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

(Extended Abstract)

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

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, approaches have historically concentrated largely on the design of polynomial-time algorithms that deliver only approximate solutions. Fixed parameter tractability has recently emerged as an alternative to this trend, and can be traced at least as far back as the work done to show, via the Graph Minor Theorem, that a variety of parameterized problems are tractable when the relevant input parameter is fixed. See, for example, [9, 10]. Such problems are now classified as FPT. We refer the reader to [8] for a wealth of information on this emergent approach to problem complexity.

Formally, a problem is FPT if it has an algorithm that runs in $O(f(k)n^c)$, where n is the problem size, k is the input parameter, and c is a constant. A well known FPT example is the parameterized Vertex Cover problem, which is posed as an undirected graph G and a parameter k. Here the problem size n is the number of vertices. The question is if there is a set C of k or fewer vertices such that every edge of G has at least one endpoint in C. This problem can be solved in time $O(1.2852^k + kn)$ [4] by employing the bounded search tree technique, which restricts the search space of the problem to a (search) tree whose size (number of nodes) is bounded by a function of the parameter. Vertex Cover has many real-world applications, including many in the field of bioinformatics. It can be used in the construction of phylogenetic trees, in phenotype identification, and in analysis of microarray data, just to name 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 the run times to a reasonable level for large parameter values $k \geq 200$ remains an algorithmic challenge.

This paper is concerned with a suite of techniques which can be used in conjunction or independently to reduce both the problem size n and more importantly the parameter size k. In general each technique takes as input the graph G of size n and the parameter k. A corresponding graph G' of size $n' \leq n$ and parameter size $k' \leq k$ are produced such that n' is bounded by a (often polynomial) function of k', and G has a vertex cover of size $\leq k$ if and only if G' has a vertex cover of size $\leq k'$. Such techniques are collectively referred to as *kernelization*. Amenability to kernelization seems to be a fundamental difference between problems that are FPT and other $\mathcal{NP}$-hard problems. After kernelization, the process reverts to *branching*. Empirical studies of branching are also underway. See, for example, [1].

*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.

†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

2 Kernelization Alternatives

Our vertex-cover kernelization suite consists of four independent techniques. The first is a simple method based on the elimination of high degree vertices [2]. The second and third methods, reformulate the Vertex Cover problem 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 it can be stated as a network flow problem which can be solved using an algorithm developed by Dinic [7, 11]. The fourth method, which is new in this paper, we call *crown reduction*, and is based on finding a particular independent set and its neighborhood which can both be removed from the graph. This paper develops the theoretical justification for each of these techniques, and provides examples of their performance on a sample of actual application problems.

2.1 Preprocessing Rules

All of the techniques covered in this paper can benefit from some basic simplifications of the graph that are referred to here as preprocessing. The codes used to implement the techniques all take advantage of one or more of the following preprocessing rules that are relatively computationally inexpensive and easy to perform with run times that are $O(n^2)$.

Rule 1: An isolated vertex u (vertex of degree 0) can not be in a vertex cover of optimal size. Since there are no edges associated with such a vertex, there is no benefit of including it in any vertex cover. Thus G' is created by deleting u. This reduces the problem size so that $n' = n - 1$. This rule is applied repeatedly until all isolated vertices are eliminated.

Rule 2: In the case of a pendant vertex u (vertex of degree 1), there is a vertex cover of optimal size that does not contain the pendant vertex but does contain its unique neighbor v. Thus, G' is created by deleting both u and v and their incident edges from G. It is then also possible to delete the neighbors of v whose degrees drop to 0. This reduces the problem size so that n' is n decremented by the number of deleted vertices and reduces the parameter size to $k' = k-1$. 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 0. This reduces the problem size so that n' is n decremented by the number of deleted vertices and reduces the parameter size to $k' = k - 2$. 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 so that $n' = n - 2$. The parameter size is reduced to $k' = k - 1$. This idea was first proposed in [4], 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.

2.2 Kernelization by High Degree

This simple technique (see [2]) is based on the fact that vertices with degree $> k$ must be in any vertex cover of size $\leq k$. If v is a vertex of degree $> k$ and it is not included in the vertex cover, then all of its neighbors must be included. Thus the size of the cover would also be $> k$. This algorithm is applied repeatedly until all vertices of degree $> k$ are eliminated. The run time of the algorithm is $O(n^2)$ because of the need to compute the degree of each vertex.

The following theorem is used to bound the size of the kernel which results from the application of this algorithm in combination with the preprocessing rules. Note that if this algorithm and the preprocessing rules are applied, then each remaining vertex, v, has degree, $d(v)$, such that $3 \leq d(v) \leq k'$.

Theorem 1: If G' is a graph with a vertex cover of size k' and there is no vertex of G' with degree $> k'$ or degree < 3, then $n' \leq \frac{k'^2}{3} + k'$.

Proof: Let C be a vertex cover of size k' of the graph G'. The complement, $\overline{C}$, of C is an independent set with $n' - k'$ vertices. Let F be the set of edges in G' with endpoints in $\overline{C}$. Since the elements of $\overline{C}$ have degree ≥ 3, 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. However, $|C| = k'$. Therefore, $3(n' - k') \leq |F| \leq k'^2$ and $n' \leq \frac{k'^2}{3} + k'$. ■

2.3 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 then 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 [13, 11, 12].

The solution of the linear programming problem can be used to simplify the related integer programming problem in the following manner.

Define

$P = \{u \in V | X_u > 0.5\}$,
$Q = \{u \in V | X_u = 0.5\}$ and
$R = \{u \in V | X_u < 0.5\}$,
and then use the following modification of a theorem from Nemhauser and Trotter [13, 12].

Theorem 2: 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 the vertex cover problem has been solved with a "no" instance.

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 [5, 6]. Other methods to speed LP kernelization appear in [11].

2.4 Kernelization by Network Flow

This algorithm solves the linear programming formulation of vertex cover by reducing a graph to a network flow problem. It follows the algorithm used by Niedermeier in the proof of the Nemhauser-Trotter theorem. The algorithm in the proof of the Nemhauser-Trotter theorem defines a bipartite graph B in terms of the input graph G, finds the vertex cover of B by computing a maximum matching of B, and assigns values to the vertices of G based on the cover of B. We implemented this algorithm, solving the maximum matching of B by turning B into a network flow problem and solving it with Dinic's maximum flow algorithm [11, 7]. The time complexity of this algorithm is O($m\sqrt{n}$) where n is the number of vertices in the original graph G, and m is the number of edges in G, and 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 $H = (U, F)$ as follows:
$A = \{A_v | v \in V\}$
$B = \{B_v | v \in V\}$

$U = A_v \cup B_v$
$F = \{(A_v, B_u) | (v, u) \in E \text{ or } (u, v) \in E\}$

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 a vertex cover of H.

- Case 1: In the case that all vertices are included in the matching, the vertex cover of H is either the set A or the set B.
- Case 2: In the case that not every vertex is 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. S is the set of all unmatched vertices in A. R is the set of all vertices in A which are reachable from S by alternating paths with respect to M. T is the set of neighbors of R along edges in M. The vertex cover of the bipartite graph G' 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 a 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 from 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 3: Step 5 of the Algorithm produces a valid vertex cover of H; this cover has size $|M|$.

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 $(x, y) \in G$, $x \in S$, $x \in R$, or $x \in A - S - R$.

- First consider the case where $x \in S$. Since $x \in S$, x is unmatched. Then y must be matched because otherwise the matching M would be more optimal if it contained (x, y). Hence another edge (w, y) exists that is in M. Then, $w \in R$ and $y \in T$, and therefore (x, y) is covered.
- Next consider the case where $x \in R$. Then there must exist an edge (x, w) that is in M with $w \in T$. In the case where $w = y$, $y \in T$, and we are done. In the case where $w \neq y$, we know that another edge $(z, w) \in M$ exists where $z \in R$ or S. There must also exist another edge $(v, y) \in M$, since if y were not matched, it would be more optimal to add the two edges (z, w) and (x, y) to M instead of (x, w). Then $v \in R$ and $y \in T$ and therefore (x, y) is covered.
- In the case where $x \in A - S - R$, (x, y) is covered by definition of the cover.

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

Theorem 4: Step 6 of the Algorithm produces a feasible solution to the linear programming formulation of G.

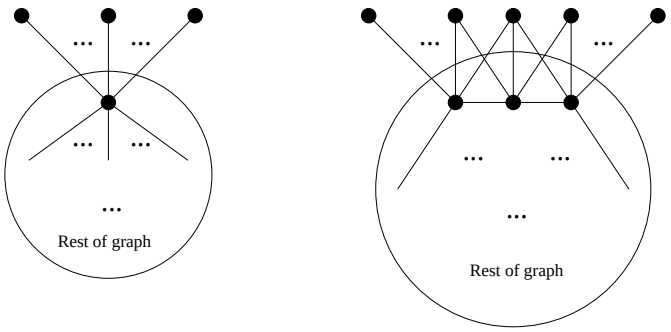
Proof: Each vertex in G is assigned a weight – either 0, 0.5, or 1. For every edge $(x, y) \in G$, we want the the sum of their weights, $W_x + W_y$, to be greater than or equal to 1. So for every edge (x, y), A_x and B_x are in the cover of H, A_y and B_y are in the cover of H, A_x and B_y are in the cover of H, or A_y and B_x are in the cover of H. If $(x, y) \in G$, $(A_x, B_y), (A_y, B_x) \in H$. Since one or both of the vertices in each edge of H is in H's cover, we know that at the least 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. Then 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 $O(n^{\frac{5}{2}})$.

2.5 Kernelization by Crown Reduction

The crown reduction is similar to the previous algorithms in that it attempts to use the structure of the graph to recognize two sets I and H of vertices so that there is an optimal vertex cover that does not contain any vertex from I but does contain every vertex from H. This process is based on the following definition, theorem, and algorithm.

A *crown* is an ordered pair (I, H) of subsets of vertices from a graph G that satisfies the following criteria: (1) $I \neq \emptyset$ is an independent set of G, (2) $H = N(I)$, and (3) there exists a matching M on the edges connecting I and H such that all elements of H are matched. H is called the *head* of the crown. The *width* of the crown is $|H|$. This notion is depicted in Figure1.



A crown of width 1 A crown of width 3

Figure 1: Sample crown decompositions.

Theorem 5: If G is a graph with a crown (I, H), then there is a vertex cover of G of minimum size that contains all the vertices in H and none of the vertices in I.

Proof: Since there is a matching M of the edges between I and H, 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. ■

Given a graph G, the following algorithm can be used to find a crown.

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 $I_{N-1} = I_N$.
4a. Let $H_n = N(I_n)$.
4b. Let $I_{n+1} = I_n \cup N_{M_2}(H_n)$.

The crown is now the ordered pair (I, H) where $I = I_N$ and $H = H_N$. Next, we show what conditions are necessary to guarantee that the crown algorithm is successful in finding a crown.

Theorem 6: 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 I and H and their adjacent edges. The problem size is $n' = n - |I| + |H|$ and the parameter size is $k' = k - |H|$. It is important to note that if a maximum matching of size $> k$ is found then there is not a vertex cover of size $\leq k$ and the vertex cover problem has been solved with a "no" 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: If both the matchings M_1 and M_2 are of size less than or equal to k then the graph G has at most $3k$ vertices that are not in the crown.

Proof: Since the size of the matching M_1 is less than or equal to k, it contains at most $2k$ vertices. Thus,

Algorithm	run time	kernel size (n')	parameter size (k')
High Degree	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.

the set O contains at least $n - 2k$ vertices. Since M_2 is less than or equal to k, there are at most k vertices in O that are matched by M_2. Thus there are at least $n - 3k$ vertices that are in O that are unmatched by M_2. These vertices are included in I_0 and are therefore in I. Thus the largest number of vertices in G that are not included in I and H is $3k$. ■

The particular crown produced by the crown reduction depends on the maximal matching M_1 that is used in its calculation. This implies that it is desirable to perform the reduction repeatedly, using 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 is also desirable to perform a preprocessing procedure after each crown reduction because the reduction may result in vertices of low degree. The most computationally expensive part of the procedure is finding the maximum matching M_2, which is done in our code by recasting the maximum matching problem on a bipartite graph as a network flow problem which is then solved using the algorithm developed by Dinic [7] with a run time bounded by $O(n^{\frac{5}{2}})$ for bipartite graphs [11].

3 Applications and Experimental Results

As previously noted, vertex cover has many applications. Our experiments were run in the context of solving some of these problems in computational biology. A common problem in many of these applications involves finding the maximum clique in a graph. However, since the Clique problem is $W[1]$-hard [8], it is not directly amenable to a Fixed-Parameter algorithmic approach. However, it is possible to find a maximum clique in a graph, G, by finding a minimum vertex cover of the complement of the graph, call it $\overline{G}$. Note that a graph G has a vertex cover of size k if and only if it's complement graph $\overline{G}$ has a clique of size $n - k$.

One of the applications to which we have applied our codes is the problem of finding phylogenetic trees based on protein domains [3]. The graphs that we utilized were obtained based on data from NCBI and SWISS-PROT, well known open-source repositories of biological data. Tables 1 through 3 indicate run times on graphs derived from the sh2 protein domain. The number after the domain name indicates the threshold used to convert the graph into an unweighted graph. In other words, the original graph is given as a complete graph in which each protein (vertex) has some correlation value with each of the other proteins. The biologists chose a threshold, and we remove all edges that have a weight below this value.

It should be noted that the high-degree method is incorporated together with the other preprocessing rules, and as noted above, the most efficient approach seems to always be to run preprocessing (including the high-degree method) before attempting any of the other kernelization methods. This can be observed by comparing the results in Table 1 and Table 2. In fact, both the network flow and the general linear programming methods were able to solve the problem without any branching in this case.

We conclude that in most cases the best way to proceed is to preprocess the graph (including the use of the high-degree algorithm) and then to run the crown reduction prior to branching. This is because sometimes, particularly on very dense graphs, the kernelization techniques, excluding the high-degree algorithm, will not reduce the graph by much, if at all. This can be observed in Table 4. Both linear programming and

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.

network flow can take a very long time to run on large graphs, yet since the crown reduction is so quick in comparison, it is almost always worthwhile to execute it before branching.

The graph in Table 4 highlights another important application for our codes. The graph comes from microarray data, and we look for very large (maximum) cliques in order to find sets of co-regulated genes. Before thresholding and complementation (which involves its own preprocessing), the graph actually had 12422 vertices, and depending on the threshold, we have often solved much larger instances of the problem than those shown here. However, because these particular graphs have thus far proven to be very dense, we have found that the use of linear programming and network flow only slow the entire process down. As can be seen in Table 4, only the high-degree algorithm reduces the graph. In fact, when first utilizing the high-degree algorithm prior to the other 3 methods, they still provide no further reduction of the graph. This was true for most of the graphs obtained from the 12422-node microarray graph.

4 A Few Conclusions

Crown reduction runs much faster in practice than linear programming, and sometimes reduces the graph just as well even though its worst-case bound on kernel size is worse. Given the methods at hand, the best approach seems to be first to run the preprocessing and high-degree method, followed by crown reduction. At that point, when the remaining problem kernel is a very dense graph (this can be checked based on the number of edges), it is best to proceed directly to branching. If the graph is fairly sparse, then either network

Algorithm	run time	kernel size (n')	parameter size (k')
High Degree	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.

flow or linear programming should be applied before proceeding.

References

[1] F. N. Abu-Khzam, M. A. Langston, and P. Shanbhag. Scalable parallel algorithms for difficult combinatorial problems: A case study in optimization. *Proceedings, International Conference on Parallel and Distributed Computing and Systems*, 2003. To appear.

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

[3] 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.

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

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

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

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

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

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

[10] 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.

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

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

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