DIMACS EDUCATIONAL MODULE SERIES

MODULE 06-1
Some problems are NP-harder than others

Date Prepared: May 26, 2006

Sally Cockburn
Hamilton College
Clinton, NY 13323
scockbur@hamilton.edu

Ben Coleman
Moravian College
Bethlehem, PA 18018
coleman@cs.moravian.edu

R. Bruce Mattingly
SUNY Cortland
Cortland, NY 13323
mattinglyb@cortland.edu

Kay Somers
Moravian College
Bethlehem, PA 18018
mekbs01@moravian.edu

Module Description Information

- **Title:**
  Some problems are NP-harder than others

- **Authors:**
  Sally Cockburn, Hamilton College
  Ben Coleman, Moravian College
  R. Bruce Mattingly, SUNY Cortland
  Kay Somers, Moravian College

- **Abstract:**
  This module discusses two problems from graph theory that have important applications - the Vertex Cover problem and the Dominating Set problem. While the problems appear to be similar, they differ in the amount of time needed to find a solution. Although both problems seem to require exponential time relative to the size of the graph, the Vertex Cover problem has special properties that allow it to be solved faster than the Dominating Set problem. We introduce an integer programming representation for each problem and discuss some methods to solve integer programs. To illustrate the difference between the Vertex Cover and Dominating Set problems, we present rules for preprocessing the Vertex Cover problem that effectively reduce its complexity.

- **Informal Description:**
  This module introduces students to the Vertex Cover problem and the Dominating Set problem to illustrate the use of approximation algorithms and heuristics on problems that are computationally difficult. To provide flexibility, the module includes introductory material on graph theory, computational complexity, linear programming and integer programming. The background of the students using the module will help determine which portions of the module will be emphasized.

  Sections 1-6 comprise the core of the module and would normally be covered by all students. Section 1 introduces basic definitions from graph theory and complexity of algorithms that are needed throughout the module. Section 2 presents the Vertex Cover and Dominating Set problem, and Section 3 discusses simple solution techniques. Section 4 introduces the use of linear programming and integer programming techniques to solve the two problems. Students who have taken a course in linear programming may cover this section quickly or skip it altogether. This section includes some small problems that may be solved by hand. Students with access to optimization software can solve larger instances of these problems, but this is not required. Section 5 introduces several preprocessing rules that may be used to reduce the time required to solve the Vertex Cover problem. While these rules are most useful for large problem instances, the exercises included are on smaller problems so that the concepts can be illustrated clearly and so that students without background in computing can solve them. Section 6 discusses the use of the preprocessing heuristics to solve the Vertex Cover problem, and presents a more theoretical discussion of computational complexity, including the concept of fixed-parameter tractability. Section 7 is an optional section containing some programming exercises. Section 8 is also optional. While the introduction includes some simple application problems, this section describes a more substantial application from bioinformatics on phylogenetic trees. Finally, Section 9 includes a few additional exercises that may be assigned at the discretion of the instructor.

- **Target Audience:**
  This module is intended for upper division undergraduate students in mathematics or computer science. It may be used as a supplement for courses such as graph theory, linear programming, or analysis of algorithms, or as the basis for an independent study course.

- **Prerequisite:**
  The prerequisite for this module is an introductory course in discrete mathematics. Section 7, which is optional, requires knowledge of elementary computer programming.

- **Mathematical Field:**
  Graph Theory, Combinatorial Optimization, Computational Complexity

- **Applications Areas:**
  An optional section discusses an application to phylogenetic trees.

- **Mathematics Subject Classification:**
  Primary: 05C69, 90C57, 90C60
  Secondary: 68Q15

- **Contact Information:**

  Sally Cockburn
  Hamilton College
  Clinton, NY 13323
  email: scockbur@hamilton.edu

  Ben Coleman
  Moravian College
  Bethlehem, PA 18018
  email: coleman@cs.moravian.edu

  R. Bruce Mattingly
  SUNY Cortland
  Cortland, NY 13045
  email: mattinglyb@cortland.edu

  Kay Somers
  Moravian College
  Bethlehem, PA 18018
  email: mekbs01@moravian.edu

- **Other DIMACS modules related to this module:**
  03-5: Communications Network Design

Contents

Some problems are NP-harder than others

Abstract

This module discusses two problems from graph theory that have important applications - the Vertex Cover problem and the Dominating Set problem. While the problems appear to be similar, they differ in the amount of time needed to find a solution. Although both problems seem to require exponential time relative to the size of the graph, the Vertex Cover problem has special properties that allow it to be solved faster than the Dominating Set problem. We introduce an integer programming representation for each problem and discuss some methods to solve integer programs. To illustrate the difference between the Vertex Cover and Dominating Set problems, we present rules for preprocessing the Vertex Cover problem that effectively reduce its complexity.

1 Introduction

Computational problems may be categorized by the amount of computing time needed to find a solution. Identifying the complexity class of a problem can help direct our search for a "best" algorithm for the problem. In this module we will investigate some ideas related to computational complexity using two graph theory problems: the Vertex Cover problem and the Dominating Set problem.

A graph is a pair of sets, $G = (V, E)$, where V is a finite set of *vertices* (or *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. The *size*, n, of a graph is the number of vertices in the graph, and is denoted $|V|$. For example, suppose $V = \{u, v, w, x, y, z\}$ and $E = \{(u, v), (u, w), (w, x), (v, x), (v, w), (v, y), (v, z)\}$. Then the graph $G = (V, E)$ has size $n = 6$ and can be drawn as shown in Figure 1.

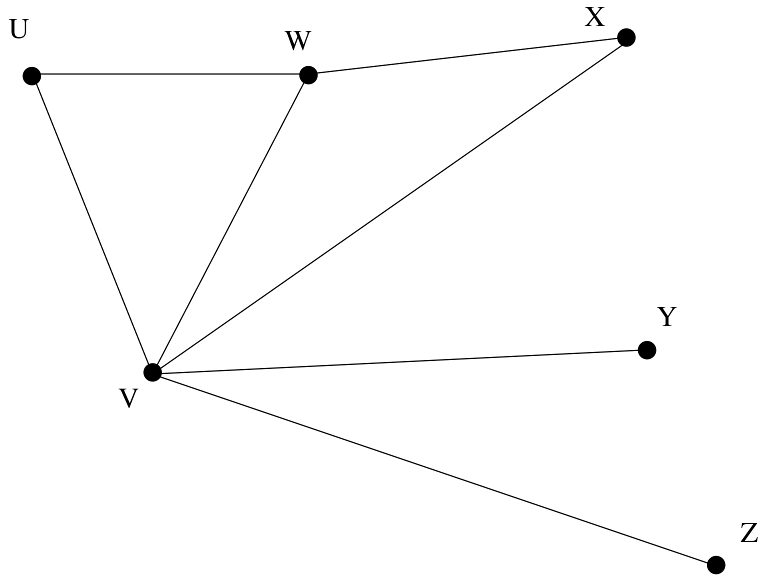


Figure 1: Graph of Size 6

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. The graph could be used to schedule project teams. As a final example, vertices could represent courses 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 to minimize conflicts.

The *complexity* of an algorithm to solve a particular model or problem is a rough measure of the maximum number of elementary computations required to obtain a solution. The field of *computational complexity* categorizes problems into classes based on the type of mathematical function that describes the best algorithm for each problem. To make these measurements, the functions are written in terms of n, the size of the problem. For Vertex Cover and Dominating Set, the size of the problem is simply the size of the graph.

The type of function used to describe the complexity of an algorithm is very important. The two most commonly used classes of functions are *polynomial* and *exponential* functions. Table 1 compares execution times for various polynomial functions with n in the base and exponential functions with n in the exponent. The values assume that one step of an algorithm can be completed in one microsecond. As n grows, the differences between the two types of functions become apparent. Exponential functions grow so fast that it is not practical to consider problems above size 40.

Complexity	**Input size n**					
Function	10	20	30	40	50	60
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec	.216 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13 min
2^n	.001 sec	1 sec	17.9 min	12.7 days	35.7 yrs	36600 yrs
3^n	.059 sec	58 min	6.5 yrs	385500 yrs	2×10^{10} yrs	1.3×10^{15} yrs

Table 1: Execution time compared to problem size for various functions

In this module, we will look at two problems involving graphs. Although the best-known solution time for each is an exponential function of the size of the graph, this mathematical categorization does not tell the whole story. There are clever tricks for one of the problems that enable us to solve it relatively efficiently, while no similar techniques are known for the other. This implies the existence of a subcategory within the exponential complexity class.

2 Vertex Cover and Dominating Set

A *vertex cover* of a graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that for each edge $(u, v) \in E$, either $u \in C$ or $v \in C$. The Vertex Cover problem is to find the size (number of vertices) of a minimum 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 minimum dominating set; the k-Dominating Set problem asks whether there is a dominating set of size (less than or equal to) k.

The Vertex Cover and Dominating Set problems occur in many real-world applications:

- 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.
- 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 form a representative committee of minimum size.
- Assume 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.

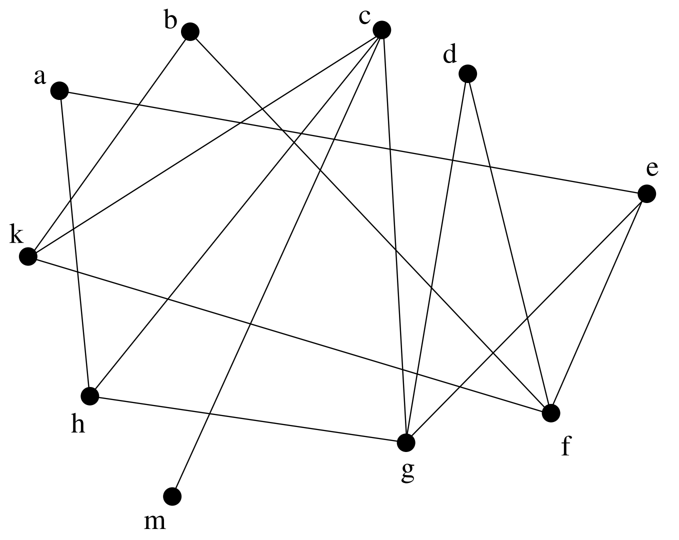


Figure 2: Graph for Exercises 1 and 2

Exercise 1. Let G be the graph given in Figure 2. 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.

1. $S = \{a, b, c, e, g, f\}$
2. $S = \{a, c, g, f, k\}$
3. $S = \{b, c, d, e, f, g, k\}$

Exercise 2. Let G be the graph given in Figure 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.

1. $S = \{c, d, e, k\}$
2. $S = \{a, c, f\}$
3. $S = \{a, b, e, h, k\}$

Exercise 3. 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 4. 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 (1) is false.

Exercise 5. Explain why the "art gallery guard" problem is an application of the Vertex Cover problem.

Exercise 6. Explain why the "representative committee" problem is an application of the Dominating Set problem.

Exercise 7. Explain which problem models the "nuclear power plant guard" problem and why.

3 Simple Techniques to Solve the Two Problems

To solve either the Vertex Cover or the Dominating Set problem, 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 subsets, determine which subsets represent vertex covers (or dominating sets), and from those, choose a subset of minimum order. In practice this is rarely practical.

Exercise 8. For the graph in Figure 1, list all subsets of V of size 3. How many subsets of V are there of size 3? Answer the equivalent question for all possible sizes of subsets, and place your answers in the following table:

Number of Vertices in subset	0	1	2	3	4	5	6
Total number of subsets							

Exercise 9. Notice that the sum of your answers in the table in the previous exercise is $64 = 2^6$. In general, a set of n elements has 2^n possible subsets. Given a set of vertices V, how many subsets of V are there if

1. $|V| = 20$?
2. $|V| = 50$?
3. $|V| = 100$?

Exercise 10. Suppose that a computer takes one microsecond (10^{-6} seconds) to consider a single subset. How long would it take to consider all subsets if

1. $|V| = 20$?
2. $|V| = 50$?
3. $|V| = 100$?

We can see that it is not practical to consider all subsets of the vertex set of a graph to find a minimum vertex cover or dominating set. Although there are a number of alternate techniques to solve these problems, these approaches retain the exponential factor 2^n for the number of operations required to solve them. Consequently, these problems are not practical to solve for large graphs, even by the fastest computers.

These two problems belong to the class of problems for which the best algorithms are exponential-time algorithms. When compared to problems with polynomial-time algorithms, we see a natural division in the world of combinatorial problems between those that are easy and those that are hard. However, we cannot simply give up on "hard" problems such as Vertex Cover and Dominating Set. They crop up in the real world in many different fields, such as telecommunications and computational biology. Even if we cannot find the best solution, it is important to get some kind of answer.

An *approximation algorithm* is a sequence of steps that produces a feasible (valid) solution to a problem in polynomial time. This solution is not guaranteed to be optimal, but it will be a valid and often good solution. For example, one (naive) approximation algorithm for the Vertex Cover problem is to number the vertices arbitrarily and then include sequentially higher numbers until the set of chosen vertices (numbers) forms a cover. The graph in Figure 3 demonstrates an instance where the approximation algorithm finds the optimal cover, $\{1, 2\}$. On the other hand, Figure 4 shows a graph where the algorithm takes all the vertices except 10 when the optimal cover is $\{9, 10\}$. The quality of the solution depends on the graph and how it is numbered.

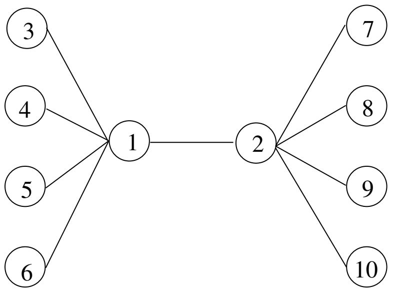


Figure 3: Graph where the approximation is optimal

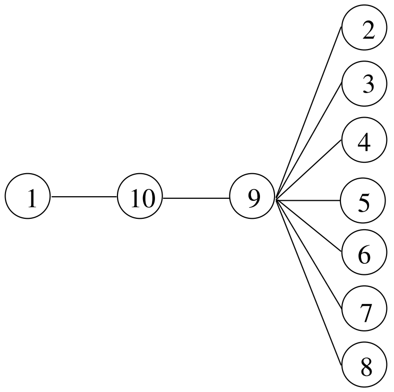


Figure 4: Graph where the approximation is poor

In the study of approximation algorithms, we want to find an algorithm that produces "good" solutions most of the time. Typically, we measure this as a ratio between the quality of the approximation to the quality of the optimal solution. For example, in Figure 3, the approximation algorithm gave a cover of size 2, which is the optimal solution. Therefore the ratio is $\frac{2}{2} = 1$. In Figure 4, the ratio is $\frac{9}{2} = 4.5$ because the algorithm gave a solution of size 9 while the optimal solution has size 2.

Exercise 11. Explain why the best ratio for the approximation algorithm described above is 1 and the worst case is $\frac{n-1}{2}$ for a graph with n vertices.

Exercise 12. Here are two approximation algorithms 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 vertices in the vertex cover. Delete these vertices and all edges incident to them. Repeat these two steps until all edges have been deleted.

Use each of these algorithms on the graph in Figure 1. What were your results?

Exercise 13. What is the worst performance ratio you can produce for a graph of n vertices for each of the approximation algorithms presented? What does this imply about each algorithm?

4 Advanced Techniques

One problem-solving strategy in mathematics involves rewriting a problem as a special case of a more general problem type. Then if we know how to solve the more general problem, we can also solve the special case. We will formulate the Vertex Cover and Dominating Set problems as *integer programming* (IP) problems. One advantage to this approach is that much research has focused on a search for sophisticated algorithms and solution methods for these problems. In fact, in recent years, much progress has been made in finding methods to handle large IP problems with special structures. Unfortunately, however, even the best algorithms still have exponential worst-case running times.

4.1 Integer programming (IP) Formulations

The mathematical model for integer programming problems is similar to that for linear programming (LP) problems: our objective is to maximize a quantity (like profit) or minimize a quantity (like cost or time) subject to a set of constraints expressed as linear inequalities. The objective function (the quantity we are maximizing or minimizing) and the constraints are linear functions of a set of *decision variables*. *Integer programming* (IP) problems have the restriction that the decision variables must be integers. *Binary integer programming* (BIP) problems have an added restriction that the decision variables must have integer values of either 0 or 1. (Situations in which the decisions are essentially yes-or-no often can be modeled by BIPs.) We will formulate the Vertex Cover and Dominating Set problems as BIPs.

Suppose a graph $G = (V, E)$ has n vertices, numbered $1, 2, 3, \ldots, n$. The BIP formulation of the Vertex Cover problem has n decision variables, x_i for $i = 1 \ldots n$. In a solution, $x_i = 0$ if vertex i is not in the cover and $x_i = 1$ if vertex i is in the cover. We can formulate the problem as:

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 a BIP using the same decision variables:

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 14. 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.

4.2 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? What's 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 Dantzig in the 1940's 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 we are going to eat only two foods: six-piece orders of McDonald's chicken nuggets (McNuggets) and small orders of McDonald's french fries. We want to maximize the protein we 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. We 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:

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

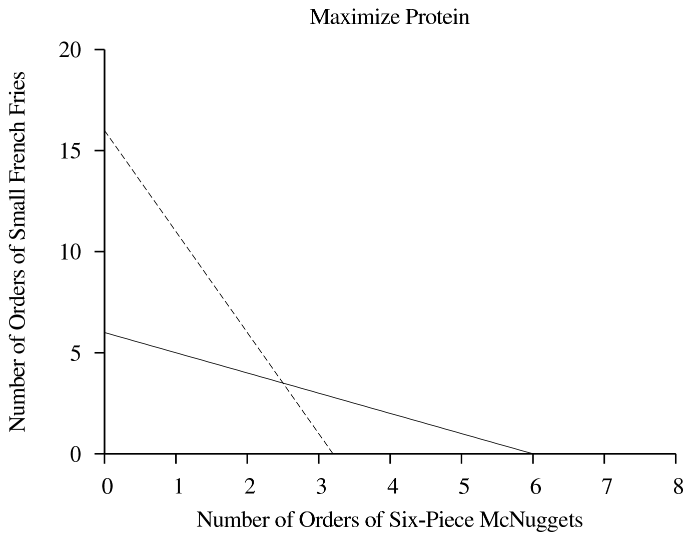


Figure 5: Feasible Region for McDonald's Example

We'll first look at the feasible solution space on the graph in Figure 5: it's 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 15. 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 16. 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 17. 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 18. 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?

Linear programming problems are generally much easier to solve than IP problems. It is almost counterintuitive that a problem with many more feasible solutions should be easier to solve (like an LP problem compared to the same problem with integer restrictions on the variables). But it is because there are all the additional feasible solutions to an LP that there will be a corner point feasible solution that is optimal. This guarantee allows linear programs to be solved much more efficiently. There is no such guarantee for an IP algorithm.

4.3 Branch and Bound

The McDonald's example suggests one way to solve an integer programming (IP) problem: ignore the fact that the decision variables are integers, and simply solve the problem as if it is a linear programming (LP) problem using a standard technique such as the simplex method. This related problem is called the *LP-relaxation*. If we are very lucky, the solution to the LP relaxation will have integer values for all of the variables. In this case, we have also solved the IP.

In general, the solution to the LP-relaxation provides us with a bound on the value of our objective function in the IP. This solution can be used as a starting point for solving IP problems. From here, we can continue to search for an integer solution using what is called the *branch and bound* method. We will illustrate this method with the BIP Vertex Cover problem. If we solve the LP-relaxation of a BIP and do not obtain an answer in which the decision variables are all zero or one, the first step is to choose a variable whose value in the current optimal solution is non-integer; this is 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 integer, then we choose another branching variable and repeat the process.

To understand this idea more clearly, consider the graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4), (1,5)\}$. This graph is shown in Figure 6.

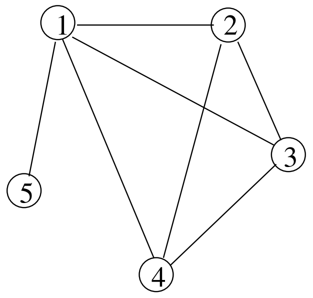


Figure 6: Branch and Bound Example Graph

We wish to find a vertex cover of minimum size by solving the following BIP.

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 constraints that the decision variables must be integers 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 integer 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 values in the optimal solution to LP1, we could choose any one of them to be a branching variable. We will 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 contains non-integers, 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; in fact, the solution to LP4 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:

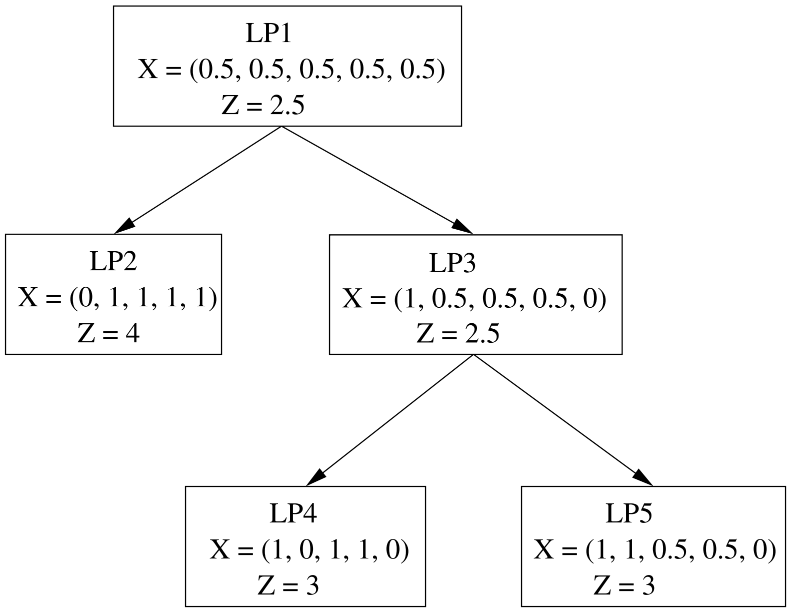


Figure 7: Summary of Branch and Bound IP example

Note that depending on the method that is used to solve the LP problems, alternative solutions may be found. For instance, $\mathbf{x} = (1, 0.5, 0.5, 0.5, 0)$ is an alternative optimal solution to LP1, and $\mathbf{x} = (1, 1, 1, 0, 0)$ is an alternative optimal solution to LP5.

Exercise 19. Let G be a graph with 5 vertices and an edge between every pair of vertices. (This is called a *complete graph*.) 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 20. Let $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{(1, 2), (2, 3), (3, 4), (1, 4), (5, 6), (6, 7), (7, 8), (8, 5), (3, 5)\}$. 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 21. Let G be the graph given in the previous question. Find a minimum dominating set from the graph. Formulate the problem as an IP and solve it using branch and bound.

The branch and bound technique is a cleverly structured enumeration procedure designed so that we only need to examine some of the feasible solutions. In general, however, we still are faced with exponential growth in computation time as a function of the problem size.

5 Preprocessing for the Vertex Cover Problem

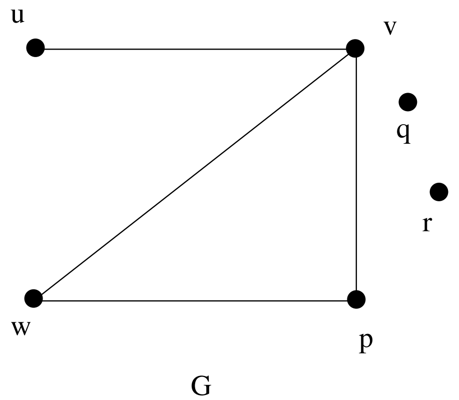
Although branch-and-bound techniques help us to "divide and conquer" large problems that are difficult to solve directly, they have not improved upon the worst-case exponential running times. We will describe and illustrate how a technique that involves *problem preprocessing* can be used on the vertex cover problem to help reduce the size of the problem. This will, in turn, have a beneficial effect on the complexity of the solution method.

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. (Recall that the k-Vertex Cover problem for a graph G asks for a "yes" or "no" answer to the question: Is there a vertex cover of size k?) 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.

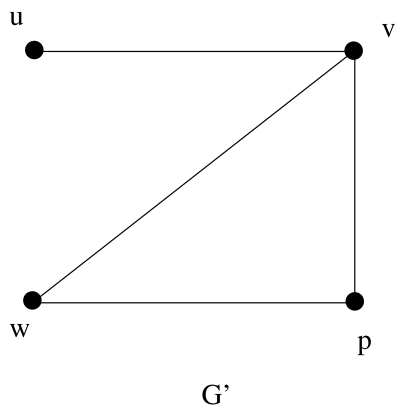
Each rule allows us to assume that certain vertices of the current graph either must be, or must not be, in the vertex cover we seek. We record those that must be included, then create a new, reduced graph. Not only will this graph be smaller in terms of number of vertices, the size of the cover we are checking for 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. (Note that 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.

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

- **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 $n' = 4$.

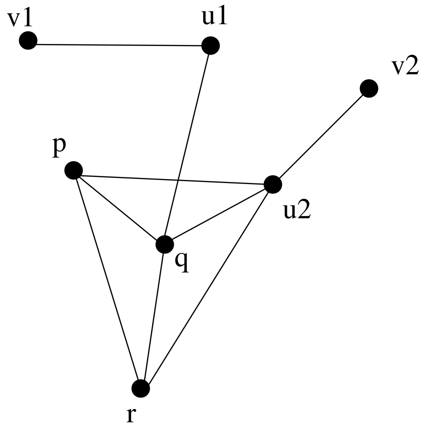


G

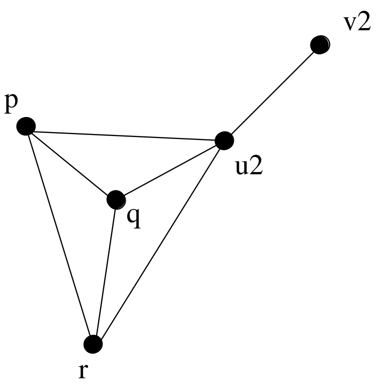


G'

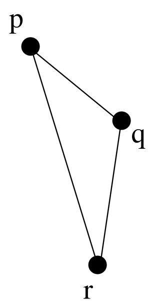
- **Rule 2**. If G has a vertex v of degree 1, then we can assume that any minimum vertex cover does not contain v, but does contain the one vertex u which is adjacent to v. To see this, note that any vertex cover must contain either u or v. Since v has degree 1, including v in a vertex cover will only 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.



G

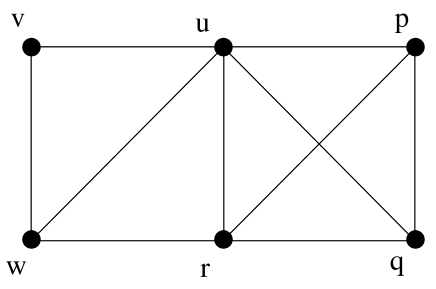


G'

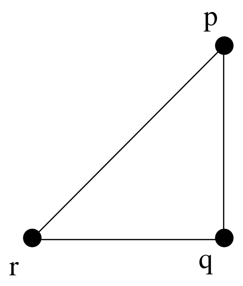


G''

- **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.



G

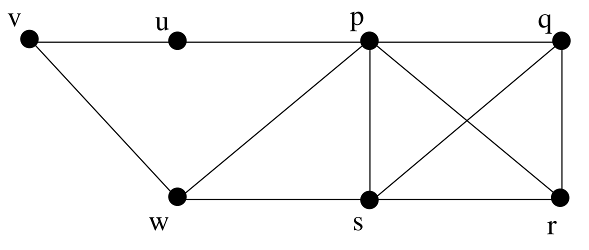


G'

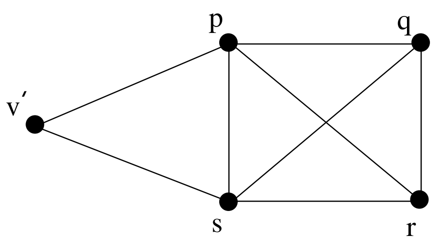
- **Rule 4**. Suppose G has a vertex v of degree 2 and its neighbors u and w are *not* adjacent. In this case, the reduced graph G' is obtained by replacing the three vertices u, v, and w with one "supervertex" v'; the neighbors of v' in G' are exactly the neighbors of u and w in G. Note that $n' = n - 2$. 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 modified G) with supervertex v' includes p, s, and either q or r (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.



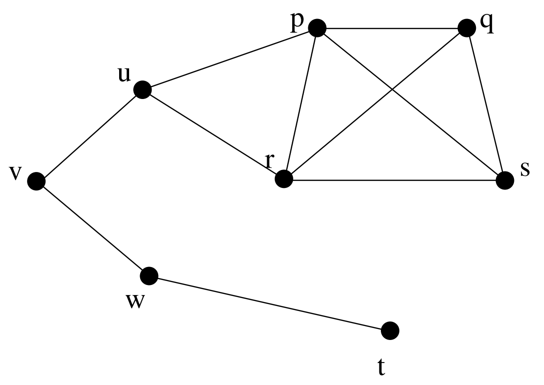
G



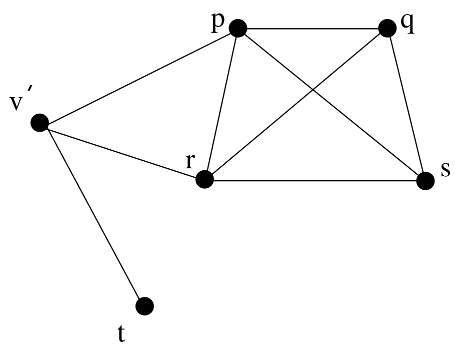
modified G

- **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.

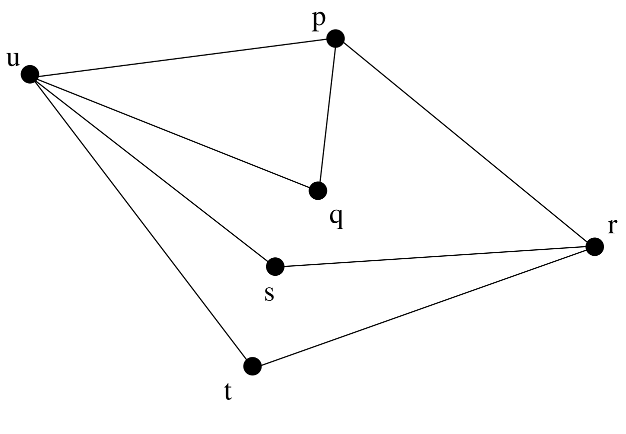


G

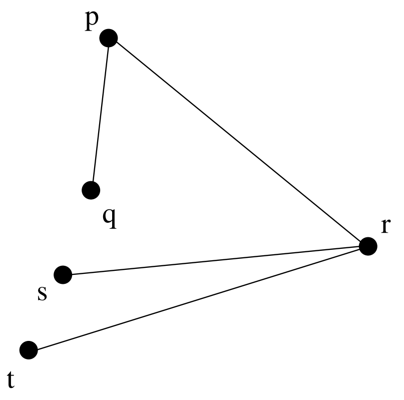


modified G

- **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).



G



G'

Exercise 22. 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 k-Vertex 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 the answer to 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 the k'-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 there is any hope that the answer to the k'-Vertex Cover problem on G' will be "yes".

Theorem 1. *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 vertex inside C' and the other vertex 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 straightforward 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.

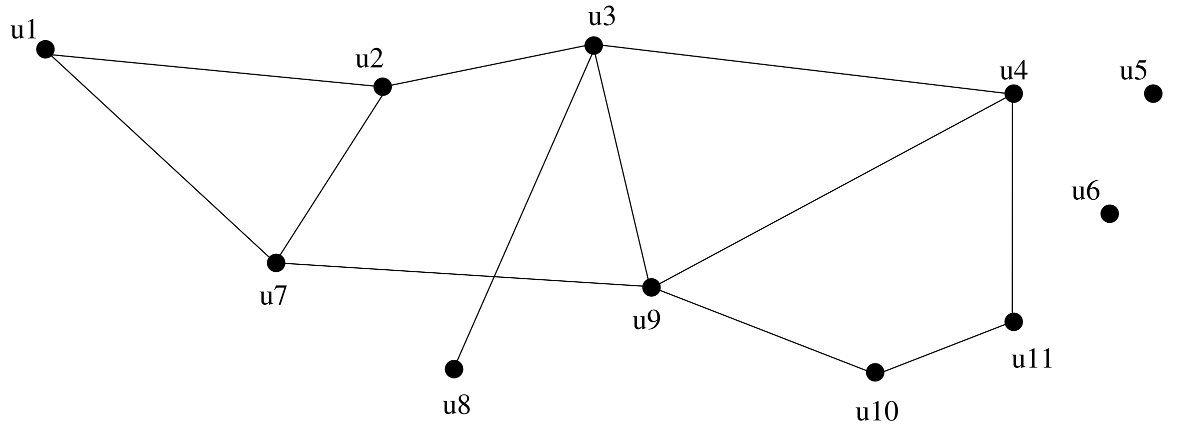


Figure 8: Graph for exercise 23

Exercise 23. Let G be the graph shown above. 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?

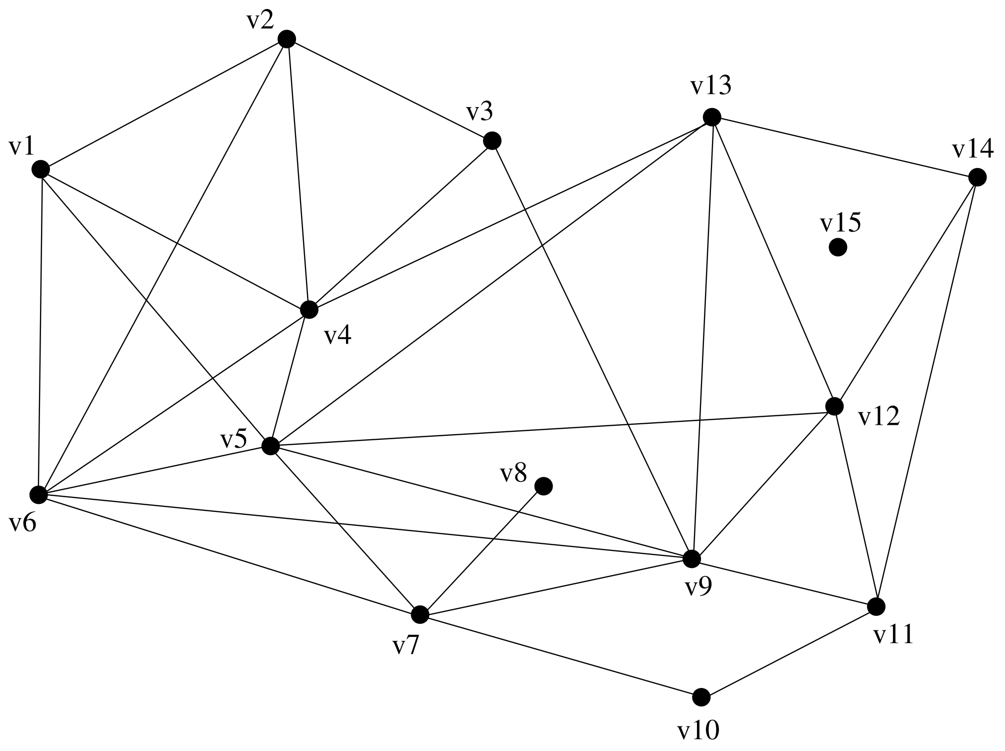


Figure 9: Graph for exercise 24

Exercise 24. Let G be the graph shown in Figure 9.

1. Using $k = 6$, apply the preprocessing rules in the order given: 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?

Exercise 25. 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

6 Solving the Vertex Cover Problem

We now turn to the issue of how to use the kernelization rules to find the optimal vertex cover of a graph. Recall that in the k-Vertex Cover problem, we are to determine if a graph has a vertex cover of size k or less. Therefore, the answer is "yes" or "no". For the more general vertex cover problem, we want to know the size of the optimal cover. The kernelization rules can be used to directly solve the k-Vertex Cover Problem. In this section, we will discuss how to utilize this result to solve the more general problem.

The application of the five kernelization rules gives us a set of vertices that must be in any vertex cover of size k or less, and a smaller graph where the degree of every vertex is greater than 2 but less than k. If we solve the Vertex Cover problem on this new graph, we can combine the solution to the new problem with the vertices removed by kernelization to obtain a valid cover, S, for the original graph. The size of S is an answer to the original k-Vertex Cover problem. If $|S| \leq k$, then the answer is "yes," and S is an optimal cover. However, if $|S| > k$, then we have a "no" instance, and the application of Rule 5 may have taken more vertices than necessary. Therefore, if the size of the cover exceeds k, we have failed to solve both the k-Vertex Cover problem and the Vertex Cover problem.

If we pick k large enough, we will be guaranteed a "yes"-answer to the k-vertex cover problem, and we will also have an optimal cover. However, for larger values of k, the size of the kernelized graph is larger. This means that the final, brute-force search for a vertex cover of the subgraph will take longer. In fact, for each additional node, the amount of time doubles.

Therefore, to solve an instance of Vertex Cover, we need a method to pick good values of k. For each value, we will use the kernelization technique to attempt to find an optimal cover. If the number of vertices is less than or equal to k, then we can stop. However, if the number of vertices exceeds k, we must choose a new value for k and begin again.

Obviously k must be non-negative, and bounded above by the number of vertices in the graph. However, we can produce a better bound on k by utilizing an *approximation algorithm.* Rather than attempting to find an exact solution to the Vertex Cover cover problem, we can utilize a *heuristic*, or guess, to produce a "good" solution. Typically, heuristics are rules that can be computed quickly and that produce solutions that are acceptably close to the optimal solution.

As discussed in Section 4, solving the IP formulation of the Vertex Cover problem as though it were a linear program produces a lower bound on the number of vertices in the cover. Further, if we round each of the decision variables, we obtain a set of vertices that is a cover. More formally, this cover is guaranteed to contain at most twice the number of nodes as the optimal cover. Therefore, using the IP relaxation technique, we have a better bound for k.

Exercise 26. Explain why the optimal value z^* of the LP relaxation of Vertex Cover provides a lower bound on k^*. Explain why this bound can be sharpened to the smallest integer greater than or equal to z^*, denoted $\lceil z^* \rceil$.

Given this tighter bound for k, the remaining step is to decide how to search within this range. We present three possibilities:

1. Pick the upper bound. This will guarantee the optimal solution is found with only a single "iteration." However, the kernelization of the graph will result in a (potentially) larger than necessary subgraph to solve.
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 middle value at each jump. This approach is similar to a binary search except that "branching" down is not possible because guessing too large produces the solution.

Exercise 27. Create a graph with at least 15 vertices and test each of the suggested methods for choosing k on your example.

Exercise 28. There is another efficient method for generating a vertex cover C for $G = (V, E)$. In this technique, we randomly select an edge $e \in E$ and add both of its vertices 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 exactly twice optimal.
3. Give an example of a graph where the size of C is the same size as the optimal.
4. Prove that this technique never gives a cover larger than twice the optimal size.

The big question is: what does preprocessing buy us in terms of computational complexity?

6.1 More on Computational Complexity

We have seen that 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 for 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. 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.

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. 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*. More formally, 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 red or blue so that no edge joins two vertices of the same color?

A decision problem is said to belong to the class NP if a "yes" answer can be quickly verified. The letters NP stand for *nondeterministic polynomial-time algorithm*; the "polynomial-time" refers only to how long it takes to check the answer. 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 vertex in this set.

Checking an answer is quite different from coming up with the answer. 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.

For some problems, there are much faster - in particular, polynomial-time - alternatives to the brute force approach. For example, a polynomial-time algorithm 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*. 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? The conjecture among experts is that P is probably a proper subset of NP, meaning that some problems require more than polynomial time.

The most important fact for our purposes is that some problems have been shown to be among the hardest of problems in NP. These are called *NP-hard* problems, and no one has ever found a algorithm solving them in polynomial time. Among the NP-hard problems are the Vertex Cover and Dominating Set problem. As we have seen, 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 know 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.

A recent approach to NP-hard problems has focused 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 definitions. 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*. 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 29. For the values of n and k listed in the following table, compute the ratio $\frac{n^{k+1}}{2^k n}$.

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

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.

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

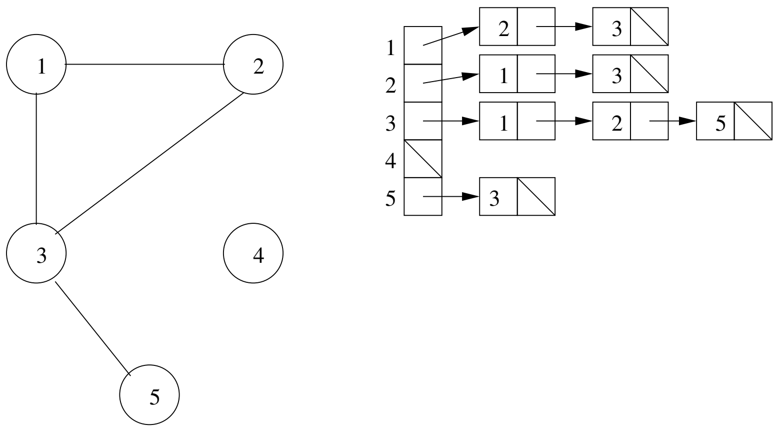
Hence, in applications where it can be assumed that the parameter is reasonably small, this class of NP-hard problems is for all practical purposes tractable.

7 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 that build upon each other to produce a complete solver for the Vertex Cover problem.

7.1 Implement a Graph Data Structure

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.



For example, in the graph shown 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.

Implement an adjacency representation of a graph and methods to perform the following operations on the graph:

- 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 understand the concept of iterators, you are strongly encouraged to include iterators over both vertices and edges.

7.2 Brute Force Vertex Cover

Even when we use the five rules to kernelize the graph, the resulting kernel must be solved using a direct method. In section 4, we presented an integer programming formulation for Vertex Cover. If you have access to an IP solver, implement the IP formulation described and write a program to convert its output to a format your graph implementation can utilize.

Alternatively, you can write a brute force algorithm to find the optimal cover to the kernel. This algorithm can be broken down into two sections, calculating subsets and testing subsets. The second part is fairly straightforward. You simply iterate over the edges of the graph and make sure that at least one vertex of each edge is in the subset.

Generating the subsets represents a more significant challenge. The most obvious approach is to recognize that there is a one-to-one mapping between binary strings of length n and a graph with n vertices. 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 an unsigned integer, we can represent all possible subsets by counting from 0 to 2^{n-1}.

7.3 Graph Kernelization

To kernelize the graph, we are really answering the question, "is there a cover of size k for graph G?" The answer is either "no," indicating that k vertices is not enough to have a cover, or a set S that actually contains the optimal cover. As we apply the five rules, certain vertices are deduced to be in any cover. These nodes must be added to S.

The most straightforward approach to implementing the rules is to attempt to apply each rule to each node in order. Repeat this process until no rules apply during a complete iteration over the vertices to guarantee you have finished.

When no more rules apply, we apply a brute force approach to the resulting kernel. The result of this function will be a set of vertices that are added to S. If the combined set of pre-processed vertices and those selected by the brute force algorithm has size less than or equal to k, then we have found a valid cover. Otherwise, the answer is "no."

Note that rule 4 is not resolved until *after* a complete cover is calculated. This rule should be implemented recursively. Be careful to handle each of the possibilities here. The recursive call will return the solution for the graph containing the super node.

7.4 Vertex Cover: 2-Approximation

In section 4, we discussed how the LP relaxation of the IP formulation produces a solution that is at worst twice as large as the optimal. Similar to the discussion of brute force above, write a program to translate the instance of a graph to the format required by an LP solver. Execute the solver, and then translate and round the solution to obtain an approximate cover. Alternatively, implement the one of the approximation schemes discussed in section 3.

8 Application to Phylogenetic Trees

An important application of the Vertex Cover problem 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 exhibiting 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.

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

$$M_0 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

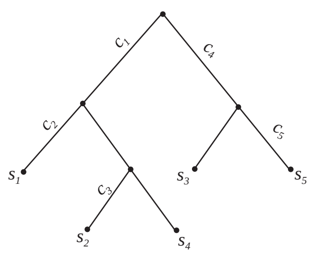


Figure 10: A perfect phylogenetic tree for M_0.

Exercise 30. 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 example 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 11; note that neither S_1 nor S_2 is contained in the other, nor are they disjoint. For perfect phylogeny, we need to avoid this situation.

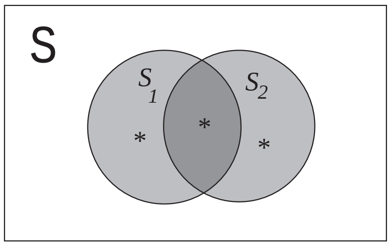


Figure 11: A small non-laminar family.

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 $B \subseteq A$.

Exercise 31. 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 12. (If S_i consists of a single species, then we need simply relabel v_i with this species.)

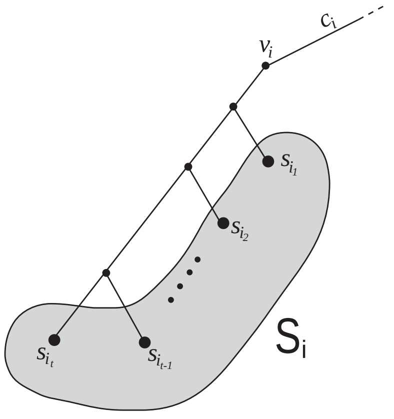


Figure 12: Binary subtree representing S_i.

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. $\square$

Exercise 32. 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.

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

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

9 Additional Problems

Exercise 35. 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 36. 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.

Exercise 37. 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.)

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*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, January 2004, ACM/SIAM, Proc. Applied Mathematics 115, L. Arge, G. Italiano and R. Sedgewick, eds.

[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 parallel 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, *Parameterized 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.

[13] www.mcdonalds.com