Combinatorial Optimization: Introductory Problems and Methods

Erin Brownell
erin.brownell@uconn.edu

COMBINATORIAL OPTIMIZATION: INTRODUCTORY PROBLEMS AND METHODS

ERIN BROWNELL

Abstract. This paper will cover some topics of combinatorial optimization, the study of finding the best possible arrangement of a set of discrete objects. These topics include the shortest path problem and network flows, which can be extended to solve more complex problems. We will also briefly cover some basics of graph theory and solving linear programming problems to give context to the reader.

Contents

1. Acknowledgements

I would like to thank my honors thesis advisor, Myron Minn-Thu-Aye, for introducing me to this topic, helping me to understand all of the concepts and proofs presented, and aiding me in the writing process.

Date: **February 27, 2019**.

2. Introduction

As a discipline of discrete mathematics, combinatorial optimization is a relatively young field. Before linear and integer programming became available in the 1950's, the relationships between problems like optimum assignment, the shortest spanning tree, transshipment and transportation, the traveling salesman problem, and others were not fully realized. Thousands of real-world problems can be abstracted to known combinatorial optimization problems. Thus, our concentration will be on the most basic of these, which can be represented naturally through the use of graphs. Hence, before we look at the theory behind our basic problems, we will introduce a few concepts of graph theory.

Next, we move on to cover a few basic problems of combinatorial optimization, starting with the shortest path problem. This is an elementary and relatively easy problem to solve, consisting of finding shortest path lengths, with the restriction that graphs must not contain negative directed cycles. The methods covered for solving this problem include those developed by Dijkstra, Bellman, and Ford in the 1950's.

Finally, we will discuss network flow problems, with a focus on maximal flows and the resulting Max-Flow Min-Cut Theorem. Some interesting results, particularly Menger's Theorem, will arise. Maximal flow problems can be solved efficiently when interpreted as linear programming problems; thus we will give some procedures and examples involving linear programming and duality. These concepts were critical to establishing the subject as a whole.

3. Background

3.1. Graphs.

First, we will discuss some basic concepts in graph theory, which will be necessary for understanding the content of this paper. In particular, we will need some relevant definitions:

Definition 1. A *graph* or *undirected graph* $G = (V, E)$ is a structure made up of a finite set of vertices (or nodes), V, and a set of edges (or arcs), E, each of which is represented by an unordered pair of vertices. In our illustration of such a graph, we show each edge as a line between two nodes.

A *digraph* or *directed graph* $G = (V, E)$, is similar, but each edge is represented by an *ordered* pair of vertices. We will represent an edge in the set E by $e = (i, j), i, j \in V$ for both directed and undirected graphs. In our illustration of a digraph, we use an arrow from node i to node j to represent an edge, as opposed to a straight line.

Definition 2. A *path* in G from vertex s to vertex t, or simply an *s-t path*, is a sequence of edges $(s, v_1), (v_1, v_2), ..., (v_k, t)$.

Definition 3. A *cycle* in G is an *(s,s)*-path containing at least one edge with no vertex repeated except for s.

Definition 4. Two vertices s and t are called *connected* if there exists an s-t path. A graph G is *connected* if all pairs of vertices in G are connected.

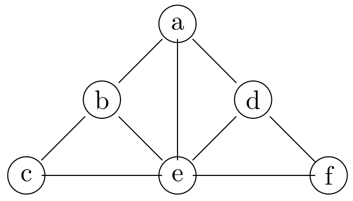
Definition 5. A *subgraph* $G' = (V', E')$ of $G = (V, E)$ is a graph with $V' \subseteq V$ and $E' \subseteq E$. In other words, *G'* contains only vertices and edges found in G.

A *component* of G is a maximal connected subgraph of G. That is, any two vertices in the component are connected by a path.

Definition 6. A *tree* is a graph that is connected and contains no cycles. The *spanning tree* of a graph G is a subgraph containing all the vertices of G and the minimum number of edges required to maintain the properties of a tree. In this paper, the concept of the spanning tree will be useful when discussing shortest paths.

Definition 7. A *separating set* of graph $G = (V, E)$ is a subset $C \subseteq E$, such that $G' = (V, E - C)$ contains more components than G.

Example 1. Separating sets for the graph below include $\{(a, b), (a, e), (a, d)\}$, $\{(a, b), (a, e), (d, e), (e, f)\}$, and $\{(b, c), (c, e)\}$ among others.



Definition 8. A *cocycle* is a minimal separating set. That is, a cocycle contains the minimum number of edges needed to separate the graph into more components than it currently contains.

Example 2. For the graph above, there are two cocycles: $\{(b, c), (c, e)\}$ and $\{(c, f), (d, f)\}$.

Definition 9. A *cutset* is a separating set determined by a certain partition of vertices into sets S and T. More specifically, an *(s,t)-cutset* is any cutset (S, T) where $s \in S$ and $t \in T$. Cutsets will become important when we discuss the Max-Flow Min-Cut Theorem in Section 4.

Example 3. For the graph above, we partition vertices by $S = \{b, c, e\}$ and $T = \{a, d, f\}$. The separating set $\{(a, b), (a, e), (d, e), (e, f)\}$ is an *(s,t)*-cutset.

Note that separating sets, cocycles, and cutsets are all subsets of *edges* in this context.

4. The Shortest Path Problem

Suppose each edge (i, j) of a directed graph has a length e_{ij}. Our goal is to find the shortest path from vertex s to vertex t, i.e. the path such that the sum of all lengths of edges in the path is minimal. This is one of the most fundamental problems of combinatorial optimization.

It is important to notice that this problem is very different when we restrict edge lengths to positive values and when we allow negative values. In particular, when there exist directed cycles of negative length, there is no efficient algorithm that gives the solution to the shortest path problem.

4.1. **Bellman's Equations.** Suppose we wish to find the shortest paths from an origin vertex s to all other vertices. Let

$$\begin{array}{rcl} e_{ij} & = & \text{the length of edge } e = (i,j) \text{ if } e \text{ is in the set of edges in our graph,} \\ & & \text{and } \infty \text{ otherwise.} \\ u_j & = & \text{the length of a shortest path from the origin to vertex } j. \end{array}$$

The following system of equations is referred to as Bellman's Equations:

$$\begin{array}{rcl} u_1 & = & 0 \\ u_j & = & \min_{k \neq j}\{u_k + e_{ij}\} \qquad (j = 2, 3, \ldots n), \end{array}$$

where k is the vertex for which $\min_{k \neq j}\{u_k + e_{ij}\}$ is as small as possible.

Theorem 1. *Bellman's Equations give a unique solution to the shortest path problem.*

To prove that Bellman's equations give a solution to the shortest path problem, we will show that the equations are 1) necessarily satisfied by and 2) sufficient to determine the lengths of the shortest paths.

First, we argue that the shortest path lengths must satisfy the equations. By assumption, there are no negative directed cycles in our graph, so we can state that $u_1 = 0$. For each edge $j \neq 1$ in the path from 1 to j there is a final edge (k, j) with length e_{kj}. In order for the overall path to be a shortest path, the path u_k must be the shortest possible path from 1 to k. Hence $u_j = u_k + e_{kj}$.

Vertex k must be a vertex in our graph with $k \neq j$. The set of vertices is finite by definition, so our choice of k must be such that $u_k + e_{kj}$ is as small as possible. Thus, the path lengths must satisfy the above equations.

Second, we prove that the equations are sufficient to determine the lengths of the shortest paths. We assume there exists a finite path from the origin to all other vertices and that the graph contains no directed cycles of negative length (Note that a path on such a negative cycle would decrease in length the more times the cycle is traversed). Thus all shortest paths are both finite and well-defined. We argue that under these assumptions, Bellman's equations give a unique, finite solution of shortest path lengths. To prove uniqueness, we must first prove the following preliminary result:

Theorem 2. *If the network contains no nonpositive directed cycles, then there exists a tree rooted from the origin, such that the path in the tree from the origin to each of the other nodes is a shortest path. (We call such a tree a tree of shortest paths.)*

Proof. Suppose paths $u_1, u_2, \ldots, u_n$ satisfy Bellman's equations. Construct paths to vertices $1, 2, \ldots, n$ having these lengths by the following procedure:

To find the path to vertex j of length u_j, find edge (k, j) such that $u_j = u_k + e_{kj}$. Next, find (l, k) such that $u_k = u_l + e_{lk}$, and so on until the origin is reached ($u_1 = 0$).

Repeating this process for all vertices j, the number of edges that can be selected as part of any shortest path is exactly $n - 1$. These $n - 1$ edges form a tree rooted at the origin. We've seen that there is a tree for any finite solution to Bellman's equations, and that the shortest path lengths are such a solution to Bellman's equations. Thus, there exists a tree of shortest paths as desired. □

Now, we prove uniqueness:

Theorem 3. *If the network contains no nonpositive cycles, and if there is a path from the origin to each of the other nodes, then there is a unique finite solution to Bellman's equations, where u_j is the length of a shortest path from the origin to vertex j.*

Proof. By way of contradiction, let $u_1, u_2, \ldots, u_n$ be the lengths of the shortest paths and $u'_1, u'_2, \ldots, u'_n$ be any other finite solution to Bellman's equations such that for some vertex j, $u_j \neq u'_j$. It must be that $u'_j > u_j$, since u_j is the length of the shortest path to j. Choose j such that $u_k = u'_k$, where (k, j) is an edge in the tree of shortest paths (there will be at least one such edge as $u_1 = u'_1 = 0$). Then we have $u'_j > u'_k + e_{kj}$, contradicting our assumption that $u'_1, u'_2, \ldots, u'_n$ is a solution to Bellman's equations. □

4.2. **Dijkstra's Method.** Dijkstra's method solves the shortest path problem given all edge lengths are positive. The main idea of this method is simply to build up a solution by finding the next closest vertex from our source and computing the shortest distance to that vertex.

All vertices in the graph, $1, 2, \ldots, n$, are labeled either "permanent" or "tentative" and belong to sets P or T, respectively. We use the notation u_i to represent a label. A permanent label on a vertex represents the length of a shortest path to the vertex i, and a tentative label represents an upper bound on the length of the shortest path to i. The following algorithm terminates when all vertices have been permanently labeled, and outputs the lengths of the shortest paths, $u_1, u_2, \ldots, u_n$:

```
           $u_1 = 0.$
initialize $u_j = e_{1j}$,   for $j = 2, 3, \ldots, n$,
           $P = \{1\}$, $T = \{2, 3, \ldots, n\}$.
while $T \neq \emptyset$ do
    Select a vertex $k \in T$ such that $u_k = \min_{j \in T}\{u_j\}$;
    Update $T = T - k$, $P = P + k$ ;
    Update $\min\{u_j, u_k + e_{ij}\}$   for all $j \in T$ ;
end
```

Algorithm 1: Dijkstra's Algorithm

Theorem 4. *For all $j \in P$, u_j is the shortest path from 1 to j.*

Proof. We prove the validity of this method by induction on the size of P, the set of vertices permanently labeled, claiming that for all $j \in P$, u_j is the shortest path from 1 to j.

Initially, the size of P is 1, that is when $P = \{1\}$. Clearly, $u_1 = 0$ is a shortest path. Assume our claim holds for P of size $i \geq 1$.

We now increase the size of P to $i + 1$ by adding vertex k. We will show that the claim still holds. Suppose edge (l, k) is the final edge on a path with length u_k and $l \in P$. By our assumption, u_l is a shortest path length from vertex 1 to l. Consider any other path from 1 to k with length u'_k. We must show $u'_k \geq u_k$.

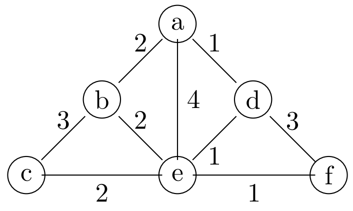
By way of contradiction, assume $u'_k < u_k$. Then the path of length u'_k must contain vertices not in P. Let y be the first vertex on this path such that $y \notin P$.

Then it cannot be the case that $u'_k < u_k$. The length of the path from 1 to y must be at least as large as u_k, because if it were less then the algorithm would have added y to P instead of adding k to P. Hence it must be the case that $u'_k \geq u_k$.

This completes our induction; we have shown that Dijkstra's algorithm is valid.

□

Example 4. Obtain the shortest path from a to c using Dijkstra's method.



We first set $u_a = 0$. The steps of the algorithm follow the rows in this table:

$P,\ T$	u_b	u_c	u_d	u_e	u_f
$P = \{a\}, T = \{b, c, d, e, f\}$	2	∞	1	4	∞
$P = \{a, d\}, T = \{b, c, e, f\}$	2	∞	1	2	4
$P = \{a, d, b\}, T = \{c, e, f\}$	2	5	1	2	4
$P = \{a, d, b, e\}, T = \{c, f\}$	2	4	1	2	3
$P = \{a, d, b, e, f\}, T = \{c\}$	2	4	1	2	3
$P = \{a, d, b, e, f, c\}, T = \emptyset$	2	4	1	2	3

The first row of the table is our initialization step. The values of u_j for each vertex j are derived from e_{ij}, the length of the edge from 1 to j if such an edge exists, and infinity otherwise.

For the second step, we must find vertex k such that $u_k = \min_{j \in T}\{u_j\}$. Vertex d satisfies this condition. So, we add d to the set of permanently labeled vertices, and update all the other labels according to $\min\{u_j, u_k + e_{ij}\}$. In other words, we find the shortest path length from a to all other vertices, allowing "traversal" through vertex d.

We repeat the previous step until all vertices have been permanently labeled.

The final row gives the resulting shortest path lengths from vertex a to all other nodes. Thus, the shortest path from a to c has length equal to 4.

4.3. **Bellman-Ford Method.**

Now, we find a general solution to Bellman's equations by way of successive approximations. Specifically, we approximate the shortest path lengths allowing one edge per path, then we recompute them allowing two edges, and so on, slowly converging to the correct shortest path lengths to each vertex. Recall that using Dijkstra's method we only considered the specific case where edge lengths are positive. In our generalization, we assume only that there are no negative directed cycles.

Define

$u_j^{(m)}$ = the length of a shortest path from the origin to vertex j, subject to the condition that the path contains no more than m edges.

To solve Bellman's equations, we use successive approximations as follows:
Initially,

$$\begin{aligned} u_1^{(1)} &= 0 \\ u_j^{(1)} &= e_{1j}, j \neq 1, \end{aligned}$$

Then from the m-th order approximation, we can compute the $(m+1)$-st order approximation by:

$$u_j^{(m+1)} = \min\{u_j^{(m)}, \min_{k \neq j}\{u_k^{(m)} + e_{kj}\}\}.$$

These must be solved for $m = 1, 2, \ldots, n-2$.

From the rule stated above, it is clear that for each vertex j, the successive approximations of u_j are monotone nonincreasing. That is,

$$u_j^{(1)} \geq u_j^{(2)} \geq u_j^{(3)} \geq \ldots.$$

Theorem 5. *For each vertex j, the successive approximations converge to the correct value of* u_j.

Proof. We argue by induction on m, the order of approximation. Clearly, $u_j^{(1)}$ gives the correct value of the shortest paths containing no more than one edge to every other vertex.

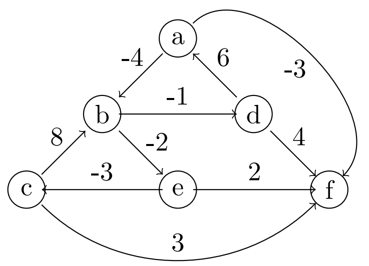
Suppose for our m-th order approximation, $u_j^{(m)}$ gives our desired values for shortest paths.

Now, there are two possibilities for when we restrict to at most $m+1$ edges: 1) a shortest path of at most $m+1$ edges from the origin to j has at most m edges, or 2) it contains $m+1$ edges and has some final edge (k, j). In the first case, the shortest path has length $u_j^{(m)}$. In the second case, the shortest path is equal to the length of the path from the origin to vertex k, $u_k^{(m)}$, plus the length of the final edge, e_{kj}. We choose the smallest possible value of the sum $u_k^{(m)} + e_{kj}$ over all possible choices of k. The minimum of these two values gives our approximation for $u_j^{(m+1)}$. $\square$

Example 5. Compute the shortest paths from origin a for the graph given by the following adjacency matrix:

$$\begin{bmatrix} 0 & -4 & \infty & \infty & \infty & -3 \\ \infty & 0 & \infty & -1 & -2 & \infty \\ \infty & 8 & 0 & \infty & \infty & 3 \\ 6 & \infty & \infty & 0 & \infty & 4 \\ \infty & \infty & -3 & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Visually,



In this example, we relabel $u_1 = u_a, u_2 = u_b$ and so on. The rows in the table below represent our values for each label as we increment m. The number of rows can be at most $n-2$.

m	$u_a^{(m)}$	$u_b^{(m)}$	$u_c^{(m)}$	$u_d^{(m)}$	$u_e^{(m)}$	$u_f^{(m)}$
1	0	-4	∞	∞	∞	-3
2	0	-4	∞	-5	-6	-3
3	0	-4	-9	-5	-6	-4
4	0	-4	-9	-5	-6	-6

We initialize $u_a^{(1)} = 0$. The first row contains this value as well the values $u_j^{(1)} = e_{1j}$ for all other nodes.

The second row is computed from the previous row using the recurrence

$$u_j^{(m+1)} = \min\{u_j^{(m)}, \min_{k \neq j}\{u_k^{(m)} + e_{kj}\}\}.$$

For example, $u_b^{(2)} = \min\{-4, \min\{\infty + 8, \infty + \infty, \infty + \infty, -3 + \infty\}\} = -4$. We compute this value for all nodes then repeat for our next increment of m, and so on.

The final row gives the shortest path lengths from vertex a to all other nodes.

5. Network Flows

Many important combinatorial optimization problems can be reduced to and solved as network flow problems. In this section, we will discuss maximal flows, which are a first problem in networks that can be adapted to solve any number of other problems.

5.1. Maximal Flows.

Suppose we have a network in the form of a directed graph. For every edge (i, j) in the graph, we assign some value $c_{ij} \geq 0$. We call this value the *capacity*, and think of it as the maximum amount that can "flow" from vertex i to vertex j.

Say we want to find the maximal flow from a *source* node s to a *sink* node t. A source node only allows flow out of the node; a sink node only allows flow into the node. For all edges, let

$$x_{ij} = \text{the amount of flow through edge } (i, j)$$
$$0 \leq x_{ij} \leq c_{ij}$$

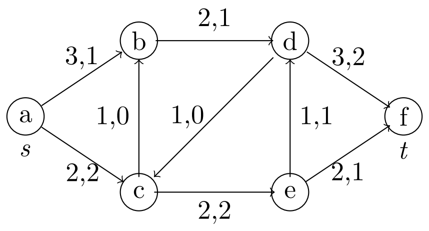
We also wish to uphold a *conservation law*; that is, what goes out of a node must be equal to what goes in, with the exception being the source and sink nodes.

A *feasible flow* is any set of edge flows x_{ij} that satisfy this law, and we represent the value of the flow by v. Observe

$$\sum_j x_{ji} - \sum_j x_{ij} = \begin{cases} -v, & \text{if } i = s \\ 0, & \text{if } i \neq s, t \\ v, & \text{if } i = t. \end{cases}$$

That is, the sum of all the flows going into node i minus the sum of all the flows going out of node i is: 1) negative if i is the source, 2) positive if i is the sink, and 3) zero otherwise, demonstrating our conservation law.

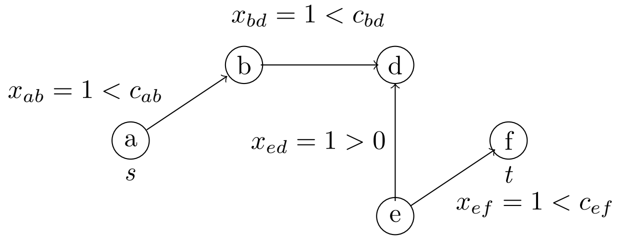
Example 6. Below is an example of a feasible flow. Each edge is labeled with its capacity and its flow: c_{ij}, x_{ij}.



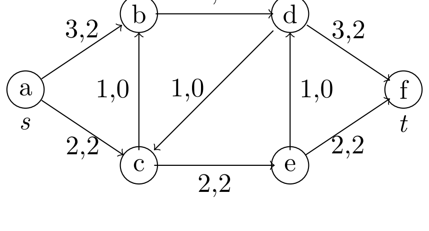
Now, let P be an undirected path from s to t. The edges in P can be either *forward* (directed from s toward t) or *backward*.

Definition 10. P is called a *flow augmenting path* with respect to a given flow $x = x_{ij}$ if $x_{ij} < c_{ij}$ for each forward edge, and $x_{ij} > 0$ for each backward edge in P.

This means that we are not at maximum capacity for each forward edge, and we have some amount of flow along each backward edge. Clearly, we can increase flow through this path by increasing forward flow and decreasing backwards flow. Hence an augmenting path can be used to produce an *augmented flow*. In our example, we have the following augmented path:



By increasing the flow by one unit in each forward edge and decreasing by one unit in each backward edge of this path we can obtain an augmented flow:



Recall from Section 2.1 that an *(s,t)*-cutset is any cutset (S,T) where $s \in S$ and $t \in T$.

Definition 11. The *capacity* of the cutset (S,T) is defined as

$$c(S,T) = \sum_{i \in S} \sum_{j \in T} c_{ij},$$

In other words, the capacity is equal to the sum of the capacities of all edges that are directed from S to T.

Lemma 1. *The value v of any (s,t)-flow cannot exceed the capacity of any (s,t)-cutset.*

Proof. Let $x = (x_{ij})$ be a flow, and (S,T) be an *(s,t)*-cutset. Using the conservation law, we find the value of any flow v by summing the net flow through all nodes in S as follows:

$$\begin{aligned} v = & \sum_{i \in S} \left(\sum_{j} x_{ij} - \sum_{j} x_{ji} \right) \\ v = & \sum_{i \in S} \sum_{j \in S} (x_{ij} - x_{ji}) + \sum_{i \in S} \sum_{j \in T} (x_{ij} - x_{ji}) \\ v = & \sum_{i \in S} \sum_{j \in T} (x_{ij} - x_{ji}) \end{aligned}$$

This tells us that the value of any flow is equal to the net flow through any cutset. However, we know $x_{ij} \leq c_{ij}$ and $x_{ji} \geq 0$ so $v \leq \sum_{i \in S} \sum_{j \in T} c_{ij} = c(S,T)$. □

Example 7. Show that the result of the lemma holds for the augmented flow in example 6, using the *(s,t)*-cutset $S = \{a, b\}$ and $T = \{c, d, e, f\}$.

Using our simplified summation in the proof, we get

$$\begin{aligned} v = & \sum_{i \in S} \sum_{j \in T} (x_{ij} - x_{ji}) \\ v = & \sum_{j \in T} (x_{aj} - x_{ja}) + \sum_{j \in T} (x_{bj} - x_{jb}) \\ v = & \ 2 + 2 \\ v = & \ 4. \end{aligned}$$

We know this value to be at most equal to the capacity of the cutset. It also happens to be equal to the value of the *(s,t)*-flow in the augmented graph. Therefore, the flow is maximal, and our cutset has minimal capacity.

Now, let us discuss three fundamental theorems in network flow theory. We will use these theorems to explore some interesting results.

First, we consider the relationship between augmenting paths and maximal flows. It seems intuitive that if there exists an augmenting flow, then the flow can be increased and therefore is not maximal, but can we say for certain that a network with no augmenting flows is indeed maximal? We argue that this is true.

Theorem 6 (Augmenting Path Theorem)**.** *A flow is maximal if and only if it admits no augmenting path from s to t.*

Proof. We show that a flow is maximal if it admits no augmenting path from s to t by examining the contrapositive: If an augmenting path exists, then the flow is not maximal. This statement is clearly true.

Conversely, suppose that x is a flow that does not admit any augmenting path. We know from the previous lemma that the flow value cannot exceed the capacity of any *(s,t)*-cutset. At most, the flow is equal to the capacity of such a cutset. We will construct our cutset (S, T) in such a way that the flow value is maximal.

Observe that

$$v = \sum_{i \in S} \sum_{j \in T} (x_{ij} - x_{ji}) = \sum_{i \in S} \sum_{j \in T} c_{ij} = c(S, T)$$

when $x_{ij} = c_{ij}$ and $x_{ji} = 0$.

This is the case when S is the set of all vertices j that admit no augmenting path from s to j and T is the complementary set, simply by the definition of an augmenting path.

We have found a cutset meeting our desired conditions, and have thus proven that the flow is maximal. □

So far we have only seen examples in which edge flows are integers, but this is not always the case. However, our main goal in any maximal flow problem is of course to find the maximal flow, which we can ensure is integral by setting all edge capacities to be integers.

Theorem 7 (Integral Flow Theorem)**.** *If all edge capacities are integers, then there exists a maximal flow which is integral.*

Proof. Let all edge capacities be integers. Let flows $x^0_{ij} = 0$ for all vertices i and j. By Theorem 4, if $x^0 = (x^0_{ij})$ is not maximal, it admits an augmenting path. Thus there is some integral flow $x^1 > x^0$. If x^1 is not maximal, it admits an augmenting path and there exists integral flow $x^2 > x^1$, and so on, until we reach a maximal flow which must be integral. □

We can use this result to prove a key theorem:

Theorem 8 (Max-Flow Min-Cut Theorem)**.** *The maximum value of an (s,t)-flow is equal to the minimum capacity of an (s,t)-cutset.*

Proof. First, assume all capacities are integers (or commensurate). By the Integral Flow Theorem, there is an integral, maximal flow. A maximal flow admits no augmenting path, and we have seen that the maximal flow value $v = c(S, T)$ for some *(s,t)*-cutset. By our lemma, $c(S, T) \geq v$; when $v = c(S, T)$ the capacity of the cutset is minimal. Thus in this case the maximal value of an *(s,t)*-flow is equal to the minimum capacity of an *(s,t)*-cutset, as desired.

We must show that the theorem holds when capacities can be any real numbers. To do so, we will present an algorithm in the next section which always computes the maximal flow in a finite number of steps. □

The max-flow min-cut theorem is a fundamental result in network theory, and leads to many other important results, including Menger's Theorem. Linear Programming, arguably the tool by which combinatorial optimization was solidified as a distinct field of study, directly results in this theorem, as we will see in the coming sections. But first, we must present the algorithm that provides proof that the theorem holds for real number capacities.

5.2. **Maximal Flow Algorithm.** Fortunately, the problem of finding the maximal capacity flow augmenting path for real number capacities is analogous to the problem of finding a shortest path. We can represent our problem in a form similar to that of Bellman's equations, but in this case we are trying to find a path in which the minimum edge length is maximum.

Let

$$\bar{c_{ij}} = \max\{c_{ij} - x_{ij}\},$$

where $c_{ij} = 0$, if there is no edge (i, j).

Let

$$u_i = \text{the capacity of a maximum capacity augmenting path from node } s \text{ to node } i.$$

Then the analogues of Bellman's equations are:

$$\begin{aligned} u_s &= \infty \\ u_i &= \max_k \min\{u_k, \bar{c_{ki}}\} \qquad i \neq s. \end{aligned}$$

Now, we give a brief sketch of how the algorithm works before defining it in its entirety.

To find the maximal flow, we need to find the maximum capacity flow augmenting paths. We solve this problem with a procedure, in which labels of the form (i^+, δ_j) or (i^-, δ_j) may be assigned to a vertex, indicating that 1) there exists an augmenting path with capacity δ_j from s to j, and (i, j) is the last edge in this path, or 2) that (j, i) is the last edge in this path (respectively).

A labeled vertex is either scanned, meaning all incident edges have been examined and labels have been applied to previously unlabeled adjacent vertices, or unscanned.

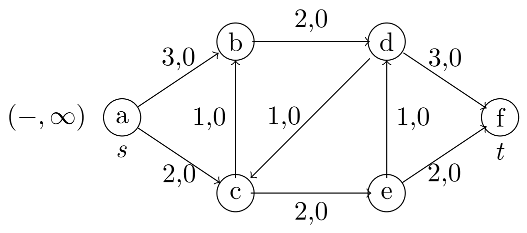
When the procedure labels t, the sink node, an augmenting path is found and the flow value can be augmented. If t remains unlabeled, then there is no augmenting path. At the end, a minimum capacity cutset (S, T) is constructed where $S =$ the set of labeled nodes and $T =$ the set of unlabeled nodes.

We refer to this method as the Maximal Flow Algorithm. The detailed procedure begins on the following page.

```
Initially, let $x = (x_{ij})$ be any integral feasible flow.;
Permanently label $s = (-, \infty)$.
while there remains a labeled node that has not been scanned do
    Find a labeled but unscanned node $i$ and scan $i$ as follows:
    for each edge $(i, j)$ do
        if $x_{ij} < c_{ij}$ and $j$ unlabelled then
            Give j the label $(i^+, \delta_j)$ where $\delta_j = \min(c_{ij} - x_{ij}, \delta_i)$
    end
    for each edge $(j, i)$ do
        if $x_{ij} > 0$ and $j$ unlabelled then
            Give j the label $(i^-, \delta_j)$ where $\delta_j = \min(x_{ij}, \delta_i)$
    end
    We are done scanning $i$.
    if node $t$ has been labeled then
        Starting at node $t$, use the index labels to construct an augmenting
          path.
        Augment the flow by increasing/decreasing edge flows by $\delta_t$, as
          indicated by the superscripts.
        Erase all labels, except on $s$.
end
The exisiting flow is maximal.
A cutset of minimum capacity is obtained by setting $S$ = set of labeled
  vertices, $T$ = set of unlabeled vertices.
```

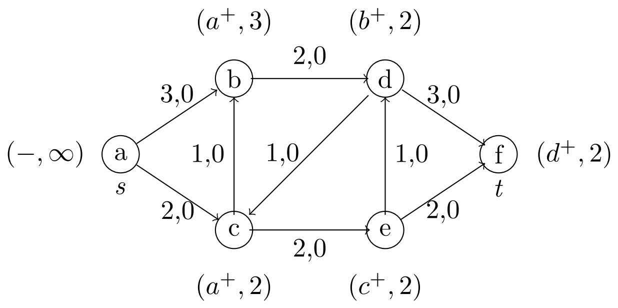
Algorithm 2: Maximal Flow Algorithm

Example 8. Consider the graph in our earlier example, but set the initial flow $x = 0$ for simplicity. Our source node s is permanently labeled with $(-, \infty)$.

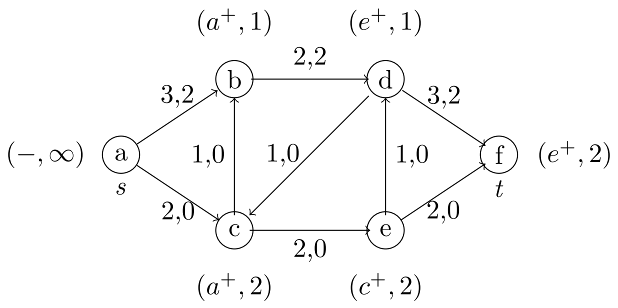


We enter the while loop of the algorithm, and begin scanning a, the source node. We find that we can label b with $(a^+, 3)$ and c with $(a^+, 2)$. Our scan of a is done. The sink node has node yet been labeled, so we continue scanning the nodes.

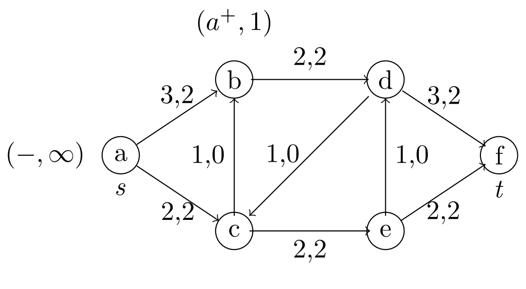
Next, we scan b and label d with $(b^+, 2)$; we scan c and label e with $(c^+, 2)$; and finally we scan d and label the sink node, f, with $(d^+, 2)$.



Now that sink node d is labeled, we construct our flow augmenting path: a, b, d, f. We increment the flow on this path by 2. At this point, we have not scanned all the labeled nodes, so we erase all labels and repeat the procedure in the while loop with our augmented graph, to obtain the following labels:



In this iteration, we find flow augmenting path a, c, e, f, and augment by 2. Again, we did not scan all nodes with labels. So, we must repeat the contents of the while loop one more time. This time, we only scan two nodes before we run out of unscanned, labeled nodes:



The existing flow value, $v = 4$, is maximal. Our minimum capacity cutset is constructed by $S = \{a, b\}, T = \{c, d, e, f\}$. These values should be familiar.

5.3. **Generalized Max-Flow Min-Cut Theorem and Results.**

We will provide a generalized version of the Max-Flow Min-Cut Theorem that is useful for interpreting many more problems, by considering networks that have node capacities. Note that our previous theorem only applied to the special case of networks in which all node capacities are infinite.

Consider a flow network with edge capacities $c_{ij} \geq 0$ and node capacities $c_i \geq 0$. Flows must satisfy conditions set by conservation laws (similar to previous conservation laws), edge capacities

$$0 \leq x_{ij} \leq c_{ij},$$

and node capacities

$$\sum_j x_{ij} \leq c_i, i \neq s, t.$$

In order to re-formulate the theorem in these terms, we need to revise some of our previous definitions involving cutsets to make sense in this context.

Definition 12. An *(s,t)-cut* is a set of edges and vertices such that any path from s to t uses at least one of its members.

Definition 13. The *capacity* of a cut is the sum of the capacities of all its members.

Our new concept of a cut is analogous to the old concept of a cutset, and we can consider them in the same way. Specifically,

Lemma 2. *In a network whose node capacities are all infinite, the minimum cut capacity is equal to the minimum cutset capacity.*

Proof. Let (S, T) be a cutset. Let C be the set of edges that connect a vertex in S to a vertex in T. By Definition 12, C is a cut, with capacity equal to the sum of the capacities of all its edges. Recall that the capacity of a cutset $(S, T) = \sum_{i \in s} \sum_{j \in T} c_{ij}$; hence the capacity of C is equal to the capacity of (S, T).

Conversely, let C be a cut containing only edges. Arbitrarily choose a source node s. Let S be the set of nodes reachable by a direct path from s, not using any edge in C. Let T be the set of remaining nodes. Then (S, T) is a cutset and C contains every edge between the nodes in S and T. Thus the capacity of (S, T) is at most the capacity of C. □

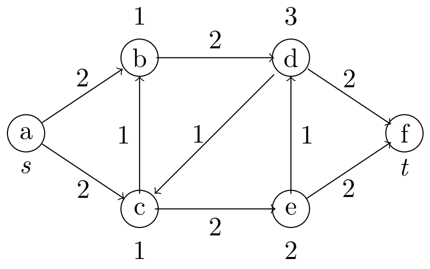
With this knowledge we can finally provide a general case of the max-flow min-cut theorem:

Theorem 9 (Generalized Max-Flow Min-Cut Theorem)**.** *In a network having node capacities as well as edge capacities, the maximum value of an (s,t)-flow is equal to the minimum capacity of an (s,t)-cut. Moreover, if all capacities are integers, there is a maximal flow that is integral.*

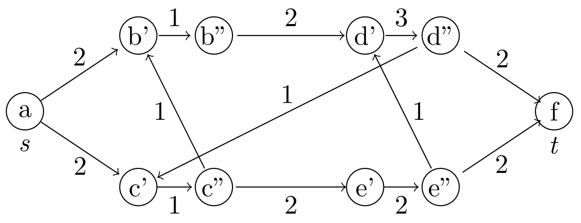
Proof. We will "expand" the network by replacing each interior node i (interior nodes are nodes that are neither a source nor a sink) by an *in-node i'* and an *out-node i"*, and an edge (i', i'') of capacity c_i. Let $s = s' = s''$ and $t = t' = t''$.

For each edge (i, j) of the original network there is an edge (i'', j') of capacity c_{ij} in the expanded network. Now, we have a network of nodes with infinite capacity as before. The original Max-Flow Min-Cut Theorem applies. □

Example 9. Convert the following network to its expanded form.



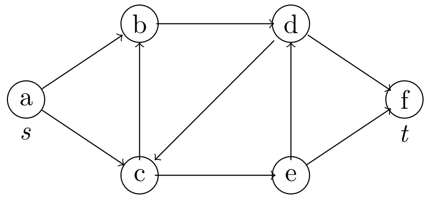
expands to



Several other interesting results are implied by this generalization. We will provide proofs of Menger's theorem, proved in 1927, which characterizes the connectivity of the network. Again, we will need some additional definitions for context.

Definition 14. A digraph G is said to be *k-connected from s to t* if for any set C of $k-1$ nodes missing s and t there is a directed path from s to t missing C. In other words, it is not possible to disconnect s from t by removing any fewer than k nodes.

Example 10. The following is 2-connected from s to t. For each of the nodes in $\{b, c, d, e\}$, there is a path from s to t that does not go through that node. If you were to remove nodes d and e, the digraph would be disconnected (or d and c, b and e, or b and c).



Definition 15. Two paths are called *independent* if they have no nodes in common except s and t.

Theorem 10 (Menger)**.** *If digraph G is k-connected from s to t and does not contain edge (s,t), then G admits k independent directed paths from s to t.*

Proof. Let each node have a capacity of 1, and each edge have infinite capacity. The minimum cut capacity is finite, since we assume no edge (s,t) in the network. If the digraph is k-connected, then the minimum cut capacity is at least k. From the generalized Max-Flow Min-Cut Theorem, the integral maximal flow has value at least k. It must be structured such that the flow from s to t is along k pairwise independent directed paths. □

Theorem 11 (Menger)**.** *The maximum number of edge-disjoint (s,t) paths in an undirected graph G is equal to the minimum number of edges in an (s,t)-cutset.*

Proof. Construct a flow network from the graph G in which each edge of G is replaced by two symmetric pairs of edges, (i, j) and (j, i), with a capacity of one. An integral maximal flow exists from s to t, in which at least one arc of each pair must be void. Then the value of this maximal flow must be equal to the maximum number of edge disjoint *(s,t)*-paths, since each disjoint path can pass at most only one unit of flow (any joint paths can be reduced to one disjoint path without changing the amount of flow). Then by the max-flow min-cut theorem, we know that this value is equal to the minimum capacity of an *(s,t)*-cutset. Clearly, since each edge is of capacity one, then this value is also equal to the minimum number of edges in an *(s,t)*-cutset. Thus concludes our proof of Menger's Theorem. □

6. Linear Programming Problems

The maximal flow problem can be transformed into a *linear programming problem*, and the max-flow min-cut theorem follows from the theorem of strong duality, which will be introduced in this section. In order to discuss the max-flow min-cut theorem in these terms, it is necessary to understand what a linear programming problem is, how to solve one, and how to find the associated dual problem. So, in this section we provide some brief introduction to this set of problems.

The general form of a linear programming problem is:

Optimize $z = \sum_{j=1}^{n} c_j x_j$

Subject to the constraints

$$\begin{aligned} \sum_{j=1}^{n} a_{ij}x_j \geq b_i, &\quad i = 1, 2, \dots, p, \\ \sum_{j=i}^{n} a_{ij}x_j = b_i, &\quad i = p+1, p+2, \dots, m, \\ x_j \geq 0, &\quad j = 1, 2, \dots, q, \\ x_j \text{unrestricted}, &\quad j = q+1, q+2, \dots, n. \end{aligned}$$

Any linear programming problem can be reduced to a problem involving only equality constraints in nonnegative variables, through the use of *slack variables.* For example, if we have the constraint $ax \geq b$, an equivalent linear equality is $ax - s = b$, where s is a nonnegative slack variable. Thus, without going into detail, we can introduce slack variables into our general problem stated above until we obtain the following:

minimize $z = cx$

subject to

$$\begin{aligned} Ax &= b \\ x &\geq 0, \end{aligned}$$

where $c = (c_1, c_2, \dots, c_n)$ is the *cost vector*, cx is the *objective function*, $A = (a_{ij})$ is an m x n coefficient matrix, and $b = (b_1, b_2, \dots, b_m)$ is the *constraint vector.* From this form, we can more easily discuss possible solutions.

Definition 16. A vector $\bar{x} \geq 0$ for which $A\bar{x} = b$ is called a *feasible solution.*

Definition 17. A feasible solution x^* is an *optimal solution* if there exists no other feasible solution $\bar{x}$ such that $c\bar{x} < cx^*$.

Solving this problem and finding feasible solutions will involve some elements of linear algebra, including matrix manipulation, but in the simple examples we will consider only a rudimentary understanding is necessary.

Definition 18. Recall from linear algebra that any m linearly independent columns of A are referred to as a *basis* of the linear system $Ax = b$. Let B represent the submatrix of A corresponding to a given basis.

There is a unique basic solution, x^B, associated with each basis B for a given matrix. A basic solution which is feasible ($x^B \geq 0$) is called a *basic feasible solution*, and a basic solution which is optimal is called a *basic optimal solution.*

Theorem 12. *If there exists a feasible solution to our linear programming problem in matrix form, then there exists a basic feasible solution.*

Theorem 13. *If there exists an optimal solution to our problem, then there exists a basic optimal solution.*

Proof. For proof, see page 42 of *Combinatorial Optimization: Networks and Matroids*, by Eugene Lawlor. The full citation can be found in the References □

From these theorems, we can see that the optimal solution we wish to find is among the basic solutions. For an n x m linear system with $m \leq n$, there are no more than n choose m bases, and for each basis we can find a unique basic solution. Thus our search for an optimal solution has been reduced to a a finite combinatorial problem.

6.1. **The Simplex Method.** The Simplex method is a method for finding the optimal basic solution, by starting from one basic feasible solution and moving to another, with the objective function value getting closer to the optimal value with each step. When we have reached the optimal value, and no improvement can be made, then the final basic solution is optimal. Rather than giving the general form of the procedure, we will demonstrate the Simplex Method by example.

Example 11. Carry out the simplex method for the following:
minimize $z = -3x_1 - 2x_2$
subject to

$$\begin{aligned} -2x_1 + x_2 &\leq 1 \\ x_1 &\leq 2 \\ x_1 + x_2 &\leq 3 \\ x_1, x_2 &\geq 0. \end{aligned}$$

First, we add slack variables to obtain a system of linear equalities:
minimize $z = -3x_1 - 2x_2$
subject to

$$\begin{aligned} -2x_1 + x_2 + x_3 &= 1 \\ x_1 + x_4 &= 2 \\ x_1 + x_2 + x_5 &= 3 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0. \end{aligned}$$

Next, we set up our matrix. In this example, the columns represent the variables $z, x_1, x_2, x_3, x_4, x_5$, and b, in that order, and the rows represent our first three constraint equations and the objective function (in the form $0 = -3x_1 - 2x_2 - z$).

Our matrix:

$$\begin{bmatrix} 0 & -2 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 0 & 0 & 1 & 3 \\ -1 & -3 & -2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

From this matrix, we can see that the initial basic solution is $x_1, x_2 = 0, x_3 = 1, x_4 = 2, x_5 = 3$, from the basis in columns corresponding to x_3, x_4, x_5. Our objective function evaluates to 0. We will find a column in which to pivot, to produce the next basis, giving an improved basic solution.

The Simplex Method tells us to apply the Ratio Test to the elements in the column with the most negative coefficent in the objective function. This is in the column x_1, with coeffecient -3.

We use the Ratio Test to ensure that our constants in column b remain nonnegative. The test is completed by dividing the values in column b by the corresponding values in column x_1, provided that they are nonnegative. We then take the minimum of these quotients, and pivot around the corresponding element in column x_1.

In this case, the minimum quotient comes from the second row, with value 2. Thus we pivot around the element 1, in the second row and second column. Our pivot operations include: adding (2 x the second row) to the first row, subtracting the second row from the third throw, and adding (3 x the second row) to the fourth row:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 0 & 5 \\ 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 \\ -1 & 0 & -2 & 0 & 3 & 0 & 6 \end{bmatrix}$$

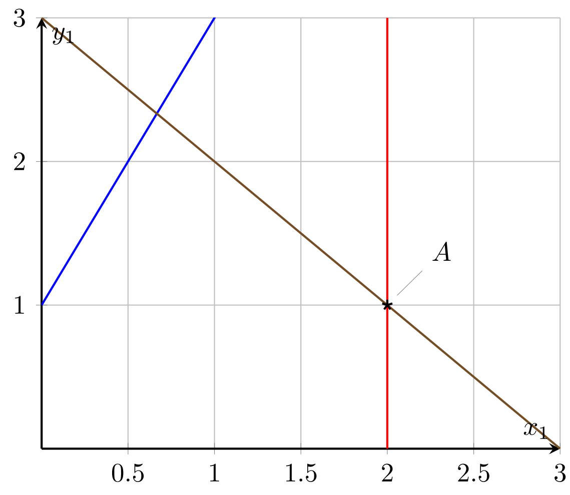
Now our basis consists of columns x_1, x_3, x_5, with improved basic solution $x_2, x_4 = 0, x_1 = 2, x_3 = 5, x_5 = 1$. Our objective function evaluates to -6, verifying that our solution is improved. We repeat the Ratio Test, this time in column x_2, and find that our pivot element is 1, in the third row, third column. Pivoting, we get:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 3 & -1 & 4 \\ 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 \\ -1 & 0 & 0 & 0 & 1 & 2 & 8 \end{bmatrix}$$

At this point, we are done because all the coefficents of the objective function are nonnegative. Our optimal basic solution is $x_4, x_5 = 0, x_1 = 2, x_2 = 1, x_3 = 4$, with value -8.

In this example, we only start out with two variables; thus it is relatively easy to give a visual, geometric interpretation of the problem. Instead of minimizing $z = -3x_1 - 2x_2$, we look at the equivalent problem of maximizing $z = 3x_1 + 2x_2$, under the same constraints.

Geometrically,



The feasible solutions to our linear program are found on the boundary and within the interior of the polygon. The maximum value of z in our feasible region is found at point A, with coordinates (2,1). This verifies our optimal solution found by the simplex method.

6.2. **Duality.** Duality Theory is essential for solving many problems, in particular the linear programming version of maximal flows. The main idea is that for every linear programming problem, there is another problem, its *dual*, such that solving one problem in effect solves the other problem. Given a linear program of the form

Minimize

$$z = \sum_{j=1}^{n} c_j x_j$$

Subject to

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

$$\sum_{j=i}^{n} a_{ij} x_j = b_i$$

$$x_j \geq 0$$

$$x_j \text{unrestricted},$$

the associated dual problem is:

Minimize

$$w = \sum_{i=1}^{m} (-b_i)u_j$$

Subject to

$$u_i \geq 0$$
$$u_i \text{ unrestricted}$$
$$\sum_{i=1}^{m} (-a_{ij})u_j \geq -c_j$$
$$\sum_{i=1}^{m} (-a_{ij})u_j = -c_j.$$

Note that minimizing $-bu$ is the same as maximizing bu, and also that the coefficient matrices of the primal and dual problems are negative transposes of each other, with the roles of the b and c vectors reversed.

Example 12. Find the dual of the problem:
minimize

$$z = x_1$$

subject to

$$\begin{aligned} x_1 + x_2 &= 1 \\ x_1 &= 2 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Its Dual:
maximize

$$w = u_1 + 2u_2$$

subject to

$$\begin{aligned} u_1 + u_2 &\leq 1 \\ u_1 &\leq 0 \\ u_1, u_2 &\text{ unrestricted.} \end{aligned}$$

Example 13. Find the dual of our linear programming problem in example 11.
maximize $z = 3x_1 + 2x_2$
subject to

$$\begin{aligned} -2x_1 + x_2 &\leq 1 \\ x_1 &\leq 2 \\ x_1 + x_2 &\leq 3 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Its Dual:
minimize $w = u_1 + 2u_2 + 3u_3$
subject to

$$\begin{aligned} -2u_1 + u_2 + u_3 &\geq 3 \\ u_1 + u_3 &\geq 2 \\ u_1, u_2, u_3 &\geq 0. \end{aligned}$$

Note that dual problems don't necessarily contain the same number of variables.

The next few results lead up to the theorem of Strong Duality, of which the max-flow min-cut theorem is a special case that directly results. We will not prove Strong Duality, as it is rather involved for the scope of this paper, but a reference will be provided to the reader. In the coming proofs, we will arbitrarily choose a pair of dual problems in the following form:

$$\begin{array}{rcl} \text{minimize} & \text{cx} & \\ \text{subject to} & & \\ \text{Ax} & \geq & \text{b} \\ \text{x} & \geq & 0. \end{array} \qquad \begin{array}{rcl} \text{maximize} & \text{ub} & \\ \text{subject to} & & \\ \text{uA} & \leq & \text{c} \\ \text{u} & \geq & 0. \end{array}$$

Theorem 14 (Weak Duality)**.** *If $\bar{x}$ and $\bar{u}$ are feasible solutions to dual problems, then $c\bar{x} \geq \bar{u}b$.*

Proof. Suppose our dual problems are in the form outlined above. We have $A\bar{x} \geq b$ and $\bar{u} \geq 0$, hence $\bar{u}A\bar{x} \geq \bar{u}b$. Similarly, $\bar{u}A\bar{x} \leq c\bar{x}$. Thus $c\bar{x} \geq \bar{u}b$.

For problems not in the form above, the proof is similar. □

Corollary 1. *If $\bar{x}$ and $\bar{u}$ are feasible solutions to dual problems and $c\bar{x} = \bar{u}b$, then $\bar{x}$ and $\bar{u}$ are optimal solutions.*

Theorem 15 (Strong Duality)**.** *If either problem of a dual pair of problems has a finite optimum, then the other does also and the two optimal objective values are equal; if either has an unbounded optimum, the other has no feasible solution.*

Proof. For proof, see page 56 of *Combinatorial Optimization: Networks and Matroids*, by Eugene Lawlor. The full citation can be found in the References. □

In the next section, we will be able to verify the optimality of the maximal flow solution using the orthogonality condition of optimal solutions.

Theorem 16 (Orthogonality of Optimal Solutions)**.** *If $\bar{x}$ and $\bar{u}$ are feasible solutions to the dual problems in the form outlined above, then $\bar{x}$ and $\bar{u}$ are optimal if and only if ($\bar{u}A$ - c)$\bar{x}$ = $\bar{u}$($A\bar{x}$ - b) = 0. That is, for $j = 1, 2, \ldots, n$,*

$$\bar{x}_j > 0 \implies \sum_{i=1}^{m} \bar{u}_i a_{ij} = c_j$$

and for $i = 1, 2, \ldots, m$,

$$\bar{u}_i > 0 \implies \sum_{j=1}^{m} a_{ij}\bar{x}_j = b_i.$$

Proof. Suppose that $(\bar{u}A - c)\bar{x} = \bar{u}(A\bar{x} - b) = 0$. Then by distributing and rearranging terms, we find that $\bar{u}A\bar{x} = c\bar{x} = \bar{u}b$. By assumption, $\bar{x}$ and $\bar{u}$ are feasible solutions. Thus by Corollary 1, they are optimal.

Conversely, suppose that $\bar{x}$ and $\bar{u}$ are optimal solutions. Then by the theorem of Strong Duality, the objective values are equal; that is, $c\bar{x} = \bar{u}b$. Since $\bar{u}A\bar{x} = c\bar{x} = \bar{u}b$, we can immediately see that $(\bar{u}A - c)\bar{x} = \bar{u}(A\bar{x} - b) = 0$. □

6.3. The Max-Flow Min-Cut Theorem as a Linear Programming Problem.

Now that we have some idea of how linear programming problems are solved, we can apply this knowledge to solving maximal flows. In fact, we can even view the max-flow min-cut theorem as a consequence of strong duality.

Recall that the primal linear programming problem is
maximize v
subject to

$$\sum_j x_{ji} - \sum_j x_{ij} = \begin{cases} -v, & \text{if } i = s \\ 0, & \text{if } i \neq s, t \\ v, & \text{if } i = t \end{cases}$$

$$x_{ij} \leq c_{ij}$$
$$x_{ij} \geq 0.$$

Then the dual problem is
minimize $\sum\limits_{i,j} c_{ij} w_{ij}$
subject to

$$\begin{aligned} u_j - u_i + w_{ij} &\geq 0 \\ u_s - u_t &\geq 1 \\ w_{ij} &\geq 0 \\ u_i \text{ unrestricted.} \end{aligned}$$

where u_i is a dual variable associated with the ith node equation and w_{ij} is a dual variable associated with the capacity constraint on edge (i, j).

Recall that for any *(s,t)*-cutset there is a corresponding flow; that is, there is a feasible solution to the dual problem where the objective function, v, is equal to the capacity of that cutset. If this is the case, then we can also say that there is an optimal solution which corresponds to an *(s,t)*-cutset. We will derive the optimal solution as follows:

For any cutset (S, T), let

$$\mathrm{u}_i = \begin{cases} 1, & \text{if } i \in S \\ 0, & \text{if } i \in T \end{cases} \qquad \mathrm{w}_{ij} = \begin{cases} 1, & \text{if } i \in S, j \in T \\ 0, & \text{otherwise} \end{cases}$$

For the optimal solution, we must construct the cutset in this way:

Assume $u_t = 0$ and $u_s = 1$. The remaining variables take on values of 0 or 1. For each edge (i, j), $w_{ij} = 1$ if and only if $u_i = 1$ and $u_j = 0$. Then let

$$S = \{i | u_i = 1\}$$
$$T = \{j | u_j = 0\}$$

The capacity of cutset (S, T) is equal to the value of the optimal dual solution. Thus we have shown that the dual problem is capable of finding a minimum capacity $(s, t)-$cutset, to produce an optimal value, as implied by the theorem of Strong Duality. This is precisely the statement of the max-flow min-cut theorem.

In addition, note that the primal and dual solutions are optimal if and only if they uphold the orthogonality condition, as presented in Theorem 14. That is,

$$x_{ij} > 0 \implies u_j - u_i + w_{ij} = 0$$
$$w_{ij} > 0 \implies x_{ij} = c_{ij}.$$

Consider the variable u_i as the "potential" of node i. Then for each edge (i, j) in an optimal pair of solutions, one of three cases can occur:

1) The potential at i is less than the potential at j; the flow in (i, j) is zero.
2) The potential at i is equal to the potential at j; the flow in (i, j) may or may not be positive.

3) The potential at j is greater than the potential at j; the flow in (i, j) is equal to the capacity of the edge, c_{ij}.

In terms of maximal flows, these cases all make intuitive sense. Thus we have shown that solving the maximal flow problem as a linear program with duality allows us to derive similar results as we did using the max flow algorithm, including a derivation of the max-flow min-cut theorem.

7. Conclusions

We have seen that there exist some interesting relationships between several problems, including finding the shortest path in a graph, finding the maximal flow, and solving linear programming problems with duality. From the methods discussed to solve such problems, we can adapt our approach to solve even more complex problems. For example, we can add a cost to each edge flow in a network, and find a minimum cost flow, or we can suppose that our system does not conserve flow at all nodes and edges; that our network sustains losses and gains. These are just a few among many examples of adaptations of network flows.

Numerous real-world problems can not only be formulated as such problems, but also solved efficiently. We did not discuss time complexity in this paper, but the methods introduced generally run in polynomial time, and there are ways to force more complex problems into polynomial-bounded running times as well. For further reading, see the first three references.

The roots of combinatorial optimization come from daily problems confronted by every society, so we can say with certainty that the importance of this subject will never fade and it is one that has far-reaching impact.

8. References

(1) Jon Kleinberg and Eva Tardos, *Algorithm Design*, Cornell University, Pearson Education Inc., 2006.

(2) Bernhard Korte and Jens Vygen, *Combinatorial Optimization: Theory and Algorithms*, 6ed., Springer, 2018.

(3) Eugene Lawlor, *Combinatorial Optimization: Networks and Matroids*, Dover, 2001.

(4) Alexander Shrijver, "On the History of the Shortest Path Problem", *Documenta Math.*, Extra Volume ISMP, 2012, p.g. 155-167.

(5) Alexander Shrijver, "On the History of Combinatorial Optimization (Till 1960)", *Handbooks in Operations Reasearch and Management Science*, Volume 12, 2005, p.g. 1-68.