Scalable Parallel Algorithms for Difficult Combinatorial Problems: A Case Study in Optimization[*]

Faisal N. Abu-Khzam, Michael A. Langston[†] and Pushkar Shanbhag
Department of Computer Science, University of Tennessee, Knoxville, TN 37996–3450

Abstract

A novel combination of emergent algorithmic methods, powerful computational platforms and supporting infrastructure is described. These complementary tools and technologies are used to launch systematic attacks on combinatorial problems of significance. As a case study, optimal solutions to very large instances of the $\mathcal{NP}$-hard vertex cover problem are computed. To accomplish this, an efficient sequential algorithm and two forms of parallel algorithms are devised and implemented. The importance of maintaining a balanced decomposition of the search space is shown to be critical to achieving scalability. With the synergistic combination of techniques detailed here, it is now possible to solve problem instances that before were widely viewed as hopelessly out of reach. Target problems need only be amenable to reduction and decomposition. Applications are also discussed.

Key Words

Algorithm Design, Parallel Computing, Optimization, Load Balancing, Applications

1 Preliminaries

An innovative technique for dealing with foundational $\mathcal{NP}$-complete problems is based on the theory of *fixed-parameter tractability*.

> A problem of size n, parameterized by k, is fixed-parameter tractable if it can be decided in $O(f(k)n^c)$ time, where f is an arbitrary function and c is a constant independent of both n and k.

The origins of fixed-parameter tractability (henceforth FPT) can be traced back some 15 odd years, to the work by Fellows and Langston on applications of well-quasi order theory, the Robertson-Seymour theorems, nonconstructivity, and in particular the minor and immersion orders. See, for example, [9, 10, 11]. Efforts at that time were motivated by the theme that, by fixing or bounding parameters of relevance to the problem at hand, one might be able to exploit a non-uniform measure of algorithmic efficiency. In the intervening years, Downey and Fellows developed the major theoretical basis of FPT [8]. More recently, something of a cottage industry in FPT algorithm design has begun to flourish, with research groups and workshops now held around the world. Despite all this activity, however, the main focus has remained on theoretical issues, especially worst-case bounds, problem restrictions and the $\mathcal{W}$-hierarchy (a fixed-parameter analog of the polynomial hierarchy). Few serious attempts have been made at large-scale practical implementations. A notable exception is the work of Cheetham et al [4].

2 Exemplar

Perhaps the best-known example of an FPT problem, and the one we use as a case study here, is *vertex cover.* In this problem, the inputs are an undirected graph G with n vertices, and a parameter $k < n$. The question asked is whether G contains a set C of k or fewer vertices that covers every edge in G, where an edge is said to be covered if either (or both) of its endpoints is in C.

In terms of worst-case analysis, the asymptotically-fastest algorithm currently known for vertex cover is due to the work of Chen et al [5], and runs in $O(1.2852^k + kn)$ time. Compare this with $O(n^k)$, the time required to examine all subsets of size k by brute force. Of course an attractive worst-case bound is no guarantee of a practical algorithm. Nevertheless, it is remarkable that the requisite exponential growth (assuming $\mathcal{P} \neq \mathcal{NP}$) has been reduced to a mere additive term.

Algorithms designed to solve FPT problems are sometimes rather loosely termed “fixed-parameter algorithms.” Such algorithms were originally intended to work only when the parameter in question was truly fixed. The algorithm described in [3], for example, was aimed solely at determining whether an input graph has a vertex cover of size at most 5.

In contrast, our interest here is on pushing the boundary of feasible computation. We seek to construct effective methods for finding optimal vertex covers in huge graphs, irrespective of any particular parameter value. To accomplish this, we exploit, build upon and implement techniques gleaned in large part from recent advances in the theory of fixed-parameter algorithm design. We detail the salient features of some of these techniques in the next section.

[*] This research is supported in part by the National Science Foundation under grants EIA–9972889 and CCR–0075792, by the Office of Naval Research under grant N00014–01–1–0608, by the Department of Energy under contract DE–AC05–00OR22725, and by the Tennessee Center for Information Technology Research under award E01–0178–261.

[†] Communicating author: langston@cs.utk.edu.

3 Reduction

The goal of problem reduction is to condense an arbitrary input instance down to a relatively small computational core. The idea is to find a core whose size depends only on k, and to find it in time polynomial in n. In the context of FPT, this operation is usually termed "kernelization," and is most often accomplished with a variety of preprocessing rules. Many of these rules are folklore. Others have been formalized in the literature [2, 5].

For example, for vertex cover it is possible with preprocessing to eliminate vertices of very low or high degree. It is trivial to eliminate vertices of degree one, because there is no gain in using a leaf to cover its only incident edge. It is straightforward to eliminate vertices of degree two. There is also a rule for handling some but not all vertices of degree three, but it is complicated and not necessarily worth the extra effort in practice. If a vertex has degree $k+1$ or more, then it must be in any satisfying cover. Otherwise, all its neighbors would be required to be in the cover, and there are simply too many of them. Therefore we can eliminate it and reduce k by one. There is also a rule for handling vertices of degree exactly k. When no more vertices can be eliminated in this fashion, the reduced graph (core) has size at most k^2. We have implemented these and several other preprocessing rules. The order in which they are performed can sometimes have an impact on the size of the core produced. Their most important attributes are probably their speed and simplicity.

Newer and more powerful (but also slower and more complicated) kernelization methods rely on linear programming relaxation and related techniques [12, 13, 14]. We have implemented and fine-tuned these and alternate methods of our own design. We have incorporated where possible highly-efficient LP codes graciously provided to us by Bill Cook at Georgia Tech [6]. The culmination of all this is a suite of routines that, when used in conjunction, quickly produce a core of size at most $2k$. Details about their use can be found in [1].

4 Decomposition

As soon as reduction is complete, the core is ready to be passed to the decomposition stage. The problem now becomes one of exploring the core's search space efficiently. In the parlance of FPT, this is known as "branching." This is an extremely challenging task. Even though the core is now of bounded size, its search space typically contains an exponential number of candidate solutions. We use an implicit tree structure and a depth-first search to organize the search for a satisfying cover. Each internal node of the tree represents a choice. For example, one might make the choice at the root by selecting an arbitrary vertex, v. Then the left subtree may denote the set of all solutions in which v is to be in the cover. The right subtree may denote the set of all solutions in which v is not in the cover. This oversimplification alone is enough to reveal one of many curiosities: solutions are often found faster should it be that v is not in the cover. This is because, when v is unused, all its neighbors must be in the cover and, if the degree of v is high, we converge much more rapidly toward a solution. Moving on down the tree, each leaf is a set of k or fewer vertices that may or may not form a valid cover, corresponding to a potential solution. Although effective, this form of decomposition is an exhaustive process to be sure. (This should come as no surprise. After all, the underlying problem is $\mathcal{NP}$-complete.)

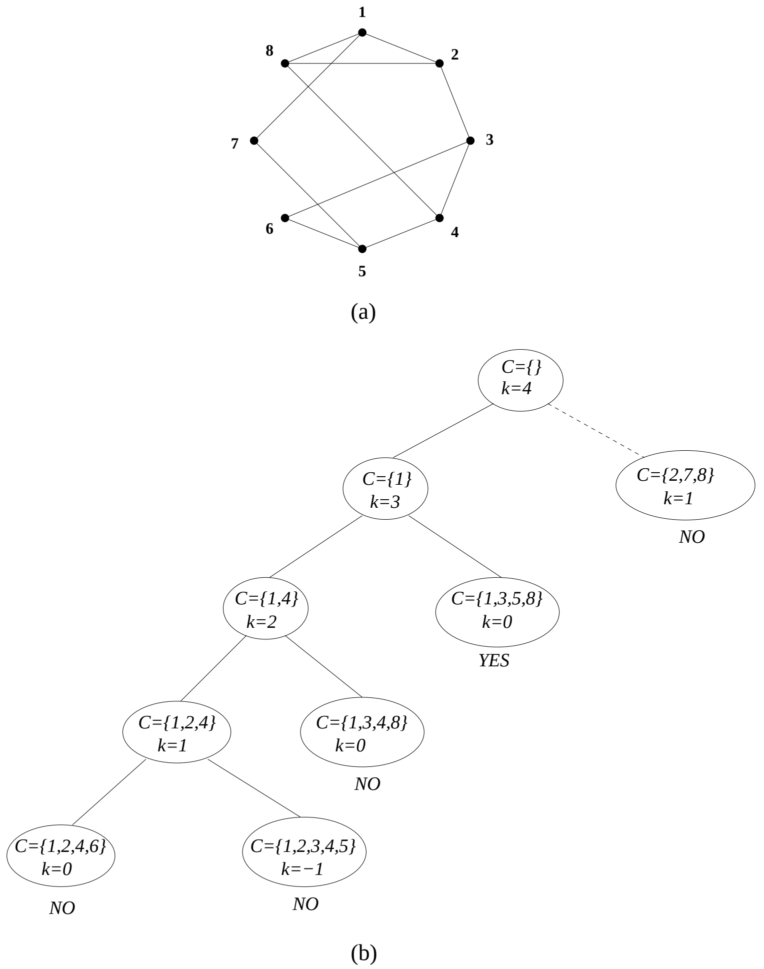
Of course reduction and decomposition need not be stand-alone tasks. As decomposition proceeds, new instances generated can sometimes be further reduced by a re-application of preprocessing rules. This technique is often termed "interleaving." See [15] for more information and analysis. At this time, our codes use interleaving in only a rudimentary way, by eliminating isolated vertices and by re-applying the degree one and degree $k + 1$ rules. More complex interleaving strategies are currently under development.

Decomposition clearly requires the lion's share of computational resources. Thus, it is important to note that the subtrees spawned off at each level can be explored in parallel. Moreover, the depth of the tree can be at most k. All that needs to be done at a leaf is to check whether the removal of the leaf's candidate solution leaves an edgeless graph (all edges are covered).

Decomposition via the branching process is illustrated in Figure 1. Figure 1(a) shows a sample graph for which we want to find a vertex cover of size four. Figure 1(b) depicts a resultant tree search, rooted at the vertex whose label is 1. At each node of the tree, C denotes the candidate cover and k denotes the maximum number of vertices that may still be added to C. (Therefore $|C| + k = 4$.) In this example, we have favored branching at a node of highest current degree. Because a depth-first search is employed, and because a solution is eventually found in the root's left subtree, the dotted edge leading to the root's right subtree need not be traversed by a sequential algorithm. This is not a property easily exploited by parallel algorithms. As we shall see, parallel algorithms may be very lucky or very unlucky as the solution space is decomposed.

5 Parallelism

Parallelization works hand-in-hand with the results of decomposition. The task of spawning processes is structured by the same tree that is used to explore the core's search space. To explicate, suppose both n and k are large, and 32 processors are available. Because the search tree has a branching factor of two, decomposition will have used the first $5 << k$ levels of its tree to split the input into 32 subgraphs, one for each processor. In turn, each processor will, in parallel, examine its subgraph using the search tree technique.



(a)

(b)

Figure 1. The use of branching to find a vertex cover.

Once spawned, these tasks are left to run in a virtually unstructured manner. They can be farmed out as the need arises, and serviced in anything from a tightly-coupled to a widely-distributed fashion. No barrier synchronization is needed. No MPI-like tools are required. A process need not even know its siblings exist. Each is free to run to completion, at its own pace, returning its result whenever it is finished. We have run our codes on several different platform/gridware combinations. Our best results have generally been obtained with minimal intervention, however, in the extreme case by directly launching secure shells (SSHs).

6 Initial Results on Synthetic Data

We first tested our algorithms on synthetic data sets. These were mostly pseudo-random graphs generated in a variety of ways. The results were always impressive. In fact our best results were obtained on synthetic data sets graciously provided by Frank Dehne at Carleton University [7]. See Table 1. Numbers listed there reflect wall clock times.

These results are intriguing. Three different graphs are listed, each with 600 vertices, and each containing a vertex cover of size 400. On them we used 32 processors, each running at 500 MHz. At first we thought the sequential routine must be hung. Traces revealed, however, that it was humming along nicely. It is just that these graphs have a lot of edges and their search spaces are huge. The average speedup we observe is something north of 30,000. Because we are only employing 32 processors, this would surely have to be characterized as super-super-linear!

We have sought to determine what factors could have caused this unusually fortuitous sort of parallel behavior. Should we abandon the streamlined sequential code, and re-write it so that it uses multiple threads or otherwise emulates the actions of the parallel version?

One factor is the way in which solutions are scattered about the search space. It has been observed before [4] that solutions tend to be highly non-uniformly distributed. In this setting, therefore, decomposition can do much more

Graph Name	Sequential Reduction	Sequential Decomposition	Parallel Decomposition
RG30	1 second	halted after two days	5 seconds
RG31	1 second	halted after two days	4 seconds
RG32	1 second	halted after two days	4 seconds

Table 1. Intriguing results on synthetic graphs of size 600.

for us than merely help guide the search and parallelize the process. One or more processors may find a solution relatively close to the root of its respective subtree. The searches occurring at other processors may be fruitless; it matters not. Our parallel run time is based solely on the time required for the earliest-finishing processor to deliver to us a solution, at which point the other processors are halted. Yet the sequential algorithm is doomed to plod along, exhaustively examining each and every subtree until it stumbles across a solution-laden region of the search space.

Although the scenario just described is plausible enough, we did a little more digging and believe the primary factor at play here is really just the makeup of the data itself. These synthetic graphs are somewhat grid-like, which for technical reasons places our parallel algorithm at a great advantage. In short, we were lucky. Thus, the super-super-linear speedups we observe are mainly artifacts of the data. Nevertheless, these results do serve an important purpose. They caution us against trying to read too much into contrived examples, and echo a familiar story: *rely only on real data*.

7 Dynamic Load Balancing

As we move toward the use of non-synthetic data, the generic parallel decomposition technique just outlined has proved useful in many of our experiments. The result has generally been much smaller runtimes than those for corresponding sequential codes. In some nagging cases, however, we would observe only very small, sometimes even negligible, speedups. In an occasional extreme case, all but one of the processors would finish quickly, leaving the lone remaining processor to do the bulk of the computation. In short, we were unlucky. Observe that we can get lucky, achieving excellent speedup, only on “yes” instances. But we can get unlucky, achieving little or no speedup at all, on both “yes” and “no” instances. Interestingly, we have found that as we iterate the decision algorithm to attain optimality, the closer we get to converging the parameter to its optimal value, the more likely we are to get unlucky.

Thus, to maintain scalability as more and more machines come on line, it has been imperative that we incorporate at least some primitive form of dynamic load balancing into our methods. We have studied a number of strategies but, as we show in the next section, even a simple scheme seems to have a tremendous impact. We hesitate to interrupt active processes, and therefore refrain from redistributing processor loads until all processors but one are idle. Of course one could probably invent data for which all processors but two, or three, or any fixed number are idle. And we are building out our system to be robust enough to handle these cases and others of their ilk. But thus far this is not what real data seems to tell us is happening.

8 Results on Non-Synthetic Data

Many applications of vertex cover rely on its relationship to *clique*. In this problem, the inputs are an undirected graph G with n vertices, and a parameter $k < n$. The question asked is whether G contains a set C of k or more vertices such that every pair of elements in C is connected by an edge in G.

Clique is not FPT (unless the $\mathcal{W}$ hierarchy collapses). Fortunately, vertex cover is a complementary dual to clique. To see this, suppose we wish to determine whether G contains a large clique, where large means of size at least $n - k$ for some suitable choice of k. Let $\overline{G}$ denote the complement of G. Then G has a clique of size at least $n - k$ if and only if $\overline{G}$ has a vertex cover of size at most k. This duality is depicted in Figure 2. Figure 2(a) shows a clique in a sample graph, G_1. Figure 2(b) shows the corresponding vertex cover in $G_2 = \overline{G_1}$.

An important use of clique is in the analysis of protein sequence data that is now widely available from a variety of sources. For example, given such data, relations between sequences can be determined using codes such as the well-known ClustalW package, which returns a score for each sequence pair. A complete graph, G, is then constructed, with vertices denoting sequences and edges labeled with the corresponding scores. The source data often contains outliers, duplicates and so forth, and so in many applications we seek first to obtain the largest possible set of closely-related sequences before proceeding with the analysis.

Since we are interested in a maximum set of closely related sequences, we need to find in G a set of vertices whose pairwise scores are greater than a certain (biolog-

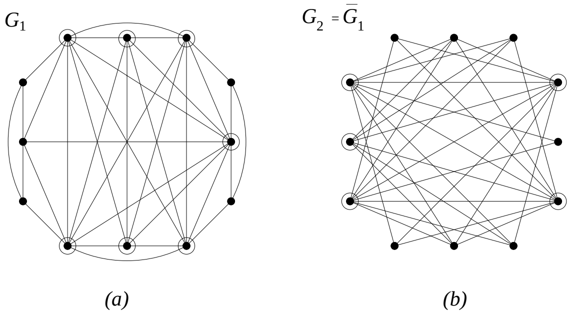


Figure 2. The duality between clique and vertex cover.

ically significant) threshold. To accomplish this, edges whose labels are less than the threshold are removed. This produces a new graph, G', and it remains to find in G' a set of vertices that are completely related. Of course this is just a restatement of the clique problem.

Working with colleagues in proteomics, we have downloaded vast assortments of sequence data against which to test our codes. These have been obtained mainly from the National Center for Biotechnology Information (NCBI). Each data set corresponds to a family of protein sequences that share a common domain. A representative set of results using data from the SH2 and SH3 domains is reported in Table 2. As before, we used 32 processors, each running at 500 MHz. Wall clock times are listed.

These results are particularly telling, because the relevant parameter is just converging on the optimal value. Note the significance of dynamic load balancing.

9 Conclusions

We believe this case study has been fruitful. By coupling algorithms based on the notion of fixed-parameter tractability with parallel computing platforms, we think we have identified an attractive way to design scalable parallel algorithms for difficult optimization problems. Certain features, however, most notably load balancing, are critical.

Some of our methods are being incorporated into the parallel, high-performance release of Clustal, dubbed Clustal-XP, due out soon. We are currently adding hardware acceleration in the form of Pilchard FPGA boards into our system, in an effort to handle particularly recalcitrant subproblems. We are also exploring mechanisms for avoiding our primitive load-balancing interrupts, for example, by allowing heavily-loaded processors to spawn subtrees out to a job queue that is maintained by a task manager.

Despite these and other growing pains, we think our results to date are worthy of attention. Problems as large as those listed in Table 2 were until recently considered by many to be hopelessly out of reach. In clique applications alone, we are now routinely returning cliques whose vertices number in the hundreds, on graphs whose vertices number in the thousands. Just imagine a straightforward $O(n^k)$ algorithm on problems of that size! In fact we recently solved a problem on DNA microarray data whose size was 12,422. The clique we returned (via vertex cover) denoted a set of 369 genes that appear experimentally to be co-regulated. This one took us a few days to solve even with our best current methods. Yet solving it at all was probably unthinkable just a short time ago. We believe the practical implications of this work are manifest.

Acknowledgments

We are grateful to Mike Fellows and Fran Rosamond for their collaboration and encouragement, to Bill Cook for providing us with the streamlined linear programming routines we employed in our reduction algorithms, and to Frank Dehne for access to some of the data sets we used in preliminary testing of our implementations.

References

[1] Faisal N. Abu-Khzam. *Topics in Graph Algorithms: Structural Results and Algorithmic Techniques, with Applications*. PhD thesis, Dept. of Computer Science, University of Tennessee, 2003.

[2] J.F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22:560–572, 1993.

[3] Kevin Cattell and Michael J. Dinneen. A characterization of graphs with vertex cover up to five. In *ORDAL*, pages 86–99, 1994.

[4] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon. Solving large FPT problems on coarse grained parallel machines. Technical report, Department of Computer Science, Carleton University, Ottawa, Canada, 2002.

Graph Name	Graph Size	Cover Size	Instance Type	Sequential Reduction	Sequential Decomposition	Parallel Decomposition	Dynamic Decomposition
SH2-5	839	399	yes	34 seconds	7 seconds	not needed	not needed
SH2-5	839	398	no	34 seconds	141 minutes	82 minutes	20 minutes
SH3-10	2466	2044	yes	203 minutes	just under five days	just under five days	140 minutes
SH3-10	2466	2043	no	203 minutes	halted after six days	halted after six days	620 minutes

Table 2. Representative results on large non-synthetic graphs.

[5] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[6] W. Cook. Private communication, 2003.

[7] F. Dehne. Private communication, 2003.

[8] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[9] M. R. Fellows and M. A. Langston. Nonconstructive advances in polynomial-time complexity. *Information Processing Letters*, 26:157–162, 1987.

[10] M. R. Fellows and M. A. Langston. Nonconstructive tools for proving polynomial-time decidability. *Journal of the ACM*, 35:727–739, 1988.

[11] M. R. Fellows and M. A. Langston. On search, decision and the efficiency of polynomial-time algorithms. *Journal of Computer and Systems Sciences*, 49:769–779, 1994.

[12] D. Hochbaum. *Approximation Algorithms for $\mathcal{NP}$-hard Problems*. PWS, 1997.

[13] S. Khuller. The vertex cover problem. *ACM SIGACT News*, 33:31–33, June 2002.

[14] G.L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[15] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter tractacle algorithms. *Information Processing Letters*, 73:125–129, 2000.