12-2004

Crown Reductions and Decompositions: Theoretical Results and Practical Methods

William Henry Suters, III
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by William Henry Suters, III entitled "Crown Reductions and Decompositions: Theoretical Results and Practical Methods." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael A. Langston, Major Professor

We have read this thesis and recommend its acceptance:

Robert C. Ward, Bruce J. MacLennan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by William Henry Suters, III entitled "Crown Reductions and Decompositions: Theoretical Results and Practical Methods." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael A. Langston

Major Professor

We have read this thesis and
recommend its acceptance:

Robert C. Ward

Bruce J. MacLennan

Acceptance for the Council:

Anne Mayhew

Vice Chancellor and Dean of Graduate Studies

(Original signatures are on file with official student records.)

Crown Reductions and Decompositions: Theoretical Results and Practical Methods

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

William Henry Suters, III

December 2004

Dedication

This thesis is dedicated to my wife, Dr. Leslie Suters.

Acknowledgments

I wish to thank all of those who assisted in the completion of this thesis. In particular I would like to thank my advisor Micahel Langston as well as Faisal Abu-Khzam and the rest of Graph Algorithms Research Group. I would also like to thank Bill Cook for sharing his LP codes. Finally, I wish to extend a special thank you to Michael Fellows and Peter Shaw for their many helpful ideas and suggestions.

Abstract

Two kernelization schemes for the vertex cover problem, an $\mathcal{NP}$-hard problem in graph theory, are compared. The first, *crown reduction*, is based on the identification of a graph structure called a *crown* and is relatively new while the second, LP-kernelization has been used for some time. A proof of the crown reduction algorithm is presented, the algorithm is implemented and theorems are proven concerning its performance. Experiments are conducted comparing the performance of crown reduction and LP-kernelization on real world biological graphs. Next, theorems are presented that provide a logical connection between the crown structure and LP-kernelization. Finally, an algorithm is developed for decomposing a graph into two subgraphs: one that is a crown and one that is *crown free*.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since solutions to $\mathcal{NP}$-hard problems have many important applications, a wide variety of algorithmic techniques have been developed to deal with the computational challenge they pose. One common approach is to develop polynomial-time methods that approximate exact solutions. The approach of this paper is different; here we exploit the fact that many problems have solution algorithms that are polynomial-time with respect to all input parameters, with the exception of a single key parameter. These problems become tractable when this key input parameter is fixed or bounded, as is often the case in practice. This idea is motivated by the Graph Minor Theorem and its many applications [5, 13, 17]. This strategy has matured considerably and such problems are now called "fixed parameter tractable" (henceforth FPT).

Suppose (S, k) defines a problem where S is some structure of size n and k is a parameter relevant to S. The problem in question is said to be FPT if it has a solution

algorithm whose run time is $O(f(k)n^c)$, where c is a constant that is independent of both n and k. These algorithmic bounds mean that, when k is fixed, it is possible to find exact solutions to the problem in polynomial time. Unfortunately, the associated constants of proportionality are often large enough to make the FPT algorithm impractical to implement. To help deal with this problem, it is critical to develop techniques to reduce the size of the problem n and, more importantly, to reduce the size of the parameter k. These techniques are collectively referred to as "kernelization." Generally, the goal of kernelization is to take the problem (S, k) and produce another instance of the problem (S', k') where S' has size $n' << n$ and where $k' \leq k$. This must be done in such a way that (S', k') has a solution if and only if (S, k) has a solution. For more information on FPT we refer to [10].

1.1 FPT Hierarchy

With the introduction of FPT problems, it is desirable to reconsider the classical polynomial complexity hierarchy. The polynomial hierarchy places problems is the sequence

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \Sigma_2^P \subseteq \Sigma_3^P \subseteq \cdots \subseteq \text{PSPACE}.$$

It is the case that some $\mathcal{NP}$-hard problems are in FPT while others are fixed parameter intractable. It should also be noted that all problems that are in $\mathcal{P}$ are also FPT since they can be solved in polynomial time regardless of which parameter is fixed. Problems

can be broken into FPT hardness classes in a manner that is similar to the polynomial hierarchy [10].

The FPT hierarchy is

$$\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq \text{XP}.$$

To understand these categories it is helpful to define a *decision circuit* as a network of logic gates with a sequence of input variables and one output. The gates are categorized as being "large" or "small" depending on the number of input lines to the gate. This determination is based on some arbitrary fixed size bound. The *depth* of the circuit is the maximum number of gates on any path between the input and output. The *weft* is maximum number of "large" gates on any path between the input and output. We define $L_{\mathcal{F}(k,h)}$ to be the parameterized language associated with the family of decision circuits of weft k and depth h. Now, $W[k]$ is the hardness class of problems that are fixed parameter reducible to $L_{\mathcal{F}(k,h)}$ for some h.

In general, each problem in the FPT hierarchy is characterized a solutions algorithm with a run time that is $O(f(k)n^{g(k)})$ for an arbitrary computable function f and where the restrictions on g become more relaxed the higher the problem is in the hierarchy. A problem is in XP if it has a solution algorithm with a run time that is $O(f(k)n^{g(k)})$ for arbitrary computable functions f and g. Downey and Fellows present a complete discussion of the FPT hierarchy as well as examples of problems that fall into the various complexity classes [10].

1.2 Clique and Vertex Cover Problems

One $\mathcal{NP}$-hard problem that has many applications in biology is the *clique* problem. An instance of this problem is a graph $G = (V, E)$ and a parameter k. The question is to determine if there is a complete subgraph V' of size k. This problem is known to be $W[1]$-complete [10], and so is not amenable to fixed parameter techniques. Now, let G^* be the dual graph of G. Determining if there is an independent set of size k in the graph G^* is equivalent to finding a clique of size k in G. Thus, the *independent set* problem is also $W[1]$-complete.

The situation is improved, however, by considering what is perhaps the best known example of an FPT problem,*vertex cover*. There are also a myriad of known applications for this foundational combinatorial problem [3]. If we can find a vertex cover of the graph G^* then the complement of the vertex cover is an independent set. We can use the FPT character of the vertex cover problem to enable us to develop an efficient algorithm to find large cliques. It should be noted that the fixed parameter in the vertex cover problem is $n-k$, using the notation developed for the clique problem. This difference in parameter is what allows us to describe a solution to a problem that is $W[1]$-complete in terms of the solution to an FPT problem.

For the remainder of this paper, we will redefine our notation to simplify the vertex cover problem. Given an undirected graph G and a parameter k, we seek to decide whether G contains a set C of k or fewer vertices such that every edge in G has at least one endpoint in C. Using a bounded search tree approach it is easy to show that the

vertex cover problems is FPT.

Step 1: Create an arbitrary ordering of the edges in the graph G.

Step 2: Select the first uncovered edge $\{u, v\}$. This edge must be covered by either u or by v. Thus there are two possible cases; either u is in the vertex cover or v is in the vertex cover. Form a new branch of the binary search tree for each of these possibilities.

Step 3: Repeatedly apply rule 2 until either a vertex cover is found or all candidate sets with k or fewer vertices have been eliminated.

Since the depth of the binary search tree is at most k, this algorithm has a running time that is $O(2^k)$ and so the problem is FPT.

This produces a practical but inefficient method for solving the vertex cover problem when k is fixed. Kernelization is used to simplify the problem as much as possible before beginning the branching process. Once kernelization has been accomplished, vertex cover can be solved in time $O(1.2852^{k'} + k'n)$ using the bounded search tree approach [8].

1.3 Kernelization

The techniques for solving the vertex cover problem are aided by a variety of preprocessing rules. All of these rules take the problem (G, k) where G has n vertices and produce another instance of the problem (G', k') where G' has n' vertices with $n' < n$

and $k' \leq k$. This is done in such a way that (G', k') has a solution if and only if (G, k) has a solution. These rules are computationally inexpensive, requiring at most $O(n^2)$ time with very modest constants of proportionality.

1.3.1 Reduction Rules

The first kernelization rules that we use are based on identifying low degree vertices. These rules can be used to remove all vertices of degree < 3.

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 [8], 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.

1.3.2 Kernelization by High Degree

This simple technique (see [6]) 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. This algorithm is superlinear ($O(n^2)$) only because of the need to compute the degree of each vertex.

The following result from [1] is used to bound the size of the kernel that 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 $3 \leq d(v) \leq k'$. Using this it can be shown that $n' \leq \frac{k'^2}{3} + k'$. Put together, these rules are straightforward to implement, but not especially effective because they can require quadratic time (in n) and produce kernels of quadratic size (in k).

1.3.3 LP-kernelization

One powerful, well-known kernelization technique is based on an integer programming formulation of the optimization version of vertex cover. We assign a weight $X_u \in \{0, 1\}$ to each vertex u of the graph $G = (V, E)$ so that the following conditions are met.

(1) Minimize $\sum_u X_u$.

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

We can relax this to a linear programming problem by replacing the constraint $X_u \in \{0, 1\}$ with $X_u \geq 0$. The linear programming problem can be solved using a general LP package, or it can be posed as a network flow problem which can be solved using network flow techniques.

The solution to the linear programming problem is used to kernelize the original vertex cover problem in the following manner. Let $P = \{u \in V | X_u > 0.5\}$, $Q = \{u \in V | X_u = 0.5\}$ and $R = \{u \in V | X_u < 0.5\}$. There is an optimal vertex cover that is a superset of P and that is disjoint from R. This is a modification from [15] of a theorem

originally proved in [16]. Furthermore, there is a solution to this problem in which $X_u = 0$ for all $u \in R$, $X_u = 1$ for all $u \in P$ and $X_u = 0.5$ for all $u \in Q$.

The time complexity of LP-kernelization is $O(n^3)$ if a general LP package is used. When a network flow approach is used, it is is $O(m\sqrt{n})$ where m is the number of edges in G. If there is a vertex cover of size k, then the total of the weights assigned to vertices in the LP-solution must be $\leq 2k$. Thus, if the resulting weight total is $> 2k$ then there is no vertex cover of size k. This implies that LP-kernelization either results in a linear kernel of size $\leq 2k$ or a "no" answer to the vertex cover problem.

Chapter 2

Crown Reduction

Although the LP-kernelization technique is very effective in reducing the size of the vertex cover problem, its run time can still pose a significant computational challenge. Thus it is desirable to find faster kernelization techniques that could be used both in replacement of and in conjunction with LP-kernelization. The method we present in this chapter, *crown reduction*, can be used to make the LP-kernelization process much more efficient. All the vertices identified by a crown reduction could, potentially, be identified by some LP-kernelization. Crown reduction is in practice much faster, but produces more variable results than LP-kernelization.

2.1 Definition

This newer kernelization technique relies on the identification of a "crown structure" in the graph [2]. The method identifies two vertex subsets, H and I, in such a way that

there is an optimal vertex cover that is both a superset of H and disjoint from I. To define a crown, it is helpful to note that a *matching* is a collection of edges that do not share any vertices. Letting $N(S)$ denote the neighborhood of S, 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

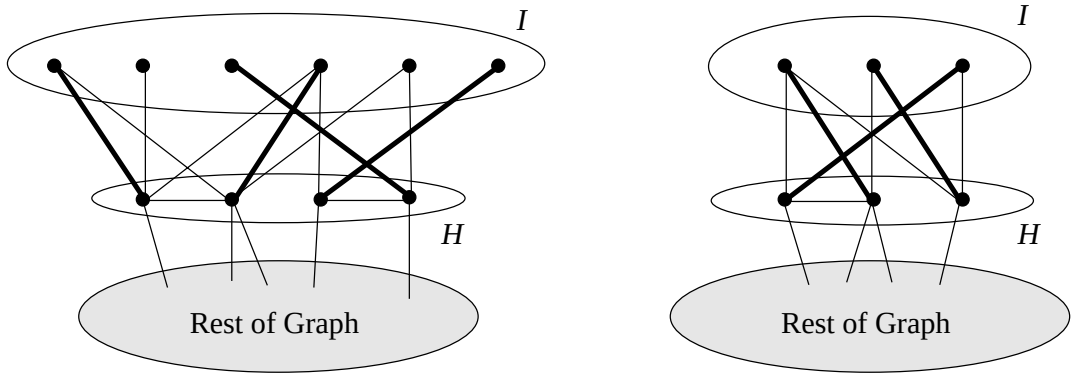
(3) there exists a matching M on the edges connecting I and H such that all elements of H are matched.

H is called the *head* of the crown. The *width* of the crown is $|H|$.

A crown that is a subgraph of another crown is called a *subcrown*. A *straight crown* is a crown (I, H) that satisfies the condition $|I| = |H|$. A *flared crown* is a crown (I, H) that satisfies the condition $|I| > |H|$. Notice that if (I, H) is a crown, then I is an independent set and H is a cutset between I and the rest of the graph. Examples are flared and straight crowns of different widths are show in figure 2.1.

2.2 Kernelization of Vertex Cover

If we can identify a crown in a graph, then this structure can be used as a tool to kernelize the vertex cover problem in a manner analogous to the pendant vertex rule presented in chapter 1 (rule 2). This is shown in the following theorem.



Flared Crown of width 4 Straight Crown of width 3

Bold edges denote matchings.

Figure 2.1: Flared and straight crowns of different widths.

Theorem 1 *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. ■

The kernelized graph G' is produced by removing vertices in I and H and their adjacent edges. The resulting problem size is $n' = n - |I| - |H|$ and the resulting parameter size is $k' = k - |H|$.

2.3 Algorithm

We now need a method for identifying crown structures in a graph. The following algorithm can be used to find a crown in an arbitrary graph. Before stating the algorithm, it is useful to note that a *maximal matching* is a matching to which no edges can be added without forcing two edges to share a vertex and that a *maximum matching* is a matching with the maximum possible number of edges. Letting $N_M(H)$ denote the neighbors of H using only the edges contained in the matching M is helpful in the definition of the algorithm.

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: If every vertex in $N(O)$ is matched by M_2, then $H = N(O)$ and $I = O$ form a crown, and we are done.

Step 4: Let I_0 be the set of vertices in O that are unmatched by M_2.

Step 5: Repeat steps 5a and 5b until $n = N$ so that $I_{N-1} = I_N$.

5a. Let $H_n = N(I_n)$.

5b. Let $I_{n+1} = I_n \cup N_{M_2}(H_n)$.

Step 6: $I = I_N$ and $H = H_N$ form a flared crown.

Theorem 2 shows that the success of the algorithm in finding a crown depends on which maximal matching, M_1, is identified in step 1. In general, the algorithm will succeed in finding a crown as long as at least one of the following conditions is met:

(1) I_0 is nonempty in Step 4 or

(2) every vertex in $N(O)$ is matched in Step 3 of the algorithm.

Crown reductions may be repeatedly applied to the same graph. When this is done it may be helpful to use a different maximal matching for each repetition. Finally, it may also be helpful to apply the reduction rules introduced in chapter 1 between repeated applications of crown reduction.

2.3.1 Proof of the Crown Reduction Algorithm

We now present a theorem to demonstrate the correctness of the crown reduction algorithm.

Theorem 2 *The algorithm produces a crown as long as either (1) the set I_0 of unmatched outsiders is not empty or (2) every vertex in $N(O)$ is matched by M_2.*

Proof. First, since M_1 is a maximal matching, the set O, and consequently its subset I, are both independent. Next consider the case where condition (2) holds. In this case $H = N(O) = N(I)$ and every vertex in H is matched by M_2. Thus by definition, H and I form a crown and we are done. Now consider the case where condition (1) holds.

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. ■

2.3.2 Performance

The most computationally expensive part a crown reduction is finding the maximum matching, M_2, which is done in this case by recasting the maximum matching problem on the bipartite graph of edges between O and $N(O)$ as a network flow problem. This network flow problem is then solved using the algorithm developed by Dinic [9] with a run time bounded by $O(m\sqrt{n})$ for bipartite graphs with m being the number of edges and n being the number of vertices [14]. In our case were are only performing the

network flow process on the bipartite graph with edges between O and $N(O)$. Thus, the time complexity of crown reduction is $O(m^*\sqrt{n^*})$, where m^* is the number of edges between O and $N(O)$ and n^* is the number of vertices in O and $N(O)$. Recall, the time complexity of LP-kernelization is $O(n^3)$ if a general LP package is used. When a network flow approach is used, it is is $O(m\sqrt{n})$ where m is the number of edges in G. Asymptotically, the behaviors of the crown reduction and LP-kernelization methods are similar. In practice, however, m^* and n^* are generally much smaller than m and n. Extensive experimental results indicate that crown reduction is in fact much faster than LP-kernelization [2], especially on large problem instances.

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 3 *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, 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 $3k$. ■

Both LP-kernelization and crown reduction result in kernels whose sizes are linear in k. The kernel that results from LP-kernelization has size at most $2k$. The kernel that results from crown reduction is (perhaps loosely) bounded above by $3k$. It should be noted that the particular crown produced by crown reduction depends on the maximal matching identified in step 1 of the algorithm. Thus, the crown reduction may be repeated, potentially resulting in smaller and smaller kernels.

Chapter 3

Experiments

The following experiments were run on graphs resulting from experimental data from computational biology, where *clique* is a common problem. Recall that clique is $W[1]$-hard [10], however, it is possible to find a maximum clique in a graph, G, by finding a minimum vertex cover of the dual graph, G^*.

3.1 LP-kernelization Can Outperform Crown Reduction

One of the applications to which we have applied our codes is the problem of finding phylogenetic trees based on protein domains [7]. The graphs that we utilized were obtained based on data from NCBI and SWISS-PROT, well known open-source repositories of biological data. The results in table 3.1 indicates run times on graph sh2-5 derived from the sh2 protein domain. The number after the domain name indicates the threshold used to convert the input into an unweighted graph.

Table 3.1: Graph sh2-5.dim with $k = 450$, $n = 839$ and 26612 edges.

Procedure	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
Crown Reduction	PP	CR	CR	CR	LP	BR
followed by	$k = 438$	$k = 415$	$k = 415$	$k = 415$	$k = 302$	cover
LP-Kernelization	$n = 810$	$n = 738$	$n = 733$	$n = 733$	$n = 496$	found
Total time 49.25	$t = 0.71$	$t = 1.39$	$t = 1.23$	$t = 1.19$	$t = 38.89$	$t = 5.84$
LP-Kernelization	PP	LP	BR			
only	$k = 438$	$k = 301$	cover			
	$n = 810$	$n = 494$	found			
Total time 51.28	$t = 0.70$	$t = 44.79$	$t = 5.79$			

Times are given in seconds. PP indicates preprocessing, CR indicates crown reduction, LP indicates LP-kernelization, and BR indicates branching.

In this implementation, the high degree kernelization and low degree reduction rules introduced in chapter 1 have been combined into a single preprocessing step. An attempt has been made to represent the manner in which these methods could be applied to a practical problem. Preprocessing is always the initial reduction attempted. Further, a single crown reduction can remove several vertices, resulting in a reduction in the degree of many of the remaining vertices. Thus, a reapplication of preprocessing may be productive. Since preprocessing is inexpensive, it is always run after a each iteration of crown reduction. The resulting reduction in graph size and run time are included in the time results for the associated crown reduction. Crown reduction is repeatedly applied until the first time it fails to further reduce the graph. Finally, LP-kernelization is applied to the graph to determine if further reductions are possible and the problem is then solved using a branching approach. For the purposes of comparison, the same graph

is reduced again using preprocessing followed only by LP-kernelization and branching, without the benefit of crown reduction.

Notice, in the case of the sh2-5 graph in table 3.1, that while a both methods produced similar results, and the method without crown reduction had fewer steps, the total time was slightly shorter with crown reduction. Also notice that LP-kernelization was successful in further reducing the graph after the application of repeated crown reductions. This is an unusual case, for reasons that will be explored in chapter 4. Since there is an element of chance in the crown reduction algorithm, it is possible that if this experiment were repeated the results would be different. However, the fact remains that the failure of crown reduction to reduce a graph does not imply that LP-kernelization will also fail.

3.2 Crown Reductions Useful In Identifying "NO" Instances

In contrast to the sh2-5 graph, there are examples where crown reduction can reduce a graph further than LP-kernelization. Such an example is the sh2-10 graph produced from the same data as sh2-5 with a different threshold value. The results for sh2-10 are show in table 3.2. This is an example in which not only did a sequence of crown reductions simplify the graph more than an LP-kernelization, crown reductions where able to completely solve the problem. This shows one of the strengths of crown reductions; it gives a relatively inexpensive method for attempting to identify "no" instances of the vertex cover problem.

Table 3.2: Graph sh2-10.dim with $k = 500$, $n = 839$ and 129697 edges.

Procedure	Step 1	Step 2	Step 3	Step 4	Step 5
Crown Reduction	PP	CR	CR	CR	CR
	$k = 340$	$k = 188$	$k = 188$	$k = 74$	no cover
	$n = 678$	$n = 471$	$n = 462$	$n = 295$	instance
Total time 4.66	$t = 1.07$	$t = 1.39$	$t = 1.61$	$t = 0.53$	$t = 0.06$
LP-Kernelization	PP	LP	BR		
only	$k = 340$	$k = 300$	no cover		
	$n = 678$	$n = 573$	instance		
Total time 64.11	$t = 1.06$	$t = 56.01$	$t = 7.04$		

Times are given in seconds. PP indicates preprocessing, CR indicates crown reduction, LP indicates LP-kernelization, and BR indicates branching.

3.3 Crown Reduction Augments LP-Kernelization

An example with more typical behavior is the graph, sh3-5, derived from data gathered about the sh3 protein domain. A comparison of the performance of the crown reduction and LP-kernelization is given in table 3.3. Although both methods produce similar reductions in the graph, crown reduction is significantly faster. Because of the expense of performing an LP-kernelization, it is helpful to note that crown reductions could be used in place of LP-kernelization or as a method to reduce the graph before attempting LP-kernelization. In this case, the time for doing preprocessing and three crown reductions was 45.49 seconds. The LP-kernelization that followed did not produce any further reduction and took 1847.62 seconds. If LP-kernelization was done without any preceding crown reductions the time required was 2416.33 seconds. Thus using crown reduction

Table 3.3: Graph sh3-5.dim with $k = 1300$, $n = 2466$ and 364703 edges.

Procedure	Step 1	Step 2	Step 3	Step 4	Step 5
Crown Reduction	PP	CR	CR	CR	LP
	$k = 1272$	$k = 1261$	$k = 1261$	$k = 1261$	k = 1261
	$n = 2398$	$n = 2312$	$n = 2305$	$n = 2305$	n = 2305
Total time 1893.11	$t = 7.76$	$t = 12.58$	$t = 12.66$	$t = 12.49$	$t = 1847.62$
LP-Kernelization	PP	LP			
only	$k = 1272$	$k = 1263$			
	$n = 2398$	$n = 2309$			
Total time 2424.14	$t = 7.81$	$t = 2416.33$			

Times are given in seconds. PP indicates preprocessing, CR indicates crown reduction, LP indicates LP-kernelization, and BR indicates branching.

in place of LP-kernelization can dramatically reduce run times. Using crown reduction prior to LP-kernelization usually produces more modest reductions in run time. Despite the reductions to the graph, the kernel remains quite large. Due to the computational cost, branching was not attempted on this kernel.

3.4 Dense Graphs and Kernelization

Neither crown reduction nor LP-kernelization is likely to work well for densely connected graphs. One example of this is the graph RMA-85-280. This graph is derived from microarray data, where a maximum clique corresponds to a set of putatively co-regulated genes. This graph has 1737 vertices and 1011051 edges, over 67% of the edges in a completely connected graph with the same number of vertices. When searching for

a vertex cover of size $k = 1457$, an initial preprocessing step reduced n to 1210 and k to 930. After this, neither crown reduction nor LP-kernelization were successful in producing further reductions. The run time for crown reduction was 6.84 seconds while LP-kernelization took 164.45 seconds. Thus, it may be useful to use crown reduction as a test to determine if an LP-kernelization is likely to be successful in reducing the graph.

For very large dense graphs, it may not even be practical to run LP-kernelization. The run time and memory requirements for LP-kernelization increase rapidly as the number of edges increases. One example of this is the graph U74-0.5-369. This graph is based on similar data to RMA-85-280 and has 5680 vertices and 13736738 edges, over 85% of the edges in a complete graph with the same number of vertices. Preprocessing reduced n to 3447 and k to 3078. Subsequent attempts at crown reductions where unsuccessful in further reducing the problem, but only took 92.23 seconds. LP-kernelization was rendered impractical due to its excessive memory and time requirements. Thus, particularly for large dense graphs, crown reduction may be a practical alternative when LP-kernelization is too expensive.

Chapter 4

Crown Decomposition

Because the crown reduction and LP-kernelization are based on different algorithmic techniques, there was considerable suspicion that the methods would prove to be orthogonal [12]. This is not the case, however. The sets P and R identified by LP-kernelization turn out, surprisingly, to be a crown. Since LP-kernelization is less variable than our crown reduction algorithm, it can be used to decompose a graph into a crown and a subgraph that contains no crowns. This is what we refer to as a *crown decomposition*. The results of this chapter are also presented in [4].

4.1 LP-kernelization: Finding Crowns

The fact that LP-kernelization is, in fact, another method of crown reduction is proven by the following theorem and an example is shown in figure 4.1

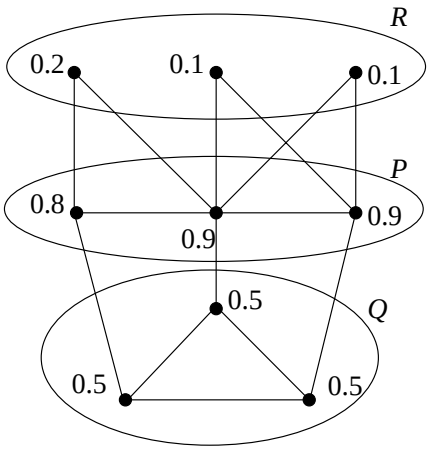


Figure 4.1: LP-solution used to identify a crown.

Theorem 4 *If LP-kernelization is applied to a graph G and finds a set R of vertices to exclude from the vertex cover and a set P of vertices to include in the cover, then (R, P) is a crown.*

Proof. Let us look at the requirements for a crown. Since $X_u < 0.5$ for all vertices $u \in R$, we know, because of the edge constraint $X_u + X_v \geq 1$, that there cannot be any edges $\{u, v\}$ with both u and v in R. Thus R is an independent set.

Suppose there is a vertex $u \in P$ where $u \notin N(R)$. Then every neighbor v of u has $X_v \geq 0.5$. Thus we could improve the LP solution by imposing $X_u = 0.5$ without violating any of the constraints $X_u + X_v \geq 1$. This cannot happen so we can conclude that $P \subseteq N(R)$. Similarly, suppose there is a $u \in N(R)$ where $u \notin P$. Then $X_u \leq 0.5$ while u has a neighbor v where $X_v < 0.5$. The edge $\{u, v\}$ must violate the constraint

$X_u + X_v \geq 1$. This too cannot happen, so we conclude that $N(R) \subseteq P$ and thus $P = N(R)$.

Let M be a maximum matching on the edges between R and P. We now show that every vertex in P must be matched by contraction. Let $C_0 \subset P$ be the set of vertices in P that are unmatched by M and suppose $C_0 \neq \emptyset$. Let $D_0 = N(C_0) \cap R$ and $C_1 = N_M(D_0) \cup C_0$. Repeat this process, setting $D_n = N(C_n) \cap R$ and $C_{n+1} = N_M(D_n) \cup C_n$, until $C = C_{N+1} = C_N$ and $D = D_N$.

Since M is a maximum matching, alternating paths with an odd number of edges that begin and end at unmatched vertices are impossible. Thus any alternating path beginning with a vertex in C_0 has an even number of edges (and an odd number of vertices), beginning and ending in C. Since very vertex in D must be part of such an alternating path, this implies that C must be larger than D. This can be most easily seen by noting that the matching M gives a natural one to one association between the elements of D and $N_M(D)$. If this were not true an alternating path with an odd number of edges would result. Furthermore $C_0 \neq \emptyset$ and $N_M(D)$ are disjoint and $C = N_M(D) \cup C_0$. Thus C is larger than D.

Notice that for any set $P' \subset P$ we know that $N(P') \cap R$ must be larger than P' since otherwise we could improve the LP solution by setting $X_u = 0.5$ for all $u \in P'$ and $u \in N(P') \cap R$. However we have already shown that $C \subset P$ is larger than $N(C) \cap R = D$ so this is a contradiction. Thus $C_0 = \emptyset$ and every vertex in P must be matched. Therefore (R, P) is a crown. ■

4.1.1 Types of Crowns Identified

We now determine the types of crowns that can be identified by LP-kernelization. This task is complicated by the fact that LP-solutions are not unique. Different optimal LP-solution techniques may produce different results on the same graph. An example of this nonuniqueness is shown in figure 4.2. In order to characterize the types of crowns identified by LP-kernelization, it is useful to prove the following lemma which helps refine the relationship between straight and flared crowns.

Lemma 1 *If (I, H) is a flared crown then there is another crown (I', H) that is straight and where $I' \subset I$.*

Proof. Since (I, H) is a crown there is a matching M on the edges between I and H so that all elements of H are matched. Since (I, H) is flared there must be at least one unmatched vertex in I. Let I' be the set of matched vertices in I. It is clear that $I' \subset I$ and that I' is an independent set. It is also clear that M is a matching between I' and

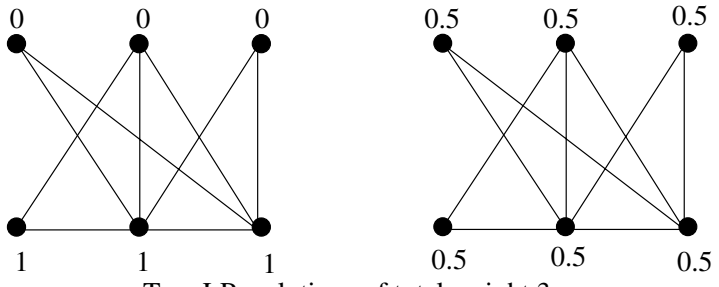


Figure 4.2: A crown is missed by one optimal LP-solution and found by another.

H and that every vertex in H is matched. Thus (I', H) is a crown. Finally, the M forms a one-to-one association between the vertices in I' and H. Thus $|I'| = |H|$ and the crown is straight. ■

This lemma allows us to break a flared crown into two subcrowns. There is a straight subcrown and a flared subcrown. We are now ready to prove that LP-kernelization eliminates all flared crowns from the graph. It does this by either finding all of the flared crowns, or by *straightening* the flared crown by identifying the flared part of the crown, but leaving the straight subcrown unrecognized. An example of this is showing in figure 4.3. One solution technique may identify an entire flared crown, while another technique may only straighten the crown.

Theorem 5 *If LP-kernelization is performed then only straight crowns remain.*

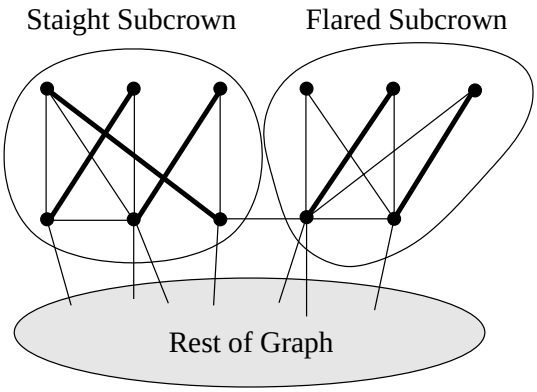


Figure 4.3: Straightening a flared crown.

Proof. Suppose there is a crown (I, H) with $|I| > |H|$ that is not identified by LP-kernelization. Vertices u not removed by LP-kernelization are given weights $X_u = 0.5$. Thus $X_u = 0.5$ for every $u \in I \cup H$. Since $|I| > |H|$ we can improved the LP-solution by assigning $X_u = 1$ for every $u \in H$ and $X_v = 0$ for every $v \in I$.

We must demonstrate that this new weight assignment is an LP-solution by showing that it still meets the edge constraints. Since $N(I) = H$ the condition $X_u + X_v \geq 1$ is satisfied by for all edges (u, v) where either u or v is in I. Next consider edges that have an endpoint in H but not in I. These edges have had their total weight increased and so still meet the edge constraints. Finally, edges that do not connect either to I or to H are unaffected by the new weights and so still satisfy the edge constraints. ■

4.1.2 Every Crown Identified by Some LP-Solution

Even though a given LP-solution may or may not recognize a particular straight crown, there is an LP-solution that does. This is proven in the following theorem.

Theorem 6 *If (I, H) is a crown, then there is an optimal LP-solution that identifies a crown of which (I, H) is a subcrown.*

Proof. Suppose there is a particular optimal LP-solution that does not identify a crown with (I, H) as a subcrown. This implies that the crown must be straight and $|I| = |H|$ by Theorem 5. Let us construct another LP-solution by assigning $X_v = 0$ for all $v \in I$ and $X_u = 1$ for all $u \in H$ and leaving the weight of the other vertices unchanged.

We first show that it is an LP-solution by showing that it still meets the edge

constraints. Since $N(I) = H$ the condition $X_u + X_v \geq 1$ is satisfied by for all edges (u, v) where either u or v is in I. Next consider edges that have an endpoint in H but not in I. These edges have had their total weight increased and so still meet the edge constraints. Finally, edges that do not connect either to I or to H are unaffected by the new weights and so still satisfy the edge constraints.

Finally, we need to show that the solution is optimal. Since $|I| = |H|$ the value of the objective function $\sum_{u \in V} X_u$ is unchanged. Therefore this is a new solution to the LP-problem which identifies the crown. ■

4.1.3 Finding All Crowns in a Graph

Even though each crown has an LP-solution that would identify it, we need to determine if there is a particular LP-solution that would eliminate all crowns from the graph. We present a lemma and series of theorems that can be used to design a procedure for identifying all crowns in a graph.

First, we need to show that if a crown is identified and removed from a graph and a second crown is identified among the remaining vertices, then these two crowns can be combined to form a single crown. To do this it is useful to prove the following lemma that allows us to restrict the number of values that must be considered in the LP-solution. It is previously known that there are optimal solutions to the LP-problem that only use weights 0, 0.5, and 1 [15, 16]. Nevertheless, it is useful to recast this result here in term of crowns, using Theorem 4, and then to demonstrate a method for modifying any LP-solution to use only these weights. An example of the application of

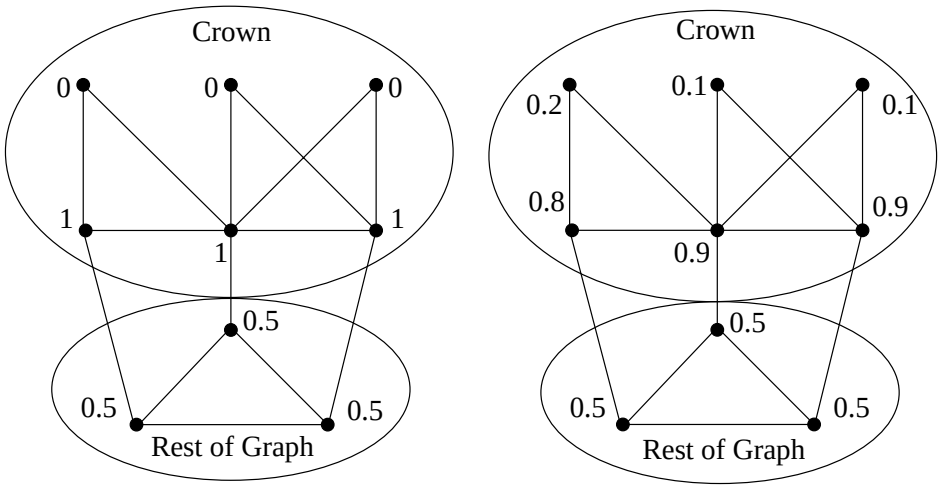


Figure 4.4: Two optimal LP-solutions, one with values restricted to 0, 0.5 and 1.

this lemma is shown in figure 4.4.

Lemma 2 *If there is an optimal solution to the LP-kernelization problem that assigns weight X_u to each vertex $u \in V$ and we define $R = \{u \in V | X_u < 0.5\}$, $Q = \{u \in V | X_u = 0.5\}$, and $P = \{u \in V | X_u > 0.5\}$, then there is another optimal solution to the LP-kernelization problem that assigns weights $X'_u = 0$ if $u \in R$, $X'_u = 0.5$ if $u \in Q$, and $X'_u = 1$ if $u \in P$.*

Proof. By Theorem 4 we know that (R, P) forms a crown and so there is a matching M between R and P so that every element in P is matched. The total weight contribution of $R \cup P$ must be at least $|M| = |P|$ in any LP-solution since the edges in M must have total edge weight ≥ 1. We can achieve this lower bound by making the weight assignments $X'_u = 0$ if $u \in R$ and $X'_u = 1$ if $u \in P$. The weights of the remaining

vertices are unchanged by the assignment $X'_u = 0.5$ if $u \in Q$, so the total edge weight of the graph is either unchanged or reduced by these assignments.

We now show that this new assignment is in fact an LP-solution by considering the edge constraints. The edge constraints are met for all edges $\{u, v\}$ with $u \in P$, since in this case $X'_u = 1$. Edges $\{u, v\}$ with $u \in R$ must have $v \in P$ and so we have already met the edge constraint. This is because (R, P) is a crown and so $N(R) = P$. Finally, we consider edges $\{u, v\}$ with $u \in Q$. Such an edge cannot have $v \in R$ since $N(R) = P$ and so we only need to examine cases where $v \in P$ or $v \in Q$. If $v \in P$ then we have already met the edge constraint. If $v \in Q$ then $X'_u = 0.5$ and $X'_v = 0.5$ and so the edge constraint is met in this final case. ■

Now we prove that if a crown is identified and removed from a graph and a second crown is identified among the remaining vertices, then these two crowns can be combined to form a single crown.

Theorem 7 *Suppose a graph $G = (V, E)$ has a crown (I, H) identified by LP- kernelization and that when (I, H) is removed the induced subgraph $G' = (V', E')$ has another crown (I', H'), then $(I \cup I', H \cup H')$ forms a crown in G.*

Proof. Let S be the optimal LP-solution for G where $X_v = 0$ for every $v \in I$, $X_u = 1$ for every $u \in H$, and $X_w = 0.5$ for every $w \notin H \cup I$. We know that such an optimal LP-solution exists by Lemma 2.

We construct a new optimal LP-solution S'. For any $u \in V$ we set X'_u as follows:

(1) If $u \in H \cup H'$ then $X'_u = 1$.

(2) If $u \in I \cup I'$ then $X'_u = 0$.

(3) If $u \notin H \cup H' \cup I \cup I'$ then $X'_u = 0.5$.

Notice that all of the vertices in $I' \cup H'$ remain when (I, H) is removed so $I' \cup H'$ and $I \cup H$ are disjoint. We already know that I and H are disjoint and that I' and H' are disjoint. This implies that I, H, I', and H' are all mutually disjoint. Thus there are no contradictions in the definition of S'.

We now show that S' is in fact an LP-solution by verifying the edge constraints for an arbitrary edge $(u, v) \in E$.

Case 1: One of the endpoints is in $H \cup H'$. Without loss of generality assume $v \in H \cup H'$, then $X'_v = 1$ and the edge constraint is met.

Case 2: One of the endpoints is in I. Without loss of generality assume $u \in I$. We know that $v \in H$ since $N(I) = H$. Thus the problem reduces to case 1 and the edge constraint is met.

Case 3: One of the endpoints is in I'. Without loss of generality assume $u \in I'$. Since (I, H) is removed before (I', H') is identified, there are two possibilities $v \in I \cup H$ or $v \notin I \cup H$. If $v \in I \cup H$, we know that $N(I) = H$ and $u \notin H$ so we can restrict our attention to the case were $v \in H$. Thus the problem reduces to case 1 and the edge constraint is met. If $v \notin I \cup H$ then v is a vertex in G' and in this graph $N(I') = H'$. Thus $v \in H'$, the problem reduces to case 1 and the edge constraint is met.

Case 4: Neither u nor v is in $H \cup H' \cup I \cup I'$. In this case $X'_u = 0.5$ and $X'_v = 0.5$ and the edge constraint is met.

Finally, we show that S' is an optimal LP-solution for G. We know that S is an optimal LP-solution for G. Notice that the total edge weight in S is $|H| + 0.5|V - (H \cup I)| = |H| + 0.5|V'|$ and that an optimal LP-solution for G' has total edge weight $0.5|V'|$. Also notice that since (I', H') is a crown, there is another optimal LP-solution that identifies this crown and uses weights 0, 1, and 0.5. We know this is possible by the proof of Lemma 2. The total edge weight of this new LP-solution for G' is $|H'| + 0.5|V' - (H' \cup I')|$. However, since these are both optimal solutions for G' we know that $0.5|V'| = |H'| + 0.5|V' - (H' \cup I')|$.

Now consider the total edge weight for S' which is $|H \cup H'| + 0.5|V - (H \cup H' \cup I \cup I')| = |H \cup H'| + 0.5|V - (H \cup I) - (H' \cup I')| = |H \cup H'| + 0.5|V' - (H' \cup I')|$. Since H, H' are disjoint, we know that $|H \cup H'| = |H| + |H'|$. Thus the total edge weight for S' is $|H| + |H'| + 0.5|V' - (H' \cup I')| = |H| + 0.5|V'|$ which is the total edge weight for S. Thus, both S and S' are optimal LP-solutions for G. Thus by Theorem 4 we know that the sets identified by S', namely $I \cup I'$ and $H \cup H'$ must form a crown. ■

Finally, we need to show that identifying two different crowns cannot result in any serious conflicts. That is, if a graph has two different crowns, then the two crowns can be combined to form a single crown. This is not a matter of simply taking the union of the two crowns. There may be vertices in the independent set of one crown that are included in the cutset of the other crown. Such conflicts always occur in straight crowns that are subcrowns of the two original crown and that can be reversed without disrupting the crown properties. An example is shown in figure 4.5.

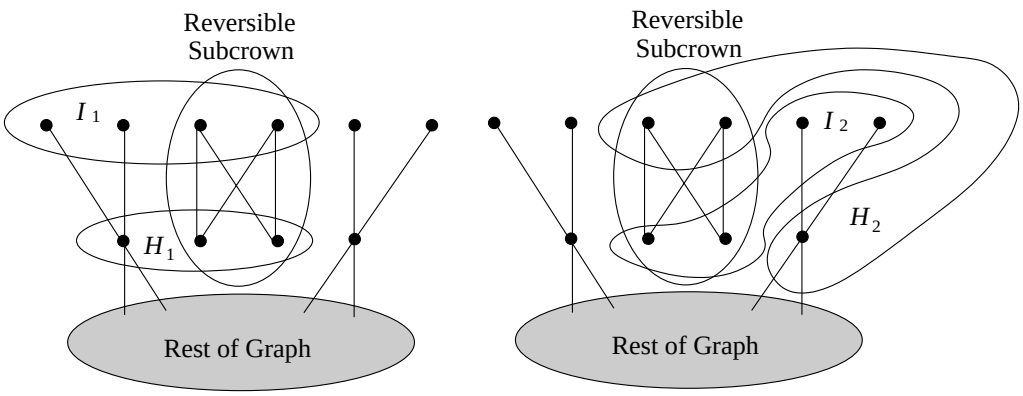


Figure 4.5: Two crowns, (I_1, H_1) and (I_2, H_2) reverse a straight crown.

Theorem 8 *If (I_1, H_1) and (I_2, H_2) are crowns of a graph G that are identified by two different LP-solutions, then there is a crown (I, H) that contains all the vertices in $I_1 \cup I_2 \cup H_1 \cup H_2$ and where $I_1 \subseteq I$ and $H_1 \subseteq H$.*

Proof. For each vertex u in G, let X_u^1 designate the weight of u in an optimal LP-solution that identifies (I_1, H_1) so that $X_u^1 = 0$ if and only if $u \in I_1$, $X_u^1 = 1$ if and only if $u \in H_1$, and $X_u^1 = 0.5$ otherwise. We know such a solution exists by Lemma 2. Similarly find X_u^2 for an optimal LP-solution that identifies (I_2, H_2).

We now create another optimal LP-solution by defining, for each vertex u, $X_u^* = \frac{X_u^1 + X_u^2}{2}$. Notice the total weight $\sum_u X_u^* = \sum_u X_u^1 = \sum_u X_u^2$ is still optimal. If $\{u, v\}$ is an edge in G, the $X_u^* + X_v^* = \frac{X_u^1 + X_u^2}{2} + \frac{X_v^1 + X_v^2}{2} \geq 1$ since $X_u^1 + X_v^1 \geq 1$ and $X_u^2 + X_v^2 \geq 1$. Thus X^* defines an optimal LP-solution. By Lemma 2 we can modify these values so that $X_u^* = 0$, 0.5, or 1 for all vertices u in G.

The only vertices in the original two crowns that do not have X^* weights of 1 or 0 are those in $I_1 \cap H_2$ and $I_2 \cap H_1$. Let G' be the graph that remains when all vertices with X^* weights of 0 or 1 are removed from the original graph. In this graph let $u \in I_1 \cap H_2$ and v be a neighbor of u. Since $u \in I_1$ and $H_1 = N(I_1)$ we know $v \in H_1$. Thus $X^1_v = 1$ and since we know $X^*_v = 0.5$ we also know $X^2_v = 0$. Therefore $v \in I_2$ and $v \in H_1 \cap I_2$. Thus $N(I_1 \cap H_2) \subseteq (I_2 \cap H_1)$ in graph G'. A similar argument shows that $N(I_2 \cap H_1) \subseteq (I_1 \cap H_2)$ in G'.

Finally, we show that $(I_1 \cap H_2, I_2 \cap H_1)$ forms a straight crown that can be reversed. Notice that if $|I_1 \cap H_2| \leq |I_2 \cap H_1|$ we would not increase the size of the LP-solution by assigning weight 1 to the vertices in $I_1 \cap H_2$ and 0 to the vertices in $I_2 \cap H_1$. On the other hand if $|I_2 \cap H_1| \leq |I_1 \cap H_2|$ then we would not increase the size of the LP-solution by assigning weight 1 to the vertices in $I_2 \cap H_1$ and 0 to the vertices in$I_1 \cap H_2$. In either case we have a new LP-solution that identifies $(I_1 \cap H_2, I_2 \cap H_1)$ as a crown but X^* does not. By Theorem 5, this implies that $(I_1 \cap H_2, I_2 \cap H_1)$ is a straight crown. Since $N(I_1 \cap H_2) \subseteq (I_2 \cap H_1)$ and $N(I_2 \cap H_1) \subseteq (I_1 \cap H_2)$ in G' we know that this crown has no neighbors. Thus it can be reversed without loss of generality.

We select the independent set of the straight crown to be $I_1 \cap H_2$ and the cutset of the straight crown to be $I_2 \cap H_1$ so that the weight agree with the crown (I_1, H_1). Notice that all of the remaining vertices in the two original crowns were identified by the LP-solution defined by X^*. Thus, by Theorem 7, we can union the crown identified by X^* and the straight crown to obtain (I, H) where $I = I_1 \cup (I_2 - I_1)$ and $H = H_1 \cup (H_2 - I_1)$. ■

4.2 Finding All Crowns in Polynomial Time

We can now use the results we have just proven to produce a polynomial time algorithm that will find all possible crowns in an arbitrary graph.

Theorem 9 *There is a polynomial time algorithm for processing a graph G to produce an induced subgraph G' that has no crowns.*

Proof. We present such a polynomial time algorithm.

Step 1: Perform LP-kernelization. By Theorem 5, this can be used to eliminate all flared crowns by either removing the entire crown, or by removing enough vertices so that all the crowns are straight. Let $G_1 = (V_1, E_1)$ be graph induced by the set of vertices that remain in the kernel. Since these vertices were not removed by LP-kernelization, we know that $X_u = 0.5$ for all $u \in V_1$. We also know that the total weight in the optimal LP-solution for G_1 is $0.5|V_1|$.

Step 2: Pick a vertex $w \in V_1$ and test it to see if it is in the independent set of some crown by finding the optimal solution of the following LP-problem.

Assign a value $X_u \geq 0$ to each vertex $u \in V_1$ so that the following conditions hold.

(1) Minimize $\sum_{u \in V_1} X_u$.

(2) Satisfy $X_u + X_v \geq 1$ whenever $uv \in E_1$.

(3) $X_w = 0$

Step 3: If the total weight is still $0.5|G_1|$, then this too is an optimal solution of the

original LP-problem and we have identified a straight crown. We remove the straight crown from the graph in the usual manner producing an induced subgraph $G_2 = (V_2, E_2)$ where we know that $X_u = 0.5$ for all $u \in V_2$ and the total weight in the optimal LP-solution for G_2 is $0.5|V_2|$. If the total weight is larger, then we have not identified a crown and we let $G_2 = G_1$.

We repeatedly apply steps 2 and 3 until all vertices have been checked or eliminated, producing the graph G'. Theorem 7 guarantees that removing a crown does not create new crowns from vertices not previously in a crown. Theorem 8 guarantees that the order in which crowns are identified does not significantly change the final result. ■

Thus we only need to check each vertex once and when this process is complete, there can be no crowns and we will have identified all possible crowns in the graph. The total run time of the LP-solution procedure is $O(mn^{3/2})$ where m is the number of edges and n is the number of vertices in G if the network flow approach is used. This is a worst case scenario, in which the original graph has no crowns, and the process is repeated n times, once for each vertex.

4.3 Crown Decomposition

The algorithm in Theorem 9 allows us to find a single large crown that breaks the graph into a crown and a subgraph without any crowns. We state this in the form of the following corollary.

Corollary 1 *The union of the crowns found in the algorithm in Theorem 9 forms a single large crown and is, up to reversals of straight crowns, unique.*

Proof. Theorems 7 and 8. ■

This is equivalent to the following corollary that allows us to decompose every graph into a large crown and a subgraph that has no crowns.

Corollary 2 *Every graph G can be decomposed in to two subgraphs, C and K where C is a crown and K has no crowns.*

Decomposing graphs into crowns and subgraphs that are crown free is not only useful in reducing the kernel size of the vertex cover problem. It may also be helpful in kernelizing a variety of packing problems, the $n - k$ coloring problem and many other $\mathcal{NP}$-hard problems [11]. Because the notion of crown decompositions is so new, there may well be other applications that have yet to surface.

Chapter 5

Conclusion

The results of this paper are both theoretical and practical. We have introduced the crown structure and presented two methods for identifying these structures. One method, LP-kernelization, was already in use and the other, crown reduction, is a more recent innovation. Once either method has been implemented and a crown structure identified, the structure can be used to reduce the size of instances of the vertex cover problem. Further, we have produced an algorithm for decomposing a graph into a single crown and a crown free subgraph.

5.1 Comparison of LP and Crown Kernelization

Since both LP-kernelization and Crown decomposition can be used to identify crown structures, it is helpful to compare their strengths and weaknesses. LP-kernelization provides a thorough but computationally expensive investigation of the graph, identify-

ing or straightening all flared crowns. This produces a kernel size that is $O(k^2)$ where k is the size of the vertex cover. Crown reduction on the other hand is a much less expensive method whose results are somewhat less predictable. A kernel size that is $O(k^3)$ is guaranteed on a single iteration. Unlike LP-kernelization, crown reduction may be repeatedly applied. In general it would seem to be good practice to seek to kernelize a problem using one or more crown reductions. This is particularly true when a rapid identification of a "no" instance is possible. Afterward, depending on the size of the kernel required and the results of the crown reductions, a decision could be made on the benefits of attempting LP-kernelization.

5.2 Applications and Areas for Further Study

Finding solutions of the clique problem is of great interest in computational biology. The kernelization of vertex cover via the identification of crown structures is a useful tool in reducing the computational expense of solving these problems. In addition, there are variations on the crown structure that could be investigated, such as double crowns, and crowns with particular subgraph (such as K_3 or P_2) in place of the vertices of the independent set. Extensions of these ideas may be useful in kernelizing other $\mathcal{NP}$-hard problems such as *set cover* and *hitting set*. Finally, decomposing graphs into crowns and subgraphs that are crown free may be helpful in kernelizing a variety of packing problems [11]. Because the notion of crown decompositions is so new, there may well be other applications that have yet to surface.

Bibliography

Bibliography

[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, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings, Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2004.

[3] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons. Scalable parallel algorithms for FPT problems. Technical Report UT-CS-04-524, Department of Computer Science, University of Tennessee, 2004.

[4] F. N. Abu-Khzam, M. A. Langston, and W. H. Suters. Fast, effective vertex cover kernelization: A tale of two algorithms. In *Proceedings, ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2005.

[5] D. Bienstock and M. A. Langston. Algorithmic implications of the graph minor theorem. In *Handbook of Operations Research and Management Science: Network Models*, pages 481–502. North-Holland, 1995.

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

[7] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon. Solving large FPT problems on coarse grained parallel machines. *Journal of Computer and System Sciences*, 67:691–706, 2003.

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

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

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

[11] M. R. Fellows. personal correspondence.

[12] M. R. Fellows. Blow-ups, win/wins, and crown rules: Some new directions in FPT. *Lecture Notes in Computer Science*, 2880:1–12, 2003.

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

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

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

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

[17] N. Robertson and P. D. Seymour. Graph minors XXIII, the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, pages 65–110, 1995.

Appendix

Instructions

The following code implements the crown reduction algorithm. It is broken into two parts. The first, crown1.c, takes an input graph and an integer k as arguments. The input file must be in DIMACS-A format. It finds a maximal matching of the graph and produces a network flow problem that can be solved to find the needed maximum matching. There are several software packages that can be used to solve the network flow problem. The output of crown1.c is formatted for the Dinic package and the output file is named "dinic.in". The second program, crown2.c, completes the crown reduction. It takes an integer k, the same input file as crown1.c and the names of two output files as arguments. The first output file will contain the resulting cover. The other will contain a record of the reduction performed in order to allow for the later reconstruction of a vertex cover. This program also requires the output file from the Dinic package, "dinic.out."

Crown1.c

```
/* crown1--the initial step in a crown reduction algorithm      */
/* This program takes an integer k as an argument and attempts  */
/* to find a crown that can be used to find a vertex cover of   */
/* size less than or equal to k. It also takes the name of a    */
/* DIMACS-A (DIMACS without the characters) file that describes */
/* the graph as an argument. The output is a related network    */
/* flow problem in strict DIMACS format that can be solved to   */
/* find a needed maximum matching. This output is stored in     */
/* dinic.in. The maximum matching can be found using            */
/* the program dinic. Once this has been done the remaining     */
/* parts of the crown algorithm are performed in program        */
/* crown2.                                                      */
/* Henry Suters, June 16, 2003                                  */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int n, k, edges, i, j, u, v, outedges, matchcount;
  int **graph;
  int *degree, *matched;
  FILE *in, *out, *outsiders, *tmp;

  /* get and check arguments */

  if (argc < 3) {
    fprintf(stderr, "Usage: %s k input_file \n", *argv);
    exit(1);
  }

  k = atoi(argv[1]);
```

```
in = fopen(argv[2], "r");
if (in == NULL) {
  perror("Unable to open file for input\n");
  exit(1);
}

out = fopen("dinic.in", "w");
if (out == NULL) {
  perror("Unable to open file for output\n");
  exit(1);
}

outsiders = fopen("crown.outside", "w");
if (outsiders == NULL) {
  perror("Unable to open file for output\n");
  exit(1);
}

/* allocate data structures */

fscanf(in, "%d %d\n", &n, &edges);

graph = (int **) malloc(n * sizeof(int *));
if (graph == NULL){
  perror("Out of memory\n");
  exit(1);
}
for (i = 0; i < n; i++){
  graph[i] = (int *) calloc(n, sizeof(int));
  if (graph[i] == NULL){
    perror("Out of memory\n");
    exit(1);
  }
}

degree = (int *) calloc(n, sizeof(int));
if (degree == NULL){
  perror("Out of memory\n");
  exit(1);
}

matched = (int *) calloc(n, sizeof(int));
```

```
if (matched == NULL){
  perror("Out of memory\n");
  exit(1);
}

/* read in graph data and find maximal matching */

matchcount = 0;
for (i = 0; i < edges; i++){
  fscanf(in, "%d %d\n", &u, &v);
  graph[u][degree[u]] = v;
  graph[v][degree[v]] = u;
  degree[u]++;
  degree[v]++;
  if(matched[u] == 0 && matched[v] == 0){
    matched[u] = 1;
    matched[v] = 1;
    matchcount++;
  }
}

fclose(in);

 /* is there a vertex cover of size <= k? */

if (matchcount <= k){

  /* determine the number of edges in subgraph defined by outsiders */
  /* and their neighbors. */

  outedges = 0;
  for(i = 0; i < n; i++)
    if (matched[i] == 0)
      outedges += degree[i];

  /* print problem line of DIMACS format */
  fprintf(out, "p max %d %d\n", n + 2, outedges + n);
  fprintf(out, "n %d s\n", n+1);
  fprintf(out, "n %d t\n", n+2);

  /* print edge lines of DIMACS format */
  /* note: cardmp numbers verticies 1 through n */
```

```
  for(i = 0; i < n; i++)
    if(matched[i] == 0){
      fprintf(outsiders, "%d ", i);
      fprintf(out, "a %d %d 1\n", n+1, i+1);
      for(j = 0; j < degree[i]; j++)
        fprintf(out, "a %d %d 1\n", i+1, graph[i][j]+1);
    }
    else
      fprintf(out, "a %d %d 1\n", i+1, n+2);
  fprintf(outsiders, "\n");

  tmp = fopen("tmp", "w");
  if (tmp == NULL) {
    perror("Unable to open file for input\n");
    exit(1);
  }

  fprintf(tmp, "%d\n%d\n", n, k);

  fclose(tmp);

}
else {
  printf("There is no vertex cover of size %d\n", k);

  tmp = fopen("tmp", "w");
  if (tmp == NULL) {
    perror("Unable to open file for input\n");
    exit(1);
  }

  fprintf(tmp, "%d\n%d\n", -1, 0);

  fclose(tmp);
}

fclose(outsiders);
fclose(out);

/* release memory */
for (i = 0; i < n; i++)
```

```
    free(graph[i]);
  free(graph);

  free(degree);
  free(matched);

  return 0;
}
```

Crown2.c

```
/* crown2--the final step in a crown reduction algorithm        */
/* This program takes an integer k as an argument and attempts  */
/* to find a crown that can be used to find a vertex cover of   */
/* size less than or equal to k. It also takes the names of     */
/* two files as arguments. The first is the name of a DIMACS-A  */
/* (DIMACS without the characters) file that describes the      */
/* graph. This should be the same file used in crown1. The      */
/* second is the name of the output file. The output is the     */
/* remainder of the graph to be searched for a vertex cover in  */
/* DIMACS-A format. The program also requires an input file,    */
/* dinic.out, that contains the DIMACS file that defines is     */
/* solution to a network flow problem used to find a            */
/* maximum matching on an appropriate subgraph. This file can   */
/* be generated using crown1 and dinic. The finally, the        */
/* program produces a recover file, crown.recover, that can be  */
/* used to recover the vertex cover once it has been found.     */
/* Henry Suters, June 10, 2003                                  */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int n, k, edges, i, j, u, v, done;
  int newedges, transcount, junk2, matchsize, num_neighbors;
  int **graph;
  int *match, *I, *H, *translate, *Outsider, *Neighbor;
  char junk;
  FILE *in, *out, *matchin, *recover, *outsiders, *tmp;

  /* get and check arguments */

  if (argc < 5) {
```

```
  fprintf(stderr, "Usage: %s k input_file output_file cover_file\n", *argv);
  exit(1);
}

k = atoi(argv[1]);

in = fopen(argv[2], "r");
if (in == NULL) {
  perror("Unable to open file for input\n");
  exit(1);
}

matchin = fopen("dinic.out", "r");
if (matchin == NULL) {
  perror("Unable to open file for input\n");
  exit(1);
}

outsiders = fopen("crown.outside", "r");
if (outsiders == NULL) {
  perror("Unable to open file for input\n");
  exit(1);
}

out = fopen(argv[3], "w");
if (out == NULL) {
  perror("Unable to open file for output\n");
  exit(1);
}

recover = fopen(argv[4], "w");
if (recover == NULL) {
  perror("Unable to open file for output\n");
  exit(1);
}

/* allocate data structures */

fscanf(in, "%d %d\n", &n, &edges);

graph = (int **) malloc(n * sizeof(int *));
if (graph == NULL){
```

```
  perror("Out of memory\n");
  exit(1);
}
for (i = 0; i < n; i++){
  graph[i] = (int *) calloc(n, sizeof(int));
  if (graph[i] == NULL){
    perror("Out of memory\n");
    exit(1);
  }
}

match = (int *) calloc(n, sizeof(int));
if (match == NULL){
  perror("Out of memory\n");
  exit(1);
}

translate = (int *) calloc(n, sizeof(int));
if (translate == NULL){
  perror("Out of memory\n");
  exit(1);
}

I = (int *) calloc(n, sizeof(int));
if (I == NULL){
  perror("Out of memory\n");
  exit(1);
}

H = (int *) calloc(n, sizeof(int));
if (H == NULL){
  perror("Out of memory\n");
  exit(1);
}

Outsider = (int *) calloc(n, sizeof(int));
if (H == NULL){
  perror("Out of memory\n");
  exit(1);
}

Neighbor = (int *) calloc(n, sizeof(int));
```

```
if (H == NULL){
  perror("Out of memory\n");
  exit(1);
}

/* read in graph data */

for (i = 0; i < edges; i++){
  fscanf(in, "%d %d\n", &u, &v);
  graph[u][v] = 1;
  graph[v][u] = 1;
}

fclose(in);

/* read in maximum matching data */
/* note: dinic numbers verticies 1 through n */

fscanf(matchin, "%c %d\n", &junk, &i);

for(i = 0; i < n; i++)
  match[i] = -1;

matchsize = 0;
while(!feof(matchin)){
  fscanf(matchin, "%c %d %d %d\n", &junk, &u, &v, &junk2);
  if(u != n+1 && v != n+2){
    matchsize++;
    match[v-1] = u-1;
    match[u-1] = v-1;
  }
}

fclose(matchin);

if (matchsize <= k){

  /* read in outsiders and find unmatched outsiders*/

  if (matchsize > 0){
    while (!feof(outsiders)){
      fscanf(outsiders, "%d", &i);
```

```
    Outsider[i] = 1;
    for (j = 0; j < n; j++)
      if (graph[i][j] == 1)
        Neighbor[j] = 1;
    if (match[i] == -1)
      I[i] = 1;
  }
}

fclose(outsiders);

/* count neighbors */

num_neighbors = 0;
for (i = 0; i < n; i++)
  num_neighbors += Neighbor[i];

/* determine crown */

if (num_neighbors == matchsize){
  printf("Simple Crown\n");
  /* we already have a crown */
  free(I);
  free(H);

  I = Outsider;
  H = Neighbor;
}
else{
  printf("Complex Crown\n");
  /* we need to isolate a crown */
  free(Neighbor);
  free(Outsider);
  done = 0;
  while(done == 0){
    done = 1;
    for(i = 0; i < n; i++)
      if(I[i] == 1){
        for(j = 0; j < n; j++)
          if(graph[i][j] == 1 && H[j] == 0){
            done = 0;
            H[j] = 1;
```

```
            if (I[j] == 1) printf("error %d\n", j);
          }
      }
    for(i = 0; i < n; i++)
      if (H[i] == 1)
       I[match[i]] = 1;
  }
}

/* determine vertices in cover, not in cover, or yet to be determined */

transcount = 0;
newedges = 0;
fprintf(recover, "%d\n", n);

for(i = 0; i < n; i++){

  /* if a vertex is in H then it is in the vertex cover */

  if(H[i] == 1){
    fprintf(recover, "%d ", 1);
    k--;
  }

  /* if a vertex is in I then it is not in the vertex cover */

  if(I[i] == 1)
    fprintf(recover, "%d ", -1);

  /* if a vertex is not in either H or I then its status is undetermined */
  /* count the number of undetermined verticies and uncovered edges */
  /* determine the compacted numbering of vertices in this subgraph */

  if (H[i] == 0 && I[i] == 0){
    fprintf(recover, "%d ", 0);
    translate[i] = transcount;
    transcount++;
    for(j = i + 1; j < n; j++)
      if(graph[i][j] == 1 && H[j] == 0 && I[j] == 0)
        newedges++;
  }
}
```

```
  fprintf(recover, "\n");

  fclose(recover);

  /* print out the new graph */

  fprintf(out, "%d %d\n", transcount, newedges);
  for(i = 0; i < n; i++)
    for(j = i + 1; j < n; j++)
      if (graph[i][j] == 1 && H[i] == 0 && I[i] == 0 && H[j] == 0 && I[j] == 0)
        fprintf(out, "%d %d\n", translate[i], translate[j]);

  fclose(out);
}
else{
  printf("There is no vertex cover of size %d\n", k);
  tmp = fopen("tmp", "w");
  if (tmp == NULL) {
    perror("Unable to open file for input\n");
    exit(1);
  }

  fprintf(tmp, "%d\n%d\n", -1, 0);

  fclose(tmp);
}

/* release memory */

for (i = 0; i < n; i++)
  free(graph[i]);
free(graph);
free(I);
free(H);
free(match);
free(translate);

printf("The new cover size is %d out of %d verticies\n", k, transcount);

tmp = fopen("tmp", "w");
if (tmp == NULL) {
  perror("Unable to open file for input\n");
```

```
    exit(1);
  }

  fprintf(tmp, "%d\n%d\n", transcount, k);

  fclose(tmp);

  return 0;
}
```

Vita

William Henry Suters, III was born in Berea, Kentucky, on January 28, 1967. He attended the community school in Berea and graduated from Berea College in 1989 with a BA with a double major in mathematics and physics. He then attended graduate school in mathematics at Duke University where he earned an M.A. in 1991 and a Ph.D. in 1994. After receiving his doctorate, he obtained a faculty position at Carson-Newman College in Jefferson City, Tennessee, where he currently teaches mathematics and computer science courses and holds the rank of associate professor. In 2002 he began pursuing an M.S. degree in computer science at the University of Tennessee.