

Chapter 8

Approximation Algorithms

As we have seen, there are many important problems that are **NP**-Complete. In fact, these problems are too important to simply ignore: we need to develop methods to deal with these problems. Our main technique for doing so will be to study approximation algorithms: techniques for finding a solution that is guaranteed to be close to optimal. Before we do so, however, we shall briefly examine two other techniques: assuming a probability distribution over the input, and restricting the allowed inputs to special cases.

8.1 Other possible approaches

8.1.1 Restrict the input

One thing we can do is to restrict the input. In other words, instead of solving the problem in its full generality, we attempt to solve only certain classes of inputs. One such example is 2-SAT, which is solvable in polynomial time. We can think of 2-SAT as a restriction of SAT, the more general satisfiability problem.

There are a number of different possible restrictions that can be made on a graph. For example, we might restrict our attention to graphs that are acyclic, graphs that have bounded degree, or graphs that are planar (i.e., can be drawn on the plane in such a way that none of the edges of the graph intersect). It turns out that many **NP**-Complete problems can be solved fairly efficiently with these kinds of restrictions. For example, if we look at the CLIQUE problem, the CLIQUE problem turns out to be fairly easy for all of these three possible restrictions on the graph.

Another type of restriction that we might make is to assume that the integers that appear in the input are polynomial in the input size. This is actually something we've done for the knapsack problem. For that problem, we designed an algorithm that had a running time that was polynomial in the size of the input if we assumed that all of the integers that appear in the input, e.g. the knapsack capacity, were polynomial in the size of the input. So this does give us a polynomial-time algorithm of a restricted kind, but this restricted kind can be quite useful in practice.

Unfortunately many of these restrictions are not very useful. For example, it turns out that in many cases, the graph problems that we really want to solve do not adhere to the restrictions described above. However, the restriction on the integers in the input actually does end up being useful often. Perhaps for this reason, there is some terminology that goes along with this particular restriction, which we describe here briefly.

We have the following definitions:

Definition 83 *An algorithm runs in pseudo-polynomial time if the running time is polynomial in the input size and any integer in the input.*

Example 1: For the knapsack problem we have a pseudo-polynomial time algorithm (using dynamic programming) to solve it.

Example 2: The Subset-Sum problem is an **NP**-Complete problem. But it is also solved in pseudo-polynomial time by using a dynamic programming algorithm similar to the knapsack solutions.

It is not the case that all problems with integers in the input can be solved in pseudo-polynomial time. In particular, we have the following definition:

Definition 84 *A problem is Strongly NP-Complete if it remains NP-Complete even when all integers in an input of length n are polynomial in n .*

For example, let's say that we reduce 3-SAT to a problem Π . If all the integers that appear in the instance of Π are polynomial in the size of the original input to 3-SAT, then we have shown that Π is strongly **NP**-Complete. Note that this is not the case for the reduction to the subset sum problem that we saw earlier, as the integers that we produce could have size exponential in the size of the original problem.

Exercise: If a strongly NP-complete problem can be solved in pseudo-polynomial time, then $P=NP$.

One example of a strongly NP-complete problem is called the Bin-Packing problem.

Definition: The Bin-Pack problem (strongly NP-complete).

Input: A set of items of integer size, bin size B , and an integer K .

Question: can we partition the items into at most K sets (bins) such that no bin has total size larger than B (i.e., the size of each set $\leq B$)?

Another term that is not closely related, but sounds similar to pseudo-polynomial time is *quasi-polynomial* time. Pseudo-polynomial time is a description of what the running time is polynomial in. Quasi-polynomial time, on the other hand, is a description of how fast the running time is. In particular, quasi-polynomial time lies between polynomial time and exponential time, as follows:

- Polynomial: $n^{O(1)} = 2^{O(\log n)}$.
- Quasi-polynomial: $2^{(\log n)^{O(1)}}$.
- Exponential: $2^{O(n)}$.

8.1.2 Assume a probability distribution over the input

We also can assume that we have some probability distribution over the input. For example, for the clique problem, we could assume a random graph and attempt to find the largest clique in this random graph.

The good news is that many **NP**-Complete problems turn out to be efficiently solvable in the average case. For example, if we assume that every possible input graph is equally likely, then it is quite likely that we can find the largest clique in a given graph in polynomial time.

There are two reasons why this is not usually an acceptable solution. First, it is not at all clear what the right probability distribution to use should be. Certainly there are many instances where the uniform

probability distribution is not appropriate, but, even worse, there are many times when we are unable to make any predictions as to what inputs are likely and what inputs are unlikely. Second, it turns out that the worst case inputs can be quite important. In particular, people have observed over the years that the problems that really do show up in practice, e.g., the clique problems that we want to solve, or the vertex cover problems that we want to solve, actually are instances that are difficult to solve.

8.2 Introduction to approximation algorithms

If we have an **NP-Complete** problem, we cannot expect to find the optimal solution. However, we might be able to find a solution that is only a little bit worse than the actual optimal. In fact, in some cases, we can design algorithms that have a provable bound on how close solution returned by the algorithms is to the optimal value. When this is possible, it is often the best strategy for dealing with these problems. As a result, this area of algorithms research has seen quite a bit of attention in recent years.

8.2.1 Greedy approximation algorithm for Vertex-Cover problem

As an example of this, we next consider the vertex-cover problem. In the vertex-cover problem, we want to find a set of vertices that are incident to every edge in the graph. Note that up to now, we've only considered the decision version of the vertex-cover problem. Since it does not make any sense to approximate a "yes" or "no" answer, when we discuss an approximation algorithm, we focus on the optimization problem.

Example: Optimization version of the vertex-cover problem

Input: A graph $G = (V, E)$.

Output: A set $U \subseteq V$ of minimum size such that $\forall e \in E$, e has at least one endpoint in U .

We first give a natural approximation algorithm for this problem, but we then demonstrate that the resulting algorithm is actually not the best approach. A very natural approach for this problem is to start with the vertex having the highest degree. More generally, we could build up a solution, one vertex at a time, by always adding the vertex that covers the most edges that have not been previously covered. This leads to the following algorithm:

Greedy Algorithm:

1. $U = \emptyset$
2. while $E \neq \emptyset$ do
3. let v be the maximum degree vertex
4. $U = U + v$;
5. $G = G - v$;
6. return U

Note that when we add a vertex to U , we remove all the edges from E that are incident to that vertex. The approximation solution of the problem instance in Figure 8.1(a) has size of 4 (shown in Figure 8.1(b)). Notice that this is not an optimal solution. The optimal solution has size of 3 as shown in Figure 8.1(c).

It turns out that the performance of this greedy algorithm can actually be fairly poor. We next construct an input where this greedy algorithm does quite poorly. The specific input is shown in Figure 8.2. Here, the number of the vertices in each column is shown in the first row. The graph has m vertices in the first column. The second column has $\lfloor \frac{m}{2} \rfloor$ vertices, and, in general, the k -th column has $\lfloor \frac{m}{k} \rfloor$ vertices, where

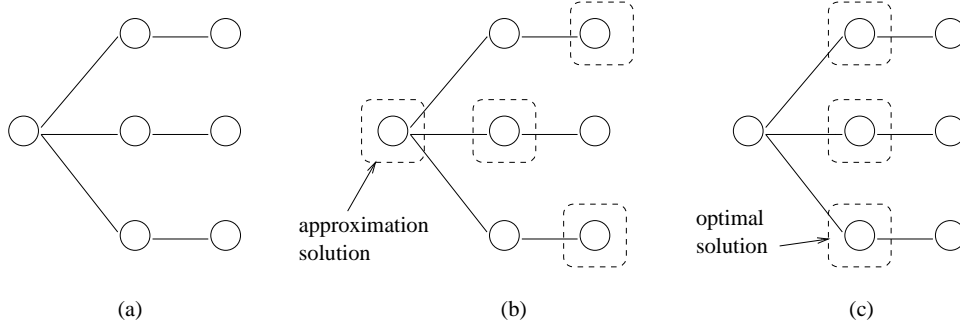


Figure 8.1: Comparison of greedy approximation solution and optimal solution.

$k = 1, 2, \dots, m$. Thus there is a single vertex in the last column. Every vertex in the k -th column ($k > 1$) has k edges to the vertices in the first column. There is at most one edge from the vertices in any given column to any vertex in the first column. The single vertex in the last column has m edges, all of which go to distinct vertices in the first column. Also note that every vertex in the first column has degree $\leq (m - 1)$.

In the above graph, all the edges go to the first column, so that the vertices in the first column cover all the edges. This is the minimum vertex cover, and thus the optimal solution has size of m .

How well does the greedy algorithm do? Since the vertex in the last column has degree m , which is the maximum degree, the greedy algorithm will start with this vertex. After this vertex is put into U , and all the edges incident to that vertex are removed, the vertices in the first column now have degree at most $(m - 2)$, which is less than the degree of vertices in the $(m - 1)$ -st column. Thus, in the next step, the greedy algorithm will pick the vertices in the $(m - 1)$ -th column, and so on. The greedy algorithm ends up with a solution consisting of all vertices of the graph, except those in the first column. Thus, the size of the vertex cover returned by the greedy algorithm is:

$$\begin{aligned} 1 + \lfloor \frac{m}{m-1} \rfloor + \dots + \lfloor \frac{m}{3} \rfloor + \lfloor \frac{m}{2} \rfloor &= (m + \lfloor \frac{m}{2} \rfloor + \lfloor \frac{m}{3} \rfloor + \dots + \lfloor \frac{m}{m-1} \rfloor + \lfloor \frac{m}{m} \rfloor) - m \\ &\geq m(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{m}) - 2m \\ &= \Theta(m \log m) \end{aligned}$$

The inequality above comes from the fact that each of the fractions is at most 1 larger than the value obtained by rounding down.

8.2.2 Performance ratio for approximation algorithms

To measure whether an approximation algorithm performs well, we use the *performance ratio* of an algorithm. Our definition depends on whether the problem is a maximization problem or a minimization problem.

Definition 85 *The performance ratio of an algorithm is*

$$\begin{aligned} \max_{x, |x|=n} \frac{C_{alg}(x)}{C_{opt}(x)} &\quad \text{for a minimization problem,} \\ \text{and } \max_{x, |x|=n} \frac{C_{opt}(x)}{C_{alg}(x)} &\quad \text{for a maximization problem,} \end{aligned}$$

where $C_{alg}(x)$ is the cost of the algorithm solution on input x , and $C_{opt}(x)$ is the cost of the optimal solution on input x .

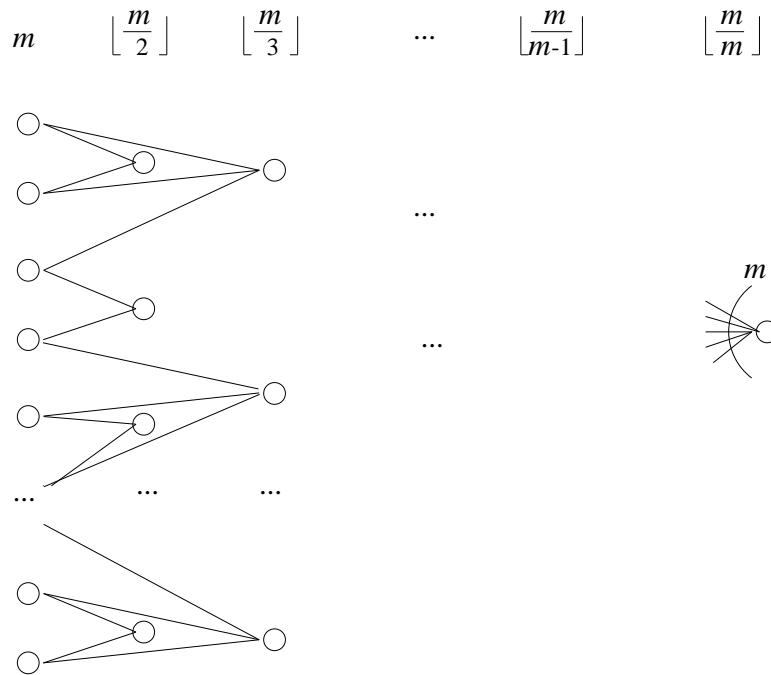


Figure 8.2: Specifically constructed vertex-cover problem where greedy algorithm does poorly.

The reason that we have different measures for the two types of problems is to allow us to compare minimization problems and maximization problems on a similar scale. For both of these measures, large values denote bad approximations and small values denote good approximations.

Example: According to the definition above, the performance ratio of the greedy approximation algorithm for the Vertex-Cover problem is

$$\Omega\left(\frac{m \log m}{m}\right) = \Omega(\log m) \quad (8.1)$$

Why is this stated as $\Omega(\log m)$ instead of $\Theta(\log m)$? The reason is that there may be some other input that has the same size, but has an even worse performance ratio, e.g., m^2 .

Definition 86 An $f(n)$ -approximation is an approximation with a performance ratio of $f(n)$.

Example: It turns out that the bound in equation (19.1) is tight. Thus the greedy approximation algorithm for vertex-cover problem is a $\log n$ -approximation.

8.2.3 A better approximation for the vertex-cover problem:

Algorithm:

1. $S = \emptyset$;
2. while $E \neq \emptyset$ do
3. pick any edge $e = (u, v)$
4. $S = S + u + v$;

5. $G = G - u - v$;
6. return S ;

When we remove the two endpoints of the chosen edge, any edges that are incident to the endpoints are also removed. Also, note that at step 3, any edge in the graph can be chosen: it does not matter what procedure we use for this step.

Claim 87 *This algorithm is a 2-approximation.*

Proof: To prove this claim, we need to argue that for any input, we have a solution that is at most a factor of 2 worse than the optimal solution for that input.

Let E' be the set of edges chosen by the algorithm. The cost of the algorithm is:

$$C_{alg}(G) = 2 \cdot |E'| \quad (8.2)$$

Note that this result does not make any assumption about how we actually choose the edges. Whenever the algorithm chooses an edge, it removes both of the endpoints as well as any other edges that are touching those endpoints. So any pair of edges in E' do not share a vertex. Thus E' is a matching. For the matching E' , each edge in E' has to be covered by some vertex. So for every edge in E' , one of the two endpoints has to be in the optimal solution, and there is no overlap of these endpoints. Therefore, the cost of the optimal solution on any graph G has to be at least the size of E' , i.e.,

$$C_{opt}(G) \geq |E'|. \quad (8.3)$$

We see that (8.2) and (8.3) imply that for any G , $C_{alg}(G) \leq 2C_{opt}(G)$.

Finally, we show that the performance ratio is at least 2, by exhibiting one particular input where the performance ratio of the algorithm is no better than 2. Such an input is a matching: the algorithm will take all of the vertices as its solution, and this is exactly twice as many nodes as the optimal solution of just choosing one endpoint of every edge in the matching. ■

This simple algorithm is very close to the best known approximation algorithm for the vertex-cover problem.

8.2.4 Approximation for Independent-Set problem

We know every problem in **NP** can be reduced to a vertex-cover problem. So, in some sense, everything in **NP** is no harder than the vertex-cover problem. Since we have a 2-approximation for the vertex-cover problem, does this mean that we also have a 2-approximation for every problem in **NP**? Unfortunately not: the vertex-cover approximation algorithm does not provide an approximation for every problem in **NP**. We next see an example of this by examining the *Independent-Set* problem.

Independent-Set problem:

Input: An undirected graph $G = (V, E)$.

Output: A set $U \subseteq V$ of maximum size such that no two vertices in U are connected by an edge.

The Independent-Set problem is equivalent to the clique problem on the complement graph. Also, the decision version of the independent-set problem is polynomial-time reducible to the decision version of the vertex-cover problem.

Claim 88 *Independent-Set \leq_p Vertex-Cover.*

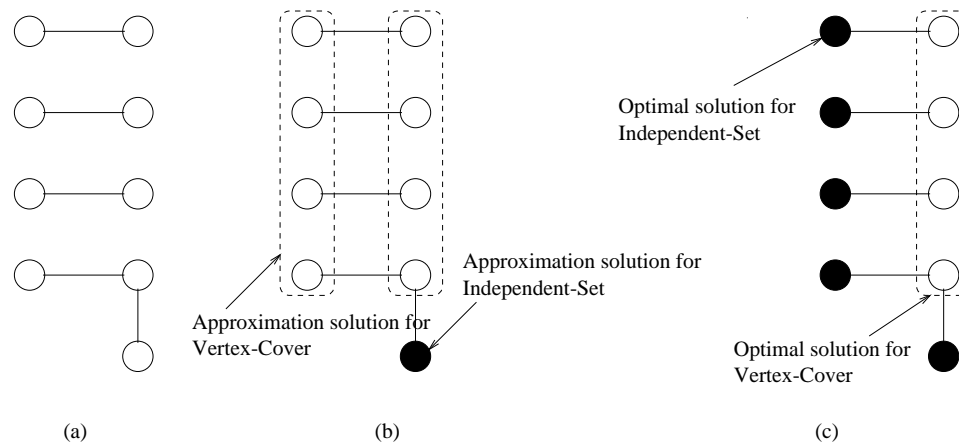


Figure 8.3: Illustration that the 2-approximation for the Vertex-Cover problem does not provide a good approximation for the Independent-Set problem.

Proof: If $U \subseteq V$ is an independent set, from the definition of an independent set, we see that no edge in the graph G has both endpoints in the set U . Thus, if we have an independent set U , then every edge in G has at least one endpoint in $V - U$, which gives us that $V - U$ is a vertex cover. Thus U is an independent set if and only if $V - U$ is a vertex cover. This implies that a graph G has an independent set of size k if and only if it has a vertex cover of size $|V| - k$. Thus, we can do the following reduction, which can be completed in polynomial time:

1. $G \rightarrow G$;
2. $k \rightarrow |V| - k$;

This is guaranteed to map “yes” instances of the Vertex-Cover problem to “yes” instances of the Independent-Set problem, and “no” instances to “no” instances. Therefore, the Independent-Set problem is polynomial time reducible to the Vertex-Cover problem. ■

Since we have a 2-approximation for the vertex-cover problem, we next try to use this 2-approximation algorithm in conjunction with this reduction to provide an approximation algorithm for the independent-set problem.

Approximation algorithm for the Independent-Set problem:

1. Find S , the approximation solution of Vertex-Cover problem, using the 2-approximation algorithm for the Vertex-Cover problem.
2. Return $V - S$.

This algorithm certainly gives us a valid independent set. But how does this algorithm perform? Let’s look at the example in Figure 8.3 (a). Here, the performance ratio of the algorithm for the vertex-cover problem is roughly 2. The size of the vertex-cover found by this algorithm is $n - 1$ (Figure 19.4 (b)), i.e., $|S| = n - 1$ ($n = |V|$), since it puts every vertex but one into the cover. Then the algorithm will only return one vertex as the approximate solution for the Independent-Set problem, i.e., $C_{alg}(G) = 1$. On the other hand, the cost of the optimal independent set on this particular graph is $\frac{n+1}{2}$ (Figure 8.3 (c)), i.e., $C_{opt}(G) = \lceil \frac{n}{2} \rceil$. Thus, we cannot have better than a $\frac{n}{2}$ -approximation. This is really a terrible approximation, since a naive algorithm would be to always return a single vertex, which is always an independent set of size 1, and thus is guaranteed to be an n -approximation. The designed algorithm is only a factor of 2 better than this naive algorithm.

It turns out that we are not able to do much better than this naive algorithm, and thus there is a very big difference between these two different **NP**-Complete problems as to how well they can be approximated.

8.3 Approximation for the max-cut Problem

Given an input graph $G = (V, E)$, the solution to a max-cut problem is a bi-partition of the vertices in the graph G such that the number of edges between the two cells of the partition is maximized. Though the max-cut problem is **NP**-Complete, it is possible to construct a 2-approximation: a solution that is guaranteed to be within a factor of 2 of the optimal solution. The algorithm starts with two sets such that one set contains all the vertices in the graph and the other set is empty. The vertices are subsequently moved from one set to another as long as the size of the cut increases. The 2-approximation for the max-cut problem is given as follows:

1. $S = \emptyset$ /* set of vertices on one side */
2. **while** $\exists v$ such that switching v to other side improves the cut:
3. switch v .
4. **return** the resulting cut, S .

Note that a vertex v can be switched multiple times: it is possible that switching v once improves the cut, but then after other vertices have also been switched, switching v back now improves the cut again.

We need to show that the running time of this algorithm is polynomial in the input size and that it always returns a cut that has size within a factor of 2 of the optimal solution. For each *switch* v operation in the above algorithm, the size of the cut increases by at least 1. As the maximum size of any cut is $|E|$, the number of iterations can be at most $|E|$. The algorithm therefore runs in polynomial time.

Theorem 89 *The max-cut algorithm is a 2-approximation.*

Proof: The size of the cut found by the max-cut algorithm needs to be shown to be within a factor of 2 of the optimal solution. Let $G = (V, E)$ be the input graph. Let $a(v)$ be the set of edges from $v \in V$ that cross the cut. Let $b(v)$ be the set of edges from $v \in V$ that do not cross the cut. Note that $\forall v \ a(v) \geq b(v)$, since otherwise v would have been moved by the algorithm. Summing over all vertices $v \in V$, we get

$$\sum_{v \in V} a(v) \geq \sum_{v \in V} b(v). \quad (8.4)$$

The LHS (left-hand-side) of Equation 8.4 counts each edge that crosses the cut twice: once for each endpoint. Similarly the RHS counts each edge that does not cross the cut twice. Thus, Equation 8.4 implies that more edges cross the cut than do not cross the cut, or in other words that $C_{alg} \geq |E|/2$. Furthermore, the optimal solution can at best have all of the edges crossing the cut: $C_{opt} \leq |E|$. Therefore the above described max-cut algorithm is a 2-approximation. ■

It should be noted here, however, that there is a more complicated approximation algorithm that achieves a 1.1383 performance ratio.

8.4 Approximation for the Metric Traveling Salesman Problem (MTSP)

The Metric Traveling Salesman Problem (MTSP) is:

Input: n cities and a distance function, d , such that the distance function is restricted to conform to the triangle inequality: $\forall i, j, k \quad d_{ij} \leq d_{ik} + d_{kj}$.

Output: A tour of minimum length that passes through every city exactly once.

Some NP-Complete problems turn out to be in P after we make a restriction such as the triangle inequality on the input. Thus, even though it is known that the Traveling Salesman Problem is NP-Complete, the MTSP problem may be in P, or it may be NP-Complete. However, it turns out that we can show that it is NP-Complete by reducing the hamiltonian-cycle problem to the MTSP problem. The decision version of the hamiltonian-cycle problem can be described as:

Input: An undirected graph $G = (V, E)$.

Problem: Is there a cycle that goes, exactly once, through each vertex in the graph?

Theorem 90 *MTSP is NP-complete.*

Proof: Given a subset of edges in the graph and an integer k , it can be verified in polynomial time if the edges pass through every vertex in the graph exactly once and if the cost of the tour is $\leq k$. Therefore $MTSP \in NP$.

We next show that hamiltonian-cycle \leq_p MTSP. Let $G = (V, E)$ be the input to the hamiltonian-cycle problem. To transform the input G to an input of MTSP, let cities be equivalent to the vertices in V . Let

$$d_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 2 & \text{otherwise} \end{cases}$$

The maximum value of d_{ij} is 2. Additionally, the shortest path in the graph consisting of two edges has length 2. Therefore the triangle inequality holds.

To see that we do indeed have a valid reduction, note that if there is a Hamiltonian cycle in G , the corresponding MTSP tour exists with a length of $|V|$ since the distance of each edge is 1. Similarly, if a MTSP tour of length $|V|$ exists, the tour uses only edges with a distance of 1. Therefore a Hamiltonian cycle exists. ■

8.4.1 First Algorithm for MTSP

We start with a simple approximation algorithm for the MTSP problem, called MTSP1. This algorithm is a 2-approximation, but we shall see shortly how to improve this to a $\frac{3}{2}$ -approximation, using a slightly more complicated technique. MTSP1 works as follows: we think of the cities as the vertices of a completely connected graph $G = (V, E)$, and we assign a distance to each edge i, j as the TSP distance d_{ij} between the vertices that edge connects.

1. Compute a Minimum Spanning Tree (MST) on G .
2. Construct a pseudo-tour of the MST by starting at an arbitrary vertex v and by walking around the perimeter of the MST with the MST edges always to the right. The tour uses each edge twice as shown by the directed-loop in Figure 8.4 around the MST. Since the MST contains all the vertices in the graph G and there are no cycles in a MST, the tour is guaranteed to visit all the vertices.

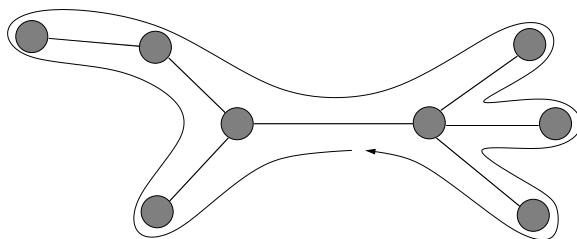


Figure 8.4: MTSP1: Pseudo-tour

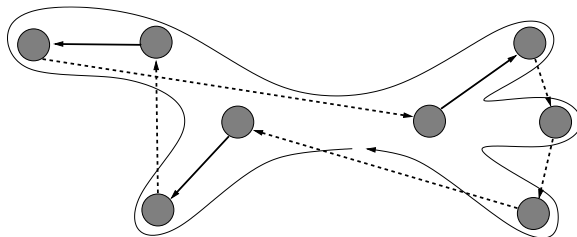


Figure 8.5: MTSP1: Extracted Tour

3. Extract a tour by short-cutting the repeated vertices in the pseudo-tour. In particular, a vertex in the pseudo-tour is skipped if it has already been visited. In the resulting tour, each vertex will be used exactly once. Figure 8.5 shows the extracted tour in **bold solid** and dashed lines. The solid lines represent the edges common to both the pseudo-tour and the extracted tour. The dashed lines represent the short-cutting used to construct the extracted tour.

Claim 91 *The MTSP1 algorithm is a 2-approximation.*

Proof: As the pseudo-tour visits each edge twice, the cost of the pseudo-tour is twice the cost of the MST. The extracted tour has cost at most twice the cost of the MST as the triangle inequality means we can take a short cut without increasing the cost. Additionally, the cost of an optimal tour on a given graph G cannot be less than the MST of the graph, since we can always construct a spanning tree by taking a tour and removing any edge of that tour. Thus, we get the following:

$$\text{length of extracted tour} \leq \text{length of pseudo-tour} = 2 \times \text{cost}(MST)$$

Since

$$\begin{aligned} \text{cost}(MST) &\leq \text{length of optimal tour} \\ \text{length of extracted tour} &\leq 2 \times \text{length of optimal tour} \end{aligned}$$

Thus, algorithm *MTSP1* is a 2-approximation. ■

8.4.2 Second Algorithm for MTSP (MTSP2: a $\frac{3}{2}$ -Approximation)

A better algorithm for *MTSP* can identify a tour of length at most $\frac{3}{2}$ times the length of optimal tour. Such an algorithm uses a *Eulerian tour*. A Eulerian tour (Euler 1736) can be defined as follows.

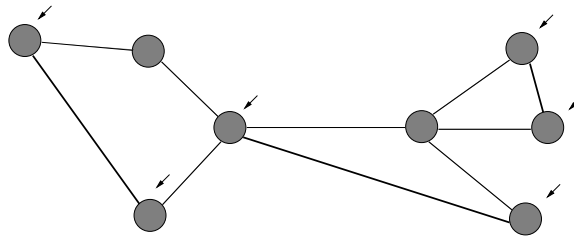


Figure 8.6: MTSP2: MST and Matchings

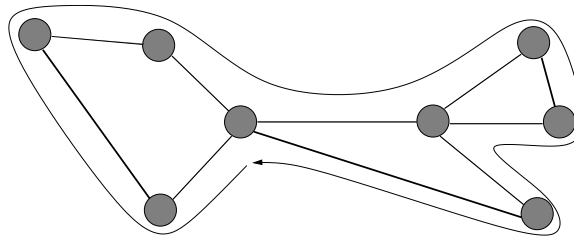


Figure 8.7: MTSP2: Eulerian Tour

Definition 92 *A Eulerian tour is a path that traverses every edge of the graph exactly once and returns back to the initial vertex.*

A Eulerian tour can thus visit any vertex in G several times. A graph G contains a Eulerian tour if and only if G is a connected graph and every vertex in the graph G has even degree. Furthermore, if a graph contains a Eulerian tour, we can find one in polynomial time. The following algorithm is due to Christofides [NCF76]. We again start with a completely connected graph $G = (V, E)$, and a distance function d as defined by the TSP distance.

MTSP2 Algorithm

1. Compute a Minimum Spanning Tree(MST) on G .
2. A Eulerian tour cannot be constructed on a MST as there are odd degree vertices in any tree, since a leaf node has odd degree. However, the MST can be augmented in the following way such that every vertex in the resulting graph has even degree.
 - a. The set D of odd degree vertices is identified from the MST. The arrows in Figure 8.6 show the odd degree vertices. Note that $|D|$ is even, since the sum of all the degrees in D is even. This is the case since the sum of the degrees of all vertices in the graph G is equal to twice the edges in G and therefore is an even number. Removing the even degree vertices still leaves us with an even number.
 - b. A minimum weight matching on D is computed and is added to the MST. Such a matching can be found in polynomial time. An example matching is shown in Figure 8.6 as **bold** lines on the MST. It should be noted that it is permissible to use a matching that contains an edge already present in the original MST. In such cases, the newly found edge must be considered as a distinct edge on the graph.
3. The resulting graph is connected as it contains a MST and each vertex in the graph now has even degree. Thus, a Eulerian tour can be identified on the resulting graph. Such a tour is guaranteed to

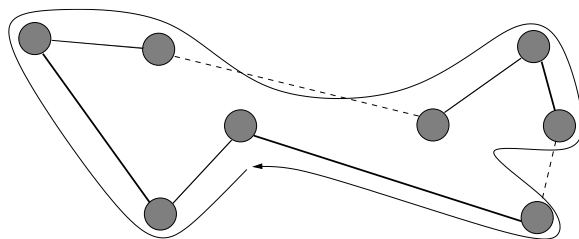


Figure 8.8: MTSP2: Extracted Tour

visit all vertices in the graph as the resulting graph contains a MST. Figure 8.7 shows a Eulerian tour on the resulting graph.

4. A tour is extracted from the Eulerian tour by short-cutting the repeated vertices in the Eulerian tour such that each vertex is used exactly once. Starting at an arbitrary vertex, we follow the sequence of vertices visited by the Eulerian tour, except that a vertex in the Eulerian tour is neglected if it has already been visited. Figure 8.8 shows the Eulerian tour and the extracted tour. The solid lines represent the edges common to both the pseudo-tour and the extracted tour. As before, the dashed lines represent the short-cutting used to construct the extracted tour.

Claim 93 *The MTSP2 algorithm is a $\frac{3}{2}$ -approximation.*

Proof: The cost of the Eulerian tour is equal to the sum of the cost of the matching edges and the cost of the MST. Furthermore, the triangle inequality allows us to short-cut repeated vertices without increasing cost, and thus the cost of the extracted tour is at most the cost of the Eulerian tour. In other words,

$$\text{cost of the extracted tour} \leq \text{cost of the Eulerian tour} \leq \text{cost}(MST) + \text{cost of matching} \quad (8.5)$$

We already saw that

$$\text{cost}(MST) \leq \text{cost of optimal tour} \quad (8.6)$$

The cost of the matching on D can be computed by initially computing the cost of an optimal tour on the vertices in D . By the triangle inequality, the cost of an optimal tour on the vertices in D is less than or equal to the cost of the optimal tour on all the vertices in G . Since D consists of an even number of vertices, any tour on D can be partitioned into two distinct matchings. One of these matchings must have at most half of the weight of the original tour. From this, we see that

$$\text{cost of } D \text{ matching} \leq \frac{1}{2} \times \text{cost of optimal tour} \quad (8.7)$$

From Equations 8.5, 8.6 and 8.7,

$$\text{cost of extracted tour} \leq \frac{3}{2} \times \text{cost of optimal tour}$$

The MTSP2 algorithm is therefore a $\frac{3}{2}$ -approximation. ■

This algorithm leads us to two natural questions: (1) does this approximation algorithm extend to more general versions of the TSP problem, and (2) can we do better for more specific versions of the TSP?

To address question (1), we prove the following:

Claim: For general version of TSP, there is no k -approximation for any constant k , unless $\mathbf{P} = \mathbf{NP}$.

Proof: The proof can be provided by a reduction from the Hamiltonian-cycle problem. The Hamiltonian-cycle input is a graph $G = (V, E)$. The corresponding TSP input is obtained by assigning the cities to vertices V in G . The distance function d is defined as $d_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ (k+1) \cdot |V| & \text{otherwise} \end{cases}$

If there is a Hamiltonian cycle, then there is a TSP tour of length $|V|$. However, if G doesn't have a Hamiltonian cycle, the tour is of length $> (k+1) \cdot |V|$. Thus, if we have a k -approximation, we can tell the difference between the two possible cases. This would give us a polynomial time algorithm that decides whether or not a Hamiltonian cycle exists. ■

Note that this reduction assumes that we know the value of k . Think about this as follows: assume that someone describes to you a k -approximation. This k -approximation has some specific value of k associated with it. You can then use this value of k to construct the reduction. Thus, for any k -approximation, there is a corresponding reduction that uses that value of k . In fact, the combination of the approximation algorithm and the reduction can be used to solve the Hamiltonian-cycle problem in polynomial time.

To address question (2) above, we note that if a stronger restriction on the distance function d is used, better approximation algorithms can be obtained. Consider the Euclidean TSP, where cities are points in a plane. Such a problem is still \mathbf{NP} -Complete. However, for $\forall \epsilon > 0$, a $(1 + \epsilon)$ -approximation can be obtained. The algorithm runs in $n^{O(1+1/\epsilon)}$ time.

8.5 Approximation for the set-cover problem

We next consider an approximation algorithm for the following *set-cover* problem.

Input: a finite set U and a collection $C = \{S_1, S_2, \dots, S_m\}$ of subsets of U .

Output: a minimum sized cover $C' \subseteq C$ such that every element of U is contained in at least one element of C' .

Here is a small example:

$$\begin{aligned} U &= \{e_1, e_2, e_3, e_4\} \\ C &= \{\{e_1, e_2\}, \{e_1, e_3\}, \{e_1, e_4\}, \{e_1, e_2, e_4\}\} \\ C' &= \{\{e_1, e_2\}, \{e_1, e_3\}, \{e_1, e_4\}\} \\ C'' &= \{\{e_1, e_3\}, \{e_1, e_2, e_4\}\} \end{aligned}$$

Both C' and C'' are solutions but C'' has the minimum possible size. Another example is when the set U is defined to be the edges of a graph, and C contains one set S_i for each vertex i , such that S_i contains all of the edge that are incident to the vertex i . Note that this second example is actually the vertex-cover problem we considered earlier. This example demonstrates that the set-cover problem is \mathbf{NP} -Complete.

We here consider approximation algorithms for the set-cover problem. In fact, we shall consider the more general case of this problem, where each set S_i has a weight w_i , and our objective is to minimize the total weight of the set-cover C' . Although we saw for the vertex-cover problem that a greedy approach is not always best, we shall introduce a greedy algorithm for set-cover. It turns out that for set-cover, we do not expect to do better than this simple approach.

For a greedy algorithm, a natural objective is to choose the set that covers the largest number of previously uncovered elements. For example, a natural first step would be to choose the largest set to be in the cover. However, we also want to take the weight of the sets into account. Thus, another natural first step would be to choose the set of minimum weight. We shall combine these two objectives by choosing the set that minimizes the quantity $w_i/|S_i|$. The idea is that we want to minimize the cost (weight) per element covered. This leads us to the following algorithm:

Approximation Algorithm for Set-cover:

Set $R = U, C^* = \emptyset$

While $R \neq \emptyset$

Let S_i be the set minimizing $w_i/|S_i \cap R|$.

$R = R - S_i$.

$C^* = C^* \cup \{S_i\}$.

Return C^* .

Theorem 94 Let $d^* = \max_i |S_i|$. This algorithm is an $H(d^*)$ -approximation, where $H(d^*) = \sum_{i=1}^{d^*} \frac{1}{i}$.

Proof: We shall formalize the intuition we described for the greedy algorithm. In particular, $\forall e \in U$, let $c_e = \frac{w_i}{|S_i \cap R|}$, where S_i is the first set that is chosen that covers the element e , and the elements in the set R are those that are in the set R at the time that S_i is chosen (right before the elements of S_i are removed from R .) We can think of the cost c_e as the cost paid for covering element e .

Claim 95

$$\sum_{S_i \in C^*} w_i = \sum_{e \in U} c_e.$$

This claim follows from the fact that the weight of each selected set is distributed in cost over the elements that are newly covered by that set. Thus, the weight of each set is completely accounted for. Note that the left hand side of Claim 95 is the cost of the solution returned by the algorithm. Thus, this Lemma says we can argue about the cost of the algorithm in terms of the right hand side as well.

Lemma 96

$$\forall S_k, \sum_{e \in S_k} c_e \leq H(|S_k|) \cdot w_k.$$

Note that this Lemma refers to all sets S_k : both those chosen by the greedy algorithm, as well as those sets not chosen by the greedy algorithm. Roughly, this Lemma says the following: the elements that are in the set S_k could have been covered with total cost w_k . However, instead, the algorithm uses total cost $\sum_{e \in S_k} c_e$. The Lemma says that the algorithm cost cannot be more than the cost by using the set w_k by more than a factor of $H(|S_k|)$. We shall make this more formal below, after we prove this Lemma.

Proof: Let d be the number of elements in the set S_k . We can assume through renumbering that $S_k = \{e_1, e_2, \dots, e_d\}$. Furthermore, we can also assume that the elements of S_k are numbered by the order that they are covered by the algorithm, where ties (i.e., two elements being covered during the same iteration of the algorithm) are broken arbitrarily.

Note first that for all j , $1 \leq j \leq d$, in the iteration where e_j is covered, it must be the case that $\frac{w_k}{|S_i \cap R|} \leq \frac{w_k}{d-j+1}$. This is because the elements are numbered in the order they are covered, and thus none of the elements e_k, \dots, e_d can be covered yet. Note that it is possible that $\frac{w_k}{|S_i \cap R|} < \frac{w_k}{d-j+1}$, since some of the

elements before e_j may be covered at the same time that e_j is covered (for example, consider what happens when the first set chosen is S_k itself).

Let S_i be the set chosen to cover the element e_j . Since e_j was not covered prior to S_i being chosen, the set S_k has not been chosen yet. The set S_k may be chosen at this iteration, or a set with a smaller cost per newly covered element may be chosen. Thus, $\frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|}$. This implies that $c_{e_j} \leq \frac{w_k}{d-j+1}$. This gives us that

$$\sum_{j=1}^d c_{e_j} \leq \sum_{j=1}^d \frac{w_k}{d-j+1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \dots + \frac{w_k}{1} = H(d) \cdot w_k.$$

■

We are now ready to prove the theorem. Let C^{opt} be the sets in the optimal set cover (that minimizes the total weight). Let w^{opt} be the cost of this optimal solution, and so $w^{opt} = \sum_{S_i \in C^{opt}} w_i$. From Lemma 96, we see that $\forall S_i \in C^{opt}, w_i \geq \frac{1}{H(d^*)} \sum_{e \in S_i} c_e$. This gives us that

$$w^{opt} \geq \sum_{S_i \in C^{opt}} \frac{1}{H(d^*)} \sum_{e \in S_i} c_e = \frac{1}{H(d^*)} \sum_{S_i \in C^{opt}} \sum_{e \in S_i} c_e \geq \frac{1}{H(d^*)} \sum_{e \in U} c_e = \frac{1}{H(d^*)} \sum_{S_i \in C^*} w_i.$$

Here, the second step (equality) follows from a rearrangement of the terms of summation. The third step (\geq) follows from the fact that the optimal solution must be a set cover, and thus each element of the set U must appear in at least one set of C^{opt} , and hence at least once in the double summation. The final step (equality) follows from Claim 95. ■

We point out that we have not shown here that the greedy algorithm does not do better than an $H(d^*)$ -approximation. However, our very first attempt at an approximation algorithm was a greedy algorithm for the vertex-cover problem. The greedy algorithm considered there was actually exactly the greedy algorithm we considered here for the set-cover problem, for the special case of set-cover described above that corresponds to the vertex cover problem. As a result, the lower bound of $\Omega(\ln n)$ we described for the approximation ratio of that algorithm also applies to the greedy algorithm we consider here. (Recall that $H(d^*) \approx \ln d^*$.) Similarly, the analysis of the greedy set-cover algorithm we provide here, demonstrates that the greedy algorithm for vertex-cover does not do much worse than it does for the example we considered there.

We point out that for the vertex-cover problem, we saw a rather simple way to improve the approximation ratio from $\ln n$ to 2. However, this does not seem to be possible for the more general set-cover problem (even without weights). As we shall see below, the performance of this rather simple greedy algorithm is basically the best that we can hope for.

8.6 Polynomial Time Approximation Schemes and the knapsack problem

Definition 97 *A problem has a polynomial time approximation scheme (PTAS) if and only if $\forall \epsilon > 0$ it has a polynomial time $(1 + \epsilon)$ -approximation.*

An example of this is the Euclidian TSP problem, where $\forall \epsilon > 0$ there is a $(1 + \epsilon)$ approximation with a running time of $n^{O(1+\frac{1}{\epsilon})}$. The running time for this PTAS is exponentially dependent on $\frac{1}{\epsilon}$. Thus, for large values of ϵ (which do not give a very good approximation) the running time is reasonable, but if we let ϵ get arbitrarily small, the running time for this PTAS increases considerably. A specific type of PTAS with better running times than the Euclidian TSP PTAS is a *fully* polynomial time approximation scheme (FPTAS), where the running time remains polynomial in $\frac{1}{\epsilon}$.

Definition 98 A problem has a fully polynomial time approximation scheme (FPTAS) if and only if $\forall \epsilon > 0$ it has $(1 + \epsilon)$ -approximation, such that the runtime of the $(1 + \epsilon)$ -approximation is polynomial in $\frac{1}{\epsilon}$ and also polynomial in the size of the input.

An example of this is the FPTAS algorithm for the knapsack problem we shall examine next. This algorithm has a running time of $O(\frac{n^3}{\epsilon})$.

8.6.1 FPTAS for the Knapsack Problem

Many problems which are not strongly NP-Complete do have an FPTAS. An example of this is the knapsack problem. Recall that the knapsack problem is defined as follows:

The knapsack problem:

Input: A set of items numbered as $1, 2, \dots, n$;
 a weight w_i for each item i ;
 a value v_i for each item i ;
 and a knapsack capacity, C .

Output: A subset B of the items with the maximum total value such that $\sum_{i \in B} w_i \leq C$.

Previous Knapsack Algorithm

We have already seen a dynamic programming algorithm to solve the knapsack problem. It was described for the simplified version of the Knapsack problem where the weight of each item is equal to the value of the item, but the idea behind that algorithm also works for the more general version of the problem considered here. To do so, define $knap(i, w)$ to be the maximum value obtained using items 1 to i and capacity at most w . We see that

$$Knap(i + 1, w) = \max(Knap(i, w), Knap(i, w - w_{i+1}) + v_{i+1}).$$

We can compute $knap(i, w)$ row by row in a table of size $n \cdot C$. We obtain the final solution by looking at the entry $knap(n, C)$. The runtime of this algorithm is $O(n \cdot C)$, and it will always give an exact solution. (Notice that C can be as large as exponential in the input size.)

It turns out that this algorithm does not lead directly to FPTAS - we shall see shortly why. However, a similar approach can be modified to give us an FPTAS.

New Knapsack Algorithm

We now look at an alternative algorithm which we then convert to an approximation scheme. In this alternative algorithm, instead of taking $knap(i, w)$ to be the maximum value of a subset with items 1 through i with a given capacity we define $vknap(i, v)$ to be the minimum weight required to achieve a value of at least v using items 1 to i . A way of computing these values row by row is by using the following formula:

$$vknap(i + 1, v) = \min\{vknap(i, v), vknap(i, v - v_{i+1}) + w_{i+1}, \}$$

where if $v < v_{i+1}$, $vknap(i, v - v_{i+1}) = 0$

1				·	·	·		
2				·	·	·		
				·	·	·		
				·	·	·		
n-1				·	·	·		
n				·	·	·		

$\underbrace{\hspace{10em}}_{nV}$

Figure 8.9: $vknap(i, v)$.

To see that this is the case, note that to achieve value v with items 1 through $i + 1$, either we do so using only items 1 through i , or we achieve value $v - v_{i+1}$ with items 1 through i , and the remaining value with item $i + 1$. For the first row of the table, we use the following:

$$vknap(1, v) = \begin{cases} w_1 & \text{for } v \leq v_1 \\ \infty & \text{for } v > v_1 \end{cases}$$

In the old version of the algorithm, the number of columns was the total capacity. In the new version we define $V = \max_i(v_i)$. The maximum possible value we can achieve is $n \cdot V$. Thus, we only need to consider values of v up to V , and so we now have nV columns. To find the optimal solution (the maximum value that does not exceed the knapsack capacity), we examine the last row of the table, scanning from the left until we find a value that exceeds the Knapsack capacity. The optimal solution is the value before it. In other words we find the largest column, v_{max} , such that $vknap(n, v_{max}) \leq C$. (Notice that $vknap(n, v)$ is non-decreasing as v increases.)

Running time of the algorithm. The size of the table is n^2V (n rows by $n \cdot V$ columns). Every entry in the table can be computed in constant time, and the scan over the last row takes time $O(nV)$. Thus, the runtime of this algorithm is $O(n^2V)$.

Knapsack Approximation

The idea of the approximation algorithm for Knapsack is to reduce the precision with which we compute the value, and thus reduce the number of columns. In particular, we set the k least significant bits to 0 in every value that appears in the input. This does not affect the optimal value that can be achieved too much because the lower order bits do not add up to very much compared to the higher order bits, but reducing the number of bits reduces the amount of time required by the algorithm. By truncating the last k bits of all the values we reduce the number of columns (and hence the amount of work done) in the table by a factor 2^k .

The tradeoff here is that when k is small, our approximation is good (ϵ is small), while when k is large, the runtime is reduced. The solution we obtain will still be valid because we have not changed the weights: the same solution is guaranteed to still fit into the knapsack. While the value achieved will not necessarily be optimal, it will not be too far off from the optimal value.

Since this is a maximization problem, we need to find k such that $\frac{C_{opt}}{C_{alg}} \leq (1 + \epsilon)$, for a given ϵ .

- Let $v'_i = v_i$ with the k lowest order bits set to zero.
- Let C_{alg} be the value of the solution returned by this algorithm.
- Let C'_{alg} be the optimal value of the modified problem.

Let C_{opt} be the optimal value of the original problem.

Let B' be the optimal subset of items in the modified problem.

Let B be the optimal subset of items in the original problem.

First, note that since the value of each item can only be decreased by truncating bits, it must be the case that $C_{alg} \geq C'_{alg}$. Furthermore

$$C'_{alg} = \sum_{i \in B'} v'_i.$$

Since the set B' represents the maximum solution to the modified problem, the solution formed by the set B must be less than or equivalent to it (in the modified problem). Therefore:

$$C_{alg} \geq C'_{alg} = \sum_{i \in B'} v'_i \geq \sum_{i \in B} v'_i$$

Since each v'_i has the k smallest bits set to 0, $v'_i > v_i - 2^k$ for all v_i , and so

$$\sum_{i \in B} v'_i > \sum_{i \in B} (v_i - 2^k) \geq C_{opt} - n \cdot 2^k$$

Therefore:

$$\begin{aligned} C_{alg} &\geq C_{opt} - n \cdot 2^k \\ \frac{C_{opt}}{C_{alg}} &\leq \frac{C_{opt}}{C_{opt} - n \cdot 2^k} \\ &= \frac{C_{opt} - n \cdot 2^k + n \cdot 2^k}{C_{opt} - n \cdot 2^k} \\ &= 1 + \frac{n2^k}{C_{opt} - n2^k} \\ &\leq 1 + \frac{n2^k}{V - n2^k} \end{aligned}$$

Since we are looking for a $(1 + \epsilon)$ -approximation, we want a value of k such that $(1 + \frac{n2^k}{V - n2^k})$ will be no larger than $(1 + \epsilon)$.

Claim 99 *If $k \leq \log(\frac{\epsilon V}{2n})$ and $\epsilon \leq 1$, then $(1 + \frac{n2^k}{V - n2^k})$ is at most $(1 + \epsilon)$.*

Proof:

$$\frac{C_{opt}}{C_{alg}} \leq 1 + \frac{n2^k}{V - n2^k}$$

$$\text{As } k \leq \log(\frac{\epsilon V}{2n})$$

$$\begin{aligned} \frac{C_{opt}}{C_{alg}} &\leq 1 + \frac{n \frac{\epsilon V}{2n}}{V - n \frac{\epsilon V}{2n}} \\ &\leq 1 + \frac{\frac{\epsilon V}{2}}{V - \frac{\epsilon V}{2}} \end{aligned}$$

As $\epsilon \leq 1$,

$$\frac{\epsilon V}{2} \leq \frac{V}{2}$$

Therefore,

$$\begin{aligned} \frac{C_{opt}}{C_{alg}} &\leq 1 + \frac{\frac{\epsilon V}{2}}{\frac{V}{2}} \\ &\leq 1 + \epsilon \end{aligned}$$

■

To recap, the FPTAS for the Knapsack problem is as follows: Given a problem instance and an approximation ratio $(1 + \epsilon)$:

1. Set the value of k , i.e., let $k = \lfloor \log(\frac{\epsilon V}{2n}) \rfloor$;
2. Truncate the last k bits from all the values (not the weights);
3. Solve the smaller problem (using the vknap version), and use the items in the resulting solution.

Running time of the algorithm: The running time is still the size of the table, with each table entry taking constant time. In the algorithm, we shall actually remove the last k bits of the values, rather than simply setting them to 0. Each value would then be shifted right k bits, and the new maximum value is $\frac{V}{2^k}$. We still have n rows, but now we only have $\frac{nV}{2^k}$ columns, so the table has size $n \cdot \frac{nV}{2^k}$. Substituting in our value for k , the run time is $O(\frac{n^2 V}{2^k}) = O(\frac{n^2 V}{\frac{\epsilon V}{2n}}) = O(\frac{n^3}{\epsilon})$, which is polynomially dependent on $1/\epsilon$.

We now briefly mention why we defined a modified version of the dynamic programming algorithm, instead of using the original version. In the original version of the algorithm, the running time is proportional to the number of different weights we need to consider, and thus, to design an FPTAS, we would need to round the weights of the items. We can either round up or down. If we round down, then we may end up with a subset of items whose (unrounded) weights exceed the knapsack's capacity. If we round up the weights we get a valid solution but the effect of the rounding is that we might not include an item with a small weight, but very large value. This means that it is not possible to bound the approximation ratio if the weights are rounded up.

8.7 Hardness results for approximability

NP-Complete problems seem to fall into very distinct categories in terms of how well they can be approximated. In order of decreasing quality of approximation, these include, but are not limited to:

1. FPTAS - This is the best we can hope for, since the running time is polynomial in $\frac{1}{\epsilon}$. The example of this we have seen is the knapsack problem.
2. PTAS - This is still very good, but it is not polynomial in $\frac{1}{\epsilon}$. The example of this we mentioned was the Euclidean TSP.
3. Constant - This is worse, but still quite respectable. We have seen several examples of this: Max-Cut, Vertex-Cover, and Metric TSP. We have seen a 2-approximations for the first two, and a $\frac{3}{2}$ -approximation for the third. For all three problems, it is known that there is a constant such that the problem cannot be approximated better than that constant (unless $\mathbf{P} = \mathbf{NP}$).

4. $\log n$ - Here n is the size of the input. The example of this we have seen is the set-cover problem; again, it is quite likely that this approximation algorithm is the best possible.
5. Worse than $\log n$ - A number of problems are such that the best known is considerably worse than $\log n$, and often as bad as n^ϵ , for some constant ϵ . This type of approximation is not very useful. However, for some problems, such as the clique problem, this is the best that we can do (unless $\mathbf{P} = \mathbf{NP}$).

8.7.1 Problems without an FPTAS

We first demonstrate that $\mathbf{P} \neq \mathbf{NP}$, then there is not an FPTAS for every problem. In fact, relatively few problems have an FPTAS. The following theorem demonstrates (roughly) that if there exists an FPTAS for any *strongly NP-Complete* problem, then $\mathbf{P} = \mathbf{NP}$. Specifically:

Theorem 100 Consider a problem π such that,

1. π is strongly **NP-Complete**.
2. all the values in the input and the output are integers.
3. \forall inputs I , $C_{opt}(I)$ is polynomial in $|I|$ and the largest number in I , where $C_{opt}(I)$ is the cost of the optimal solution for the input I .

If there is an FPTAS for π then $\mathbf{P} = \mathbf{NP}$.

For example, a problem with these properties is the clique problem with integer weights (where the objective is to find the largest total weight clique). This problem is strongly **NP-Complete** (since it is **NP-Complete** even with unit weights). An upper bound on C_{opt} is the product of the largest weight and the number of vertices, which is polynomial in the size of the input graph and the largest number in the input. In fact, all of the strongly **NP-Complete** problems we have seen are covered by this theorem.

Proof: We here assume a maximization problem; the case of a minimization problem is similar. Since π is strongly **NP-Complete**, to show that $\mathbf{P} = \mathbf{NP}$, we need to show that we can design an algorithm for π that runs in polynomial time for all inputs I where all integers in I are polynomial in the input size.

An algorithm for the problem on input I is:

1. Let $\epsilon = \frac{1}{U(I)+1}$, where $U(I)$ is the upper bound on the optimal solution on input I . Note that we can assume that $U(I)$ is polynomial in $|I|$.
2. Run $\text{FPTAS}(\epsilon, I)$, and use the solution returned by the approximation algorithm.

Let $C_{alg}(I)$ be the cost of the solution that the algorithm gives for input I . Let $C_{opt}(I)$ be the cost of the optimal solution for the input I . Since $(1 + \epsilon)C_{alg} \geq C_{opt}$, we get $C_{opt} - C_{alg} \leq \epsilon C_{alg}$. As $\epsilon C_{alg} < 1$, $C_{opt} - C_{alg} < 1$. Thus, since all values are integers, the solution given by the algorithm is the optimal solution. Since $U(I)$ is polynomial in $|I|$, and we have an FPTAS, the running time is polynomial in the input size. Thus if there is an FPTAS for a strongly **NP-Complete** problem of the type described in the theorem statement, then we can solve the problem exactly. ■

8.7.2 Problems without a PTAS

Unfortunately, even problems with a PTAS do not seem to occur as frequently as we would like: there are many problems where it is known that the best possible approximation algorithm (assuming $\mathbf{P} \neq \mathbf{NP}$) is a constant. In the last few years, there has been considerable work and progress in determining exactly what is the best approximation ratio possible for a number of problem. Most of these results (as well as the even stronger lower bounds described below) are based on Probabilistically Checkable Proofs.

In some ways, these techniques lead to a similar high level strategy as we saw in proving problems to be \mathbf{NP} -Complete. There, we showed that 3-SAT was \mathbf{NP} -Complete, and then used that to show that many other problems were also \mathbf{NP} -Complete. Here, Probabilistically Checkable Proofs are used to show that a small number of problems are \mathbf{NP} -Complete, and then those problems are reduced to other problems, using a type of reduction suitable for showing results on approximability.

To show lower bounds of a constant on approximability, we use as the starting problem a variant of 3-SAT. While it does not make any sense to approximate 3-SAT (as it has a solution of the form Yes/No), the variation of the 3-SAT that can be approximated is the Max-3-SAT. The Max-3-SAT problem is defined as:

Input: A boolean formula ψ in 3-CNF form.

Output: The maximum number of clauses that can be simultaneously satisfied.

Note that if we use a random assignment to the variables, then every clause in a 3-CNF formula has a $\frac{7}{8}$ probability of being satisfied. This gives us a $\frac{7}{8}$ approximation for the Max-3-SAT: simply use a random assignment of the variables. The expected number of clauses that will be satisfied by this algorithm is $\frac{7}{8}$ of all the clauses. Perhaps surprisingly, we can show that this is the best that we can do. Specifically, using Probabilistically Checkable Proofs, we can show that $\forall \epsilon > 0$, if $\mathbf{P} \neq \mathbf{NP}$, there is no $(\frac{7}{8} - \epsilon)$ -approximation for the Max-3-SAT problem.

We next give a simple example of using this result to prove the hardness of approximation for other problems. In particular, we prove the following result for the clique problem. We point out that there are actually much stronger inapproximability results known for this problem, as are described below.

Claim 101 *If $\mathbf{P} \neq \mathbf{NP}$, the clique problem cannot be approximated better than $\frac{8}{7}$.*

Proof: To show the clique problem was \mathbf{NP} -Complete, we reduced the 3-SAT problem to the Clique problem. We showed that all the m clauses of a boolean formula are satisfied if and only if a clique of size m exists in the corresponding graph. However, from the same reduction, it also follows that $\forall k$, k clauses of a Boolean formula can be simultaneously satisfied if and only if there is a clique of size k in the corresponding graph.

Thus, if we can approximate the clique problem better than $\frac{8}{7}$ we can also approximate Max-3-SAT better than $\frac{8}{7}$. So, if $\mathbf{P} \neq \mathbf{NP}$, clique cannot be approximated better than $\frac{8}{7}$. ■

8.7.3 Problems with no constant factor approximation

For the set-cover problem, we saw a $\ln |U|$ -approximation, but we did not see a constant factor approximation. In fact, the simple greedy algorithm we saw is likely to be the best possible for this problem. In particular, we know that if $\mathbf{P} \neq \mathbf{NP}$, then there is no polynomial time $c \ln |U|$ -approximation algorithm for this problem, for some $c > 0$. Furthermore, we know that if $\mathbf{NP} \not\subseteq \mathit{DTIME}(n^{\log \log n})$, then there is no polynomial time $(1 - \epsilon) \ln |U|$ -approximation to the set-cover problem, for any $\epsilon > 0$. $\mathit{DTIME}(n^{\log \log n})$ is the class of problems that can be solved in deterministic time $O(n^{\log \log n})$; it is almost as unlikely that \mathbf{NP} is contained in this class as that it is contained in \mathbf{P} , and thus it is quite likely that the greedy approximation is optimal.

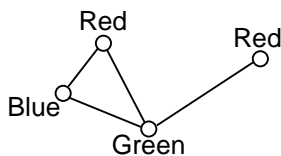


Figure 8.10: A graph whose chromatic number is 3

The set-cover problem serves the same purpose for problems where $\Theta(\log n)$ is the best approximation ratio possible as max-3-sat did for problems where a constant approximation ratio was the best possible. In particular, the set-cover problem can be used to show lower bounds on approximability for a number of other problems.

8.7.4 Problems with no $\log n$ -approximation

Examples in this class include *Clique*, *Independent Set*, and the *Graph-Coloring* problem. Problems in this class have no good approximations (unless $\mathbf{P}=\mathbf{NP}$).

- *Clique*

Claim 102 If $\mathbf{P} \neq \mathbf{NP}$, then there is no $|V|^{\frac{1}{2}-\epsilon}$ -approximation algorithm for clique, for any $\epsilon > 0$.

Claim 103 If $\mathbf{NP} \neq \mathbf{ZPP}$, then there is no $|V|^{1-\epsilon}$ -approximation algorithm for clique, for any $\epsilon > 0$.

Definition 104 **ZPP** is the class of problems that can be solved by Las Vegas randomized algorithms in polynomial time.

The best known approximation for *Clique* is a $O(\frac{|V|}{\log^2 |V|})$ -approximation.

- *Max Independent Set*

In the *independent-set* problem, we want to find the largest set of vertices without any edges between any pair of vertices in the set. This problem is really just a disguised version of the clique problem: to convert from one to the other, just use \overline{G} , the complement of the graph G . Thus, this is just as hard to approximate as clique.

- *Graph-Coloring*

The *graph-coloring* problem is also in this class. For this problem, the input is an undirected graph $G = (V, E)$ and we wish to find the *chromatic number* of G , which is the minimum number of colors required to color G . A coloring is a mapping f :

$$f : V \longrightarrow \text{colors}$$

such that if v_1 is adjacent to v_2 then $f(v_1) \neq f(v_2)$. That is, adjacent vertices must have different colors. The graph in Figure 8.10 is an example of a graph whose chromatic number is 3.

Graph-coloring is clearly related to clique since the chromatic number of a graph is at least as large as the maximum clique.

Claim 105 If $\mathbf{P} \neq \mathbf{NP}$, then there is no $|V|^{\frac{1}{7}-\epsilon}$ -approximation algorithm for Graph-Coloring, for any $\epsilon > 0$.

Claim 106 *If $\mathbf{NP} \neq \mathbf{ZPP}$, then there is no $|V|^{(1-\epsilon)}$ -approximation for any $\epsilon > 0$.*

An interesting special case of this problem is planar graphs. By the Four Color Theorem, such graphs are always 4-colorable. We can determine 1-colorability and 2-colorability in polynomial time. However, determining whether the chromatic number of a planar graph is 3 or 4 is **NP**-Complete.

8.7.5 Useful resources

The website <http://www.nada.kth.se/~viggo/problemlist/compendium.html> maintains a very useful list of many important problems and specifically what is currently known about how well these problems can be approximated. It includes information on general problems and special cases.

Chapter 9

Linear Programming

Linear Programming is a widely-used and very powerful technique. For example, some of the problems we have developed algorithms for in this course (such as the maximum flow problem) can be solved using Linear Programming.

9.1 Introduction

We will start with a specific example of Linear Programming and then use this to motivate the general form of the problem. This example is the Diet Problem: design a diet such that a daily nutritional requirement will be satisfied with minimum cost.

- input: an integer n , the number of different foods;
 an integer m , the number of required nutrients;
 an $m \times n$ matrix whose entries are a_{ij} , the amount of nutrient i in a single unit of food j ;
 an m -vector whose entries are b_i , the minimum daily requirement of nutrient i ; and
 an n -vector whose entries are c_j , the cost of a unit of food j .
- goal: Find the vector $x = \{x_1, x_2, \dots, x_n\} \in \mathfrak{R}^n$ that minimizes $\sum_{j=1}^n c_j x_j$, where x_j is the quantity of food j consumed per day. Note that this is equivalent to minimizing the cost spent per day on food.
- requirements: For each $i \in \{1, \dots, m\}$, $\sum_{j=1}^n a_{ij} x_j \geq b_i$, (which means that the amount of nutrient i consumed in a day is at least the minimum required) and
 for each $j \in \{1, \dots, n\}$, $x_j \geq 0$ (which means that the intake of each food items is non-negative).

We see that this formulation expresses our goal of spending as little money as possible, but still meeting the minimum daily nutritional requirements.

9.1.1 General Linear Programming problems

General Linear programming problems have the following form:

Let c , x , and a_i be vectors. Minimize $c \cdot x$, subject to the following constraints:

$$a_i \cdot x \geq b_i, \text{ or } a_i \cdot x = b_i, \text{ or } a_i \cdot x \leq b_i \text{ and}$$

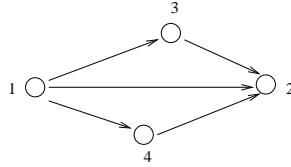


Figure 9.1: A simple graph

$$x_j \leq 0, \text{ or } x_j \text{ unconstrained, or } x_j \geq 0$$

An important requirement here is that each component of the vector x can be any real number. Note that if x can only take integer values, then the problem becomes NP-Complete. On the other hand, with real values, we can solve the problem in polynomial time.

9.1.2 Another example: Maximum flow

The familiar Max-Flow problem can be solved by stating it as a linear programming problem:

Input: A directed graph $G = (V, E)$,
 a matrix $C_{u,v}$ holding the edge capacities (0 if no edge from u to v), and
 source node s and sink node t .

We convert this to the following Linear Programming problem:

Variables: f_{uv} : flow from u to v for every pair of vertices (u, v) . We shall use f to denote the vector of all f_{uv} s.
 F : value of the flow.
 Objective function: Maximize variable F .
 Subject to constraints: capacity: $f \leq C$
 skew-symmetry: $f_{uv} + f_{vu} = 0$, and
 conservation of flow: $B \cdot f + F \cdot d = 0$.

We next explain the conservation of flow constraints. The rows of the matrix B represent vertices of G and the columns represent ordered pairs of vertices (u, v) such that $u \neq v$. For every column (u, v) of B , if $(u, v) \in E$ then row u is a $+1$, row v is a -1 , and all other rows are 0. If $(u, v) \notin E$ then that column contains only zeroes.

For example, for the graph depicted in Figure 9.1, the matrix B would be (all skipped columns are 0's):

B	(1, 2)	(1, 3)	(1, 4)	(2, 1)	...	(3, 2)	...	(4, 2)	...
1	+1	+1	+1	0	...	0	...	0	...
2	-1	0	0	0	...	-1	...	-1	...
3	0	-1	0	0	...	+1	...	0	...
4	0	0	-1	0	...	0	...	+1	...

and d is a $|V|$ -vector which has $+1$ at vertex t , -1 at vertex s and 0 everywhere else.

The columns of the matrix B and the entries of the vector f are ordered in such a way that the column of B corresponding to the pair (u, v) lines up with the entry of f corresponding to the flow from u to v . As a result, the constraint corresponding to each row i of the matrix B represents the conservation of flow constraint for vertex i . This is because the $+1$ entries cause the total flow out of a node to be a positive value, and the -1 entries cause the total flow into a node to be a negative value. For their sum to be 0 these quantities have to exactly cancel.

9.1.3 Standard Form of a Linear Program

In order to solve linear programs, we consider a specific form for expressing the problem.

Definition 107 *The Standard Form for Linear Programming is as follows:*

Minimize: $c \cdot x$
Subject to: $Ax = b$, and $x \geq 0$,

where: x is a n -vector, c is a n -vector, b is a m -vector, and A is a $m \times n$ matrix. We assume that the rows of A are linearly independent (since otherwise we can remove some constraints). Also, we assume that $m < n$ (if $m = n$ we can simply invert matrix A), and that a feasible solution exists.

Every Linear Programming problem can be converted to this form. To do so, we use the following conversion process:

- If $x_j \leq 0$ is one of the constraints then replace it with $x_j \geq 0$, and multiply all appearances of x_j by -1 . In other words, multiply the j th column of A by -1 and the j th entry of c by -1 .
- If x_j is unconstrained then replace all occurrences of it with $x'_j - x''_j$ and add constraints $x'_j \geq 0$ and $x''_j \geq 0$, where x'_j and x''_j are new variables.
- If $a_i x \leq b_i$ is one of the constraints then replace it with $a_i x + y = b_i$ and $y \geq 0$, where y is a new variable.
- If $a_i x \geq b_i$ is one of the constraints then use the same method: replace it with $a_i x - y = b_i$ and $y \geq 0$.

9.2 Feasible Solutions

Definition 108 *The set of feasible solutions of an LP problem is the set $F = \{\mathbf{x} \in \mathbb{R}^k \text{ such that } \mathbf{A} \cdot \mathbf{x} = \mathbf{b} \text{ and } x_i \geq 0, 1 \leq i \leq n\}$.*

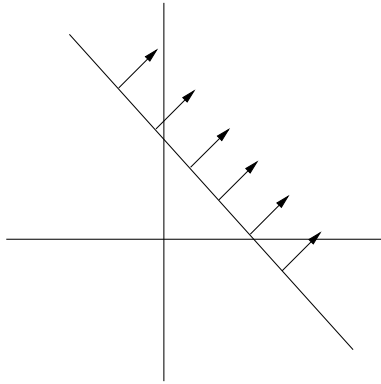
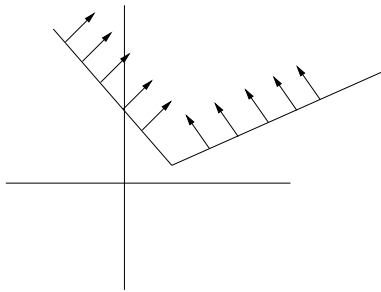
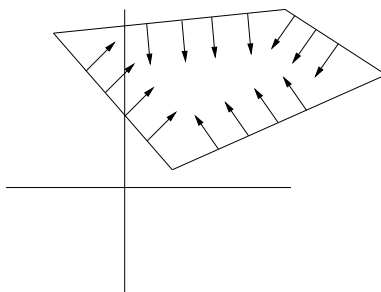
In order to solve the LP problem, we have to find the elements of the set F that minimize the optimization function $\mathbf{C} \cdot \mathbf{x}$; for the time being, we will ignore optimization and simply consider F . The following definitions are depicted in Figures 9.2, 9.3, and 9.4.

Definition 109 *A half-space in \mathbb{R}^k is a set of the form*

$$\{\mathbf{x} \in \mathbb{R}^k : \sum_{j=1}^k a_{ij} x_j \leq b_i\} \quad (9.1)$$

Definition 110 *A polyhedron in \mathbb{R}^k is the intersection of half-spaces.*

Definition 111 *A polytope is a bounded polyhedron.*

Figure 9.2: A half-space in \mathfrak{R}^2 Figure 9.3: A polyhedron in \mathfrak{R}^2 Figure 9.4: A polytope in \mathfrak{R}^2

9.2.1 F as a \mathfrak{R}^{n-m} polyhedron

Claim 112 *Any linear program in standard form with n variables and m equations has a feasible set that can be described as a polyhedron in \mathfrak{R}^{n-m} .*

Although the problem is given in standard form, to demonstrate this claim, we consider linear programs given in *canonical form*:

Given an $m \times n'$ matrix \mathbf{A} , an m -vector \mathbf{b} , and an n' -vector \mathbf{c} , find an n' -vector \mathbf{x} such that $\mathbf{c} \cdot \mathbf{x}$ is minimized and

$$\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b} \quad (9.2)$$

$$\mathbf{x} \geq \mathbf{0} \quad (9.3)$$

Note that $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ defines m half-spaces, and so the set of feasible solutions to a canonical form LP problem consists of the intersection of m half-spaces in $\mathfrak{R}^{n'}$. Thus, to prove the claim we only need to show that any LP problem given in standard form with n variables and m equations is equivalent to a LP problem in canonical form with $n - m$ variables and m equations.

Proof: Consider an LP problem in standard form. Assume that the m rows of \mathbf{A} are linearly independent—otherwise there are some redundant constraints and we can reduce the number of rows of \mathbf{A} . There must therefore be m linearly independent columns of \mathbf{A} ; assume that they are the first m ; we can ensure this through reordering of the variables. Then we can write \mathbf{A} as

$$\mathbf{A} = \left[\begin{array}{c|c} & \mathbf{B} \\ \hline & \end{array} \right] \quad (9.4)$$

where \mathbf{B} is an invertible $m \times m$ matrix.

After left-multiplying both sides of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ by \mathbf{B}^{-1} , we have $\mathbf{B}^{-1} \cdot \mathbf{A} \cdot \mathbf{x} = \mathbf{B}^{-1} \cdot \mathbf{b}$ or $\mathbf{A}' \cdot \mathbf{x} = \mathbf{b}'$, and the first m columns of \mathbf{A}' form the identity matrix \mathbf{I}_m :

$$\left[\begin{array}{c|c} & \mathbf{I}_m \\ \hline & \end{array} \right] \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ \vdots \\ b'_m \end{bmatrix} \quad (9.5)$$

Looking at a single row i in this equation we have

$$x_i + \sum_{j=m+1}^n a'_{ij} x_j = b'_i \quad (9.6)$$

for all i in $1 \dots m$. In order to convert the LP in standard form to canonical form, the “=” sign has to be converted into a “ \leq ”. We have assumed that $x_i \geq 0$ for $1 \leq i \leq n$; so for all i we can drop x_i from the above equation resulting in

$$\sum_{j=m+1}^n a_{ij} x_j \leq b'_i \quad (9.7)$$

for all i in $1 \dots m$ which is a linear programming problem in canonical form with the $n - m$ variables $x_{m+1} \dots x_n$ and m equations. Thus, the feasible set can be represented as the intersection of half-spaces in \mathfrak{R}^{n-m} . This completes the proof. ■

9.2.2 An Example

Consider the set of constraints

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 3 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.8)$$

$$\mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 6 \end{bmatrix} \quad (9.9)$$

giving a set of equations

$$\begin{array}{rcccccccl} x_1 & +x_2 & +x_3 & +x_4 & & & & = & 4 \\ x_1 & & & & +x_5 & & & = & 2 \\ & & x_3 & & & +x_6 & & = & 3 \\ & 3x_2 & +x_3 & & & & +x_7 & = & 6 \end{array} \quad (9.10)$$

It is easy to convert these constraints into canonical form; since the 4 rightmost columns of \mathbf{A} already form an identity matrix, we only need to remove x_4 from the first equation, x_5 from the second, and so on, to get a set of equations in 3 variables:

$$\begin{array}{rcccccl} x_1 & +x_2 & +x_3 & \leq & 4 \\ x_1 & & & \leq & 2 \\ & & x_3 & \leq & 3 \\ & 3x_2 & +x_3 & \leq & 6 \end{array} \quad (9.11)$$

The solutions to these equations form a polyhedron in \mathfrak{R}^3 . Note that since $n = 7$ and $m = 4$ this is \mathfrak{R}^{n-m} . Each constraint defines a half-space in \mathfrak{R}^{n-m} . Figure 9.5 shows the polyhedron defined by the intersection of these half spaces, which in this case is a polytope. The feasible set F is the set of points in the polytope.

Consider any point of this polytope. An important aspect of this representation to keep in mind is that any such point corresponds to a specific value for all n of the original variables. In this example, it is clear that the point defines x_1, x_2 and x_3 . Furthermore, once those three variable are defined, the remaining four variables are also defined. For example, $x_4 = 4 - x_1 - x_2 - x_3$.

Thus, we have two different ways to view the linear programming constraints: algebraically, i.e., in terms of the equations as represented by (9.10) and (9.11), or geometrically, i.e., in terms of the polyhedron as represented by Figure 9.5. These two interpretations are equivalent; understanding both as well as the correspondence between the two will be central to our understanding of linear programming.

To further develop this relationship, note that in the polyhedron, there is a plane corresponding to each of the n variables being zero, so n planes form the boundary of the polytope. Clearly the origin planes represent one of $x_1 \dots x_3$ being zero. If we look at the original set of equations, we can see that the other planes represent the values of $x_1 \dots x_3$ required to satisfy a row of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ when one of the $x_4 \dots x_7$ is set to 0. So the plane $x_3 = 3$ represents the solutions to the equation $x_3 + x_6 = 3$ when $x_6 = 0$, $3x_2 + x_3 = 6$ represents the solutions to the equation $3x_2 + x_3 + x_7 = 6$ when $x_7 = 0$, and so on.

From this, we see that a vertex in a feasible polyhedron can be described both geometrically and algebraically as follows:

Geometrical description: A vertex is an intersection of at least $(n - m)$ hyperplanes. A vertex is called *degenerate* if it is an intersection of more than $(n - m)$ hyperplanes. For example, in three-dimensional space

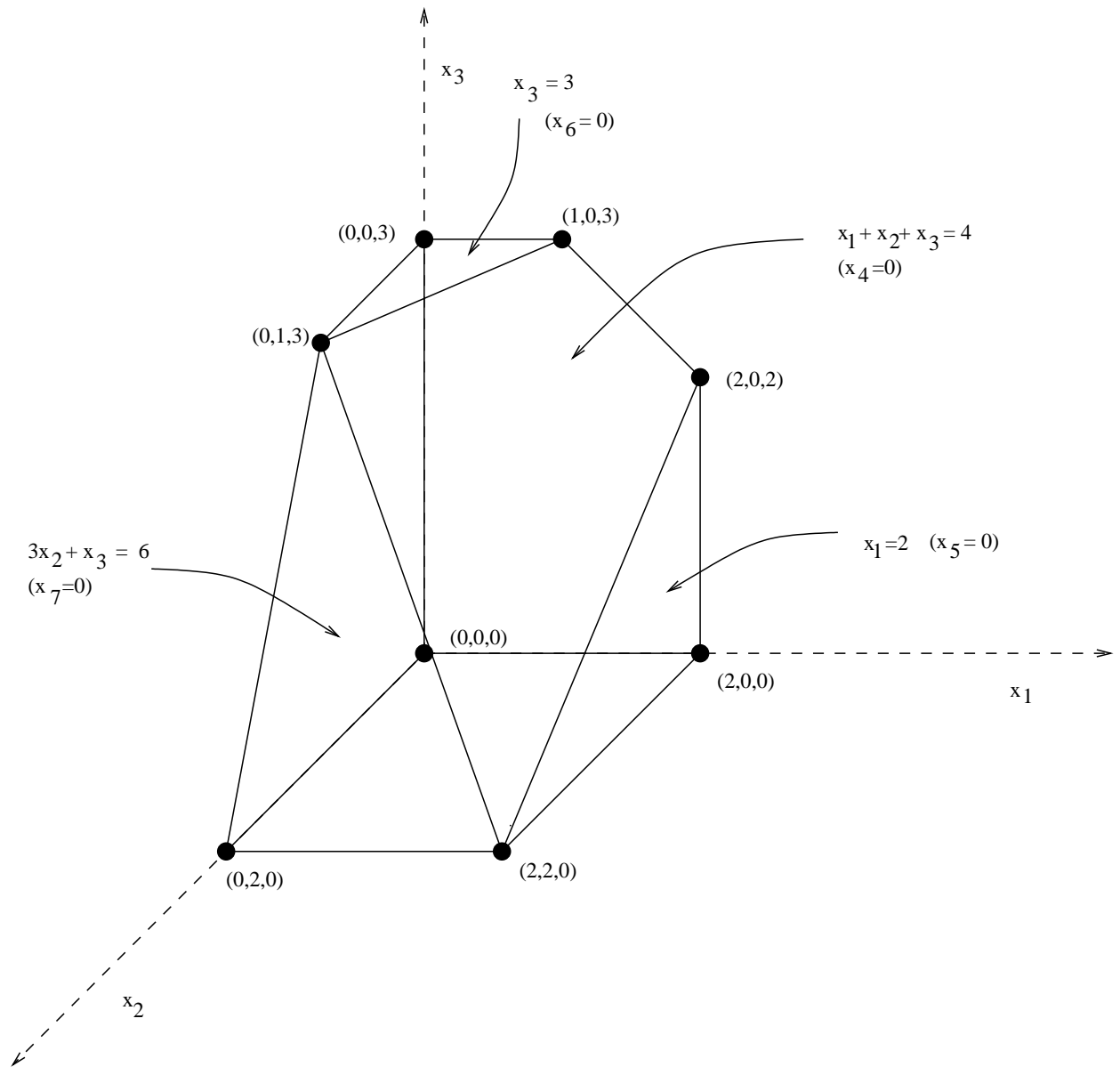


Figure 9.5: Example Polyhedron

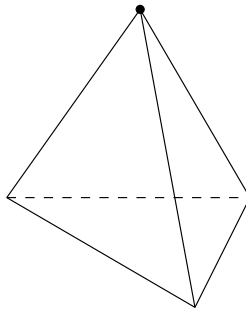


Figure 9.6: Non-Degenerate vertex

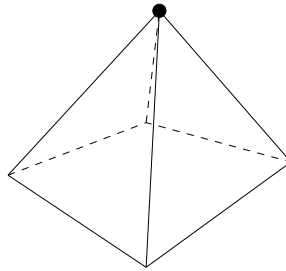


Figure 9.7: Degenerate vertex

($n - m = 3$), a pyramid shaped polyhedron has a degenerate vertex in its apex, which is an intersection of 4 faces of the pyramid (see Figures 9.6 and 9.7). In what follows we will assume that all vertices are *non-degenerate* for simplicity.

Algebraic description: Since a hyperplane corresponds to a variable being set to 0, a vertex corresponds to a feasible solution in which at least $(n - m)$ variables are set to zero. A non-degenerate vertex has exactly $n - m$ variables set to zero.

Now that we have some understanding of the feasible set F , we return to our original optimization problem.

9.3 Finding the Optimal Value

Lemma 113 *If F is a polytope, $\mathbf{c} \cdot \mathbf{x}$ is minimized at a vertex of the polytope; if F is not a polytope, either $\mathbf{c} \cdot \mathbf{x}$ is minimized at a vertex, or $\mathbf{c} \cdot \mathbf{x}$ can be made arbitrarily small.*

When $\mathbf{c} \cdot \mathbf{x}$ can be made arbitrarily small, the problem is called an unbounded problem.

Instead of a formal proof, we here only give an intuitive argument as to why this is the case. Let's consider an LP problem in two-dimensions whose objective function is $\mathbf{c} \cdot \mathbf{x}$. Any equality of the form $\mathbf{c} \cdot \mathbf{x} = d$, where d is a constant, is a line in 2D space, and varying d produces parallel lines. As we decrease the value of d the line moves in a preferred direction as shown in Figure 9.8. As we move the line in the direction such that $\mathbf{c} \cdot \mathbf{x}$ is minimized, the last point or set of points in the polyhedron that we see is either a vertex or an edge that is parallel to the line $\mathbf{c} \cdot \mathbf{x} = d$. If there is such a parallel edge (as in Figure 9.9), the set of points still contains at least one vertex. Therefore, the minimum is still achieved at a vertex.

We can imagine an analogous situation in three dimensions, in which we move a plane through three di-

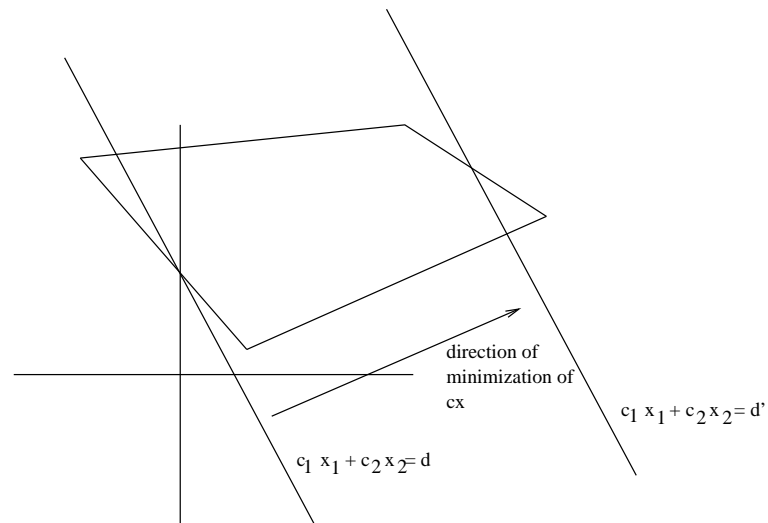


Figure 9.8: Optimization on a 2D Feasible Set

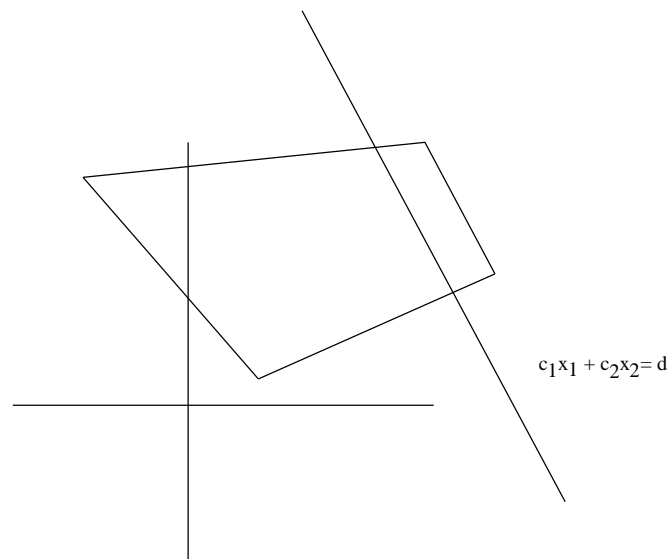


Figure 9.9: Optimization of a 2D Feasible Set - Special Case

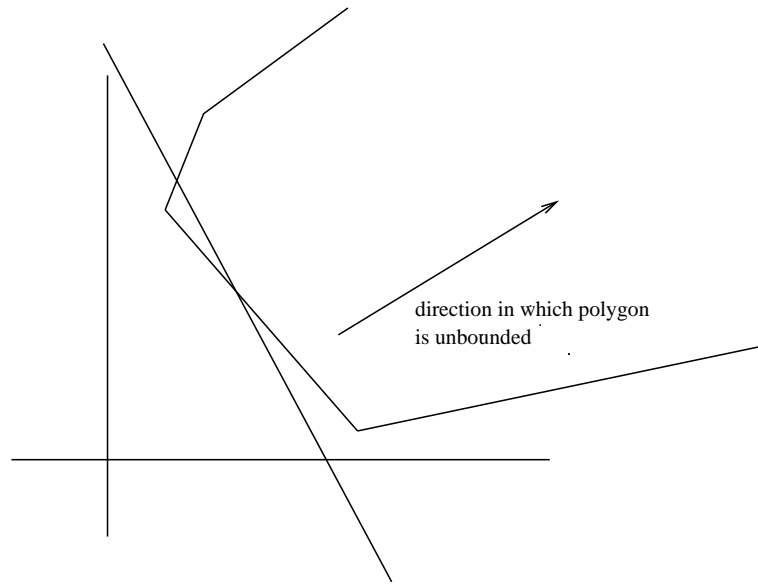


Figure 9.10: Optimization of a non-Polytope Feasible Set

mensional space; we might have a line or a plane parallel to the moving plane, but we will have at least one vertex in the set of points which minimizes $\mathbf{c} \cdot \mathbf{x}$. The same is true for higher dimensions as well.

If F is not a polytope (as in Figure 9.10) then, in some cases, it is possible to move the line indefinitely in the direction of optimization, if the polyhedron is unbounded in this direction. If the polyhedron is bounded in the direction of optimization, then $\mathbf{c} \cdot \mathbf{x}$ is minimized at a vertex. Therefore, for the LP problem to be unbounded it is necessary, but not sufficient, for F to be unbounded.

9.3.1 A simple algorithm

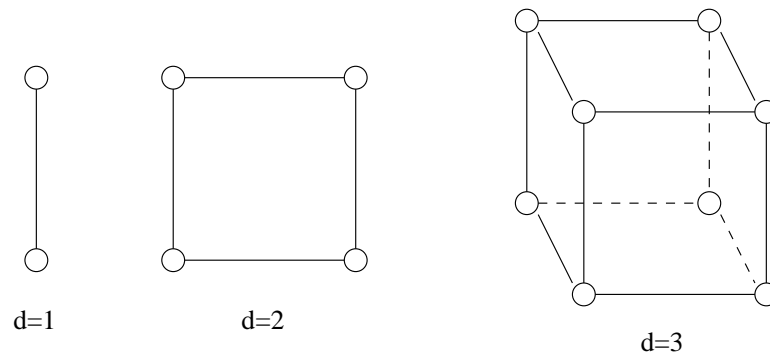
Lemma 113 suggests the following simple algorithm for finding the optimal solutions to an LP problem:

1. For each vertex \mathbf{x} of F
2. Compute $\mathbf{C} \cdot \mathbf{x}$
3. Return minimum value found.

This algorithm will give us a correct answer but has exponential running time. The number of vertices to be considered can be exponential in the degree $n - m$. To illustrate this, let's consider the example of a d -dimensional cube (Figure 9.11). For $d = 1$, we have a single face (in this case, an edge) and two vertices. For $d = 2$, we have a square with 4 faces and 4 vertices. In a three-dimensional cube, we have 6 faces and 8 vertices. Thus, we notice that on the addition of each dimension, the number of vertices increases by a factor of 2, and therefore is exponential in d .

9.3.2 Simplex algorithm

1. Start at some vertex \mathbf{x}

Figure 9.11: d -dimensional hypercubes

2. Repeat:
 3. Find a neighboring vertex \mathbf{y} (i.e., a vertex reachable from \mathbf{x} via an edge of the polyhedron) such that $\mathbf{C} \cdot \mathbf{y} < \mathbf{C} \cdot \mathbf{x}$
 4. $\mathbf{x} = \mathbf{y}$
 5. until no such \mathbf{y} is found
 6. Return \mathbf{x} (or unbounded)

This algorithm returns the correct answer because polyhedron F is convex (since it is the intersection of half-spaces). This means that if a vertex is locally optimal, it is globally optimal as well. Furthermore, it is guaranteed to examine each vertex at most once, since the cost function is strictly increasing during the course of the algorithm. Its running time will be dependent on the number of vertices that are visited.

In practice, this algorithm works very well (often as fast as linear time) for many pivot rules (rules according to which the vertex \mathbf{y} is chosen when there is more than one vertex matching the condition in step 3). However, for each of the standard pivot rules, it is possible to construct a pathological example for which this algorithm runs in exponential time.

Representing Vertices

In order to actually implement the Simplex algorithm, we take a closer look at what vertices really represent in terms of the algebraic interpretation of the linear programming problem. Recall that a vertex is the intersection of at least $m - n$ hyperplanes. For simplicity, we here assume that our vertices are the intersection of exactly $n - m$ hyperplanes and will not deal with the degenerate case of an intersection of more than $n - m$ hyperplanes. Recall that a hyperplane is a set of points in the feasible region where one of the variables of \mathbf{x} has been set to zero. Thus, a vertex has the property that $n - m$ variables of \mathbf{x} are set to zero, or equivalently, that there are m non-zero variables, which we refer to as $x_{B(1)}, x_{B(2)}, \dots, x_{B(m)}$.

Note that if we have some subset of our variables in \mathbf{x} set to zero then we only need concern ourselves with the columns of A corresponding to the non-zero variables: $A_{B(1)}, A_{B(2)}, \dots, A_{B(m)}$. Also note that there are $\binom{n}{m}$ ways to pick m of our n variables to be non-zero, but not all of the $\binom{n}{m}$ possibilities will correspond to actual vertices of our polyhedron. Stating the same thing in terms of the geometric interpretation of the problem, it is not the case that $n - m$ hyperplanes always define a vertex of the polyhedron. There are two reasons why this might occur:

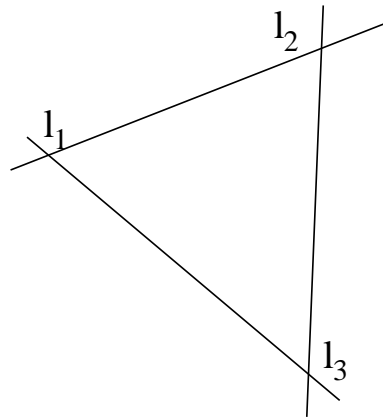


Figure 9.12: Three planes in \mathcal{R}^3 that do not intersect at a point.

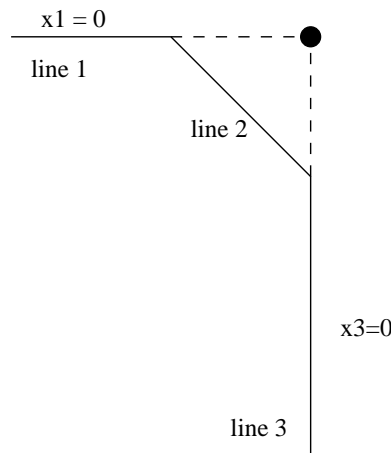


Figure 9.13: Intersection of hyperplanes not a vertex

1. The hyperplanes either intersect in an infinite number of points or never intersect at all (e.g. parallel lines in \mathcal{R}^2).

Example. In three dimensions, consider the projection of three planes in \mathcal{R}^3 depicted in Figure 9.12. Each point of intersection corresponds to a shared line: l_1, l_2 , and l_3 . In \mathcal{R}^3 , these lines are perpendicular to this piece of paper.

The algebraic interpretation of this is that the columns of A corresponding to the non-zero variables are not linearly independent.

Definition 114 A basic solution is a set of m non-zero variables such that the corresponding columns are linearly independent.

2. The corresponding hyperplanes may intersect, but the intersection may occur outside of the feasible region. In this case we call the intersection an “infeasible solution”.

Example. Figure 9.13 shows a polyhedron in \mathcal{R}^2 . If x_1 and x_3 are set to zero, we get the point that is the intersection of lines 1 and 3. However, this point is outside the feasible set.

The algebraic interpretation of this is that the point where the hyperplanes intersect (i.e., when we solve for x), some of the coordinates of x are negative.

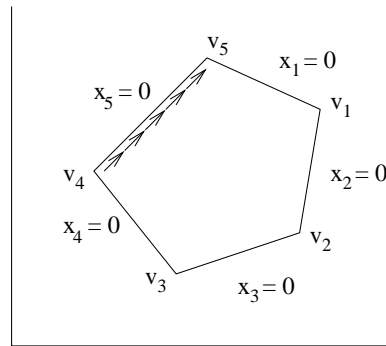


Figure 9.14: walking from vertex to vertex

Definition 115 A basic feasible solution (BFS) is a basic solution that lies in the feasible region.

A basic feasible solution corresponds to a vertex in our polyhedron. Furthermore, we see the following:

Claim 116 A BFS is uniquely specified by a set of m non-zero variables.

Proof: Let $\mathbf{B} = [\mathbf{A}_{\mathbf{B}(1)} \mathbf{A}_{\mathbf{B}(2)} \dots \mathbf{A}_{\mathbf{B}(m)}]$. \mathbf{B} is called the basis \mathbf{B} for the BFS. Let $\hat{\mathbf{x}} = \{\mathbf{x}_{\mathbf{B}(1)}, \mathbf{x}_{\mathbf{B}(2)}, \dots, \mathbf{x}_{\mathbf{B}(m)}\}$. Since all the x_i not in $\hat{\mathbf{x}}$ are zero, we have: $\mathbf{A} \cdot \mathbf{x} = \mathbf{B} \cdot \hat{\mathbf{x}} = \mathbf{b}$. Since we have a BFS, the columns of \mathbf{B} are linearly independent. Thus \mathbf{B} must have an inverse, and we can solve for $\hat{\mathbf{x}}$:

$$\hat{\mathbf{x}} = \mathbf{B}^{-1} \cdot \mathbf{b} \quad (9.12)$$

Thus, from Equation 9.12, we can determine the values of the m non-zero variables. Since the remaining variables are all set to zero, the values for all n variables for a BFS can be obtained from the set of non-zero variables. ■

This gives us another interpretation of linear programming. Out of all possible $\binom{n}{m}$ choices of m non-zero variables, we restrict ourselves to those where (a) the corresponding set of columns of A are linearly independent, and (b) when we use these columns to solve for x , the values of x that we obtain are all non-negative. Out of these sets of choices, we want to find the choice that minimizes the objective function.

To start the Simplex algorithm, we must find a set of m variables that gives us a BFS. Finding linearly independent columns is not difficult: we can do this in a greedy fashion. However, ensuring that the solution is feasible (i.e., that the values of x are non-negative) is harder, and we don't want to look at all $\binom{n}{m}$ possibilities, since this would be exponentially many. However, once we have a starting vertex, it is much easier to move from that vertex to find a neighboring vertex. We shall discuss that problem first, and then return to the problem of finding an initial vertex.

Finding a Neighboring Vertex

Assuming we can find a BFS we want a method for moving from that vertex to a neighboring vertex in a high-dimensional space.

Example. Consider the polyhedron in \mathfrak{R}^2 depicted in Figure 9.13, where $n = 7$ and $m = 5$. We wish to move from v_4 to v_5 . v_4 corresponds to the constraints: $x_4 = 0$ and $x_5 = 0$. The idea is to relax the constraint

$x_4 = 0$. To do so, we gradually increase x_4 until some other variable (in this case x_1) gets set to 0. This corresponds to starting at v_4 and walking along the edge (v_4, v_5) of our polyhedron until we reach v_5 .

We now generalize to a higher-dimensional space. We start with a set of m non-zero variables. We want to exchange a new variable that was initially set to zero with one of our non-zero variables. As in the example above, we increase our new variable until one of our original non-zero variables becomes zero. Because we require that all of our x_i are greater than zero, we can be sure that by increasing a variable we will move in the correct direction.

Let $x_{B(1)}, x_{B(2)}, \dots, x_{B(m)}$ be our nonzero variables and let $A_{B(1)}, A_{B(2)}, \dots, A_{B(m)}$ be their corresponding columns. Let x_j be the new variable we want to exchange and let A_j be its corresponding column. Then given our constraint $A\mathbf{x} = \mathbf{b}$ we know

$$\sum_{i=1}^m x_{B(i)} A_{B(i)} = \mathbf{b}. \quad (9.13)$$

Since the $A_{B(i)}$ s are m linearly independent columns with m entries each, they form a basis for \mathfrak{R}^m . Thus we can express A_j as a linear combination of these columns:

$$A_j = \sum_{i=1}^m t_{ij} A_{B(i)}, \quad t_{ij} \in \mathfrak{R}. \quad (9.14)$$

Adding and subtracting λA_j from (9.13) we get:

$$\sum_{i=1}^m x_{B(i)} A_{B(i)} + \lambda A_j - \lambda \sum_{i=1}^m t_{ij} A_{B(i)} = \mathbf{b}. \quad (9.15)$$

Rearranging terms in (9.15) we get:

$$\lambda A_j + \sum_{i=1}^m (x_{B(i)} - \lambda t_{ij}) A_{B(i)} = \mathbf{b} \quad (9.16)$$

Increasing the parameter λ corresponds to moving along an edge of the polyhedron. As we increase λ , we add in more of the variable x_j , and as a result we also change the other variables to ensure that the constraint $A\mathbf{x} = \mathbf{b}$ is still satisfied. As we increase λ , if there is some t_{ij} that is positive, then for some i , $x_{B(i)} - \lambda t_{ij}$ will eventually be 0. This corresponds to reaching another vertex along the edge of travel. Define

$$\lambda_0 = \min_{\{i \mid t_{ij} > 0\}} \left\{ \frac{x_{B(i)}}{t_{ij}} \right\}. \quad (9.17)$$

λ_0 will be the first value of λ at which $x_{B(i)} - \lambda t_{ij}$ becomes zero. Note that if there is a vertex in the direction of movement, then we will have $\{i \mid t_{ij} > 0\}$ nonempty, and if all $t_{ij} < 0$, then the problem is unbounded in that direction.

Let l be the value of i that achieves the minimum in (9.17). Then x_l is the variable that gets set to zero first. In our new solution we will replace x_l with x_j , which corresponds to setting $B(l) = j$. We can summarize this process with the following equation:

$$x'_{B(i)} = \begin{cases} \lambda_0, & \text{if } i = l \\ x_{B(i)} - \lambda_0 t_{ij}, & \text{if } i \neq l \end{cases} \quad (9.18)$$

Finding an initial vertex

Here we go back to the problem of finding the initial vertex so that we can proceed with the main iteration. We want to find some point \mathbf{x} that satisfies $A\mathbf{x} = \mathbf{b}$ which has m non-zero variables. For this step, we do not care about minimizing the optimization criteria. Consider the following approach. First, recall the original problem in standard form:

$$\begin{aligned} \text{Objective Function:} & \quad \min \mathbf{c} \cdot \mathbf{x} \\ \text{Subject to:} & \quad A\mathbf{x} = \mathbf{b} \\ & \quad \mathbf{x} \geq 0 \end{aligned}$$

Now, let's consider an alternative linear programming problem defined as follows:

$$\begin{aligned} \text{Objective Function:} & \quad \min y_1 + y_2 + \dots + y_m \\ \text{Subject to:} & \quad A\mathbf{x} + \mathbf{y} = \mathbf{b} \\ & \quad \mathbf{y} \geq 0 \text{ and } \mathbf{x} \geq 0 \end{aligned}$$

where \mathbf{y} is a vector of length m . In this new problem, finding an initial BFS is trivial: $\mathbf{x} = 0$, $\mathbf{y} = \mathbf{b}$. Thus, we can easily start the Simplex algorithm for this new problem, and find an optimal solution to it.

The important thing to note here is that if there is an optimal solution to the alternative problem that achieves cost 0, it must be the case that $\sum_{i=1}^m y_i = 0$. Since that point must have m non-zero variables, these m non-zero variables must be variables of \mathbf{x} . Thus, the point that serves as the optimal solution to the new problem provides a BFS solution to the original problem. On the other hand, if there is no solution for the alternative problem that has cost 0, then there is no BFS for the original problem.

9.3.3 Running Time Analysis

Despite intense study, the running time of the Simplex algorithm has not been completely understood. We do know that a move between two vertices can be performed in $O(mn)$ time (linear in the input size). This is accomplished with the use of a data structure called a **tableau**[K91]. Thus, the total running time of the algorithm is $O(mn)$ times the number of traversed vertices.

The number of traversed vertices depends on the pivot rule. The pivot rule is the method used to decide which edge to traverse, given the choice between multiple possible edges. There are various alternatives for pivot rules that one can consider. Here are some of them:

- We could choose the edge that gives us the steepest gradient in the cost function.
- We could choose the edge that gives us the largest total change.
- We could choose randomly from possible candidates.

It turns out that these alternatives are not the best thing to do in some special cases. Klee and Minty [KM72] showed that in the case of a perturbed hypercube (also known as the Klee-Minty polytope), Simplex with

the first pivot rule visits all vertices even though the cost function is monotonically decreasing throughout the course of the Simplex algorithm. For most natural pivot rules, there are example inputs that are known to require exponential time.

An open question remains: Is there an efficiently computable pivot rule such that Simplex is guaranteed to run in polynomial time?

Despite all the debate about the running time, in practice, the Simplex algorithm is very efficient and is widely used in many mathematical or data analysis packages. Very recent work on trying to explain the discrepancy between the worst case running time and why Simplex is so efficient in practice can be found in [ST2001].

We have not shown that linear programming is in P , but this is the case. The first polynomial time algorithm was introduced by Khachian [K79], which uses the **ellipsoid method**. Despite its theoretical importance, this algorithm has a high degree polynomial running time, and is generally considered impractical. A better alternative was later introduced by Karmarkar [K84], which uses the **interior-point method**. For further information on Simplex and these related algorithms, please see: [PS82],[S88], or [K91].

9.4 Duality

We next cover the concept of *duality*. The dual of a problem is a very closely related problem such that the solutions to the two problems are tightly coupled. An example of this we saw earlier in the course is the Max-Flow Min-Cut Theorem, where we saw that the maximum flow in a graph between a pair of vertices is equal to the minimum cut between those two vertices. This is an important concept in the design of algorithms, and has a number of uses. In particular, it can be used for:

1. Designing polynomial time algorithms for a problem.
2. Proving the optimality of a solution (as we did for the Max-Flow problem).
3. Finding the solution to a linear program efficiently.

Recall that any linear program with m constraints over n variables can be expressed in *canonical form* as follows:

$$\begin{array}{ll} \text{Minimize} & \mathbf{C} \cdot \mathbf{x} \\ \text{such that} & \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \quad (9.19)$$

This is known as the original, or *primal*, representation of the problem. For each such LP, there is also a related *dual* linear program, which has the following form:

$$\begin{array}{ll} \text{Maximize} & \mathbf{y} \cdot \mathbf{b} \\ \text{such that} & \mathbf{y} \cdot \mathbf{A} \leq \mathbf{C} \\ & \mathbf{y} \geq \mathbf{0} \end{array} \quad (9.20)$$

Observe that this is a just a transformation of the original problem 9.19, with the following modifications:

1. We maximize instead of minimize.

2. We exchange the roles of C and b
3. The constraints are “ \leq ” instead of “ \geq ”.
4. The vector y that is being solved for has m components and n constraints, instead of the vector x that is used for the original problem, which has n components and m constraints.
5. yA is computed in place of Ax .

It turns out that the dual of the dual is the original primal. We leave it as an exercise to show this.

We show shortly that the optimal answer (if one exists) for both problems is the same. This characteristic is exploited by *primal-dual* algorithms, which run both the primal and dual versions of a problem simultaneously and uses the solution from the first of these two to complete. This can avoid some instances where the input would be slow to converge on the optimal solution in one of the two problems.

9.4.1 Vitamin Seller’s Problem

Recall the original diet problem from lecture 23:

- x_i is the amount of food i to consume.
- Cx is the cost of the diet, which should be minimized.
- $Ax \geq b$ represents the nutritional constraints of the diet.
- $x \geq 0$ specifies that the diet must consist of non-negative food quantities.

The objective is to find a diet satisfying the specified nutritional goal while minimizing the cost incurred by that diet. This problem has a dual which we will call the Vitamin Seller’s Problem. This dual describes the nutrient problem from the perspective of a seller of vitamins who wishes to maximizing the cost of the vitamins being sold, while staying competitive with the cost of buying the nutrients in food form. This new problem is defined as follows:

- y_i is the selling price of nutrient i .
- yb is the total cost of the daily required nutrients.
- $yA \leq C$ represents the cost constraints.
- $y \geq 0$ specifies the cost must be non-negative.

The objective function is the cost of the required nutrients, and we wish to maximize this cost. The cost constraints represent the fact that the price of each vitamin must be set to be competitive with the cost of real food. In particular, the cost of selling the vitamins in a given food item must be no more expensive than actually buying that food item.

9.4.2 Weak and Strong Duality

Note that in the optimal solution to the vitamin sellers problem, the cost of buying those vitamins can be no more expensive than satisfying the equivalent vitamin requirements by consuming real food. This is actually part of a general property of duality, expressed in the following theorem:

Theorem 117 Weak Duality. *If x_0 and y_0 are feasible solutions for the primal and dual problems, then $Cx_0 \geq y_0b$.*

Proof: First, we can use the fact that $y_0 \geq 0$,

$$\begin{aligned} Ax &\geq b \\ y_0 Ax_0 &\geq y_0 b \end{aligned}$$

Similarly, using $x_0 \geq 0$,

$$\begin{aligned} yA &\leq C \\ y_0 Ax_0 &\leq Cx_0 \end{aligned}$$

And therefore,

$$Cx_0 \geq y_0 Ax_0 \geq y_0 b$$

■

Furthermore, it is possible to make a stronger claim about duality.

Theorem 118 Strong Duality. *Exactly one of the following is true:*

1. *The primal and dual have finite optima which are equal*
2. *The primal is unbounded, and the dual is infeasible*
3. *The primal is infeasible, and the dual is unbounded*

The proof of the Strong Duality theorem will not be covered in this course due to lack of time, but may be found in [PS82]. Note that this theorem supports the primal-dual algorithm strategy described in the previous section, because a solution from either the primal or dual can be used to infer the solution to the other. Next, we will present the dual of a familiar problem, Max-Flow.

9.4.3 Max-Flow Linear Programming Dual

Consider the following linear programming problem for the network flow problem:

Variables:	u_i for each vertex
	w_{ij} for each edge
Objective function:	minimize $\sum_{ij} c_{ij} w_{ij}$ where c_{ij} are the edge capacities of the input graph
Subject to constraints:	$\forall i, j \ w_{ij} \leq u_j - u_i$
	$u_t - u_s \geq 1$
	$w_{ij} \geq 0$

Exercise: show that this is the dual of the linear program described as equivalent to the Max-Flow problem.

Based on our previous experience with this problem, our intuition is that the dual of the Max-Flow problem should correspond to the Min-Cut problem. This is in fact the case.

Claim 119 *The optimal solution of the dual of the Max-Flow problem corresponds to the Min-Cut problem.*

Although not shown here, the basic idea is that the optimal assignment of our variables will correspond to a minimum cut. In particular, there is an optimal solution of the following form: