Some Problems are NP-Harder than Others

1. Introduction

In this module we will investigate some ideas related to computational complexity using two classic graph theory problems: the Vertex Cover problem and the Dominating Set problem. Formally, a graph is a pair of sets, $G = (V, E)$, where V is a finite set of *vertices* (sometimes called *nodes*) and E is a set of unordered pairs of vertices called *edges*. If $e = (u, v) \in E$, then we say that u and v are *adjacent* vertices (or *neighbors*), and that e is *incident* to both u and v; u and v are called the *endvertices* of e. The *size* (or *order*) of a graph is the number of vertices in the graph, $n = |V|$. For example, suppose $V = \{u, v, w, x, y, z\}$ and $E = \{(u, v), (u, w), (v, x), (v, w), (v, y), (v, z)\}$. Then the graph $G = (V, E)$ has size $n = 6$ and can be represented with a diagram as follows.

FIXME graph 1

Graphs can be used to construct models that help us analyze and solve real world problems. For example, vertices can represent locations within a city, with edges between vertices corresponding to locations connected by a road. We could then use the graph to help us find the most efficient way to visit all of the locations. As another example, vertices could represent students in a class, with an edge between two vertices indicating that the corresponding students are willing to work on a project together. As a final example, vertices could represent course offered at a university during a particular term, with an edge between two vertices indicating that there is at least one student taking both of the corresponding courses. We could use the graph to schedule final exams so that students have no conflicts.

2. The Problems

A *vertex cover* of a graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that for each edge $e = (u, v) \in E$, either $u \in C$ or $v \in C$. The Vertex Cover problem is to find the size (number of vertices) of a smallest vertex cover. A related problem is the k-Vertex Cover problem, which asks whether there exists a vertex cover of size k (where k is a positive integer). Note that if G has a vertex cover of size strictly less than k, then the answer to this question is "yes". (Why?)

A *dominating set* in $G = (V, E)$ is a set of vertices $D \subseteq V$ such that every vertex $v \in V$ is either itself an element of D or is adjacent to an element of D. The Dominating Set problem is to find the size of a smallest dominating set; the k-Dominating Set problem asks whether there is a dominating set of size (less than or equal to) k.

Exercise 1: Let G be the graph given in the figure below.

FIXME graph here

(1) For each set S, determine if S is a vertex cover for G. Are any of the sets a minimum vertex cover? Give reasons for your answers.
   (a) $S = \{a, b, c, e, g, f\}$
   (b) $S = \{a, c, g, f, k\}$

(c) $S = \{b, c, d, e, f, g, k\}$

(2) For each set S, determine if S is a dominating set for G. Are any of the sets a minimum dominating set? Give reasons for your answer.
   (a) $S = \{c, d, e, k\}$
   (b) $S = \{a, c, f\}$
   (c) $S = \{a, b, e, h, k\}$

Exercise 2: The *degree of a vertex* v is the number of edges incident to v in the graph. Suppose a graph G has a vertex v of degree 0; such a vertex is called *isolated*.

(1) Will v be in a minimum vertex cover of G? Why or why not?
(2) Will v be in a minimum dominating set for G? Why or why not?

Exercise 3: Suppose G is a graph with no vertices of degree 0.

(1) Prove that any vertex cover of G must also be a dominating set for G.
(2) Give a counterexample to show that the converse of the statement in part (a) is false.

Exercise 4: A subset of vertices in a graph is called *independent* if no two vertices in the subset are adjacent. Let C be a vertex cover in the graph $G = (V, E)$. Show that $I = V \backslash C$ (*i.e.* the complement of C in V) is an independent set. Conclude that the problems of finding a maximum independent set and a minimum vertex cover are equivalent.

Exercise 5: The *complement* of a graph $G = (V, E)$ is the graph $\bar{G} = (V, \bar{E})$, where $\bar{E}$ consists precisely of all edges *not* in G. A subset of vertices is called a *clique* if any two vertices in the subset are adjacent. Show that a vertex subset J in G is independent if and only if J is a clique in $\bar{G}$. Conclude that the problems of finding a a maximum independent set and a maximum clique are equivalent.

3. Applications of the problems

Suppose the edges of a graph represent display hallways in an art gallery, and the vertices represent intersections of those hallways. We want to position security guards at the intersections in such a way that each gallery hallway is covered, using the fewest total number of guards possible.

Suppose a group is trying to form a committee that will be representative of and responsive to the group's needs and ideas. Each member of the group is represented by a vertex. Each person in the group designates individuals whom he or she feels would represent his or her ideas on the committee. An edge between two vertices indicates that the two people designated each other (we assume that such designations are always reciprocated). The group would like to forma representative committee of minimum size.

Suppose the vertices of a graph represent locations in a nuclear power plant. An edge between vertices indicates that a guard at either location can see a warning light at the other location (we assume that a guard can see a warning light at his or her own location). We want to find the minimum number of guards needed to oversee all locations in the plant.

Exercise 4

(1) Explain why the "art gallery guard" problem is an application of the VERTEX COVER problem.
(2) Explain why the "representative committee" problem is an application of the DOMINATING SET problem.
(3) Explain which problem models the "nuclear power plant guard" problem.

4. Integer programming (IP) Formulations

To solve either VERTEX COVER or DOMINATING SET, several approaches can be used. Since the solution to either problem involves finding a subset of the vertex set V, it would be theoretically possible to list all possible subsets, determine which subsets represent vertex covers (or dominating sets), and from those, choose a subset of minimum order. In practice this is rarely feasible.

Exercise 5: Recall that a set of n elements has 2^n possible subset. Given a set of vertices V, how many subsets of V are there if

a. $|V| = 20$ b. $|V| = 50$ c.$|V| = 100$

Exercise 6: In the previous exercise, suppose that a computer takes 10^{-6} seconds (*i.e.* a millionth of a second) to compute each subset. How long would it take to compute all subsets in parts (a) through (c)?

Another approach is to formulate these problems as integer programming problems. Let us begin with the vertex cover problem. Suppose a graph $G = (V, E)$ has n vertices, numbered $1, 2, 3, \ldots, n$. The decision variables are, for $i = 1, ..., n$ $x_i = 0$ if vertex i is not in the cover and $x_i = 1$ if vertex i is in the cover. We want to solve the following problem.

minimize

$$\sum_{i=1}^{n} x_i$$

subject to

$$\begin{aligned} x_i + x_j &\geq 1 \text{ for every } (i,j) \in E \\ x_i &= 0 \text{ or } 1 \text{ for } i = 1, 2, \ldots, n \end{aligned}$$

The dominating set problem can be similarly formulated as an integer programming problem. Again, the decision variables are x_i, for $i = 1, \ldots, n$; $x_i = 0$ if vertex i is not in the dominating set and $x_i = 1$ if vertex i is in the dominating set. For this problem, we want to:

minimize

$$\sum_{i=1}^{n} x_i$$

subject to:

$$\begin{aligned} x_i + \sum_{(i,j)\in E} x_j &\geq 1 \text{ for each } i = 1, \ldots, n \\ x_i &= 0 \text{ or } 1 \text{ for } i = 1, 2, \ldots, n \end{aligned}$$

Exercise 7: Consider the given IP formulations of the two problems.

(1) Explain why the two IP formulations have the same objective function.
(2) For each model, explain why the constraints are appropriate.

Exercise 8 A *matching* in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that no two edges in M have a common end vertex. The MAXIMUM MATCHING problem is to find the maximum size of matching. Give an IP formulation of this problem. (*Hint.* Use one decision variable for each edge in G.)

5. Linear Programming and Integer Programming

The example below will allow us to explore some of the ideas behind the relationship between a linear programming problem (LP) and the integer programming problem (IP) with the same objective function and constraints. If our variables must be integers, can we solve the IP by rounding the LP solution? By rounding to the nearest feasible solution? Whats different about the IP problem?

We will look at a small, very simplified, linear programming problem involving a diet. Interestingly, a diet problem, with many more variables and constraints than the one below, was one of the first problems used to test the simplex algorithm. This algorithm was devised by George Danzig in the 1940s and is still the most widely used algorithm for solving linear programming problems. Here is our problem, put together using information from www.mcdonalds.com.

Suppose you are going to eat only two foods: six-piece orders of McDonalds chicken nuggets (McNuggets) and small orders of McDonalds french fries. You want to maximize the protein you eat while keeping calories and sodium intake within reasonable limits. Each six-piece order of nuggets contains 15 grams of protein, 250 calories and 800 mg of sodium. Each small order of french fries contains 3 grams of protein, 220 calories and 150 mg of sodium. You want to have no more than 1500 calories and no more than 2500 mg of sodium per day from these foods.

Let m represent the number of six-piece orders of McNuggets and f, the number of small orders of french fries.

maximize:

$$p = 15m + 3f$$

subject to:

$$\begin{aligned} 250m + 220f &\leq 1500 \text{ (calorie constraint)} \\ 800m + 150f &\leq 2500 \text{ (sodium constraint)} \end{aligned}$$

with:

$$m > 0, \quad n > 0$$

Well first look at the feasible solution space on the graph that follows: its the region below and to the left of both lines.

The optimal solution to the LP problem occurs at one of the corner points of the feasible solution space: $(0, 0), (3.125, 0), (2.3466, 4.1516)$ or $(0, 6.8182)$. We can compute p at each of these points, or we can look at the slope of the objective equation for a fixed value of p to determine at which of these points the optimal solution occurs. Either method will show that the optimal solution occurs at the intersection of the two constraint lines, where $m = 2.3466$ and $f = 4.1516$. The optimal value is $p = 15 * 2.3466 + 3 * 4.1516 = 47.6538$ grams of protein.

Exercise 9: Why might these solutions involving non-integer values of m and f be undesirable?

Now suppose we want to limit our solution to integer values of m and f. Is choosing the closest integers to the optimal LP solution, namely $(m, f) = (2, 4)$ going to be the optimal integer solution? Maybe, maybe not!

One way to be sure is to evaluate the objective function at *all* of the feasible integer solutions: $(0, 0), (0, 1), ...(0, 6); (1, 0), (1, 1), ...(1, 5); (2, 0), (2, 1), ...(2, 4); (3, 0)$. We can no longer just look at the boundary of the feasible solution space. In fact, the boundary doesn't contain any integer solutions! It turns out that the optimal integer solution occurs at $(3, 0)$. At this solution, the amount of protein $p = 45$ grams.

Exercise 10: What is the solution to the LP problem if we keep all requirements the same, except we change the limit on the amount of sodium to no more than 2300 mg per day?

Exercise 11: What is the solution to the IP problem if we keep all requirements the same, except we change the limit on the amount of sodium to no more than 2300 mg per day?

Exercise 12: What implications do these computations have for deducing the solution of an LP problem with integer restrictions on the decision variables (i.e. an IP problem) from the solution of the LP problem with no such restrictions?

6. Branch and Bound

As we have seen, one way to solve an integer programming (IP) problem is to ignore the fact that the decision variables are integers, and simply solve the problem as if it is a linear programming (LP) program using standard techniques such as the simplex method. This related problem is called the *LP-relaxation*. In some instances, we may be very fortunate in that the solution to the LP relaxation will have integer values for all of the variables. In this case, we have also solved the

IP. If not, the solution to the LP-relaxation provides us with a lower bound on the value of our objective function in the IP. This solution can be used as a starting point for solving IP problems. One method that we will only mention briefly is to add new constraints, called *cutting planes*, that systematically eliminate non-integral solutions until a solution to the IP is found.

A different approach that we will describe in this section is called the *branch and bound* method. If we solve the LP-relaxation of an IP and do not obtain an integral answer, the first step is to choose a variable whose value in the current optimal solution is non-integral; this is called the *branching variable*. We can form two new LP relaxation problems in which the branching variable is set equal to 0 and 1 respectively. If the solutions to these problems are not integral, then we choose another branching variable and repeat the process.

To understand this idea more clearly, let us consider a specific problem. Consider the graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4), (1, 5)\}$. (Note that G can be constructed by forming a complete graph on 4 vertices and then adding the edge $(1, 5)$). We wish to find a vertex cover of minimum size by solving the following IP.

minimize

$$\sum_{i=1}^{5} x_i$$

subject to:

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ x_1 + x_3 &\geq 1 \\ x_1 + x_4 &\geq 1 \\ x_2 + x_3 &\geq 1 \\ x_2 + x_4 &\geq 1 \\ x_3 + x_4 &\geq 1 \\ x_1 + x_5 &\geq 1 \\ x_i &= 0 \text{ or } 1 \text{ for } i = 1, 2, 3, 4, 5 \end{aligned}$$

We form a linear programming problem by relaxing the integrality constraint on the decision variables to $0 \leq x_i \leq 1$. We will refer to this problem as LP1. If we solve LP1, we obtain an optimal value of $z = 2.5$, with optimal solution $\mathbf{x} = (0.5, 0.5, 0.5, 0.5, 0.5)$. Since we did not obtain an integral solution, we have not solved the IP. However, we know that the minimum value of z cannot be any less than 2.5. This is the *bounding* part of the method.

Since none of the variables had integer value in the optimal solution to LP1, we could choose any one of them to be a branching variable. Let us choose x_1 as our first branching variable. We form a new linear programming problem, LP2, by adding the constraint $x_1 = 0$ to LP1. Similarly, we form a third problem, LP3, by adding the constraint $x_1 = 1$ to LP1. An optimal solution to LP2 is $\mathbf{x} = (0, 1, 1, 1, 1)$ which gives $z = 4$. This is a feasible solution to our IP, but we do not yet know if it is optimal. Solving LP3 produces an optimal solution $\mathbf{x} = (1, 0.5, 0.5, 0.5, 0)$ which gives $z = 2.5$.

Since the solution to LP3 is not integral, we must choose another branching variable. Suppose we pick x_2. Branching from LP3, we form two new problems. In problem LP4 we have $x_2 = 0$ and

in LP5 we have $x_2 = 1$. An optimal solution to LP4 is $\mathbf{x} = (1, 0, 1, 1, 0)$ which gives $z = 3$. An optimal solution to LP5 is $\mathbf{x} = (1, 1, 0.5, 0.5, 0)$ which also gives $z = 3$.

The solution to LP4 is integral which provides us with another feasible solution to the IP, and in fact, provides us with a new lower bound on the solution, since we obtained $z = 3$ as opposed to $z = 4$ in LP2. Although the solution to LP5 is not integral, further branching is not necessary. Note that the value of z in LP5 is also equal to 3, which does not improve upon our lower bound. If we add new constraints to LP5 by requiring other variables to have integer values, the value of z will either remain the same or increase, but can never decrease. Therefore, we have used bounding to limit our search for optimal solutions to the IP. Although we have checked only 2 of the 25 feasible solutions to the IP, we can state with confidence that we have found an optimal solution.

The following diagram summarizes our solution to this example problem:

Note that depending on the method that is used for solving linear programs, other optimal solutions may be found. For instance, $\mathbf{x} = (1, 0.5, 0.5, 0.5, 0)$ is another optimal solution to LP1, and $\mathbf{x} = (1, 1, 1, 0, 0)$ is another optimal solution to LP5.

Exercise 13: Let G be a complete graph with 5 vertices. Find a minimum vertex cover by formulating an IP and using the branch and bound method. Draw a diagram that illustrates the branches formed during your solution.

Exercise 14: Let $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{(1, 2), (2, 3), (3, 4), (1, 4), (5, 6), (6, 7), (7, 8), (
Draw a diagram of this graph and try to find a minimum vertex cover by inspection. Formulate the problem as an IP and solve it using branch and bound.

Exercise 15: Let G be the graph given in the previous question. Try to find a minimum dominating set by inspection. Formulate the problem as an IP and solve it using branch and bound.

7. Computational Complexity

Although Vertex Cover and Dominating Set are easy to state, they quickly become very difficult to solve as the size of the graph grows, even by the fastest computers. Formulating them as IP's does not help matters much; there are no fast algorithms for integer programming. These two problems share this property with a number of other classic problems in discrete mathematics. What is curious is that for some of these problems, there are clever tricks that enable us to solve them efficiently, whereas others have defied all attempts to find similar shortcuts. There appears to be a natural division in the world of combinatorial problems between those that are easy and those that are hard. In this section, we make this idea more precise.

The study of these issues is called *computational complexity*. A general problem, such as Vertex Cover, is formally a collection of *instances* of the problem, in this case one for each particular graph $G = (V, E)$. An algorithm for solving the problem must be general enough to handle all possible instances as input. The *complexity* of an algorithm is a rough measure of the maximum number of elementary computations required to obtain a solution, as a function of the size of the input; this is used to give an indication of the worst-case running time of the algorithm (one that is independent of the increasing speed of computers). When the input is a graph $G = (V, E)$, for example, the size is usually taken to be $n = |V|$. It is often difficult to specify the complexity

function $f(n)$ very precisely, but certainly we expect it to be an increasing function (*i.e.* the bigger the size of the input, the longer the algorithm will take). Of paramount importance is *how* quickly the complexity grows as n increases. In particular, is the growth no worse than polynomial in nature, or is it exponential? It is common to describe the complexity of an algorithm solely on the basis of its 'order of magnitude' in terms of growth, using what is called *big-O notation*. This is formally defined below.

Definition 1. *Let* $\mathbb{N}$ *represent the natural numbers (*i.e. *the positive integers) and* $\mathbb{R}^+$ *the positive real numbers, and let* f *and* g *be functions* $\mathbb{N} \to \mathbb{R}^+$*. Then* $f(n)$ *is* $O(g(n))$ *if* $f(n)$ *is eventually dominated by some positive multiple of* $g(n)$*; that is, there exist* $m \in \mathbb{N}$ *and* $C \in \mathbb{R}^+$ *such that* $f(n) < Cg(n)$ *for all* $n \geq m$*.*

For simplicity, we can restrict our attention to the class of *decision problems*, those for which the answer is either 'yes' or 'no'. Some examples are:

- PRIME: Is a given positive integer composite (as opposed to prime)?
- SATISFIABILITY: Given a compound logical expression, is there a set of truth values for the constituent propositions that make the entire compound expression true?
- 2-COLORABILITY: Can all the vertices of a graph be colored with red and blue so that no edge joins two vertices of the same color?

Any optimization problem can be solved indirectly by solving a number of decision problems. For example, we could determine the minimum size of a vertex cover of a graph G by repeatedly asking if there exists a vertex cover of size k, for a series of intelligent choices of positive integer k. If we use binary search, then because the minimum vertex cover has size at most $n = |V|$, we need to solve such a decision problem at most $\lfloor \log_2 n \rfloor$ times. This is such a slow-growing function of n that the complexity of the optimization problem is essentially that of the associated decision problem. This tactic can in fact be applied to any integer programming (IP) problem.

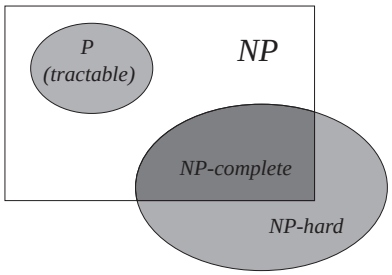
A decision problem is said to belong to the class NP if a 'yes' answer can be quickly verified by means of a *certificate*. The letters NP stand for *nondeterministic polynomial-time algorithm*; the 'polynomial-time' refers only to how long it takes to check the certificate. For example, we could certify that, yes, n is composite by providing two integers s and t that satisfy $n = st$; this multiplication can be verified very quickly by a computer. We could certify that, yes, the compound expression p is satisfiable by providing a list of appropriate truth values for the constituent propositions; this can be quickly verified using binary multiplications and additions. We could certify that, yes, G has a vertex cover of size k by listing the vertices of one such cover; we then simply have to check that each edge has one endvertex in this set.

Checking a certificate is quite different from coming up with a certificate. A brute force approach to the k-VERTEX COVER problem would involve investigating whether each k-subset of vertices is a cover; for a graph with n vertices, there are $\binom{n}{k}$ such subsets and in a worst-case scenario we would have to test each one of them. A crucial point is that this number gets very large very quickly as n and k grow; while it may be possible to handle relatively small instances with this method, large instances become intractable even for the fastest computers of today and of the foreseeable future.

Exercise 16 For the values of n and k listed in the table below, compute how long a computer would take to perform $\binom{n}{k}$ elementary computations, assuming it can perform a billion such computations per second. Express your last three answers in terms of years.

n	k
50	10
100	10
100	50
200	50
200	100

For some problems, there are much faster - in particular, polynomial-time - alternatives to the brute force approach. For example, a polynomial-time algorithms exists for determining whether a graph is 2-colorable. In 2002, a polynomial-time algorithm was finally discovered for determining whether a number is composite or prime. This subclass of problems within the class NP is labeled P; more informally, such problems are called *tractable*. Another example of a problem in P is MAXIMUM MATCHING, encountered in Exercise 6. A famous open problem in mathematics asks whether in fact $P = NP$. That is, will we eventually find quick algorithms for solving *every* problem in NP?



One step towards answering this question is the identification of a set of problems that have the amazing property that if any *one* of them can be solved in polynomial time, then so can *every* problem in NP. Such problems are called *NP-hard*. A problem that is both NP-hard and is itself in NP is called *NP-complete*; see the Venn diagram below. In fact, these terms are often used interchangeably. The first problem shown to be NP-complete was SATISFIABILITY (in 1971). Over the years, many bright minds spent a lot of time and effort trying to find efficient algorithms for various NP-hard problems with no success, and as a result the consensus among experts is that P is probably a proper subset of NP.

There are by now many optimization problems that have been shown to be NP-hard, among them VERTEX COVER and DOMINATING SET. Formulating these problems as IP's does not help, because in general, integer programming is also NP-hard. However, we cannot simply give up on large instances of these and other NP-hard problems. They crop up in the real world in many different fields, such as telecommunications and computational biology, where it is important to get some kind of answer. This has led to the development of techniques like branch and bound. It is important to now that while branch and bound is often successful at solving large-scale IP's in a reasonable amount of time, it comes with no guarantees. It is quite possible to bogged down investigating an exponentially increasing number of branches. Interestingly, however, general linear programming (LP) is in P!

A recent approach to NP-hard problems has focussed on determining what component of the input size is responsible for the computational intractability. For example, the first improvement in an algorithm for k-VERTEX COVER had running time $O(2^k n)$; currently the best algorithm has

complexity $O(1.271^k + kn)$. Note that in both cases only k contributes to the exponential growth in the running time. This leads to the following definition.

Definition 2. (1) *A* parameterized *problem is one for which the size of an instance is an ordered pair (k, n); k is referred to as the* parameter *and n as the* main input size.
(2) *A parameterized problem is* fixed-parameter tractable *(or FPT) if there exists an algorithm that solves the problem with complexity $f(k)p(n)$, where p is a polynomial in n (and f is an unrestricted function of k).*

(Note that the latest algorithm for k-Vertex Cover satisfies this definition, because for all positive integers n, $1.271^k + kn \leq (1.271^k + k)n$, and complexity is concerned only with putting an upper limit on the running time.) By contrast, there are no known algorithms for the k-Dominating Set problem that do any better than simply determining whether each possible k-subset of the vertices is a dominating set, and this has complexity $O(n^{k+1})$. In this case, we cannot isolate the parameter k in one multiplicative factor. The following exercise demonstrates what a difference this makes to running time.

Exercise 17 For the values of n and k listed in the table in Exercise 16, compute the ratio $\frac{n^{k+1}}{2^k n}$.

One might hope that for k-Dominating Set, it is not k that is causing the computational complexity, but some other parameter. Maybe if we just reconfigure the input, with some other part of the input size playing the role of the parameter, we can get an algorithm of the required form. In fact, parameters need not even be numbers; they can reflect not just one but several structural properties of the input. However, it has been shown that it is highly unlikely that Dominating Set is FPT - as unlikely, in some sense, as $P = NP$.

Even though the table in the exercise shows that $2^k n$ is clearly more manageable than n^{k+1}, it is still possible for this number to get too large to be computable in a reasonable amount of time. However, a purely empirical observation is that for most parameterized problems encountered in the real world, the value of the parameter k is comparatively small (for example, under 100). This keeps the value of $2^k n$ manageable, while n^{k+1} remains unwieldy.

In fact, the advantage of having a small value of k gets compounded. A crucial feature of FPT problems is that they are particularly amenable to clever pre-processing techniques that (quickly) reduce each instance to a smaller instance of the same problem. If the reduction is drastic enough, it is still practical to apply even exponential-time algorithms to solve the smaller instance. The following makes this more precise.

Definition 3. *A parameterized problem P is* kernelizable *if there are rules for transforming any instance I of size (k, n) in polynomial time to another instance I' of size (k', n') satisfying $k' \leq k$ and $n' \leq h(k')$, where h is an (unrestricted) function of k'.*

The reduced instance I' is the 'kernel' of the original instance; implied in this definition is that a solution to the kernel I' can be 'lifted' to a solution of I. Note that the main input size n' of I' is bounded by a function of the parameter k', which is itself bounded by k; the main input size of the original instance has completely disappeared! This demonstrates that the time required to solve an instance of the problem ultimately depends *only* on the size of the parameter.

The result below shows that this reduction strategy works for all FPT problems.

Theorem 1. *A parameterized problem is kernelizable if and only if it is FPT.*

Hence, in applications where it can be assumed that the parameter is reasonably small, this class of NP-hard problems is for all practical purposes tractable.

8. Preprocessing for the Vertex Cover Problem

We now present some preprocessing rules that allow us to kernelize an instance of the k- Vertex Cover problem on a graph G of size n. These rules are applied iteratively to obtain smaller and smaller instances of the problem, until we obtain one so small that it can reasonably be solved even with an exponential time algorithm.

Each rule allows us to assume that certain vertices of the current graph either must be, or must not be, in the vertex cover we seek. We record those that must be included, then create a new, reduced graph. Not only will this graph be smaller in terms of number of vertices, the size of the cover we are checking for (k) will also usually be smaller. For notational purposes, we will use $G = (V, E)$, n and k to denote the 'current' graph, its size and parameter. We let S denote the set of vertices of G that must be contained in a minimum vertex cover of G (before we apply any rules, $S = \emptyset$). We use G', n' and k' to denote the graph, size and parameter obtained after a preprocessing rule has been applied.

Recall that we are tackling the k-Vertex Cover problem only as a means of answering the original Vertex Cover problem. Thus, although we are officially checking for vertex covers of size k, we are in fact interested in finding vertex covers of minimum size.

- **Rule 1**. If G has a vertex v of degree 0, it cannot be in a vertex cover of minimum size. Since v does not "cover" any edges, we don't gain anything by including v in a vertex cover (as you should have discovered in Exercise 2). Thus we eliminate all isolated vertices from the graph to obtain G'; in addition, $n' = n -$ (the number of deleted vertices), and $k' = k$. The set S is unchanged. In the example that follows, $n = 6$ and $n' = 4$.
  FIXME graph 3
- **Rule 2**. If G has a vertex v of degree 1, then we can assume that any minimum vertex cover does not contain v, but does contain the one vertex u which is adjacent to v. To see this, note that any vertex cover must contain either u or v. Since v has degree 1, including v in a vertex cover will only cover one edge. Including u in a vertex cover will cover that edge and may cover other edges as well. Thus including u in the vertex cover will do no worse than including v and might do significantly better, as far as covering other edges is concerned. Thus, we add u to S, then delete both u and v and any edges incident to them to get the reduced graph G'. By Rule 1, we can also delete any other neighbor of U whose degree drops to 0. Next we set $n' = n -$ (the number of deleted vertices), and $k' = k - 1$ (because we've included the vertex in the vertex cover). In the graph G below, we apply the rule once to obtain the graph G'. The vertex cover, so far, includes u_1. After the first application of Rule 2, we have $n' = n - 2 = 5$. We can apply the rule again to get the reduced graph with three vertices. Now the vertex cover, so far, includes u_1 and u_2.
  FIXME graph 4
- **Rule 3**. Suppose G has a vertex v of degree 2 and its two neighbors, u and w, are adjacent. Then any vertex cover must contain at least two of these three vertices in order to cover the

three edges $(u, v), (u, w)$ and (v, w). Note that v only covers two of these edges (since v has degree 2), while u and w may cover other edges as well. Thus we may assume that the two adjacent neighbors u and w are in any minimum vertex cover, and v is not. Hence, we add u and w to S, then delete all three vertices v, u, and w and their incident edges. We also delete each neighbor whose degree drops to 0. We set $n' = n -$ (number of deleted vertices) $= n - 3$, and $k' = k -$ (number of vertices added to S) $= k - 2$. In the graph G below, we delete vertices u, v, and w and all incident edges to obtain the reduced graph G'. Then $n' = n - 3 = 3$ and the vertex cover, so far, includes u and w.

FIXME graph 5

- **Rule 4**. Suppose G has a vertex v of degree 2 and its neighbors u and w are *not* adjacent. In this case, the reduced graph G' is obtained by replacing the three vertices u, v, and w with one "supervertex" v'; the neighbors of v' in G' are exactly the neighbors of u and w in G. Note that $n' = n - 2$. Determining what changes need to be made to S and k requires more subtle investigation. Suppose C' is a minimum vertex cover of G'. There are two cases to consider.
  - **Case 1**: If $v' \notin C'$, then the edges incident to the 'supervertex are covered by other vertices in C'. Then the set $C = C' \cup \{v\}$ forms a minimum vertex cover of the original graph G; adding v ensures that the edges (u, v) and (w, v) (which are in G but not G') are covered. In this case, we add v to S and set $k' = k - 1$.
    In graph G below, a minimum vertex cover of the reduced graph (shown as graph H) with supervertex v' includes p, r, and either q or s (but not v'). Thus, Case 1 applies and only v (of the vertices u, v, and w) is in the corresponding vertex cover of the original graph.
    FIXME graph 7
  - **Case 2**: If $v' \in C'$, then v' must be needed to cover some of the edges incident to v'. In this case, we can create a minimum vertex cover for the original graph by replacing the supervertex v' in C' with the two vertices u and w; that is, we let $C = (C \backslash \{v'\}) \cup \{u, w\}$. In this case, we are adding both u and w to S, but we still have $k' = k - 1$.
    An example follows. If we replace vertices u, v, and w in graph G below by a supervertex v' and then apply Rule 2, we see that v' would be in a vertex cover of the modified graph, so Case 2 applies. This means that vertices u and w would be included in the vertex cover of G.
    FIXME graph 8
- **Rule 5**: If G has a vertex of degree greater than k, then that vertex must be included in any vertex cover of size less than or equal to k. To see why this rule is true, suppose G has a vertex u of degree $m > k$ and let C be a vertex cover. If $u \notin C$, then each of its m neighbors must be in C, to ensure that all edges incident to u are covered. But then $|C| \geq m > k$. Hence we add to S all vertices of degree greater than k. To obtain the reduced graph G', we remove these vertices and all edges incident to them. We set $n' = n -$ (number of deleted vertices) and $k' = k -$ (number of deleted vertices).

Exercise 18 Prove that if a graph G has more than k vertices of degree greater than k (*i.e.* more than k vertices are removed when Rule 5 is applied), then the answer to the kVertex Cover problem on G is "no".

As mentioned earlier, we continue iteratively invoking these rules until none are applicable. Let G' be the final reduced graph, of size n' with parameter k'; this is the kernelized version of the original instance of the problem. The next step is to find a minimum vertex cover C' of G' and to check whether $|C'| \leq k'$. It should be clear from the description of the rules that the answer

to k'-VERTEX COVER on G' is "yes" if and only if k-VERTEX COVER on G is "yes". In this case, we can in fact reconstruct a minimum vertex cover for the original graph G. If Rule 4 was never invoked in the preprocessing phase, then simply set $C = C' \cup S$. If Rule 4 was invoked, then we must 'lift' C carefully, keeping track of whether any supervertices are included in any of the intermediate vertex covers. Hence, a "yes' answer to thek'-VERTEX COVER on G' allows us to also solve the optimization problem VERTEX COVER on G. Note, however, that if the answer to k'-VERTEX COVER on G' is "no", then we have not yet solved VERTEX COVER on G. We must begin again with a larger value of the parameter k.

A major concern with this strategy is how long it will take to find a minimum vertex cover on the kernelized graph G' using an exponential time algorithm. We could be wasting a lot of time if it turns out that the answer to k'-VERTEX COVER on G' is "no". Fortunately, there is a quick way of checking whether this is any hope that the answer will be "yes".

Theorem 2. *If G' has a vertex cover of size k', then $n' \leq \frac{k'^2}{3} + k'$.*

Proof. Let C' be a vertex cover in G' of size k'. Since no preprocessing rules apply to G', the degree of every vertex u must satisfy $3 \leq deg(u) \leq k'$. Let F denote the set of edges that have one endvertex inside C' and the other endvertex outside C'. Note that no two vertices outside C' can be adjacent to each other, by definition of a vertex cover. Since each of the $n' - k'$ vertices outside C' has degree at least 3, we can say $3(n' - k') \leq |F|$. On the other hand, since each of the k' vertices in C' has degree at most k', we can say $|F| \leq (k')^2$. Combining these two inequalities gives $3(n' - k') \leq (k")^2$; the rest of the proof is elementary algebra.

□

We must be careful when applying this theorem. If $n' > \frac{k'^2}{3} + k')$, then we can skip the search, and start anew with a larger initial parameter k. If $n' \leq \frac{k'^2}{3} + k')$, it is worth our while to search for the minimum vertex cover in G', but we may still find that it has size greater than k', and therefore have to start again.

Exercise 19: Let G be the graph shown below. Apply preprocessing rules 1 through 4. (Note that you don't need to specify a value of k to carry out these rules.) Have you identified a minimum vertex cover of G?

FIXME graph 9

Exercise 20: Let G be the graph shown below.

(1) Using $k = 6$, apply the preprocessing rules in the order give: Rule 1, then 2, etc. Have you identified a vertex cover of size k or less? What is the next step?
(2) Using $k = 6$, apply preprocessing rule 5 first, then apply the other rules in order: rule 1, then 2, etc. Have you identified a vertex cover of size k or less? What is the next step?

FIXME graph here

Exercise 21: Here are two approximation algorithms (that is, algorithms that give us good, but not necessarily optimal solutions) for the VERTEX COVER problem.

- Algorithm A: Add to the vertex cover a vertex of maximum remaining degree; delete this vertex and all edges incident to it. Repeat these two steps until all edges have been deleted.
- Algorithm B: Choose an arbitrary edge and include both endvertices in the vertex cover. Delete these vertices and all edges incident to them. Repeat these two steps until all edges have been deleted.

(1) Use each of these algorithms on the graph of Exercise 1 at the beginning of this module. What were your results?
(2) Create a variety of different types of graphs and test these algorithms on them. Which algorithm seems to perform better, in general?

Exercise 22: Specify preprocessing rules for the k-Dominating Set problem that are applicable in the following cases.

(1) G has a vertex of degree 0
(2) G has a vertex of degree 1

9. Solving the Vertex Cover Problem

Let us review our overall strategy. We want to solve Vertex Cover on a graph G. We start by guessing a value of k and then seek to answer the k-Vertex Cover problem on G. The preprocessing rules allow us to reduce this problem to a smaller graph G' with a smaller parameter k'. We then attack this smaller problem by finding for a minimum vertex cover C' on G'. If it is of size less than or equal to the reduced parameter k', then we are done. Not only can we answer both decision problems with "yes", we can also 'lift' C" to a minimum vertex cover C on G. If not, then we have to start again with a new, larger value of k.

It would therefore seem strategic to start with a very large value of k. However, the larger the value of k, the smaller the set of vertices we remove when we apply Rule 5. This means the size of the kernelized graph G' is larger, which in turn means that the exponential-time search for a minimum vertex cover in G' will take longer. In fact, for each additional vertex, the amount of time doubles. Hence we do not want to start with *too* large a value for k.

What we need is at this point is a heuristic to pick good values of k. Obviously k must be non-negative, and it is bounded above by n, the total number of vertices in the graph. With a little work we can produce better bounds on k.

As discussed in section 4, Vertex Cover can be formulated as an integer programming (IP) problem. Although in general IP problems are intractable, we can solve the the LP relaxation using efficient LP algorithms.

Exercise 23: (a) Explain why the optimal value z^* of the LP relaxation of Vertex Cover provides a lower bound on k^*. (b) Explain why this bound can be sharpened to $\lceil z^* \rceil$ (the smallest integer greater than or equal to z^*).

The LP relaxation can also be used to provide an upper bound. Let $\mathbf{x}^*$ denote the optimal solution to the LP relaxation, and recall that there is one decision variable per node. Let C consist of all

nodes i for which ${x_i}^* \geq 1/2$. Note that this corresponds to using conventional rounding on the components of $\mathbf{x}^*$ to create an integral vector $\mathbf{c}$; this technique does not always result in a feasible solution to the original IP, but in this case it does.

Exercise 24: Prove that C is a vertex cover. (*Hint.* For each $e = (i, j) \in E$, you must show that either $i \in C$ or $j \in C$.)

Since k^* is the minimum size of a vertex cover, clearly, $k^* \leq |C|$. An interesting side note is that vertex cover C is guaranteed to be at most twice the size of an optimal cover. This follows from the fact that the rounding process at most doubles the 'cost' of the optimal solution. More precisely, $c_i \leq 2{x_i}^*$ and so

$$|C| = \sum_{i=1}^{n} c_i \leq 2 \sum_{i=1}^{n} {x_i}^* = 2z^* \leq 2k^*.$$

Hence, $|C|/2$ is another lower bound on k^*, but from the equation above, obviously not as good a one as $\lceil z^* \rceil$.

We now know that we should choose a value of k between $\lceil z^* \rceil$ and $|C|$. The remaining step is to decide how to search within this range. We present three possibilities:

(1) Pick the upper bound as the first candidate. This will guarantee the optimal solution is found with a single "iteration." However, as mentioned above, the kernelized graph G' is potentially larger than necessary and this may result in an unreasonably long brute force search.
(2) Pick the lower bound as the first candidate, and use successively larger jumps until we find a valid k. One potential pattern is to use inverse powers of 2 as our jump distances.
(3) Pick the average of the upper and lower bound as the first candidate. If this value is too small, successively increase by half the distance to the upper bound. This approach is similar to a binary search except that we never "branching" down, because guessing too large produces the solution.

Exercise 25: There is another efficient method for generating a vertex cover C for $G = (V, E)$ which is at most twice the optimal size. (This method, like the one based on IP presented above, is called a *2-approximation algorithm* to Vertex Cover.) In this technique, we randomly select an edge $e \in E$ and add both of its endvertices to C. Next, we remove from G all edges incident to these vertices. We then repeat the process on the resulting graph and continue until there are no remaining edges.

(1) Explain why that the final set C will be a vertex cover.
(2) Give an example of a graph where the size of C is:
    (a) exactly twice optimal.
    (b) the same size as the optimal.

10. Programming Assignments

The material in this module can be broken down into a number of programming assignments. In the following sections, we present four suggested activities. In each case, a sample solution is provided.

10.1. **Write a Graph Class.** The most common method of representing a graph in memory is by using an *adjacency list*. For each vertex, we maintain a list of the other vertices to which it is adjacent. Typically, this is stored as an array of linked lists.

adj.pdf

For example, in the graph above, vertex 3 is adjacent to 1, 2, and 5. These three values are all contained in the third list of the array. Notice that the edge $(1, 3)$, it is represented in *both* 1's list and 3's list. This is important when we delete edges and vertices.

To implement some of the algorithms we have been discussing in relation to Vertex Cover, we need to focus on how to work with the adjacency list of a graph to:

- add a vertex to the graph;
- add an edge to the graph;
- remove an edge from the graph;
- remove a vertex (and all edges incident to that vertex);
- determine the degree of a vertex.

If you have studied the STL or understand the concept of iterators, you are strongly encouraged to include iterators over both vertices and edges.

10.2. **Brute Force Vertex Cover.** Recall that our basic strategy is to use the five kernelization rules to produce a much smaller graph G', on which we can reasonably use a brute force approach to find an optimal vertex cover. An algorithm to accomplish this can be broken down into three sections - generating subsets, testing to see if they are vertex covers, and finding the smallest size of a subset that is a vertex cover. The last two parts are fairly straightforward. To test, you simply iterate over the edges of the graph and make sure that at least one endvertex of each edge is in the subset. Once a first vertex cover is found, record its size in an integer variable. After each subsequent vertex cover is found, compare its size to the recorded variable and change it as necessary.

Generating subsets represents a more significant challenge. The most obvious approach is to exploit the same one-to-one mapping between binary strings of length n and the n vertices that we used in the IP formulation of Vertex Cover. Specifically, if we number the vertices $1, 2, \ldots, n$, then we can use the i^{th} bit of the string as a Boolean denoting whether or not vertex i is in the set. By treating the string as a single unsigned integer, we can represent all possible subsets by counting from 0 to 2^{n-1}.

[*****QUESTIONS: Are more details going to be added here? Is the assignment to actually produce the algorithms - the code? Or pseudocode?**********]

10.3. **Graph Kernelization.** To implement the five kernelization rules, we take an iterative approach. At each stage, we must keep a a running record of the set of vertices S that we can assume are in an optimal cover, the reduced graph G', the size n' of G' and the integer k'. The most straightforward approach is to first sort the vertices of the current graph in order of increasing degree. Consider the first vertex in this list. If it has degree 0, then apply Rule 1, update G',

n' and k' accordingly and start again. If it has degree 1, then apply Rule 2, update G', n' and k' accordingly and start again. If it has degree 2, first determine which of Rule 3 or Rule 4 is appropriate, then apply it, update the variables and start again. If the minimum vertex degree is 3 or greater, then go to the opposite end of the list and consider the vertex of maximum degree. If it has degree greater than the current value of k', then apply Rule 5.

[*********QUESTION: Is this right? Or does it make sense to first delete ALL vertices of suitably large degree in one fell swoop, as opposed to one at a time? And does it make sense to do the rules in order, as I've outlined above, or should Rule 5 be applied first? Did you actually implement this, Ben? I have no empirical experience with this, but you might.*******]

Repeat this process until no rules apply during a complete iteration over the vertices to guarantee you have finished. At this point, we apply the brute force approach developed above to the ultimate kernel G'. The final step is to 'lift' the solution to the original graph G by combining the optimal cover S' of G' with the set S. It is important in this step to have kept track of the "supervertices" generated by Rule 4; they need to be 'unconsolidated' to recover the nodes in G.

10.4. **Vertex Cover: 2-Approximation.** In section 1, we discussed how the LP relaxation of the IP formulation can be used to produce a vertex cover that is at worst twice as large as optimal. Similar to the brute force discussion above, write a program to translate the instance of a graph to the format required by an LP solver. Execute the solver, and then round the solution to obtain a vertex cover. Alternatively, implement the approximation scheme mentioned in section 2.

10.5. **Brute Force Dominating Set.** Implement an algorithm that solves the Dominating Set problem using a brute force approach.

11. Application to Phylogenetic Trees

An important application of Vertex Cover lies in the study of evolutionary relationships among species. By comparing the presence or absence of certain heritable traits or *characters* among various species, biologists attempt to deduce which ones have common ancestors and thereby construct what is known as a *phylogenetic tree*. A problem is that often real-world data present conflicting evidence; one solution is to pare down the data as little as necessary to obtain a consistent picture. In this section, we examine this application in more detail.

We begin with a set $\mathcal{S}$ of m species and a set $\mathcal{C}$ of n characters. We assume that *(a)* the common ancestor of all species in $\mathcal{S}$ exhibits none of the characters, and *(b)* once a trait emerges, it is inherited by all subsequent species in the phylogenetic tree (*i.e.* once a character changes from '0' to '1', it cannot change back again). This means that all species exibiting a certain character will have a common ancestor. We encode observed data in an associated $m \times n$ binary matrix $M = M(\mathcal{S}, \mathcal{C})$ by setting $m_{ij} = 1$ if species s_i exhibits character c_j; otherwise $m_{ij} = 0$. Intuitively, we say that M is a *perfect phylogeny* if we can construct an evolutionary tree based on the information in M that adheres to our biological assumptions. We can put this in more precise, graph theoretical terms. Formally, a tree is a connected graph with no cycles. A *rooted* tree has one vertex designated as the *root*, usually drawn at the top. Terms such as *parent, ancestor, child, descendant* have the obvious meanings. A rooted tree is *binary* if it has at most two children per vertex. A *leaf* is a vertex with no children.

Definition 4. *A binary rooted tree $T = (V, E)$ is a* perfect phylogenetic tree *of $M(\mathcal{S}, \mathcal{C})$ if*

(1) *there is a one-to-one correspondence between the species $\mathcal{S}$ and the leaves of T;*
(2) *each character $c \in \mathcal{C}$ labels exactly one edge in T (but there may be unlabeled edges in T);*
(3) *for each species, the unique path from the root to the corresponding leaf contains an edge corresponding to each character exhibited by the species.*

The figure below gives a perfect phylogenetic tree for $M_0 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$.

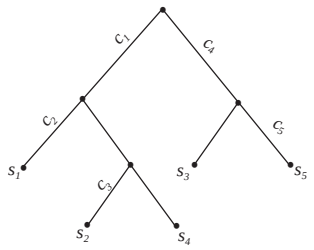


FIGURE 1. A perfect phylogenetic tree for M_0.

Exercise 26: If $M(\mathcal{S}, \mathcal{C})$ has a perfect phylogenetic tree, is it unique? Provide either a proof or a counterexample to support your answer.

Not all binary matrices admit perfect phylogenetic trees. Consider the following matrix.

$$M_c = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Any tree representing M_c must have the property that the unique path from the root to leaf s_1 contains both edges c_1 and c_2. If c_1 occurs before c_2 on this path (that is, c_1 emerged before c_2 in the evolutionary time scale), then any species exhibiting c_2 must also exhibit c_1; this is contradicted by species s_2. Similarly, species s_3 contradicts the possibility that c_2 occurs after c_1.

This examples represents in simplest form what can go wrong with the data. For any character c_j, let S_j denote the set of species exhibiting c_j. A Venn diagram illustrating the situation in the preceding example is given in Figure 1; note that neither S_1 nor S_2 is contained in the other, nor are they disjoint. For perfect phlyogeny, we need to avoid this situation.

Definition 5. *A family of subsets $\mathcal{A}$ of a given universal set U is* laminar *if for all $A, B \in \mathcal{A}$, either $A \cap B = \emptyset$ or $A \subseteq B$ or $B \subseteq A$.*

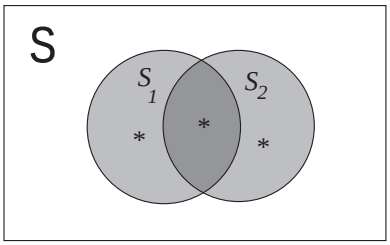


FIGURE 2. A small non-laminar family.

Exercise 27: Let A be a maximal member of a laminar family $\mathcal{A}$; that is, A is not contained in any other member of $\mathcal{A}$. Let $\mathcal{A}_A = \{B \in \mathcal{A} \mid B \subseteq A\}$. Prove that $\mathcal{A}_A$ and its complement in $\mathcal{A}$ are laminar families.

Theorem 3. *$M = M(\mathcal{S}, \mathcal{C})$ is a perfect phylogeny if and only if $\{S_j \mid c_j \in \mathcal{C}\}$ is a laminar family.*

Proof. If $\{S_j \mid c_j \in \mathcal{C}\}$ is not laminar, then we run into the contradiction mentioned above. Conversely, assume this family is laminar; we will give an iterative process for constructing a corresponding phylogenetic tree. Begin with a root vertex. Identify a maximal set S_i in the family; add two edges to the root, one labeled c_i. The vertex at the end of this edge, v_i, represents the common ancestor of all species exhibiting character c_i. If S_i is also a minimal set in the family, then we branch down from v_i in a binary manner, with a leaf for each species in S_i as shown in Figure 2. (If S_i consists of a single species, then we need simply relabel v_i with this species.)

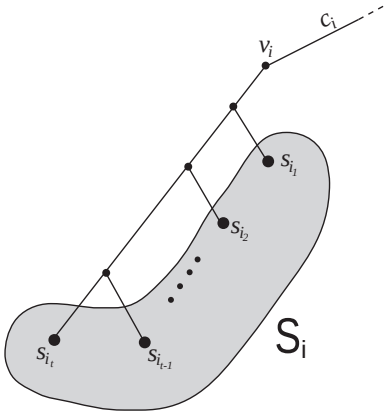


FIGURE 3. Binary subtree representing S_i.

If S_i is not minimal, then by the exercise, $\{S_j \mid S_j \subset S_i\}$ is a laminar family and we can repeat the process above with v_i as the root of the subtree corresponding to all species exhibiting character c_i. Additionally, $\{S_j \mid S_j \cap S_i = \emptyset\}$ is a laminar family and we can repeat the process above with w_i as the root of the subtree corresponding to species not exhibiting character c_i. We leave it to the reader to show that ultimately we will have a perfect phylogenetic tree of M. □

Exercise 28: Prove that a binary matrix M is a perfect phylogeny if and only if no two columns of M contain rows with all three patterns (1,1), (1, 0) and (0, 1)) (*i.e.* M does not contain some row-permutation of M_c as a submatrix).

If a species-character matrix is not a perfect phylogeny, biologists attempt to construct a phylogenetic tree that is consistent with as much of the data as possible by eliminating columns of the matrix (*i.e.* character data) which present conflicting evidence.

Definition 6. *The* conflict graph $G = (V, E)$ *associated with* $M = M(\mathcal{S}, \mathcal{C})$ *has* $V = \mathcal{C}$ *and* $e = \{c_i, c_j\} \in E$ *if and only if the corresponding columns in* M *display the patterns (1,1), (1, 0) and (0, 1).*

Note that M is a perfect phylogeny if and only if its conflict graph has no edges.

Exercise 29: Let U be a vertex cover in the conflict graph of matrix $M(\mathcal{S}, \mathcal{C})$. Prove that if $\mathcal{C}' = \mathcal{C} \backslash U$, then $M' = M(\mathcal{S}, \mathcal{C}')$ is a perfect phylogeny.

Exercise 30: By eliminating a minimum set of characters, find a phylogenetic tree consistent with a maximum amount of data in the matrix below.

$$M = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

12. References

References

[1] F. Abu-Khzam, R. Collins, M. Fellows, M. Langston, W. Suters C. Symons *Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments*, extended abstract. http://www.cs.utk.edu/ rcollins/papers/ALENIX.html

[2] J.Chen, I. Kanj, W. Jia *Vertex Cover: Further Observations and Further Improvements*, Journal of Algorithms **41** (2001), 280–301.

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

[4] G.B. Dantzig, *The Diet Problem*, Interfaces **20:4** (1990) 43-47.

[5] M. R. Fellows, *Parametrized Complexity: The Main Ideas and Connections to Practical Computing* Experimental Algorithmics (R. Fleischer et al., eds.), LCNS 2547, 51–77, Springer-Verlag, Berlin Heidelberg, 2002.

[6] D. Gusfield, S Eddhu, C. Langley, *Efficient Reconstruction of Phylogenetic Networks with Constrained Recombination*, Proceedings of the Computational Systems Bioinformatics, (CSAB '03).

[7] F. S. Hillier, G. J. Lieberman, *Introduction to Operations Research*, seventh edition, McGraw-Hill, New York, 2001.

[8] R. R. Hudson, *Generating Samples under a Wright-Fisher neutral model of genetic variation*, Bioinformatics 18 (2002) 337-338.

[9] B. Kolman, R. E. Beck, *Elementary Linear Programming with Applications*, second edition, Academic Press, San Diego, 1995.

[10] F. S. Roberts, *Graph Theory and Its Applications to Problems of Society*, Society for Industrial and Applied Mathematics, Philadelphia, 1978.

[11] D. B. West *Introduction to Graph Theory*, second edition, Prentice-Hall, Upper Saddle River, 2001.

[12] R. J. Wilson, J. J. Watkins, *Graphs An Introductory Approach*, John Wiley and Sons, Inc., New York 1990. www.mcdonalds.com