Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments*

Faisal N. Abu-Khzam†, Rebecca L. Collins‡, Michael R. Fellows§, Michael A. Langston‡, W. Henry Suters‡ and Christopher T. Symons‡

Abstract

A variety of efficient kernelization strategies for the classic vertex cover problem are developed, implemented and compared experimentally. A new technique, termed *crown reduction*, is introduced and analyzed. Applications to computational biology are discussed.

1 Introduction

The computational challenge posed by $\mathcal{NP}$-hard problems has inspired the development of a wide range of algorithmic techniques. Due to the seemingly intractable nature of these problems, practical approaches have historically concentrated on the design of polynomial-time algorithms that deliver only approximate solutions. The notion of fixed parameter tractability (FPT) has recently emerged as an alternative to this trend. FPT's roots can be traced at least as far back as work motivated by the Graph Minor Theorem to prove that a variety of otherwise difficult problems are decidable in low-order polynomial time when relevant parameters are fixed. See, for example, [10, 11].

Formally, a problem is FPT if it has an algorithm that runs in $O(f(k)n^c)$ time, where n is the problem size, k is the input parameter, and c is a constant [9]. A well-known example is the parameterized Vertex Cover problem. Vertex Cover is posed as an undirected graph G and a parameter k. The question asked is whether G contains a set C of k or fewer vertices such that every edge of G has at least one endpoint in C. Vertex Cover can be solved in $O(1.2852^k + kn)$ time [5] with the use of a bounded search tree technique. This technique restricts a problem's search space to a tree whose size is bounded only by a function of the relevant parameter. Vertex Cover has a host of real-world applications, particularly in the field of computational biology. It can be used in the construction of phylogenetic trees, in phenotype identification, and in analysis of microarray data, to name just a few. While the fact that the parameterized Vertex Cover problem is FPT makes the computation of exact solutions theoretically tractable, the practical matter of reducing run times to reasonable levels for large parameter values has remained a formidable challenge.

In this paper, we develop and implement a suite of algorithms, each of which takes as input a graph G of size n and a parameter k, and returns a graph G' of size $n' \leq n$ and a parameter $k' \leq k$. It is important that (1) n' is bounded by a function only of k' (not of n) and (2) G has a vertex cover of size at most k if and only if G' has a vertex cover of size at most k'. Each algorithm may be employed independently or in conjunction with others. The use of such techniques is called *kernelization*. An amenability to kernelization seems to be a hallmark of problems that are FPT, and a characteristic that distinguishes them from apparently more difficult $\mathcal{NP}$-hard problems. After kernelization is completed, the solution process reverts to *branching*. Large-scale empirical studies of branching methods are also underway. See, for example, [2].

2 Kernelization Alternatives

Our vertex cover kernelization suite consists of four separate techniques. The first method is a simple scheme based on the elimination of high degree vertices. The second and third methods reformulate vertex cover as an integer programming problem, which is then simplified using linear programming. This linear programming problem can either be solved using standard linear programming techniques or restated as a network flow problem that can then be solved using an algorithm developed by Dinic [8, 12]. The fourth method, which is new, we call *crown reduction*. It is based on finding a particular independent set and its neighborhood, both

*This research has been 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 E01–0178–081.

†Division of Computer Science and Mathematics, Lebanese American University, Chouran, Beirut 1102 2801, Lebanon

‡Department of Computer Science, University of Tennessee, Knoxville, TN 37996–3450, USA

§School of Electrical Engineering and Computer Science, University of Newcastle, Calaghan NSW 2308, Australia

of which can be removed from the graph. We develop the theoretical justification for each of these techniques, and provide examples of their performance on samples of actual application problems.

3 Preprocessing Rules

The techniques we employ are aided by a variety of preprocessing rules. These are computationally inexpensive, requiring at most $O(n^2)$ time with very modest constants of proportionality.

Rule 1: An isolated vertex (one of degree zero) cannot be in a vertex cover of optimal size. Because there are no edges incident upon such a vertex, there is no benefit in including it in any cover. Thus, in G', an isolated vertex can be eliminated, reducing n' by one. This rule is applied repeatedly until all isolated vertices are eliminated.

Rule 2: In the case of a pendant vertex (one of degree one), there is an optimal vertex cover that does not contain the pendant vertex but does contain its unique neighbor. Thus, in G', both the pendant vertex and its neighbor can be eliminated. This also eliminates any additional edges incident on the neighbor, which may leave isolated vertices for deletion under Rule 1. This reduces n' by the number of deleted vertices and reduces k' by one. This rule is applied repeatedly until all pendant vertices are eliminated.

Rule 3: If there is a degree-two vertex with adjacent neighbors, then there is a vertex cover of optimal size that includes both of these neighbors. If u is a vertex of degree 2 and v and w are its adjacent neighbors, then at least two of the three vertices (u, v, and w) must be in any vertex cover. Choosing u to be one of these vertices would only cover edges (u, v) and (u, w) while eliminating u and including v and w could possibly cover not only these but additional edges. Thus there is a vertex cover of optimal size that includes v and w but not u. G' is created by deleting u, v, w and their incident edges from G. It is then also possible to delete the neighbors of v and w whose degrees drop to zero. This reduces n' by the number of deleted vertices and reduces k' by two. This rule is applied repeatedly until all degree-two vertices with adjacent vertices are eliminated.

Rule 4: If there is a degree-two vertex, u, whose neighbors, v and w, are non-adjacent, then u can be folded by contracting edges $\{u, v\}$ and $\{u, w\}$. This is done by replacing u, v and w with one vertex, u', whose neighborhood is the union of the neighborhoods of v and w in G. This reduces the problem size by two and the parameter size by one. This idea was first proposed in [5], and warrants explanation. To illustrate, suppose u is a vertex of degree 2 with neighbors v and w. If one neighbor of u is included in the cover and is eliminated, then u becomes a pendant vertex and can also be eliminated by including its other neighbor in the cover. Thus it is safe to assume that there are two cases: first, u is in the cover while v and w are not; second v and w are in the cover while u is not. If u' is not included in an optimal vertex cover of G' then all the edges incident on u' must be covered by other vertices. Therefore v and w need not be included in an optimal vertex cover of G because the remaining edges $\{u, v\}$ and $\{u, w\}$ can be covered by u. In this case, if the size of the cover of G' is k' then the cover of G will have size $k = k' + 1$ so the decrement of k in the construction is justified. On the other hand, if u' is included in an optimal vertex cover of G' then at least some of its incident edges must be covered by u'. Thus the optimal cover of G must also cover its corresponding edges by either v or w. This implies that both v and w are in the vertex cover. In this case, if the size of the cover of G' is k', then the cover of G will also be of size $k = k' + 1$. This rule is applied repeatedly until all vertices of degree two are eliminated. If recovery of the computed vertex cover is required, a record must be kept of this folding so that once the cover of G' has been computed, the appropriate vertices can be included in the cover of G.

4 Kernelization by High Degree

This simple technique [3] relies on the observation that a vertex whose degree exceeds k must be in every vertex cover of size at most k. (If the degree of v exceeds k but v is not included in the cover, then all of v's neighbors must be in the cover, making the size of the cover at least $k + 1$.) This algorithm is applied repeatedly until all vertices of degree greater than k are eliminated. It is superlinear ($O(n^2)$) only because of the need to compute the degree of each vertex.

The following theorem is a special case of a more general result from [1]. It is used to bound the size of the kernel that results from the application of this algorithm in combination with the aforementioned preprocessing rules. Note that if this algorithm and the preprocessing rules are applied, then the degree of each remaining vertex lies in the range $[3, k']$.

THEOREM 4.1. *If G' is a graph with a vertex cover of size k', and if no vertex of G' has degree less than three or more than k', then $n' \leq \frac{k'^2}{3} + k'$.*

Proof. Let C be a vertex cover of G', with $|C| = k'$. C's

complement, $\overline{C}$, is an independent set of size $n' - k'$. Let F be the set of edges in G' with endpoints in $\overline{C}$. Since the elements of $\overline{C}$ have degree at least three, each element of $\overline{C}$ must have at least three neighbors in C. Thus the number of edges in F must be at least $3(n' - k')$. The number of edges with endpoints in C is no smaller than $|F|$ and no larger than $k'|C|$, since each element of G has at most k' neighbors. Therefore $3(n' - k') \leq |F| \leq k'^2$ and $n' \leq \frac{k'^2}{3} + k'$. ■

5 Kernelization by Linear-Programming

The optimization version of Vertex Cover can be stated in the following manner. Assign a value $X_u \in \{0, 1\}$ to each vertex u of the graph $G = (V, E)$ so that the following conditions hold.

(1) Minimize $\sum_u X_u$.

(2) Satisfy $X_u + X_v \geq 1$ whenever $\{u, v\} \in E$.

This is an integer programming formulation of the optimization problem. In this context the objective function is the size of the vertex cover, and the set of all feasible solutions consists of functions from V to $\{0, 1\}$ that satisfy condition (2). We relax the integer programming problem to a linear programming problem by replacing the restriction $X_u = \{0, 1\}$ with $X_u \geq 0$. The value of the objective function returned by the linear programming problem is a lower bound on the objective function returned by the related integer programming problem [12, 13, 14].

The solution to the linear programming problem can be used to simplify the related integer programming problem in the following manner. Let $N(S)$ denote the neighborhood of S, and define $P = \{u \in V | X_u > 0.5\}$, $Q = \{u \in V | X_u = 0.5\}$ and $R = \{u \in V | X_u < 0.5\}$. We employ the following modification by Khuller [13] of a theorem originally due to Nemhauser and Trotter [14].

THEOREM 5.1. *If P, Q, and R are defined as above, there is an optimal vertex cover that is a superset of P and that is disjoint from R.*

Proof. Let A be the set of vertices of P that are not in the optimal vertex cover and let B be the set of vertices of R that are in the optimal cover, as selected by the solution to the integer programming problem. Notice that $N(R) \subseteq P$ because of condition (2). It is not possible for $|A| < |B|$ since in this case replacing B with A in the cover decreases its size without uncovering any edges (since $N(R) \subseteq P$), and so it is not optimal. Additionally it is not possible for $|A| > |B|$ because then we could gain a better linear programming solution by setting $\epsilon = \min\{X_v - 0.5 : v \in A\}$ and replacing X_u with $X_u + \epsilon$ for all $u \in B$ and replacing X_v with $X_v - \epsilon$ for all $v \in A$. Thus we must conclude that $|A| = |B|$, and in this case we can replace B with A in the vertex cover (again since $N(R) \subseteq P$) to obtain the desired optimal cover. ■

The graph G' is produced by removing vertices in P and R and their adjacent edges. The problem size is $n' = n - |P| - |R|$ and the parameter size is $k' = k - |P|$. Notice that since the size of the objective function for the linear programming problem provides a lower bound on the objective function for the integer programming problem, the size of any optimal cover of G' is bounded below by $\sum_{u \in Q} X_u = 0.5|Q|$. If this were not the case, then the original linear programming procedure that produced Q would not have produced an optimal result. This allows us to observe that if $|Q| > 2k'$, then this is a "no" instance of the vertex cover problem.

When dealing with large dense graphs the above linear programming procedure may not be practical since the number of constraints is the number of edges in the graph. Because of this, the code used in this paper solves the dual of the LP problem, turning the minimization problem into a maximization problem, and making the number of constraints equal to the number of vertices [6, 7]. Other methods to speed LP kernelization appear in [12].

6 Kernelization by Network Flow

This algorithm solves the linear programming formulation of vertex cover by reducing it to a network flow problem. As in [14], we define a bipartite graph B in terms of the input graph G, find the vertex cover of B by computing a maximum matching on it, and then assign values to the vertices of G based on the cover of B. In our implementation of this algorithm, we compute the maximum matching on B by turning it into a network flow problem and using Dinic's maximum flow algorithm [8, 12]. The time complexity of the overall procedure is $O(m\sqrt{n})$, where m denotes the number of edges and n denotes the number of vertices in G. The size of the reduced problem kernel is bounded by $2k$.

The difference between this method of linear programming and the previous LP-kernelization is that this method is faster (LP takes $O(n^3)$) and is guaranteed to assign values in $\{0, 0.5, 1\}$, while LP codes assign values in the (closed) interval between 0 and 1. Given a graph G, the following algorithm can be used to produce an LP kernelization of G.

Step 1: Convert $G = (V, E)$ to a bipartite graph $H = (U, F)$. $U = A \cup B$, where $A = \{A_v | v \in V\}$ and $B = \{B_v | v \in V\}$. If $(v, w) \in E$, then we place both (A_v, B_w) and (A_w, B_v) in F.

Step 2: Convert the bipartite graph H to a network

flow graph H': Add a source node that has directed arcs toward every vertex in A, and add a sink node that receives directed arcs from every vertex in B. Make all edges between A and B directed arcs toward B. Give all arcs a capacity of 1.

Step 3: Find an instance of maximum flow through the graph H'. For this project we used Dinic's algorithm, but any maximum flow algorithm will work.

Step 4: The arcs in H' included in the instance of maximum flow that correspond to edges in the bipartite graph H constitute a maximum matching set, M, of H.

Step 5: From M we can find an optimal vertex cover of H. Case 1: If all vertices are included in the matching, the vertex cover of H is either the set A or the set B. Case 2: If not all vertices are included in the matching, we begin by constructing three sets S, R, and T. With the setup we have here ($|A| = |B|$ and all capacities are 1), if all vertices in A are matched, then all vertices in B are too. So we can assume that there is at least one unmatched vertex in A. Let S denote the set of all unmatched vertices in A. Let R denote the set of all vertices in A that are reachable from S by alternating paths with respect to M. Let T denote the set of neighbors of R along edges in M. The vertex cover of the bipartite graph H is $(A - S - R) \cup (T)$. The size of the cover is $|M|$.

Step 6: Assign weights to all of the vertices of G according to the vertex cover of H. For vertex v:
$W_v = 1$ if A_v and B_v are both in the cover of H.
$W_v = 0.5$ if only one of A_v or B_v is in the cover of H.
$W_v = 0$ if neither A_v nor B_v is in the cover of H.
In Case 1 of Step 5, where one of the sets A or B becomes the vertex cover, all vertices are returned with the weight 0.5.

Step 7: The graph that remains will be $G' = (V', E')$ where $V' = \{v | W_v = 0.5\}$ and $k' = k - x$ where x is the number of vertices with weight $W_v = 1$.

THEOREM 6.1. *Step 5 of this algorithm produces a valid optimal vertex cover of H.*

Proof. In Case 1, the vertex cover of H includes all of A or all of B. The size of the vertex cover is $|A| = |B| = |M|$. Without loss of generality assume the vertex cover is A. All edges in the bipartite graph have exactly one endpoint in A. Thus, every edge is covered, and the vertex cover is valid.

In Case 2, we have sets S, $R \subset A$ and $T \subset B$. The vertex cover is defined as $(A - S - R) \cup (T)$. For every edge $(x, y) \in H$, x must lie in S, R or $A - S - R$. Suppose x lies in S, in which case x is unmatched. Then y must be matched, because otherwise M would not be optimal. Hence another edge (w, y) exists in M. Then, $w \in R$ and $y \in T$, and therefore (x, y) is covered. We now argue that $N(R) = T$. By definition, $T \subseteq N(R)$. To see that $N(R) \subseteq T$, note that if y is contained in $N(R)$ and not in T, then y must be matched, because otherwise there would be an augmenting path (some neighbor of y contained in R is reachable from S by an alternating path). Thus, the neighbor of y contained in M must also be in R, and so y must lie in T. Thus, if x lies in R, y must lie in T and again edge (x, y) is covered. Finally, if x lies in $A - S - R$, (x, y) is covered by definition.

As for the size of the cover, $|S| = n - |M|$, where n is the number of vertices in the original graph. Because all elements of R are matched, and by definition, T is all vertices reachable from R by matched edges, $|T| = |R|$. Therefore the size of the cover $= |(A - S - R) \cup T| = |(A - S - R) \cup R| = |A - S| = |M|$. Since H is a bipartite graph with a maximum matching of size $|M|$, the minimum vertex cover size for H is $|M|$, so this cover is optimal. ■

THEOREM 6.2. *Step 6 of this algorithm produces a feasible solution to the linear programming formulation on G.*

Proof. Each vertex in G is assigned a weight – either 0, 0.5, or 1. For every edge $(x, y) \in G$, we want the sum of their weights, $W_x + W_y$, to be greater than or equal to 1. Thus, for every edge (x, y), the cover of H contains either A_x and B_x, A_y and B_y, A_x and B_y, or A_y and B_x. If $(x, y) \in G$, then $(A_x, B_y), (A_y, B_x) \in H$. Because H's cover contains one or both endpoints of every edge, we know that at the least either A_x and B_x, A_y and B_y, A_x and B_y, or A_y and B_x are in the cover. Therefore every edge $(x, y) \in G$ has a valid weight. ■

The graph G' is produced in the same manner as in the LP kernelization procedure and again we have the situation where a "no" instance occurs whenever $|G'| > 2k$. The time complexity of the algorithm is $O(m\sqrt{n})$ where m and n are the number of edges and vertices, respectively, in the graph G. Since there are at most $O(n^2)$ edges in the graph, this implies the overall method is $O(n^{\frac{5}{2}})$.

7 Kernelization by Crown Reduction

The technique we dub "crown reduction" is somewhat similar to the other algorithms just described. With it, we attempt to exploit the structure of the graph to identify two disjoint vertex sets H and I so that there is an optimal vertex cover containing every vertex of H

but no vertex of I. This process is based on the following definition, theorem, and algorithm.

A *crown* is an ordered pair (H, I) of disjoint vertex subsets from a graph G that satisfies the following criteria:

(1) $H = N(I)$,
(2) I is a nonempty independent set, and
(3) the edges connecting H and I contain a matching in which all elements of H are matched.

H is said to contain the *head* of the crown, whose *width* is $|H|$. I contains the *points* of the crown. This notion is depicted in Figure 1.

THEOREM 7.1. *If G is a graph with a crown (H, I), then there is an optimal vertex cover of G that contains all of H and none of I.*

Proof. Since there is a matching of the edges between H and I, any vertex cover must contain at least one vertex from each matched edge. Thus the matching will require at least $|H|$ vertices in the vertex cover. This minimum number can be realized by selecting H to be in the vertex cover. It is further noted that vertices from H can be used to cover edges that do not connect I and H, while this is not true for vertices in I. Thus, including the vertices from H does not increase, and may decrease, the size of the vertex cover as compared to including vertices from I. Therefore, there is a minimum-size vertex cover that contains all the vertices in H and none of the vertices in I. ■

The following algorithm can be used to find a crown in an arbitrary input graph.

Step 1: Find a maximal matching M_1 of the graph, and identify the set of all unmatched vertices as the set O of outsiders.

Step 2: Find a maximum auxiliary matching M_2 of the edges between O and $N(O)$.

Step 3: Let I_0 be the set of vertices in O that are unmatched by M_2.

Step 4: Repeat steps 4a and 4b until $n = N$ so that $I_{N-1} = I_N$.

Step 4a: Let $H_n = N(I_n)$.

Step 4b: Let $I_{n+1} = I_n \cup \{H_n$'s neighbors under $M_2\}$.

The desired crown is the ordered pair (H, I), where $H = H_N$ and $I = I_N$. We now determine the conditions necessary to guarantee that this algorithm is successful in finding such a crown.

THEOREM 7.2. *The algorithm produces a crown as long as the set I_0 of unmatched outsiders is not empty.*

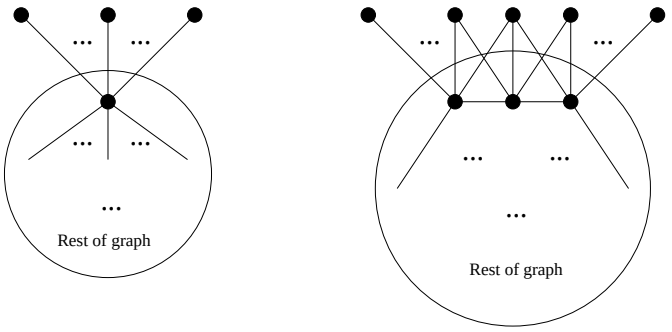
Proof. First, since M_1 is a maximal matching, the set O, and consequently its subset I, are both independent. Second, because of the definition of H, it is clear that $H = N(I_{N-1})$ and since $I = I_N = I_{N-1}$ we know that $H = N(I)$. The third condition for a crown is proven by contradiction. Suppose there were an element $h \in H$ that were unmatched by M_2. Then the construction of H would produce an augmented (alternating) path of odd length. For h to be in H there must have been an unmatched vertex in O that begins the path. Then the repeated step 4a would always produce an edge that is not in the matching while the next step 4b would produce an edge that is part of the matching. This process repeats until the vertex h is reached. The resulting path begins and ends with unmatched vertices and alternates between matched and unmatched edges. Such a path cannot exist if M_2 is in fact a maximum matching because we could increase the size of the matching by swapping the matched and unmatched edges along the path. Therefore every element of H must be matched by M_2. The actual matching used in the crown is the matching M_2 restricted to edges between H and I. ■

The graph G' is produced by removing vertices in H and I and their adjacent edges. The problem size is $n' = n - |H| - |I|$; the parameter size is $k' = k - |H|$. It is important to note that if a maximum matching of size greater than k is found, then there can be no vertex cover of size at most k, making this is a "no" problem instance. Therefore, if either of the matchings M_1 and M_2 is larger than k, the process can be halted. This fact also allows us to place an upper bound on the size of the graph G'.

THEOREM 7.3. *If M_1 and M_2 are each of size at most k, then there are no more than $3k$ vertices that lie outside the crown.*

Proof. Because the size of M_1 is at most k, it contains at most $2k$ vertices. Thus, the set O contains at least $n-2k$ vertices. Because the size of M_2 is at most k, no more than k vertices in O are matched by M_2. Thus, there are at least $n - 3k$ vertices in O that are unmatched by M_2. These vertices are included in I_0 and are therefore in I. It follows that the number of vertices in G not included in H and I is at most $3k$. ■

The particular crown produced by this decomposition depends on the maximal matching M_1 used in its calculation. This suggests that it is may be desirable



A crown of width 1 A crown of width 3

Figure 1: Sample crown decompositions.

to try to perform the decomposition repeatedly, using pseudo-randomly chosen matchings, in an attempt to identify as many crowns as possible and consequently to reduce the size of the kernel as much as possible. It may also be desirable to perform preprocessing after each decomposition, because the decomposition itself can leave vertices of low degree. The most computationally expensive part of the procedure is finding the maximum matching M_2, which we accomplish in our implementations by recasting the maximum matching problem on a bipartite graph as a network flow problem. This we then solve using Dinic's algorithm. The run time is $O(m\sqrt{n})$, which is often considerably better than $O(n^{\frac{5}{2}})$.

8 Applications and Experimental Results

Our experiments were run in the context of computational biology, where a common problem involves finding maximum cliques in graphs. Clique is $W[1]$-hard, however, and thus unlikely to be directly amenable to a fixed-parameter tractable approach [9]. Of course, a graph has a vertex cover of size k if and only if its complement has a clique of size $n - k$. We therefore exploit this duality, finding maximum cliques via minimum covers.

One of the applications to which we have applied our methods involves finding phylogenetic trees based on protein domain information, a high-throughput technique pioneered in [4]. The graphs we utilized were obtained from domain data gleaned at NCBI and SWISS-PROT, two well-known open-source repositories of biological information. Tables 1 through 3 illustrate representative results on graphs derived from the sh2 protein domain. The integer after the domain name indicates the threshold used to convert the input into an unweighted graph.

In our implementations, the high-degree method is incorporated along with the preprocessing rules. In general, we have found that the most efficient approach is to run this combination before attempting any of the other kernelization methods. To see this, compare the results of Table 1 with those of Table 2. Next, it is often beneficial to use one or more other kernelization routines. As long as the problem is not too large, network flow and linear programming are sometimes able to solve the problem without any branching whatsoever. This behavior is exemplified in Table 2. The final task is to perform branching if needed.

On very dense graphs, kernelization techniques (other than the high-degree rule) may not reduce the graph very much, if at all. Both linear programming and network flow can be computationally expensive. Because crown reduction is quick by comparison, performing it prior to branching appears to be a wise choice. This aspect of kernelization is highlighted in Table 4. Unlike the others, the graph used in this experiment was derived from microarray data, where a maximum clique corresponds to a set of putatively co-regulated genes.

9 A Few Conclusions

Crown reduction tends to run much faster in practice than does linear programming. It sometimes reduces the graph just as well, even though its worst-case bound on kernel size is larger. Given the methods at hand, the most effective approach seems to be first to run preprocessing and the high-degree algorithm, followed by crown reduction. If the remaining problem kernel is fairly sparse, then either linear programming or network flow should probably be applied before proceeding on to the branching stage. On the other hand, if the

Algorithm	run time	kernel size (n')	parameter size (k')
High Degree with Preprocessing	0.58	181	43
Linear Programming	1.15	0	0
Network Flow	1.25	36	18
Crown Reduction	0.23	328	98

Table 1: Graph: sh2-3.dim, $n = 839$, $k = 246$. Times are given in seconds.

Algorithm	run time	kernel size (n')	parameter size (k')
Linear Programming	0.05	0	0
Network Flow	0.02	0	0
Crown Reduction	0.03	69	23

Table 2: Graph: sh2-3.dim, $n = 839$, $k = 246$. Preprocessing (including the high-degree algorithm) was performed before each of the other 3 methods. Times are given in seconds.

Algorithm	run time	kernel size (n')	parameter size (k')
Linear Programming	1:09.49	616	389
Network Flow	40.53	622	392
Crown Reduction	0.07	630	392

Table 3: Graph: sh2-10.dim, $n = 726$, $k = 435$. Preprocessing (including the high-degree algorithm) was performed before each of the other 3 methods. Times are given in seconds.

Algorithm	run time	kernel size (n')	parameter size (k')
High Degree with Preprocessing	6.95	971	896
Linear Programming	37:58.95	1683	1608
Network Flow	38:21.93	1683	1608
Crown Reduction	6.11	1683	1608

Table 4: Graph: u74-0.7-75.compl, $n = 1683$, $k = 1608$, $|E| = 1,259,512$. Times are given in seconds.

kernel is relatively dense, it is probably best to avoid the cost of these methods, and instead begin branching straightaway.

Acknowledgment
We wish to thank an anonymous reader, whose thorough review of our original typescript helped us to improve the presentation of the results we report here.

References

[1] F. 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] F. N. Abu-Khzam, M. A. Langston, and P. Shanbhag. Scalable parallel algorithms for difficult combinatorial problems: A case study in optimization. In *Proceedings, International Conference on Parallel and Distributed Computing and Systems*, pages 563–568, Los Angeles, CA, November, 2003.

[3] J. F. Buss and J. Goldsmith. Nondeterminism within $\mathcal{P}$. *SIAM Journal on Computing*, 22:560–572, 1993.

[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.

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

[6] V. Chv´tal. *Linear Programming*. W.H.Freeman, New York, 1983.

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

[8] E. A. Dinic. Algorithm for solution of a problem of maximum flows in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

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

[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.