6.1 Finding the Minimum Cut

Given an undirected graph \( G = (V, E) \), \( \{A, B\} \) is called a partition of \( G \) if \( A, B \) are nonempty subsets of \( V \), \( A \) and \( B \) are disjoint, and \( A \cup B = V \). \( A \) and \( B \) are called partition classes. Given a partition, a cut \( K \) is defined to be the subset of edges such that each edge connects vertices of different partition classes. Mathematically, \( K = \{\{u,v\} \in E : u \text{ and } v \text{ are in distinct partition classes of some given partition} \} \) The min cut is the cut that has the fewest elements (edges) among all the cuts of the partitions of 2 subsets of \( G \). Note that the min cut is not unique in general.

Karger’s min cut algorithm provides a quick method of finding the minimum cut of a undirected graph with some significant probability of returning the correct answer.

6.1.1 The Algorithm

Step 1: Pick a random edge \((u, v)\).

Step 2: Collapse the two nodes \(u\) and \(v\) (See Figure 6.1)

Step 3: Repeat step one and step two until only two nodes are left.

Step 4: The edges that are left are the min cut.

6.1.2 Pseudo-code

Let \( G = (V, E) \) be the graph, where \( s \) is the starting node and \( t \) is the ending node.

```python
while (V != empty) {
    pick \((u, v)\) randomly from \(E\)
    add the list of edges of \(v\) to the list of edges of \(u\), update \(E\)
    delete all edges of \(v\) from \(E\)
    delete node \(v\) from \(V\)
}
return size of \(E\)
```

6.1.3 Analysis

Remember that we defined \( K \) to be the set of the edges in the min cut. The algorithm gives the correct answer if none of the edges picked for collapse were in \(K\).
Theorem 6.1  Let \( n \) be the number of vertices and \( m \) be the number of edges in the graph. Karger’s algorithm finds the min cut with a probability greater than or equal to \( \frac{2}{n(n+1)} \).

Proof:

Suppose the min cut has \( c \) edges. The probability the algorithm picks one of the \( c \) edges in the first iteration is \( \frac{c}{m} \). Let \( d \) be the degree of the graph (i.e., the minimum number of edges of a vertex in the graph). Then \( m \geq n \times d/2 \). Also, supposed vertex \( u \) has \( d \) edges. Then the partition at \( u \) has the cut of \( d \) elements. Since the min cut \( c \) has the least number of edges of any cut, \( d \geq c \). Thus, \( m \geq n \times d/2 \geq n \times c/2 \). So, \( \frac{c}{m} \leq \frac{c}{2n} \).

Therefore, the probability one of the min cut edges is not picked in the first round is greater than or equal to \( \frac{n-2}{n} \).

Following a similar calculation, the probability the algorithm avoids picking a min cut edge in the second round is greater than or equal to \( \frac{n-3}{n-1} \).

The probability the algorithm does not pick a min cut edge in the third round is greater than or equal to \( \frac{n-4}{n-2} \).

Therefore, the probability that the algorithm never picks a min cut edge is greater than or equal to

\[
\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \tag{6.1}
\]
which equals
\[
\frac{2}{n(n-1)} \tag{6.2}
\]
after some canceling of like terms in the numerator and denominator.

The total running time of Karger’s algorithm is \(O(n^3)\) (because the algorithm picks \(n\) edges and collapses them in \(n^2\) time). With some clever implementation tricks, the running time can be reduced to \(O(n^2)\).

### 6.2 Linear Programming

In a linear programming problem in standard form, we seek to maximize (or minimize) \(c_1 x_1 + c_2 x_2 + c_3 x_3 + \ldots + c_n x_n\), with fixed \(c_i\) being real numbers, in the set of all \((x_1, x_2, x_3, \ldots, x_n)\) that satisfies the following set of constraints:

\[
\begin{align*}
    a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n &= b_1 \\
    a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n &= b_2 \\
    \quad \ldots \\
    a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n &= b_m 
\end{align*}
\]

and \(x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0\). \(a_{ij}\) are all real numbers.

#### 6.2.1 Example: Factory Production

First, we will consider a slightly different problem: maximize \(c_1 x_1 + c_2 x_2 + c_3 x_3 + \ldots + c_n x_n\), with \(c_i\) being positive real numbers, under the following constraints:

\[
\begin{align*}
    a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n &\leq b_1 \\
    a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n &\leq b_2 \\
    \quad \ldots \\
    a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n &\leq b_m \\
    x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0
\end{align*}
\]

All \(a_{ij}\) are nonnegative real numbers.

Let’s try to solve a factory production problem using this form of linear programming. Let \(x_1, x_2\) be the number of items of product I and II respectively. Suppose product I and II are sold at the price of 1 and 2
Figure 6.2: By graphing the solution set of the linear program in the factory example (Section 6.2.1), we can easily find the optimum. We graph the regions \( x_1 \leq 200 \) and \( x_2 \leq 300 \), then the region \( x_1 + x_2 \leq 400 \). The revenue \( x_1 + 6x_2 \) is optimized when \( x_1 = 100 \) and \( x_2 = 300 \).

per unit, respectively. The revenue is \( x_1 + 6x_2 \). Suppose each unit of product I requires one unit of resource A and one unit of resource C, and each unit of product II requires one unit of resource B and one unit of resource C. We have 200, 300, and 400 units of resource A, B, and C. We seek to maximize the revenue.

This problem can be written in this mathematical form:

\[
\begin{align*}
\text{max} \quad & x_1 + 6x_2 \\
\text{s.t.} \quad & x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 \leq 400 \\
& x_1, x_2 \geq 0
\end{align*}
\]

This problem can be solved pretty easily just by graphing the solution set on the \( \mathbb{R}^2 \) plane (Figure 6.2). However, we might encounter similar problems with more variables and more constraints, which are not as easily solved by graphing. Consider this problem:

\[
\begin{align*}
\text{max} \quad & x_1 + 6x_2 + 14x_3x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 + x_3 \leq 400 \\
& x_2 + 3x_3 \leq 600 \\
& x_1, x_2 \geq 0
\end{align*}
\]

Observations:
Assume that there are feasible solutions and that this set of solutions is bounded. By the fundamental theorem of linear programming, the optimum points occur at corners.

A corner is a point where \( n \) constraints (inequalities or \( x_i \geq 0 \)) are tight. To find a corner, we pick \( n \) constraints to be tight. Now we have \( n \) equalities and \( n \) unknowns. Since there are points that satisfy all the constraints, there must be at least one solution for the \( n \) unknowns.

(Note that we also want the solution to those \( n \) equalities to be unique – i.e., we want the matrix of these \( n \) equalities to be non-singular. If we end up a non-singular matrix, or a set of non-unique solutions, we can simply add extra constraints until we obtain a unique solution. We will always be able to do so because we are assuming that the feasible solutions are bounded. If we have a non-unique solution, we can easily show that this set is in fact a non-trivial affine space which isn’t bounded, and that there is no point that achieves the maximum, because the maximum is essentially infinity).

**Simplex method:**

Observe that a corner is an optimal solution iff the value we obtain at that corner is greater than or equal to the values achieved by all the adjacent corners. (An adjacent corner shares \( n - 1 \) equalities with the original corner.) The polytopes of feasible solutions are convex, so every non-optimal corner has a greater-valued adjacent corner. (In other words, only the local optima are also global optima.) Furthermore, there are finitely many inequalities and thus finitely many corners. Thus, we can start at a corner, and check its adjacent corners. If we find an adjacent corner that achieves a greater value, we move on to this corner and check its adjacent corners. Since there are only finitely many corners and at least one of them must achieve the maximum value, this algorithm will terminate with the corner that is an optimum solution to this maximizing problem.

Also observe that if we assume that all the coefficients are nonnegative, the origin \((0,0,\ldots,0)\) satisfies all the constraints, so we can always start our algorithm at the origin.

**Algorithm:**

**Step 1:** Initialize \( x \) to be the origin (\( x \) is a vector with values of \( x_1, x_2, \ldots, x_n \))

**Step 2:** Check the value of the adjacent corners of \( x \)

**Step 3:** If one of the corners has value greater than \( x \)'s value, let \( x \) be that corner, go back to step 2. If not, return \( x \).

### 6.2.2 Reducing problems to different forms

Linear programming problems can often be reduced to other forms.

\[
\begin{align*}
\text{max} & \iff \text{min}: & \text{multiply the equation by -1} \\
\geq & \iff \leq: & \text{multiply the inequality by -1} \\
= & \iff \leq \text{and } \geq: & \text{split the equality into two inequalities} \\
\leq & \iff =: & \text{introduce a new variable}: A \leq B \text{ if and only if } A + s \leq B, s \geq 0 \\
>< & \iff =: & \text{introduce two new variables}: x >> 0 \text{ if and only if } x = a - b, a \geq 0, b \geq 0
\end{align*}
\]

The linear programming problem in the standard form can also be written in the matrix form: \( \text{max } cx \) under the constraints \( Ax = b, x \geq 0 \) (\( c \) is a 1 by \( n \) vector, \( x \) and \( b \) are \( n \) by 1 vectors, \( A \) is an \( m \) by \( n \) matrix). By the procedures listed above, one can also convert the last problem into this form: \( \text{min } c'y \) under the constraint of \( A'y = b', y \geq 0 \). In fact, if we let \( b' = c, A' = AT, c' = b \), this is exactly the dual problem of
the previous one. By the duality theorem of linear programming, we know that if one of those two problems has a finite optimal solution, so does the other, and the corresponding values of the objective functions are equal.

### 6.2.3 Max-flow and linear programming

The max-flow problem that we considered before is essentially a special case of the linear programming problem. Suppose we are given a directed graph $G = (V, E)$. The objective is to maximize the sum of flow out of the starting node, where flow must be conserved at each node (other than the starting and ending nodes), must be less than the flow constraint on that edge, and must be non-negative.

For example, the graph in Figure 6.3 can be formulated in this way:

\[
\begin{align*}
\text{max} & \quad X_{sa} + X_{st} \\
X_{sa} & \leq 3 \\
X_{sb} & \leq 3 \\
X_{ab} & \leq 1 \\
X_{at} & \leq 4 \\
X_{bt} & \leq 2 \\
X_{sa} & = X_{ab} + X_{at} \\
X_{sb} + X_{ab} & = X_{bt}
\end{align*}
\]