacity configuration $(c_1, \ldots, c_d)$, then the small jobs on this machine need to be scheduled according to the *empty capacity configuration*, i.e., the capacity configuration $(1 + \lceil 1/\epsilon \rceil) \cdot (1, 1, \ldots, 1) - (c_1, \ldots, c_d)$. Given a machine configuration M, we let $\bar{M}$ denote the corresponding machine configuration as the one obtained by taking the empty capacity configurations for each of the machines in M.

To see whether the small jobs can be scheduled according to a machine configuration $\bar{M}$, Chekuri and Khanna present an integer programming (ILP) formulation that assigns the vectors to the machines. Moreover, they show that solving the LP relaxation of this ILP formulation and distributing the fractionally assigned vectors equally over the machines result in a solution in which each dimension of each machine is only overloaded by a factor of $(1 + \epsilon)$.

Once they have found a machine configuration M according to which the large jobs can be scheduled and corresponding machine configuration $\bar{M}$ according to which the small jobs can be scheduled, Chekuri and Khanna have shown that all jobs can be scheduled such that the load of any machine does not exceed $1 + \epsilon$. If for all machine configurations M and corresponding machine configuration $\bar{M}$ the large jobs cannot be scheduled according to M or the small jobs cannot be scheduled according to $\bar{M}$, then the vectors cannot be scheduled such that the load of each machine is at most 1.

General Dimension d

For the general case in which the dimension d of the vectors is not restricted to be a constant, Chekuri and Khanna present several approximation algorithms. Also for the general case, they assume that all vectors can be scheduled such that the makespan is bounded by 1. For two algorithms, they use as a subroutine an approximation algorithm for finding a set of vectors S that maximizes the volume of these vectors, i.e., the sum of all coordinates of all these vectors $\sum_{j \in S} \sum_{k=1}^{d} (\mathbf{p_j})_k$, restricted to $\| \sum_{j \in S} \mathbf{p_j} \|_\infty \leq 1$. This resulted in the following results.

Theorem 2 *There exists a polynomial-time $O(\log^2 d)$-approximation algorithm for the vector scheduling problem.*

Theorem 3 *There exists a $O(\log d)$-approximation algorithm for the vector scheduling problem that runs in time polynomial in n^d.*

These approximation results are good when d is small compared to the number of machines m. On the other hand, Chekuri and Khanna also give a randomized algorithm, which just assigns each job uniformly at random to one of the machines, obtaining a performance guarantee that is better when d is large compared to m.

Theorem 4 *There exists a randomized algorithm that has a performance guarantee of $O(\log dm / \log \log dm)$ with high probability.*

Finally, there is also a hardness result for the vector scheduling problem.

Theorem 5 *For any constant $\rho > 1$, there is no polynomial-time approximation algorithm with a performance guarantee of ρ, unless $NP = ZPP$.*

V

Extensions

Epstein and Tassa [6, 7] extended the vector scheduling problem to deal with more general objective functions. Instead of defining the load of a machine as the maximum resource usage over all resources of the jobs that are served by this machine, they defined the load as the sum of the vectors $\mathbf{p_j}$ assigned to the machine. That is, the load itself is now also a d-dimensional vector. Letting $\mathbf{l_i} = \sum_{j \in M_i} \mathbf{p_j}$ denote the load of machine i, then in [6] they gave PTASes for several objective functions of the form $F(S) = f(g(\mathbf{l_1}), \dots, g(\mathbf{l_m}))$. Note that the vector scheduling problem as discussed in this entry is equal to the case that $f = g = \max$. In [7], they extended their results to the more general case where the function g may vary per machine, i.e., $F(S) = f(g_1(\mathbf{l_1}), \dots, g_m(\mathbf{l_m}))$.

Bonifaci and Wiese [4] extended the vector scheduling problem to the ℓ_p norm and the case of unrelated machines. That is, a job j has a d-dimensional resource usage $\mathbf{p_{ij}}$ on machine i. They considered the case in which the number of types of machines is constant: on the same type of machine, a certain job has the same resource usage. Moreover, they restricted themselves to the case of having only a constant number of resources, that is, the vectors $\mathbf{p_{ij}}$ are d dimensional for a constant d. For this setting, they developed a PTAS.

The PTAS of Chekuri and Khanna has a running time that is doubly exponential in d. Bansal, Vredeveld, and Van der Zwaan [2] showed that this double exponential dependence on d is necessary. For $\epsilon < 1$, they showed that unless the exponential time hypothesis fails, there is no $(1+\epsilon)$-approximation algorithm with running time $\exp(o(\lfloor 1/\epsilon \rfloor^{d/3}))$. Moreover, they showed that unless NP has subexponential algorithms, no $(1+\epsilon)$-approximation algorithm exists with running time $\exp(\lfloor 1/\epsilon \rfloor^{o(d)})$. These lower bounds even hold for the case that ϵm more machines are allowed, for sufficiently small $\epsilon > 0$. Moreover, they also gave a $(1+\epsilon)$-approximation algorithm with running time $\exp((1/\epsilon)^{O(d \log \log d)} + nd)$, which is the first efficient approximation scheme (EPTAS) for the problem with constant d.

Open Problems

The gap between the lower bounds and upper bounds on the running time of $(1+\epsilon)$-approximation algorithms has almost been closed. The question remains whether this is also the case when instead of the ℓ_∞-norm, the ℓ_p-norm is minimized. Furthermore, it would be interesting to know whether one can obtain better running times when the vectors are highly structured. These highly structured vectors may occur, for example, in applications of real-time scheduling; see, e.g., [1, 3].

Cross-References

- ▶ Approximation Schemes for Bin Packing
- ▶ Approximation Schemes for Makespan Minimization
- ▶ Vector Bin Packing

Recommended Reading

1. Bansal N, Rutten C, van der Ster S, Vredeveld T, van der Zwaan R (2014) Approximating real-time scheduling on identical machines. In: LATIN 2014, Montevideo. LNCS, vol 8392. Springer, Heidelberg, pp 550–561
2. Bansal N, Vredeveld T, van der Zwaan R (2014) Approximating vector scheduling: almost matching upper and lower bounds. In: LATIN 2014, Montevideo. LNCS, vol 8392. Springer, Heidelberg, pp 47–59
3. Baruah S, Bonifaci V, D'Angelo G, Marchetti-Spaccamela A, van der Ster S, Stougie L (2011) Mixed criticality scheduling of sporadic task systems. In: Proceedings of the 19th annual European symposium on algorithms (ESA), Saarbrücken. LNCS, vol 6942. Springer, Heidelberg, pp 555–566
4. Bonifaci V, Wiese A (2012) Scheduling unrelated machines of few different types. CoRR abs/1205.0974
5. Chekuri C, Khanna S (2004) On multidimensional packing problems. SIAM J Comput 33(4):837–851
6. Epstein L, Tassa T (2003) Vector assignment problems: a general framework. J Algorithms 48(2):360–384
7. Epstein L, Tassa T (2006) Vector assignment schemes for asymmetric settings. Acta Inform 42(6–7):501–514
8. Garey M, Johnson D (1978) "Strong" NP-completeness results: motivation, examples, and implications. J ACM 25:499–508
9. Graham R (1966) Bounds for certain multiprocessor anomalies. Bell Syst Tech J 45:1563–1581
10. Hochbaum DS, Shmoys DB (1987) Using dual approximation algorithms for scheduling problems theoretical and practical results. J ACM 34(1): 144–162

Vertex Cover Kernelization

Jianer Chen
Department of Computer Science, Texas A&M University, College Station, TX, USA

Keywords

Vertex cover data reduction; Vertex cover preprocessing

Years and Authors of Summarized Original Work

2004; Abu-Khzam, Collins, Fellows, Langston, Suters, Symons

Problem Definition

Let G be an undirected graph. A subset C of vertices in G is a *vertex cover* for G if every edge in G has at least one end in C. The (parametrized) VERTEX COVER problem is for each given instance (G, k), where G is a graph and $k \geq 0$ is an integer (the parameter), to determine whether the graph G has a vertex cover of at most k vertices.

The VERTEX COVER problem is one of the six "basic" NP-complete problems according to Garey and Johnson [4]. Therefore, the problem cannot be solved in polynomial time unless P = NP. However, the NP-completeness of the problem does not obviate the need for solving it because of its fundamental importance and wide applications. One approach was initiated based on the observation that in many applications, the parameter k is small. Therefore, by taking the advantages of this fact, one may be able to solve this NP-complete problem effectively and practically for instances with a small parameter. More specifically, algorithms of running time of the form $f(k)p(n)$ have been studied for VERTEX COVER, where $p(n)$ is a low-degree polynomial of the number $n = |G|$ of vertices in G and $f(k)$ is a function independent of n.

There has been an impressive sequence of improved algorithms for the VERTEX COVER problem. A number of new techniques have been developed during this research, including kernelization, folding, and refined branch-and-search. In particular, the *kernelization* method is the study of polynomial time algorithms that can significantly reduce the instance size for VERTEX COVER. The following are some concepts related to the kernelization method:

Definition 1 Two instances (G, k) and (G', k') of VERTEX COVER are *equivalent* if the graph G has a vertex cover of size $\leq k$ if and only if the graph G' has a vertex cover of size $\leq k'$.

Definition 2 A *kernelization algorithm* for the VERTEX COVER problem takes an instance (G, k) of VERTEX COVER as input and produces an equivalent instance (G', k') for the problem, such that $|G'| \leq |G|$ and $k' \leq k$.

The kernelization method has been used extensively in conjunction with other techniques in the development of algorithms for the VERTEX COVER problem. Two major issues in the study of kernelization method are (1) effective reductions of instance size; and (2) the efficiency of kernelization algorithms.

Key Results

A number of kernelization techniques are discussed and studied in the current paper.

Preprocessing Based on Vertex Degrees

Let (G, k) be an instance of VERTEX COVER. Let v be a vertex of degree larger than k in G. If a vertex cover C does not include v, then C must contain all neighbors of v, which implies that C contains more than k vertices. Therefore, in order to find a vertex cover of no more than k vertices, one must include v in the vertex cover, and recursively look for a vertex cover of $k - 1$ vertices in the remaining graph.

The following fact was observed on vertices of degree less than 3.

Theorem 1 *There is a linear time kernelization algorithm that on each instance (G, k) of vertex cover, where the graph G contains a vertex of degree less than 3, produces an equivalent instance (G', k') such that $|G'| < |G|$ and/or $k < k'$.*

Therefore, vertices of high degree (i.e., degree $> k$) and low degree (i.e., degree < 3) can always be handled efficiently before any more time-consuming process.

Nemhauser-Trotter Theorem

Let G be a graph with vertices $v_1, v_2, \ldots, v_n$. Consider the following integer programming problem:

$$\begin{aligned} \text{(IP)Minimize} \quad & x_1 + x_2 + \cdots + x_n \\ \text{Subject to} \quad & x_i + x_j \geq 1 \\ & \text{for each edge } [\mathrm{v_i}, \mathrm{v_j}] \text{ in G} \\ & x_i \in \{0, 1\}, \quad 1 \leq i \leq n \end{aligned}$$

It is easy to see that there is a one-to-one correspondence between the set of feasible solutions to (IP) and the set of vertex covers of the graph G. A natural LP-relaxation (LP) of the problem (IP) is to replace the restrictions $x_i \in \{0, 1\}$ with $x_i \geq 0$ for all i. Note that the resulting linear programming problem (LP) now can be solved in polynomial time.

Let $\sigma = \{x_1^0, \ldots, x_n^0\}$ be an optimal solution to the linear programming problem (LP). The vertices in the graph G can be partitioned into three disjoint parts according to σ:

$$I_0 = \{v_i \mid x_i^0 < 0.5\},$$

$$C_0 = \{v_i \mid x_i^0 > 0.5\}, \text{ and}$$

$$V_0 = \{v_i \mid x_i^0 = 0.5\}$$

The following nice property of the above vertex partition of the graph G was first observed by Nemhauser and Trotter [5].

Theorem 2 ***(Nemhauser-Trotter)*** *Let $G[V_0]$ be the subgraph of G induced by the vertex set V_0. Then (1) every vertex cover of $G[V_0]$ contains at least $|V_0|/2$ vertices; and (2) every minimum vertex cover of $G[V_0]$ plus the vertex set C_0 makes a minimum vertex cover of the graph G.*

Let k be any integer, and let $G' = G[V_0]$ and $k' = k - |C_0|$. As first noted in [3], by Theorem 2, the instances (G, k) and (G', k') are equivalent, and $|G'| \leq 2k'$ is a necessary condition for the graph G' to have a vertex cover of size k'. This observation gives the following kernelization result.

Theorem 3 *There is a polynomial-time algorithm that for a given instance (G, k) for the vertex cover problem, constructs an equivalent instance (G', k') such that $k' \leq k$ and $|G'| \leq 2k'$.*

A Faster Nemhauser-Trotter Construction

Theorem 3 suggests a polynomial-time kernelization algorithm for VERTEX COVER. The algorithm is involved in solving the linear programming problem (LP) and partitioning the graph vertices into the sets I_0, C_0, and V_0. Solving the linear programming problem (LP) can be done in polynomial time but is kind of costly in particular when the input graph G is dense. Alternatively, Nemhauser and Trotter [5] suggested the following algorithm without using linear programming. Let G be the input graph with vertex set $\{v_1, \ldots, v_n\}$.

1. construct a bipartite graph B with vertex set $\{v_1^L, \ldots, v_n^L, v_1^R, \ldots, v_n^R\}$ such that $[v_i^L, v_j^R]$ is an edge in B if and only if $[v_i, v_j]$ is an edge in G;
2. find a minimum vertex cover C_B for B;
3. $I_0' = \{v_i \mid \text{if neither } v_i^L \text{ nor } v_i^R \text{ is in } C_B\}$;
   $C_0' = \{v_i \mid \text{if both } v_i^L \text{ and } v_i^R \text{ are in } C_B\}$;
   $V_0' = \left\{v_i \mid \text{if exactly one of } v_i^L \text{ and } v_i^R \text{ is in } C_B\right\}$

It can be proved [5] (see also [2]) that Theorem 2 still holds true when the sets C_0 and V_0 in the theorem are replaced by the sets C_0' and V_0', respectively, constructed in the above algorithm.

The advantage of this approach is that the sets C'_0 and V'_0 can be constructed in time $O(m\sqrt{n})$ because the minimum vertex cover C_B for the bipartite graph B can be constructed via a maximum matching of B, which can be constructed in time $O(m\sqrt{n})$ using Dinic's maximum flow algorithm, which is in general faster than solving the linear programming problem (LP).

Crown Reduction

For a set S of vertices in a graph G, denote by $N(S)$ the set of vertices that are not in S but adjacent to some vertices in S. A *crown* in a graph G is a pair (I, H) of subsets of vertices in G satisfying the following conditions: (1) $I \neq \emptyset$ is an independent set, and $H = N(I)$; and (2) there is a matching M on the edges connecting I and H such that all vertices in H are matched in M.

It is quite easy to see that for a given crown (I, H), there is a minimum vertex cover that includes all vertices in H and excludes all vertices in I. Let G' be the graph obtained by removing all vertices in I and H from G. Then, the instances (G, k) and (G', k') are equivalent, where $k' = k - |H|$. Therefore, identification of crowns in a graph provides an effective way for kernelization.

Let G be the input graph. The following algorithm is proposed.

1. construct a maximal matching M_1 in G; let O be the set of vertices unmatched in M_1;
2. construct a maximum matching M_2 of the edges between O and $N(O)$; $i = 0$; let I_0 be the set of vertices in O that are unmatched in M_2;
3. repeat until $I_i = I_{i-1}$ $\{H_i = N(I_i)$; $I_{i+1} = I_i \cup N_{M_2}(H_i)$;$i = i + 1$; $\}$; (where $N_{M_2}(H_i)$ is the set of vertices in O that match the vertices in H_i in the matching M_2)
4. $I = I_i$; $H = N(I_i)$; output (I, H).

Theorem 4 *(1) if the set I_0 is not empty, then the above algorithm constructs a crown (I, H); (2) if both $|M_1|$ and $|M_2|$ are bounded by k, and $I_0 = \emptyset$, then the graph G has at most 3k vertices.*

According to Theorem 4, the above algorithm on an instance (G, k) of VERTEX COVER either (1) finds a matching of size larger than k – which implies that there is no vertex cover of k vertices in the graph G; or (2) constructs a crown (I, H) – which will reduce the size of the instance; or (3) in case neither of (1) and (2) holds true, concludes that the graph G contains at most $3k$ vertices. Therefore, repeatedly applying the algorithm either derives a direct solution to the given instance, or constructs an equivalent instance (G', k') with $k' \leq k$ and $|G'| \leq 3k'$.

Applications

The research of the current paper was directly motivated by authors' research in bioinformatics. It is shown that for many computational biological problems, such as the construction of phylogenetic trees, phenotype identification, and analysis of microarray data, preprocessing based on the kernelization techniques has been very effective.

Experimental Results

Experimental results are given for handling graphs obtained from the study of phylogenetic trees based on protein domains, and from the analysis of microarray data. The results show that in most cases the best way to kernelize is to start handling vertices of high and low degrees (i.e., vertices of degree larger than k or smaller than 3) before attempting any of the other kernelization techniques. Sometimes, kernelization based on Nemhauser-Trotter Theorem can solve the problem without any further branching. It is also observed that sometimes particularly on dense graphs, kernelization techniques based on Nemhauser-Trotter Theorem are kind of time-consuming but do not reduce the instance size by much. On the other hand, the techniques based on high-degree vertices and crown reduction seem to work better.

Data Sets

The experiments were performed on graphs obtained based on data from NCBI and SWISS-PROT, well known open-source repositories of biological data.

Cross-References

▸ Data Reduction for Domination in Graphs
▸ Local Approximation of Covering and Packing Problems
▸ Vertex Cover Search Trees

Recommended Reading

1. Abu-Khzam F, Collins R, Fellows M, Langston M, Suters W, Symons C (2004) Kernelization algorithms for the vertex cover problem: theory and experiments. In: Proceedings of the workshop on algorithm engineering and experiments (ALENEX), pp 62–69
2. Bar-Yehuda R, Even S (1985) A local-ratio theorem for approximating the weighted vertex cover problem. Ann Discret Math 25:27–45
3. Chen J, Kanj IA, Jia W (2001) Vertex cover: further observations and further improvements. J Algorithm 41:280–301
4. Garey M, Johnson D (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco
5. Nemhauser GL, Trotter LE (1975) Vertex packing: structural properties and algorithms. Math Program 8:232–248

Vertex Cover Search Trees

Jianer Chen
Department of Computer Science, Texas A&M University, College Station, TX, USA

Keywords

Branch and bound; Branch and search

Years and Authors of Summarized Original Work

2001; Chen, Kanj, Jia

Problem Definition

The VERTEX COVER problem is one of the six "basic" NP-complete problems according to Garey and Johnson [7]. Therefore, the problem cannot be solved in polynomial time unless P = NP. However, the NP-completeness of the problem does not obviate the need for solving it because of its fundamental importance and wide applications.

One approach is to develop *parameterized algorithms* for the problem, with the computational complexity of the algorithms being measured in terms of both input size and a parameter value. This approach was initiated based on the observation that in many applications, the instances of the problem are associated with a small parameter. Therefore, by taking the advantages of the small parameters, one may be able to solve this NP-complete problem effectively and practically.

The problem is formally defined as follows. Let G be an (undirected) graph. A subset C of vertices in G is a *vertex cover* for G if every edge in G has at least one end in C. An instance of the (parameterized) VERTEX COVER problem consists of a pair (G, k), where G is a graph and k is an integer (the parameter), which is to determine whether the graph G has a vertex cover of k vertices. The goal is to develop parameterized algorithms of running time $O(f(k)p(n))$ for the VERTEX COVER problem, where $p(n)$ is a lower-degree polynomial of the input size n, and $f(k)$ is the non-polynomial part that is a function of the parameter k but independent of the input size n. It would be expected that the non-polynomial function $f(k)$ is as small as possible. Such an algorithm would become "practically effective" when the parameter value k is small. It should be pointed out that unless an unlikely consequence occurs in complexity theory, the function $f(k)$ is at least an exponential function of the parameter k [8].

Key Results

A number of techniques have been proposed in the development of parameterized algorithms for the VERTEX COVER problem.

Kernelization

Suppose (G, k) is an instance for the VERTEX COVER problem, where G is a graph and k is the parameter. The *kernelization* operation applies a polynomial time preprocessing on the instance (G, k) to construct another instance (G', k'), where G' is a smaller graph (the *kernel*) and $k' \leq k$, such that G' has a vertex cover of k' vertices if and only if G has a vertex cover of k vertices. Based on a classical result by Nemhauser and Trotter [9], the following kernelization result was derived.

Theorem 1 *There is an algorithm of running time $O(kn + k^3)$ that for a given instance (G, k) for the* VERTEX COVER *problem, constructs another instance (G', k') for the problem, where the graph G' contains at most $2k'$ vertices and$k' \leq k$, such that the graph G has a vertex cover of k vertices if and only if the graph G' has a vertex cover of k' vertices.*

Therefore, kernelization provides an efficient preprocessing for the VERTEX COVER problem, which allows one to concentrate on graphs of small size (i.e., graphs whose size is only related to k).

Folding

Suppose v is a degree-2 vertex in a graph G with two neighbors u and w such that u and w are not adjacent to each other. Construct a new graph G' as follows: remove the vertices v, u, and w and introduce a new vertex v_0 that is adjacent to all remaining neighbors of the vertices u and w in G. The graph G' is said being obtained from the graph G by *folding the vertex v*. The following result was derived.

Theorem 2 *Let G' be a graph obtained by folding a degree-2 vertex v in a graph G, where the two neighbors of v are not adjacent to each other. Then the graph G has a vertex cover of k vertices if and only if the graph G' has a vertex cover of $k - 1$ vertices.*

An folding operation allows one to decrease the value of the parameter k without branching. Therefore, folding operations are regarded as very efficient in the development of exponential time algorithms for the VERTEX COVER problem. Recently, the folding operation has be generalized to apply to a set of more than one vertex in a graph [6].

Branch and Search

A main technique is the *branch and search* method that has been extensively used in the development of algorithms for the VERTEX COVER problem (and for many other NP-hard problems). The method can be described as follows. Let (G, k) be an instance of the VERTEX COVER problem. Suppose that somehow a collection $\{C_1, \ldots, C_b\}$ of vertex subsets in the graph G is identified, where for each i, the subset C_i has c_i vertices, such that if the graph G contains a vertex cover of k vertices, then at least for one C_i of the vertex subsets in the collection, there is a vertex cover of k vertices for G that contains all vertices in C_i. Then a collection of (smaller) instances (G_i, k_i) can be constructed, where $1 \leq i \leq b, k_i = k - c_i$, and G_i is obtained from G by removing all vertices in C_i. Note that the original graph G has a vertex cover of k vertices if and only if for one (G_i, k_i) of the smaller instances the graph G_i has a vertex cover of k_i vertices. Therefore, now the process can be branched into b sub-processes, each on a smaller instance (G_i, k_i) recursively searches for a vertex cover of k_i vertices in the graph G_i.

Let $T(k)$ be the number of leaves in the search tree for the above branch and search process on the instance (G, k), then the above branch operation gives the following recurrence relation:

$$T(k) = T(k - c_1) + T(k - c_2) + \cdots + T(k - c_b)$$

To solve this recurrence relation, let $T(k) = x^k$ so that the above recurrence relation becomes

$$x^k = x^{k-c_1} + x^{k-c_2} + \cdots + x^{k-c_b}$$

It can be proved [3] that the above polynomial equation has a unique root x_0 larger than 1. From this, one gets $T(k) = x_0^k$, which, up to a polynomial factor, gives an upper bound on the running time of the branch and search process on the instance (G, k).

The simplest case is that a vertex v of degree $d > 0$ in the graph G is picked. Let $w_1, \ldots, w_d$ be the neighbors of v. Then either v is contained in a vertex cover C of k vertices, or, if v is not contained in C, then all neighbors $w_1, \ldots, w_d$ of v must be contained in C. Therefore, one obtains a collection of two subsets $C_1 = \{v\}$ and $C_2 = \{w_1, \ldots, w_d\}$, on which the branch and search process can be applied.

The efficiency of a branch and search operation depends on how effectively one can identify the collection of the vertex subsets. Intuitively, the larger the sizes of the vertex subsets, the more efficient is the operation. Much effort has been made in the development of VERTEX COVER algorithms to achieve larger vertex subsets. Improvements on the size of the vertex subsets have been involved with very complicated and tedious analysis and enumerations of combinatorial structures of graphs. The current paper [3] achieved a collection of two subsets C_1 and C_2 of sizes $c_1 = 1$ and $c_2 = 6$, respectively, and other collections of vertex subsets that are at least as good as this (the techniques of kernelization and vertex folding played important roles in achieving these collections). This gives the following algorithm for the VERTEX COVER problem.

Theorem 3 *The* VERTEX COVER *problem can be solved in time* $O(kn + 1.2852^k)$.

Very recently, a further improvement over Theorem 3 has been achieved that gives an algorithm of running time $O(kn + 1.2738^k)$ for the VERTEX COVER problem [4].

Applications

The study of parameterized algorithms for the VERTEX COVER problem was motivated by ETH Zürich's DARWIN project in computational biology and computational biochemistry (see, e.g., [10, 11]). A number of computational problems in the project, such as multiple sequence alignments [10] and biological conflict resolving [11], can be formulated into the VERTEX COVER problem in which the parameter value is in general not larger than 100. Therefore, an algorithm of running time $O(kn + 1.2852^k)$ for the problem becomes very effective and practical in solving these problems.

The parameterized algorithm given in Theorem 3 has also induced a faster algorithm for another important NP-hard problem, the MAXIMUM INDEPENDENT SET problem on sparse graphs [3].

Open Problems

The main open problem in this line of research is how far one can go along this direction. More specifically, how small the constant $c > 1$ can be for the VERTEX COVER problem to have an algorithm of running time $O(c^k n^{O(1)})$? With further more careful analysis on graph combinatorial structures, it seems possible to slightly improve the current best upper bound [4] for the problem. Some new techniques developed more recently [6] also seem very promising to improve the upper bound. On the other hand, it is known that the constant c cannot be arbitrarily close to 1 unless certain unlikely consequence occurs in complexity theory [8].

Experimental Results

A number of research groups have implemented some of the ideas of the algorithm in Theorem 3 or its variations, including the Parallel Bioinformatics project in Carleton University [2], the High Performance Computing project in University of Tennessee [1], and the DARWIN project in ETH Zürich [10, 11]. As reported in [5], these implementations showed that this algorithm and the related techniques are "quite practical" for the VERTEX COVER problem with parameter value k up to around 400.

Cross-References

▶ Data Reduction for Domination in Graphs
▶ Exact Algorithms for k SAT Based on Local Search
▶ Local Approximation of Covering and Packing Problems
▶ Vertex Cover Kernelization

Recommended Reading

1. Abu-Khzam F, Collins R, Fellows M, Langston M, Suters W, Symons C (2004) Kernelization algorithms for the vertex cover problem: theory and experiments. In: Proceedings of the workshop on algorithm engineering and experiments (NLENEX), pp 62–69
2. Cheetham J, Dehne F, Rau-Chaplin A, Stege U, Taillon P (2003) Solving large FPT problems on coarse grained parallel machines. J Comput Syst Sci 67:691–706
3. Chen J, Kanj IA, Jia W (2001) Vertex cover: further observations and further improvements. J Algorithm 41:280–301
4. Chen J, Kanj IA, Xia G (2006) Improved parameterized upper bounds for vertex cover. In: MFCS 2006. Lecture notes in computer science, vol 4162. Springer, Berlin, pp 238–249
5. Fellows M (2001) Parameterized complexity: the main ideas and some research frontiers. In: ISAAC 2001. Lecture notes in computer science, vol 2223. Springer, Berlin, pp 291–307
6. Fomin F, Grandoni F, Kratsch D (2006) Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm. In: Proceedings of the 17th annual ACM-SIAM symposium on discrete algorithms (SODA 2006), pp 18–25
7. Garey M, Johnson D (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco
8. Impagliazzo R, Paturi R (2001) Which problems have strongly exponential complexity? J Comput Syst Sci 63:512–530
9. Nemhauser GL, Trotter LE (1975) Vertex packing: structural properties and algorithms. Math Program 8:232–248
10. Roth-Korostensky C (2000) Algorithms for building multiple sequence alignments and evolutionary trees. PhD thesis, ETH Zürich, Institute of Scientific Computing
11. Stege U (2000) Resolving conflicts from problems in computational biology. PhD thesis, ETH Zürich, Institute of Scientific Computing

Visualization Techniques for Algorithm Engineering

Camil Demetrescu[1,2] and Giuseppe F. Italiano[1,2]
[1]Department of Computer and Systems Science, University of Rome, Rome, Italy
[2]Department of Information and Computer Systems, University of Rome, Rome, Italy

Keywords

Using visualization in the empirical assessment of algorithms

Years and Authors of Summarized Original Work

2002; Demetrescu, Finocchi, Italiano, Näher

Problem Definition

The whole process of designing, analyzing, implementing, tuning, debugging and experimentally evaluating algorithms can be referred to as *Algorithm Engineering*. Algorithm Engineering views algorithmics also as an engineering discipline rather than a purely mathematical discipline. Implementing algorithms and engineering algorithmic codes is a key step for the transfer of algorithmic technology, which often requires a high-level of expertise, to different and broader communities, and for its effective deployment in industry and real applications.

Experiments can help measure practical indicators, such as implementation constant factors, real-life bottlenecks, locality of references, cache effects and communication complexity, that may be extremely difficult to predict theoretically. Unfortunately, as in any empirical science, it may be sometimes difficult to draw general conclusions about algorithms from experiments. To this aim, some researchers have proposed accurate and comprehensive guidelines on different aspects of the empirical evaluation of algorithms maturated from their own experience in the field (see, for example [1, 15, 16, 20]). The interested reader may find in [18] an annotated bibliography of experimental algorithmics sources addressing methodology, tools and techniques.

The process of implementing, debugging, testing, engineering and experimentally analyzing algorithmic codes is a complex and delicate task, fraught with many difficulties and pitfalls. In this context, traditional low-level textual debuggers or industrial-strength development environments can be of little help for algorithm engineers, who are mainly interested in high-level algorithmic ideas rather than in the language and platform-dependent details of actual implementations. Algorithm visualization environments provide tools for abstracting irrelevant program details and for conveying into still or animated images the high-level algorithmic behavior of a piece of software.

Among the tools useful in algorithm engineering, visualization systems exploit interactive graphics to enhance the development, presentation, and understanding of computer programs [27]. Thanks to the capability of conveying a large amount of information in a compact form that is easily perceivable by a human observer, visualization systems can help developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. Some examples of this kind of usage are described in [12].

Key Results

Systems for algorithm visualization have matured significantly since the rise of modern computer graphic interfaces and dozens of algorithm visualization systems have been developed in the last two decades [2, 3, 4, 5, 6, 8, 9, 10, 13, 17, 25, 26, 29]. For a comprehensive survey the interested reader can be referred to [11, 27] and to the references therein. The remainder of this entry discusses the features of algorithm visualization systems that appear to be most appealing for their deployment in algorithm engineering.

Critical Issues

From the viewpoint of the algorithm developer, it is desirable to rely on systems that offer visualizations at a *high level of abstraction*. Namely, one would be more interested in visualizing the behavior of a complex data structure, such as a graph, than in obtaining a particular value of a given pointer.

Fast prototyping of visualizations is another fundamental issue: algorithm designers should be allowed to create visualization from the source code at hand with little effort and without heavy modifications. At this aim, *reusability* of visualization code could be of substantial help in speeding up the time required to produce a running animation.

One of the most important aspects of algorithm engineering is the development of *libraries*. It is thus quite natural to try to interface visualization tools to algorithmic software libraries: libraries should offer default visualizations of algorithms and data structures that can be refined and customized by developers for specific purposes.

Software visualization tools should be able to animate *not just "toy programs"*, but significantly complex algorithmic codes, and to test their behavior on large data sets. Unfortunately, even those systems well suited for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck. Finding a solution to this kind of limitations is nowadays a challenge.

Advanced debuggers take little advantage of sophisticated graphical displays, even in commercial software development environments. Nevertheless, software visualization tools may be very beneficial in addressing problems such as finding memory leaks, understanding anomalous program behavior, and studying performance. In particular, environments that provide interpreted execution may more easily integrate advanced facilities in support to *debugging and performance monitoring*, and many recent systems attempt at exploring this research direction.

Techniques

One crucial aspect in visualizing the dynamic behavior of a running program is the way it is conveyed into graphic abstractions. There are two main approaches to bind visualizations to code: the event-driven and the state-mapping approach.

Event-Driven Visualization

A natural approach to algorithm animation consists of annotating the algorithmic code with calls to visualization routines. The first step consists of identifying the relevant actions performed by the algorithm that are interesting for visualization purposes. Such relevant actions are usually referred to as *interesting events*. As an example, in a sorting algorithm the swap of two items can be considered an interesting event. The second step consists of associating each interesting event with a modification of a graphical scene. Animation scenes can be specified by setting up suitable visualization procedures that drive the graphic system according to the actual parameters generated by the particular event. Alternatively, these visualization procedures may simply log the events in a file for a *post-mortem* visualization. The calls to the visualization routines are usually obtained by annotating the original algorithmic code at the points where the interesting events take place. This can be done either by hand or by means of specialized editors. Examples of toolkits based on the event-driven approach are Polka [28] and *GeoWin*, a C++ data type that can be easily interfaced with algorithmic software libraries of great importance in algorithm engineering such as CGAL [14] and LEDA [19].

State Mapping Visualization

Algorithm visualization systems based on state mapping rely on the assumption that observing how the variables change provides clues to the actions performed by the algorithm. The focus is on capturing and monitoring the data modifications rather than on processing the interesting events issued by the annotated algorithmic code. For this reason they are also referred to as “data driven” visualization systems. Conventional debuggers can be viewed as data driven systems, since they provide direct feedback of variable modifications. The main advantage of this approach over the event-driven technique is that a much greater ignorance of the code is allowed: indeed, only the interpretation of the variables has to be known to animate a program. On the other hand, focusing only on data modification may sometimes limit customization possibilities making it difficult to realize animations that would be natural to express with interesting events. Examples of tools based on the state mapping approach are Pavane [23, 25], which marked the first paradigm shift in algorithm visualization since the introduction of interesting events, and Leonardo [10] an integrated environment for developing, visualizing, and executing C programs.

A comprehensive discussion of other techniques used in algorithm visualization appears in [7, 21, 22, 24, 27].

Applications

There are several applications of visualization in algorithm engineering, such as testing and debugging of algorithm implementations, visual inspection of complex data structures, identification of performance bottlenecks, and code optimization. Some examples of uses of visualization in algorithm engineering are described in [12].

Open Problems

There are many challenges that the area of algorithm visualization is currently facing. First of all, the real power of an algorithm visualization system should be in the hands of the final user, possibly inexperienced, rather than of a professional programmer or of the developer of the tool. For instance, instructors may greatly benefit from fast and easy methods for tailoring animations to their specific educational needs, while they might be discouraged from using systems that are difficult to install or heavily dependent on particular software/hardware platforms. In addition to being easy to use, a software visualization tool should be able to animate significantly complex algorithmic codes without requiring a lot of effort. This seems particularly important for future development of visual debuggers. Finally, visualizing the execution of algorithms on large data sets seems worthy of further investigation. Currently, even systems designed for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck, such as changes of resolution and scale, selectivity, and elision of information.

Cross-References

▶ Experimental Methods for Algorithm Analysis

Recommended Reading

1. Anderson RJ (2002) The role of experiment in the theory of algorithms. In: Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges. DIMACS series in discrete mathematics and theoretical computer science, vol 59. American Mathematical Society, Providence, pp 191–195
2. Baker J, Cruz I, Liotta G, Tamassia R (1996) Animating geometric algorithms over the web. In: Proceedings of the 12th annual ACM symposium on computational geometry. Philadelphia, 24–26 May 1996, pp C3–C4
3. Baker J, Cruz I, Liotta G, Tamassia R (1996) The Mocha algorithm animation system. In: Proceedings of the 1996 ACM workshop on advanced visual interfaces, Gubbio, 27–29 May 1996, pp 248–250
4. Baker J, Cruz I, Liotta G, Tamassia R (1996) A new model for algorithm animation over the WWW. ACM Comput Surv 27:568–572
5. Baker R, Boilen M, Goodrich M, Tamassia R, Stibel B (1999) Testers and visualizers for teaching data structures. In: Proceeding of the 13th SIGCSE technical symposium on computer science education, New Orleans, 24–28 Mar 1999, pp 261–265
6. Brown M (1988) Algorithm animation. MIT Press, Cambridge
7. Brown M (1988) Perspectives on algorithm animation. In: Proceedings of the ACM SIGCHI'88 conference on human factors in computing systems. Washington, DC, 15–19 May 1988, pp 33–38
8. Brown M (1991) Zeus: a system for algorithm animation and multi-view editing. In: Proceedings of the 7th IEEE workshop on visual languages, Kobe, 8–11 Oct 1991, pp 4–9
9. Cattaneo G, Ferraro U, Italiano GF, Scarano V (2002) Cooperative algorithm and data types animation over the net. J Vis Lang Comp 13(4):391
10. Crescenzi P, Demetrescu C, Finocchi I, Petreschi R (2008) Reversible execution and visualization of programs with LEONARDO. J Vis Lang Comp 11:125–150, Leonardo is available at: http://www.dis.uniroma1.it/~demetres/Leonardo/. Accessed 15 Jan 2008
11. Demetrescu C (2001) Fully dynamic algorithms for path problems on directed graphs. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza"
12. Demetrescu C, Finocchi I, Italiano GF, Näher S (2002) Visualization in algorithm engineering: tools and techniques. In: Experimental algorithm design to robust and efficient software. Lecture notes in computer science, vol 2547, Springer, Berlin, pp 24–50
13. Demetrescu C, Finocchi I, Liotta G (2000) Visualizing algorithms over the web with the publication-driven approach. In: Proceedings of the 4th workshop on algorithm engineering (WAE'00), Saarbrücken, 5–8 Sept 2000
14. Fabri A, Giezeman G, Kettner L, Schirra S, Schönherr S (1996) The cgal kernel: a basis for geometric computation. In: Applied computational geometry: towards geometric engineering proceedings (WACG'96), Philadelphia, 27–28 May 1996, pp 191–202
15. Goldberg A (1999) Selecting problems for algorithm evaluation. In: Proceedings of the 3rd workshop on algorithm engineering (WAE'99), London, 19–21 July 1999. Lecture notes in computer science, vol 1668, pp 1–11
16. Johnson D (2002) A theoretician's guide to the experimental analysis of algorithms. In: Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS series in discrete mathematics and theoretical computer science, vol 59. American Mathematical Society, Providence, 215–250
17. Malony A, Reed D (1989) Visualizing parallel computer system performance. In: Simmons M, Koskela R, Bucher I (eds) Instrumentation for future parallel computing systems. ACM Press, New York, pp 59–90
18. McGeoch C (2002) A bibliography of algorithm experimentation. In: Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges. DIMACS series in discrete mathematics and theoretical computer science, vol 59. American Mathematical Society, Providence, pp 251–254
19. Mehlhorn K, Naher S (1999) LEDA: a platform of combinatorial and geometric computing. Cambridge University Press, Cambridge. ISBN 0-521-56329-1
20. Moret B (2002) Towards a discipline of experimental algorithmics. In: Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges. dimacs series in discrete mathematics and theoretical computer Science,

vol 59. American Mathematical Society, Providence, pp 197–214
21. Myers B (1990) Taxonomies of visual programming and program visualization. J Vis Lang Comp 1:97–123
22. Price B, Baecker R, Small I (1993) A principled taxonomy of software visualization. J Vis Lang Comp 4:211–266
23. Roman G, Cox K (1989) A declarative approach to visualizing concurrent computations. Computer 22:25–36
24. Roman G, Cox K (1993) A taxonomy of program visualization systems. Computer 26:11–24
25. Roman G, Cox K, Wilcox C, Plun J (1992) PAVANE: a system for declarative visualization of concurrent computations. J Vis Lang Comp 3:161–193
26. Stasko J (1992) Animating algorithms with X-TANGO. SIGACT News 23:67–71
27. Stasko J, Domingue J, Brown M, Price B (1997) Software visualization: programming as a multimedia experience. MIT Press, Cambridge
28. Stasko J, Kraemer E (1993) A methodology for building application- specific visualizations of parallel programs. J Parallel Distrib Comp 18:258–264
29. Tal A, Dobkin D (1995) Visualization of geometric algorithms. IEEE Trans Vis Comp Graph 1:194–204

Voltage Scheduling

Ming Min Li
Computer Science and Technology, Tsinghua University, Beijing, China

Keywords

Dynamic speed scaling

Years and Authors of Summarized Original Work

2005; Li, Yao

Problem Definition

This problem is concerned with scheduling jobs with as little energy as possible by adjusting the processor speed wisely. This problem is motivated by *dynamic voltage scaling (DVS)* (or *speed scaling*) technique, which enables a processor to operate at a range of voltages and frequencies. Since energy consumption is at least a quadratic function of the supply voltage (hence CPU frequency/speed), it saves energy to execute jobs as slowly as possible while still satisfying all timing constraints. The associated scheduling problem is referred to as min-energy DVS scheduling. Previous work showed that the min-energy DVS schedule can be computed in cubic time. The work of Li and Yao [7] considers the discrete model where the processor can only choose its speed from a finite speed set. This work designs an $O(dn \log n)$ two-phase algorithm to compute the min-energy DVS schedule for the discrete model (d represents the number of speeds) and also proves a lower bound of $\Omega(n \log n)$ for the computation complexity.

Notations and Definitions

In the variable voltage scheduling model, there are two important sets:

1. Set J (job set) consists of n jobs: $j_1, j_2, \ldots j_n$. Each job j_k has three parameters as its information: a_k representing the arrival time of j_k, b_k representing the deadline of j_k, and R_k representing the total CPU cycles required by j_k. The parameters satisfy $0 \le a_k < b_k \le 1$.
2. Set *SD* (speed set) consists of the possible speeds that can be used by the processor. According to the property of *SD*, the scheduling model is divided into the following two categories:

   Continuous model: The set *SD* is the set of positive real numbers.

   Discrete model: The set *SD* consists of d positive values: $s_1 > s_2 > \cdots > s_d$.

A schedule S consists of the following two functions: $s(t)$ which specifies the processor speed at time t and job(t) which specifies the job executed at time t. Both functions are piecewise constant with finitely many discontinuities.

A feasible schedule must give each job its required number of cycles between arrival time and deadline, therefore satisfying the property $\int_{a_k}^{b_k} s(t)\delta(k, \mathrm{job}(t))\mathrm{d}t = R_k$, where $\delta(i, j) = 1$ if $i = j$ and $\delta(i, j) = 0$ if otherwise.

The EDF principle defines an ordering on the jobs according to their deadlines. At any time t, among jobs j_k that are available for execution, that is, j_k satisfying $t \in [a_k, b_k)$ and j_k not yet finished by t, it is the job with minimum b_k that will be executed during $[t, t + \epsilon]$.

The power P, or energy consumed per unit of time, is a convex function of the processor speed. The energy consumption of a schedule $S = (s(t), \mathrm{job}(t))$ is defined as $E(S) = \int_0^1 P(s(t))\mathrm{d}t$.

A schedule is called an optimal schedule if its energy consumption is the minimum possible among all the feasible schedules. Note that for the continuous model, the optimal schedule uses the same speed for the same job.

The work of Li and Yao considers the problem of computing an optimal schedule for the discrete model under the following assumptions.

Assumptions

1. **Single processor:** At any time t, only one job can be executed.
2. **Preemptive:** Any job can be interrupted during its execution.
3. **Non-precedence:** There is no precedence relationship between any pair of jobs.
4. **Offline:** The processor knows the information of all the jobs at time 0.

This problem is called min-energy discrete dynamic voltage scaling (MEDDVS).

Problem 1 (MEDDVS$_{J,SD}$)

INPUT: Integer n, set $J = \{j_1, j_2, \ldots, j_n\}$ and $SD = \{s_1, s_2, \ldots, s_d\} \cdot j_k = \{a_k, b_k, R_k\}$.
OUTPUT: Feasible schedule $S = (s(t), \mathrm{job}(t))$ that minimizes $E(S)$.

Kwon and Kim [6] proved that the optimal schedule for the discrete model can be obtained by first calculating the optimal schedule for the continuous model and then individually adjusting the speed of each job appropriately to adjacent levels in set SD. The time complexity is $O(n^3)$.

Key Results

The work of Li and Yao finds a direct approach for solving the MEDDVS problem without first computing the optimal schedule for the continuous model.

Definition 1 An s-schedule for J is a schedule which conforms to the EDF principle and uses constant speed s in executing any job of J.

Lemma 1 *The s-schedule for J can be computed in $O(n \log n)$ time.*

Definition 2 Given a job set J and any speed s, let $J^{\geq s}$ and $J^{<s}$ denote the subset of J consisting of jobs whose executing speeds are $\geq s$ and $<s$, respectively, in the optimal schedule for J in the continuous model. The partition $J^{\geq s}$, $J^{<s}$ is referred to as the s-partition of J.

By extracting information from the s-schedule, a partition algorithm is designed to prove the following lemma:

Lemma 2 *The s-partition of J can be computed in $O(n \log n)$ time.*

By applying s-partition to J using all the d speeds in SD consecutively, one can obtain d subsets $J_1, J_2, \ldots, J_d$ of J where jobs in the same subset J_i use the same two speeds s_i and s_{i+1} in the optimal schedule for the Discrete Model ($s_{d+1} = 0$).

Lemma 3 *Optimal schedule for job set J_i using speeds s_i and s_{i+1} can be computed in $O(n \log n)$ time.*

Combining the above three lemmas together, the main theorem follows:

Theorem 1 *The min-energy discrete DVS schedule can be computed in $O(dn \log n)$ time.*

A lower bound to compute the optimal schedule for the *discrete model* under the algebraic decision tree model is also shown by Li and Yao.

Theorem 2 *Any deterministic algorithm for computing min-energy discrete DVS schedule with $d \geq 2$ voltage levels requires $\Omega(n \log n)$ time for n jobs.*

Applications

Currently, dynamic voltage scaling technique is being used by the world's largest chip companies, e.g., Intel's SpeedStep technology and AMD's PowerNow technology. Although the scheduling algorithms being used are mostly online algorithms, offline algorithms can still find their places in real applications. Furthermore, the techniques developed in the work of Li and Yao for the computation of optimal schedules may have potential applications in other areas.

People also study energy-efficient scheduling problems for other kinds of job sets. Yun and Kim [10] proved that it is NP-hard to compute the optimal schedule for jobs with priorities and gave an FPTAS for that problem. Aydin et al. [1] considered energy-efficient scheduling for real-time periodic jobs and gave an $O(n^2 \log n)$ scheduling algorithm. Chen et al. [4] studied the weakly discrete model for non-preemptive jobs where speed is not allowed to change during the execution of one job. They proved the NP-hardness to compute the optimal schedule.

Another important application for this work is to help investigating scheduling model with more hardware restrictions (Burd and Brodersen [3] explained various design issues that may happen in dynamic voltage scaling). Besides the single-processor model, people are also interested in the multiprocessor model [11].

Open Problems

A number of problems related to the work of Li and Yao remain open. In the discrete model, Li and Yao's algorithm for computing the optimal schedule requires time $O(dn \log n)$. There is a gap between this and the currently known lower bound $\Omega(n \log n)$. Closing this gap when considering d as a variable is an open problem.

Another open research area is the computation of the optimal schedule for the continuous model. Li, Yao, and Yao [8] obtained an $O(n^2 \log n)$ algorithm for computing the optimal schedule. The bottleneck for the log n factor is in the computation of s-schedules. Reducing the time complexity for computing s-schedules is an open problem. It is also possible to look for other methods to deal with the continuous model.

Cross-References

- ▸ List Scheduling
- ▸ Online Load Balancing of Temporary Tasks
- ▸ Parallel Algorithms for Two Processors Precedence Constraint Scheduling
- ▸ Shortest Elapsed Time First Scheduling

Recommended Reading

1. Aydin H, Melhem R, Mosse D, Alvarez PM (2001) Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In: Euromicro conference on real-time systems, Madrid. IEEE Computer Society, Washington, DC, pp 225–232
2. Bansalm N, Kimbrel T, Pruhs K (2004) Dynamic speed scaling to manage energy and temperature. In: Proceedings of the 45th annual IEEE symposium on foundations of computer science, Rome. IEEE Computer Society, Washington, DC, pp 520–529
3. Burd TD, Brodersen RW (2000) Design issues for dynamic voltage scaling. In: Proceedings of the 2000 international symposium on low power electronics and design, Rapallo. ACM, New York, pp 9–14
4. Chen JJ, Kuo TW, Lu HI (2005) Power-saving scheduling for weakly dynamic voltage scaling devices workshop on algorithms and data structures (WADS). LNCS, vol 3608. Springer, Berlin, pp 338–349
5. Irani S, Pruhs K (2005) Algorithmic problems in power management. ACM SIGACT News 36(2):63–76. New York

6. Kwon W, Kim T (2005) Optimal voltage allocation techniques for dynamically variable voltage processors. ACM Trans Embed Comput Syst 4(1):211–230. New York
7. Li M, Yao FF (2005) An efficient algorithm for computing optimal discrete voltage schedules. SIAM J Comput 35(3):658–671. Society for Industrial and Applied Mathematics, Philadelphia
8. Li M, Yao AC, Yao FF (2005) Discrete and continuous min-energy schedules for variable voltage processors. In: Proceedings of the National Academy of Sciences USA, Washington, DC, vol 103. National Academy of Science of the United States of America, Washington, DC, pp 3983–3987
9. Yao F, Demers A, Shenker S (1995) A scheduling model for reduced CPU energy. In: Proceedings of the 36th annual IEEE symposium on foundations of computer science, Milwaukee. IEEE Computer Society, Washington, DC, pp 374–382
10. Yun HS, Kim J (2003) On energy-optimal voltage scheduling for fixed-priority hard real-time systems. ACM Trans Embed Comput Syst 2:393–430. ACM, New York
11. Zhu D, Melhem R, Childers B (2001) Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In: Proceedings of the 22nd IEEE real-time systems symposium (RTSS'01), London. IEEE Computer Society, Washington, DC, pp 84–94

Voronoi Diagrams and Delaunay Triangulations

Rolf Klein
Institute for Computer Science, University of Bonn, Bonn, Germany

Keywords

Computational geometry; Delaunay triangulation; Distance problem; Edge flip; Minimum spanning tree; Sweepline; Voronoi diagram

Years and Authors of Summarized Original Work

1975; Shamos, Hoey
1987; Fortune

Problem Definition

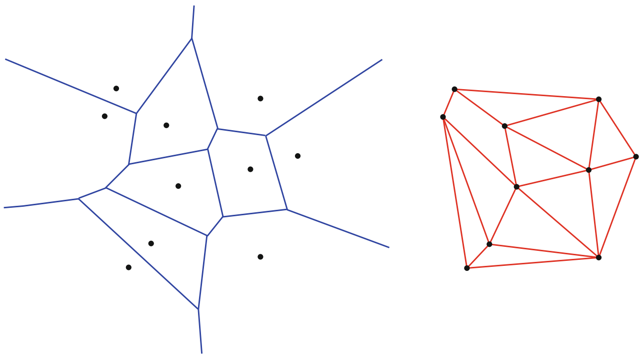
Suppose there is some set of objects p called *sites* that exert influence over their surrounding space, M. For each site p, we consider the set of all points z in M for which the influence of p is strongest.

Such decompositions have already been considered by R. Descartes [5] for the fixed stars in solar space. In mathematics and computer science, they are called *Voronoi diagrams*, honoring work by G.F. Voronoi on quadratic forms. Other sciences know them as *domains of action, Johnson-Mehl model, Thiessen polygons, Wigner-Seitz zones*, or *medial axis transform*.

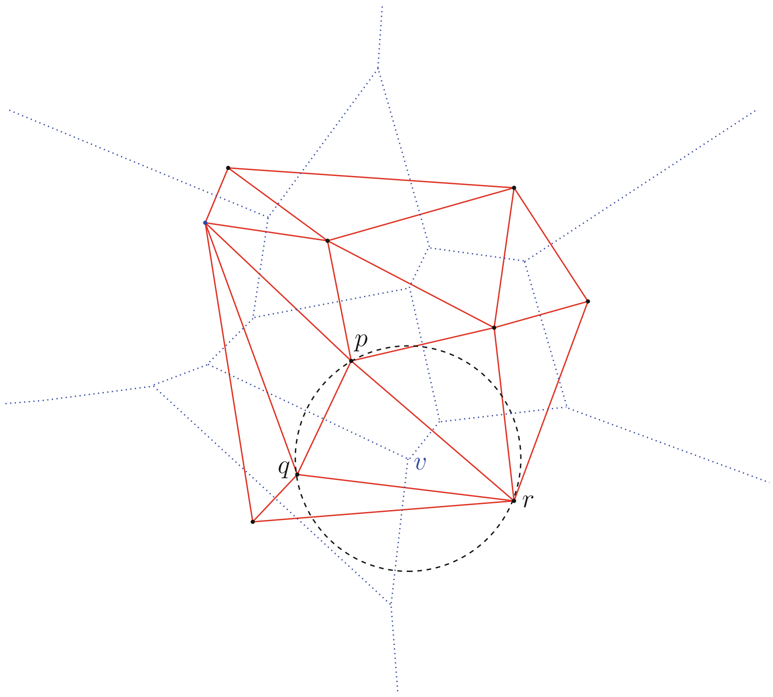
In the case most frequently studied, the space M is the real plane, the sites are n points, and influence corresponds to proximity in the Euclidean metric, so that the points most strongly influenced by site p are those for which p is the nearest neighbor among all sites. They form a convex region called the *Voronoi region* of p. The common boundary of two adjacent regions of p and q is a segment of their *bisector* $B(p,q)$, the locus of all points of equal distance to p and q. An example of 10 point sites is depicted in Fig. 1.

Let us assume that the set S of point sites is in general position, so that no three points are situated on a line, and no four on a circle. Then the Voronoi diagram $V(S)$ of S is a connected planar graph. Its vertices are those points in the plane which have three nearest neighbors in S, while the interior edge points have two. As a consequence of the Euler formula, $V(S)$ has only $O(n)$ many edges and vertices.

If we connect with line segments, those sites in S whose Voronoi regions share an edge in $V(S)$, a triangulation $D(S)$ of S results, called the *Delaunay triangulation* or *Dirichlet tessellation*; see Fig. 1. Each triangle with vertices p, q, r in S is dual to a vertex v of $V(S)$ situated on the boundary of the Voronoi regions of p, q, and r. Because p, q, r are the nearest neighbors of v in S, the circle through p, q, r centered at v contains no other point of S. Thus, $D(S)$ consists of triangles with vertices in S whose circumcircles are empty of points in S; see Fig. 2. Conversely,



Voronoi Diagrams and Delaunay Triangulations, Fig. 1 Voronoi diagram and Delaunay triangulation of 10 point sites in the Euclidean plane



Voronoi Diagrams and Delaunay Triangulations, Fig. 2 The empty circle property

each triangle with empty circumcircle occurs in $D(S)$.

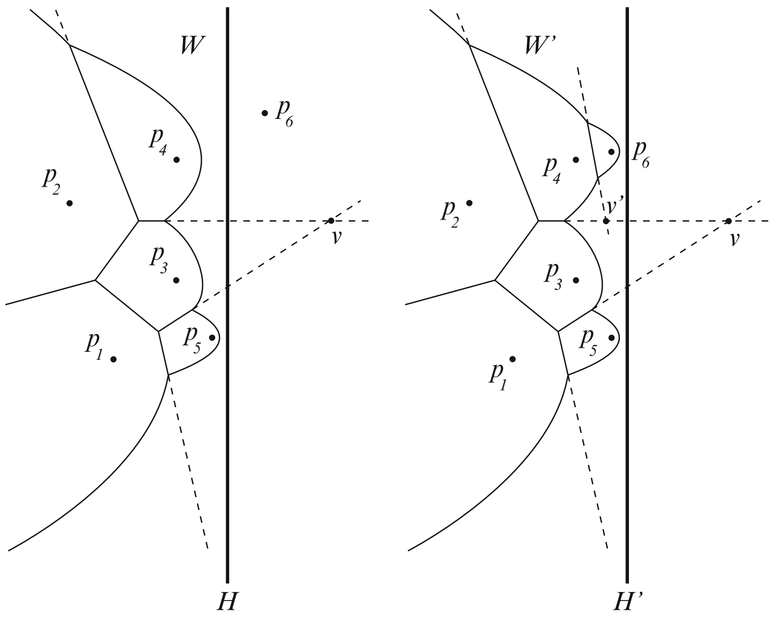
Given a set S of n point sites, the problem is to efficiently construct one of $V(S)$ or $D(S)$; the dual structure can then easily be obtained in linear time.

Generalizations

Voronoi diagrams can be generalized in several ways. Instead of point sites, other geometric objects can be considered. One can replace the Euclidean distance with distance measures more suitable to model a given situation. Instead of forming regions of all points that have the same nearest site, one can consider higher-order Voronoi diagrams where all points share a region for which the nearest k sites are the same, for some k between 2 and $n-1$. Many more variants can be found in [9] and [1]. Abstract Voronoi diagrams provide a unifying framework for some of the variants mentioned; see the corresponding chapter in this encyclopedia.

Key Results

Quite a few algorithms for constructing the Voronoi diagram or the Delaunay triangulation



Voronoi Diagrams and Delaunay Triangulations, Fig. 3 The sweepline advancing to the right

of n points in the Euclidean plane have been developed.

Divide and Conquer

The first algorithm was presented in the seminal paper [11], which gave birth to the field of computational geometry. It applies the *divide and conquer* paradigm. Site set S is split by a line into subsets L and R of equal cardinality. After recursively computing $V(L)$ and $V(R)$, one needs to compute the bisector $B(L, R)$, the locus of all points in the plane that have a nearest neighbor in L and in R. This bisector is an unbounded monotone polygonal chain. In time $O(n)$ one can find a starting segment of $B(L, R)$ at infinity, and trace the chain through $V(L)$ and $V(R)$ simultaneously. Thus, the algorithm runs in time $O(n \log n)$ and linear space, which is optimal.

Sweep

How to design a left-to-right *sweepline* algorithm for constructing $V(S)$ is not obvious. When the advancing sweepline H enters the Voronoi region of p before site p has been detected, it is not clear how to correctly maintain the Voronoi diagram along H. This difficulty has been overcome in [7] by applying a transformation that ensures that each site is the leftmost point of its Voronoi region. In [10] and [4], a more direct version of this approach was suggested. At each time during the sweep, one maintains the Voronoi diagram of all point sites to the left of sweepline H, and of H itself, which is considered a site of its own; see Fig. 3. Because the bisector of a point and a line is a parabola, the Voronoi region of H is bounded by a connected chain of parabolic segments, called the *wavefront* W. As H moves to the right, W follows at half the speed. Each point z to the left of W is closer to some point site p left of H than to H and, all the more, to all point sites to the right of H that are yet to be discovered. Thus, the Voronoi regions of the point sites to the left of W keep growing, as sweepline H proceeds, along the extensions of Voronoi edges beyond W; these *spikes* are depicted by dashed lines in Fig. 3.

There are two kinds of events one needs to handle during the sweep. When sweepline H hits a new point site (like point p_6 in Fig. 3), a new wave separating this point site from H must be added to W. When wavefront W hits the intersection of two neighboring spikes, the wave between them must be removed from W;

this will first happen in Fig. 3 when W' arrives at v'. Intersections between neighboring spikes can be determined as in the standard line segment intersection algorithm [2]. There are only $O(n)$ many events, one for each point site and one for each Voronoi vertex v of $V(S)$. Since wavefront W is always of linear size, the sweepline algorithm runs in $O(n \log n)$ time using linear space.

Reduction to Convex Hull

A rather different approach [3] obtains the Delaunay triangulation in dimension 2 from the convex hull in dimension 3, which can itself be constructed in time $O(n \log n)$. As suggested in [6], one vertically lifts the point sites to the paraboloid $Z = X^2 + Y^2$ in 3-space. The lower convex hull of the lifted points, projected onto the XY-plane, equals the Delaunay triangulation, $D(S)$.

Incremental Construction

Another, very intuitive algorithm first suggested in [8] constructs the Delaunay triangulation incrementally. In order to insert a new point site p_i into an existing Delaunay triangulation $D(S_{i-1})$, one first finds the triangle containing p_i and connects p_i to its vertices by line segments. Should p_i be contained in the circumcircles of adjacent triangles, the Delaunay property must be restored by *edge flips* that replace the common edge of two adjacent triangles T, T' by the other diagonal of the convex quadrilateral formed by T and T'. If the insertion sequence of the p_i is randomly chosen, a running time in $O(n \log n)$ can be expected. Details on all algorithms can be found in [1].

Applications

Although of linear size, Voronoi diagram and Delaunay triangulation contain a lot of information on the point set S. Once $V(S)$ or $D(S)$ are available, quite a few distance problems can be solved very efficiently. We mention only the most basic applications here and refer to [1] and [9] for further reading.

By definition, the Voronoi diagram reduces the *post office* or *nearest neighbor* problem to a point location problem: given an arbitrary query point z, the site in S nearest to z can be found by determining the Voronoi region containing z. In order to find the *largest empty circle* whose center z lies inside a convex polygon C over m vertices, one needs to inspect only three types of candidates for z, the vertices of $V(S)$, the intersections of the edges of $V(S)$ with the boundary of C, and the vertices of C. All these can be done in time $O(n+m)$.

If the site set S is split into subsets L and R, then the closest pair $p \in L$ and $q \in R$ forms an edge of the Delaunay triangulation $D(S)$ (which crosses the Voronoi edge separating the regions of p and q). This fact has nice consequences. First, the nearest neighbor of a site $p \in S$ must be one of its neighboring vertices in $D(S)$. Hence, *all nearest neighbors* and the *closest pair* in S can be found in linear time once $D(S)$ is available, because $D(S)$ has only $O(n)$ many edges. Second, $D(S)$ contains the *minimum spanning tree* of S, which can be extracted from $D(S)$ in linear time.

Remarkable and useful is the *equiangularity property* of $D(S)$. Of all (exponentially many) triangulations of S, the Delaunay triangulation maximizes the ascending sequence of angles occurring in the triangles, with respect to lexicographic order. In particular, the minimum angle is as large as possible. In fact, if a triangulation is not Delaunay, it must contain two adjacent triangles, such that the circumcircle of one contains the third vertex of the other. By flipping their common edge, a new triangulation with larger angles is obtained.

Cross-References

- ▶ 3D Conforming Delaunay Triangulation
- ▶ Abstract Voronoi Diagrams
- ▶ Delaunay Triangulation and Randomized Constructions
- ▶ Optimal Triangulation

Recommended Reading

1. Aurenhammer F, Klein R, Lee DT (2013) Voronoi diagrams and delaunay triangulations. World Scientific, Singapore
2. Bentley J, Ottman T (1979) Algorithms for reporting and counting geometric intersections. IEEE Trans Comput C-28:643–647
3. Brown KQ (1979) Voronoi diagrams from Convex Hulls. Inf Process Lett 9:223–228
4. Cole R (1989) as reported by ÓDúnlaing, oral communication
5. Descartes R (1644) Principia philosophiae. Ludovicus Elsevirius, Amsterdam
6. Edelsbrunner H, Seidel R (1986) Voronoi diagrams and arrangements. Discret Comput Geom 1:25–44
7. Fortune S (1978) A sweepline algorithm for Voronoi diagrams. Algorithmica 2:153–174
8. Green PJ, Sibson RR (1978) Computing dirichlet tessellations in the plane. Comput J 21: 168–173
9. Okabe A, Boots B, Sugihara K, Chiu SN (2000) Spatial tessellations: concepts and applications of Voronoi diagrams. Wiley, Chichester
10. Seidel R (1988) Constrained delaunay triangulations and Voronoi diagrams with obstacles. Technical report 260, TU Graz
11. Shamos MI, Hoey D (1975) Closest-point problems. In: Proceedings 16th annual IEEE symposium on foundations of computer science, Berkeley, pp 151–162