V

Valid-Utility Games

► Market Games and Content Distribution

VCG

► Generalized Vickrey Auction

Vector Sorting

► String Sorting

Vertex Coloring

► Distributed Vertex Coloring
► Exact Graph Coloring Using Inclusion–Exclusion

Vertex Cover Data Reduction

► Vertex Cover Kernelization

Vertex Cover Kernelization

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

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

Keywords and Synonyms

Vertex cover preprocessing; Vertex cover data reduction

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:

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

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:

$$\begin{aligned} I_0 &= \{v_i \mid x_i^0 < 0.5\}, \\ C_0 &= \{v_i \mid x_i^0 > 0.5\}, \text{ and} \\ V_0 &= \{v_i \mid x_i^0 = 0.5\} \end{aligned}$$

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. $I_0' = \{v_i \mid$ if neither v_i^L nor v_i^R is in $C_B\}$;
   $C_0' = \{v_i \mid$ if both v_i^L and v_i^R are in $C_B\}$;
   $V_0' = \{v_i \mid$ if exactly one of v_i^L and v_i^R is in $C_B\}$

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.: Kernelization algorithms for the vertex cover problem: theory and experiments. In: Proc. Workshop on Algorithm Engineering and Experiments (ALENEX) pp. 62–69 (2004)
2. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Ann. Discret. Math. **25**, 27–45 (1985)
3. Chen, J., Kanj, I.A., Jia, W.: Vertex cover: further observations and further improvements. J. Algorithm **41**, 280–301 (2001)
4. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-completeness. Freeman, San Francisco (1979)
5. Nemhauser, G.L., Trotter, L.E.: Vertex packing: structural properties and algorithms. Math. Program. **8**, 232–248 (1975)

Vertex Cover Preprocessing

► Vertex Cover Kernelization

Vertex Cover Search Trees

2001; Chen, Kanj, Jia

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

Keywords and Synonyms

Branch and search; Branch and bound

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
- ▶ Local Approximation of Covering and Packing Problems
- ▶ Local Search Algorithms for kSAT
- ▶ Vertex Cover Kernelization

Recommended Reading

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

Vickrey–Clarke–Groves Mechanism

- ▶ Generalized Vickrey Auction

Visualization Techniques for Algorithm Engineering

2002; Demetrescu, Finocchi, Italiano, Näher

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

Keywords and Synonyms

Using visualization in the empirical assessment of algorithms

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 im-

ages 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, R.J.: 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, pp. 191–195. American Mathematical Society, Providence, RI (2002)
2. Baker, J., Cruz, I., Liotta, G., Tamassia, R.: Animating Geometric Algorithms over the Web. In: Proceedings of the 12th Annual ACM Symposium on Computational Geometry. Philadelphia, Pennsylvania, May 24–26, pp. C3–C4 (1996)
3. Baker, J., Cruz, I., Liotta, G., Tamassia, R.: The Mocha Algorithm Animation System. In: Proceedings of the 1996 ACM Workshop on Advanced Visual Interfaces. Gubbio, Italy, May 27–29, pp. 248–250 (1996)
4. Baker, J., Cruz, I., Liotta, G., Tamassia, R.: A New Model for Algorithm Animation over the WWW, ACM Comput. Surv. **27**, 568–572 (1996)
5. Baker, R., Boilen, M., Goodrich, M., Tamassia, R., Stibel, B.: Testers and Visualizers for Teaching Data Structures. In: Proceeding of the 13th SIGCSE Technical Symposium on Computer Science Education. New Orleans, March 24–28, pp. 261–265 (1999)
6. Brown, M.: Algorithm Animation. MIT Press, Cambridge, MA (1988)
7. Brown, M.: Perspectives on Algorithm Animation. In: Proceedings of the ACM SIGCHI'88 Conference on Human Factors in Computing Systems. Washington, D.C., May 15–19, pp. 33–38 (1988)
8. Brown, M.: Zeus: a System for Algorithm Animation and Multi-View Editing. In: Proceedings of the 7th IEEE Workshop on Visual Languages. Kobe, Japan, October 8–11, pp. 4–9 (1991)
9. Cattaneo, G., Ferraro, U., Italiano, G.F., Scarano, V.: Cooperative Algorithm and Data Types Animation over the Net.J.Visual Lang.Comp. 13(4): 391– (2002)
10. Crescenzi, P., Demetrescu, C., Finocchi. I., Petreschi, R.: Reversible Execution and Visualization of Programs with LEONARDO. J. Visual Lang. Comp. **11**, 125–150 (2000). Leonardo is available at: http://www.dis.uniroma1.it/~demetres/Leonardo/. Accessed 15 Jan 2008
11. Demetrescu, C.: Fully Dynamic Algorithms for Path Problems on Directed Graphs, Ph. D. thesis, Department of Computer and Systems Science, University of Rome "La Sapienza" (2001)
12. Demetrescu, C., Finocchi, I., Italiano, G.F., Näher, S.: Visualization in algorithm engineering: tools and techniques. In: Experimental Algorithm Design to Robust and Effizient Software. Lecture Notes in Computer Science, vol. 2547. Springer, Berlin, pp. 24–50 (2002)
13. Demetrescu, C., Finocchi, I., Liotta, G.: Visualizing Algorithms over the Web with the Publication-driven Approach. In: Proc. of the 4th Workshop on Algorithm Engineering (WAE'00), Saarbrücken, Germany, 5–8 September (2000)

14. Fabri, A., Giezeman, G., Kettner, L., Schirra, S., Schönherr, S.: The cgal kernel: A basis for geometric computation. In: Applied Computational Geometry: Towards Geometric Engineering Proceedings (WACG'96), Philadelphia. Philadelphia, PA, May 27–28, pp. 191–202 (1996)
15. Goldberg, A.: Selecting problems for algorithm evaluation. In: Proc. 3rd Workshop on Algorithm Engineering (WAE'99). LNCS, vol. 1668. London, United Kingdom, July 19–21, pp. 1–11 (1999)
16. Johnson, D.: 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, 59, 215–250. American Mathematical Society, Providence, RI (2002)>
17. Malony, A., Reed, D.: Visualizing Parallel Computer System Performance. In: Simmons, M., Koskela, R., Bucher, I. (eds.) Instrumentation for Future Parallel Computing Systems. ACM Press, New York (1989) pp. 59–90
18. McGeoch, C.: A bibliography of algorithm experimentation. In: Data Struktures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 59, 251–254. American Mathematical Society, Providence, RI (2002)
19. Mehlhorn, K., Naher, S.: LEDA: A Platform of Combinatorial and Geometric Computing, ISBN 0–521-56329–1. Cambrige University Press, Cambrige (1999)
20. Moret, B.: 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, 59, 197–214. American Mathematical Society, Providence, RI (2002)
21. Myers, B.: Taxonomies of Visual Programming and Program Visualization. J. Visual Lang. Comp. **1**, 97–123 (1990)
22. Price, B., Baecker, R., Small, I.: A Principled Taxonomy of Software Visualization. J. Visual Lang. Comp. **4**, 211–266 (1993)
23. Roman, G., Cox, K.: A Declarative Approach to Visualizing Concurrent Computations. Computer **22**, 25–36 (1989)
24. Roman, G., Cox, K.: A Taxonomy of Program Visualization Systems. Computer **26**, 11–24 (1993)
25. Roman, G., Cox, K., Wilcox, C., Plun, J.: PAVANE: a System for Declarative Visualization of Concurrent Computations. J. Visual Lang. Comp. **3**, 161–193 (1992)
26. Stasko, J.: Animating Algorithms with X-TANGO. SIGACT News **23**, 67–71 (1992)
27. Stasko, J., Domingue, J., Brown, M., Price B.: Software Visualization: Programming as a Multimedia Experience. MIT Press, Cambridge, MA (1997)
28. Stasko, J., Kraemer, E.: A Methodology for Building Application-Specific Visualizations of Parallel Programs. J. Parall. Distrib. Comp. **18**, 258–264 (1993)
29. Tal, A., Dobkin, D.: Visualization of Geometric Algorithms. IEEE Trans. Visual. Comp. Graphics **1**, 194–204 (1995)

Voltage Scaling

► Speed Scaling

Voltage Scheduling

2005; Li, Yao

Minming Li
Department of Computer Science, City University of Hong Kong, Hong Kong, China

Keywords and Synonyms

Dynamic speed scaling

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 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 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 $0 \leq a_k < b_k \leq 1$.
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: $s_1 > s_2 > \ldots > s_d$.

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: $\int_{a_k}^{b_k} s(t)\delta(k, job(t))\,dt = R_k$, where $\delta(i, j) = 1$ if $i = j$ and $\delta(i, j) = 0$ otherwise.

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 $S = (s(t), job(t))$ is defined as $E(S) = \int_0^1 P(s(t))\,dt$.

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, 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 $SD = \{s_1, s_2, \ldots, s_d\}$. $j_k = \{a_k, b_k, R_k\}$.*

OUTPUT: *Feasible schedule $S = (s(t), job(t))$ 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 $\langle J^{\geq s}, J^{<s}\rangle$ 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 4 *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 5 *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 kind 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 an 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
- ► Load Balancing
- ► Parallel Algorithms for Two Processors Precedence Constraint Scheduling
- ► Shortest Elapsed Time First Scheduling

Recommended Reading

1. Aydin, H., Melhem, R., Mosse, D., Alvarez, P.M.: Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. Euromicro Conference on Real-Time Systems, pp. 225–232. IEEE Computer Society, Washington, DC, USA (2001)
2. Bansalm, N., Kimbrel, T., Pruhs, K.: Dynamic Speed Scaling to Manage Energy and Temperature, Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, pp. 520–529. IEEE Computer Society, Washington, DC, USA (2004)
3. Burd, T.D., Brodersen, R.W.: Design Issues for Dynamic Voltage Scaling, Proceedings of the 2000 international symposium on Low power electronics and design, pp. 9–14. ACM, New York, USA (2000)
4. Chen, J.J., Kuo, T.W., Lu, H.I.: Power-Saving Scheduling for Weakly Dynamic Voltage Scaling Devices Workshop on Algorithms and Data Structures (WADS). LNCS, vol. 3608, pp. 338–349. Springer, Berlin, Germany (2005)
5. Irani, S., Pruhs, K.: Algorithmic Problems in Power Management. ACM SIGACT News **36**(2), 63–76. New York, NY, USA (2005)
6. Kwon, W., Kim, T.: Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors. ACM Trans. Embed. Comput. Syst. **4**(1), 211–230. New York, NY, USA (2005)
7. Li, M., Yao, F.F.: An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules. SIAM J. Comput. **35**(3), 658–671. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2005)
8. Li, M., Yao, A.C., Yao, F.F.: Discrete and Continuous Min-Energy Schedules for Variable Voltage Processors, Proceedings of the National Academy of Sciences USA, 103, pp. 3983–3987. National Academy of Science of the United States of America, Washington, DC, USA (2005)
9. Yao, F., Demers, A., Shenker, S.: A Scheduling Model for Reduced CPU Energy, Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, pp. 374–382. IEEE Computer Society, Washington, DC, USA (1995)
10. Yun, H.S., Kim, J.: On Energy-Optimal Voltage Scheduling for Fixed-Priority Hard Real-Time Systems. ACM Trans. Embed. Comput. Syst. **2**, 393–430. ACM, New York, NY, USA (2003)
11. Zhu, D., Melhem, R., Childers, B.: Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor RealTime Systems. Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01), pp. 84–94. IEEE Computer Society, Washington, DC, USA (2001)

Voting Systems

- ► Quorums