

SIGNATURE PAGE

This thesis for honors recognition has been approved for the

Department of Mathematics.



Director

4/30/18

Date

Dr. Jodi Fasteen

Print Name




Reader

4/30/18

Date

Dr. Kay Satre

Print Name



Reader

4/30/18

Date

Dr. Amanda Francis

Print Name

Applications of Network Flows

Erica Wiens

May 1, 2018

Abstract

Have you ever played a board game or a video game where you figured out the shortest path between two locations before moving? Or have you ever been following directions from Google Maps and noticed a red patch of slow moving traffic coming up? Then you have unknowingly seen and used network flows. Network flows are used to optimize a variety of problems including the shortest path problem, the maximum flow problem, and the minimum cost problem. Through applications, we will explore a few ways to approach these problems to determine how computationally efficient these algorithms are.

Contents

1	Introduction	3
2	Big \mathcal{O} notation	5
3	Shortest Path Algorithms	6
3.1	Label Setting: Dijkstra's Algorithm	6
3.1.1	Example of Dijkstra's Algorithm	7
3.1.2	Dial's Implementation	9
3.1.3	Radix Heap Implementation	9
3.2	Label Correcting: Generic Algorithms	10
3.2.1	Modified Generic Algorithm	10
3.2.2	Implementations	11
3.3	Label Correcting: Floyd-Warshall Algorithm	11
3.3.1	Negative Cycles	12
3.4	Applying Shortest Paths to <i>Ticket To Ride</i>	12
3.5	Summary	13
4	Maximum Flow	14
4.1	Ford-Fulkerson Algorithm	14
4.1.1	Example of Ford-Fulkerson Algorithm	14
4.1.2	Edmonds-Karp Implementation	16
4.2	Push/Relabel Algorithm	17
4.2.1	Implementations	17
4.2.2	Example of Push/Relabel Algorithm	18
4.3	Blocking Flow Algorithm	20
4.4	Application: <i>Ticket To Ride</i>	21
4.5	Summary	22
5	Minimum Cost	23
5.1	Minimum Cost Concepts	23
5.2	Successive Shortest Path Algorithm	24
5.2.1	Implementations	24
5.2.2	Example	25
5.3	Application to <i>Ticket to Ride</i>	29
6	Conclusion	29
	References	30
	Appendix	31

1 Introduction

Have you ever tried to find the shortest distance from one place to another? Or have you ever wanted to know the fastest way to get somewhere and used Google Maps to find it? How about trying to find the cheapest way to move a specific number of things from one place to another?

Consider this scenario: you and some of your friends are going on a road trip. Your group wants to see as many interesting landmarks and attractions as possible, but you only have enough gas money to drive 1,000 miles round trip. We are still in college after all. How would you choose the best path to take? Would you start by making a list of all of the attractions within 500 miles of your starting destination? Or would you choose a few specific landmarks and find the shortest path between all of them? Perhaps you and your friends were instead heading up to the Great Divide Ski Area for a fun day of skiing. When looking up directions you would likely see an image similar to the one in Figure 1 that provides multiple options based on time and distance.

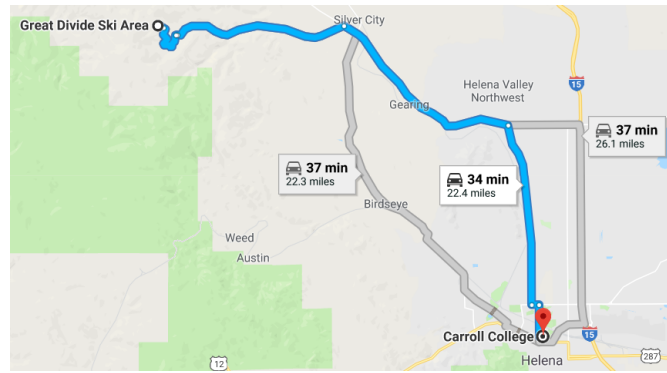


Figure 1: Directions from Carroll College to the Great Divide Ski Area

Here's another scenario. Have you ever been driving down the highway and notice that cars seem to be slowing down to a crawl? Or have you ever been using Google Maps (or an equivalent application) for directions only to see that the section of road ahead of you has turned to red on the map? We know from experience that a red section of road is not moving as fast as it could be due to road congestion. In places, such as Los Angeles, CA, people have come to expect road congestion and have built bigger roads with higher vehicle capacities to combat road congestion. In Figure 2, we can see the traffic congestion in Los Angeles on a Monday afternoon.



Figure 2: Los Angeles, CA traffic congestion on a Monday afternoon.

One final scenario to consider. Imagine that you are in charge of product distribution for a large company. You are able to ship products using a few methods from manufacturing plants,

to distribution centers, then at last to the stores where the products are sold. If we assume that we have unlimited funds, then we want to find the fastest way to ship out the products. Alternatively, we could assume that we have a finite budget in which we must distribute as many products as possible as cheaply as possible. Or perhaps there are capacities on how much we can send through each method of transportation, but we still must distribute as many products as possible.

Did you know that each of the previous three scenarios can be optimized using network flows? Now the question that arises is “What is a network flow?” Network flows are a way to symbolically represent a network that has some sort of flow through it. A tangible example of a physical network flow is a series of water pipes. Water enters the network of pipes at a source and moves, or flows, through the network to the end. Simply put a network flow is the flow through a network.

In the first road trip scenario we can apply a shortest path algorithm maximize fun within a set distance. By analyzing the network of roads extending outwards from Helena, Montana, to find the attractions within 500 miles, we can find the shortest path from a location to any other location.

When we consider the traffic scenario, we can think of the number of cars that travel along a road to be the flow on that road. Then we can optimize the number of cars that should be on the road based on the capacity of the road using maximum-flow algorithms. Instead of waiting in barely moving traffic on a street with high flow, we could be cruising along down a less crowded street with a lower flow.

Finally, the third scenario of product distribution is a classic optimization problem that can be solved using network flows [4]. Using a minimum flow algorithm we can optimize the cost of sending a fixed amount of product from factories to shops. Alternatively, we could use a minimum flow algorithm to find the shortest, in distance, or the fastest, in time, way to send products from one location to another.

Network flows apply to a variety of real world problems and are often used for optimizing routes, flows, and costs. Typically network flows will begin as a graph representation of a problem, such as LA traffic, and then be converted into a set of paths and locations. By turning the graph into sets of paths and locations, we are able to use computer algorithms to solve large network flows.

Suppose you are in charge of product distribution for a large company. You will likely be in charge of hundreds of factories, warehouses, and stores. If we were finding the best (cheapest, shortest, fastest) path for each product by hand it would take far longer than we would like. Fortunately, there are a number of computer algorithms that have been developed to solve the shortest path, maximum-flow, and minimum-cost problems.

The shortest path problem is, as the name suggests, focused on trying to find the shortest path from point A to point B. This problem alone has many real world applications and we unconsciously use less formal versions of shortest path algorithms to find the shortest path between two places. For example, imagine that you are leaving your house to head to the grocery store and part of the route you normally takes has been closed due to road construction. We can use shortest path algorithms to find the shortest path to the grocery store even with the closed roads.

We will explore two types of shortest path algorithms: label setting and label correcting. Label setting algorithms set distance labels as permanent on each iteration through the algorithm whereas label correcting algorithms only set distance labels as permanent at the conclusion of the algorithm. The three algorithms that we will explore are Dijkstra’s Algorithm, a generic

label correcting algorithm, and the Floyd-Warshall algorithm. These algorithms will help us to optimize travel from one place to another based on cost, distance, time, or other factors.

The maximum flow problem aims to optimize the flow that passes through a system. Think of a network of water pipes. The goal is send as much water through the network as possible. By using a maximum flow algorithm we can figure out just how much water can be sent through the pipes. Maximum-flow algorithms can also help to optimize the flow of text messages through cell phone towers or e-mails through internet networks.

The maximum flow algorithms that we will explore include the Ford-Fulkerson algorithm, the Push/Relabel or Preflow-Push algorithm, and the Blocking Flow algorithm. Each of these algorithms has its own merits as well as implementations to reduce the worst case run time for the algorithm.

The final type of network flow problem that we will explore is the minimum cost problem. This problem refers to sending a set amount of flow as cheaply as possible. Network flow problems are one of the most common types of optimization problems. Consider a company that has to transfer products from the factory through distribution centers to the stores where the products are sold.

Each of these three types of problems use network flows to optimize cost, distance, time, or some other factor. Let's go!

2 Big \mathcal{O} notation

Throughout the course of our exploration we will be coming back to the idea of Big \mathcal{O} notation or Landau's symbol. This notation has been adopted by mathematicians and computer scientists alike to explain the asymptotic behavior of an algorithm [8]. In other words big \mathcal{O} notation explains how the worst case run time of an algorithm changes with larger networks and the complexity of the algorithm.

By definition, big \mathcal{O} notation states that when we have two functions $f(x)$ and $g(x)$, which are defined on some domain in the real numbers, then

$$f(x) = O(g(x))$$

if and only if there exist constants N and C such that for all $x > N$,

$$|f(x)| \leq C|g(x)|.$$

That is to say when x is sufficiently large the value of $C|g(x)|$ will be greater than or equal to the value of $f(x)$.

For example if an algorithm is said to have complexity $\mathcal{O}(1)$ then the algorithm will always run in the same amount of time. However if the algorithm has a complexity of $\mathcal{O}(n)$ then the time required to complete the algorithm grows linearly with the number of nodes in the network. When an algorithm has a complexity of $\mathcal{O}(n^2)$ then when the number of nodes in the network is doubled the time it takes to complete the algorithm increases by a multiple of 4.

In the following sections, we will use big \mathcal{O} notation as a method for comparing the algorithms that solve each problem.

3 Shortest Path Algorithms

Shortest Path Algorithms were designed to find the shortest path from one place to another in a network. The shortest path could be shortest by distance, time, cost, or another factor that we wish to minimize. There are multiple ways to find the shortest path from one place to another on a graph. It is important to note here that the nodes in a network do not have to be physical locations. They could be the path that an e-mail or text message takes as it travels to the recipient from the sender. There are two ways to approach the shortest path problem: label setting and label correcting.

Label setting algorithms, such as Dijkstra's algorithm, break the nodes on the graph up into two sets, those with temporary distance labels and those with permanent distance labels. These algorithms typically begin with every node except source node in the temporary set and move the nodes into the permanent set when it is clear that the distance label has been minimized. As we will explore later, these algorithms set a new permanent label on every iteration through the algorithm.

In comparison, label correcting algorithms keep every node with a temporary distance label until the algorithm is complete. When a label correcting algorithm is complete, all of the nodes become permanently labeled nodes. These algorithms correct labels until the algorithm is completed rather than setting labels as permanent throughout.

3.1 Label Setting: Dijkstra's Algorithm

Dijkstra's Algorithm is one of the most well known ways to find the shortest path from the source node of a network to any other node in the network. It was developed by Dutch computer scientist Edsger W. Dijkstra and published in 1956 [1]. Dijkstra's algorithm is an example of a label setting algorithm.

For Dijkstra's algorithm, each edge of a network has a non-negative distance and each node has a distance label. When setting up the network, all of the node distance labels, except for the source, are set to infinity. The source has a distance label of 0 because we are starting there. The distance labels are broken up into temporary and permanent labels. Permanent labels represent the actual shortest distance to a node from the source where temporary labels represent an upper bound of the shortest distance to the source.

On each iteration of Dijkstra's algorithm, we begin at the most recent node to be permanently labeled. On the first iteration this will be the source. We then move out along the edges connected to the most recent permanent node and update the temporary distance labels on those nodes. If the distance labels decrease the label will have a subscript letter, ex: 4_s , to denote the previous node. The previous node, or predecessor node is often denoted as $pred(j) = i$ where $i, j \in N$. Then we find the minimum temporary distance label and make it a permanent distance label. Since it is the minimum temporary distance it is not possible to reduce the distance further by traversing another path. Thus it is safe to say that the minimum temporary distance is permanent.

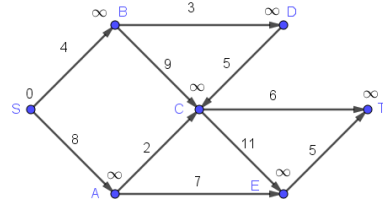
The next iteration of the algorithm will begin with the newest permanent distance label. The algorithm continues until every node has a permanent distance label. In other words the algorithm will have $n - 1$ iterations and will run in $\mathcal{O}(n^2)$ time in the worst case where n is the number of nodes in the network [10].

3.1.1 Example of Dijkstra's Algorithm

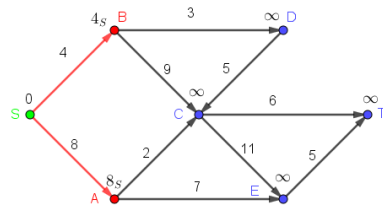
To better understand how Dijkstra's algorithm works, we will work through an example using the network shown below.

There are two ways to keep track of the iterations through Dijkstra's algorithm. One is to update distance labels on the network itself and the other is to use a table. For the sake of this example we will update the distance labels on a network and show the table at the end. As we iterate through the algorithm, green nodes on the network are permanent, blue nodes are temporary, and red shows that the node is being evaluated on that iteration of Dijkstra's algorithm.

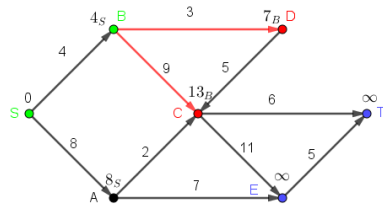
Before we begin Dijkstra's Algorithm, we must first set up the temporary and permanent distance labels. The only permanent distance label is on the source node, set to 0, and the remaining nodes are temporary distances of ∞ . In our table at the end this step is referred to as iteration 0.



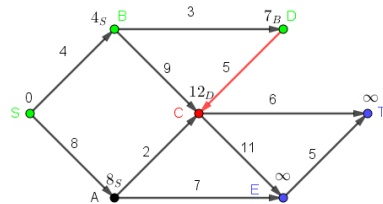
On the first actual iteration of Dijkstra's algorithm, we begin at the source node, S , and extend out along the arcs leaving S to nodes A and B . Nodes A and B are then updated with 8_S and 4_S respectively. The subscript letter indicates the node that we came from to achieve that distance. Since B has the lowest temporary distance label, B is added to the set of permanently labeled nodes.



Now, on the second iteration through Dijkstra's algorithm, we begin at node B , the most recent permanently labeled node. We can see that there are two arcs leaving node B going to nodes C and D . Since neither distance label has been updated away from ∞ yet, they are updated to 13_B for C and 7_B for D . The lowest temporary distance label is now node D with a distance of 7. Node D now has a permanent distance label.



The third iteration, will begin with node D . From here, we only extend out to node C and check its temporary distance label. The temporary label on C is updated to 12_D . The lowest temporary distance is on node A , so this distance label becomes permanent.

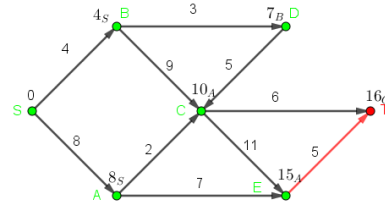
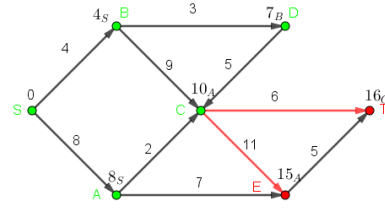
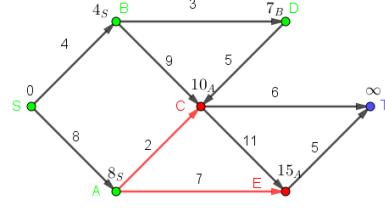


At this point through the algorithm, nodes S , A , B and D have permanent labels and nodes C , E , and T have temporary labels. We will have at most three more iterations until every point is labeled with the shortest distance to that point from the source.

On our fourth iteration we follow the arcs leaving node A to nodes C and E . From A the distance to C would be 10 which is less than the current temporary distance on B , so B is updated to 10_A . However, node E has not yet been updated so it gets updated to 15_A . The lowest temporary distance label is 10 on C , so 10_A becomes a permanent label on C .

We begin with node C for the fifth iteration through Dijkstra's algorithm. C connects to E but does not reduce the temporary distance to E . C also connects to T , which has not yet been updated, so the distance label on T is updated to 16_T . Now we move on to finding the lowest temporary distance label. We can see that the lowest temporary distance label is on node E . Then the distance on E becomes permanent and we begin our next iteration from E .

On our sixth and final iteration of Dijkstra's algorithm we follow the arc leaving E to T . As we can see, moving from E to T does not decrease the distance to T so T will not be updated. Since there is only one temporary node left, it becomes permanent and we have completed Dijkstra's algorithm on this network.



The final distance labels on the network can be seen in Figure 3. Did you notice how none of the labels were updated after the fourth iteration? This is because at that point we had already found the shortest distance from the source to each point. By using Table 1 to follow each iteration of Dijkstra's algorithm, it is easier to see that none of the nodes are being updated. In the table bold values represent permanent distance labels.

	S	A	B	C	D	E	T
0	0	∞	∞	∞	∞	∞	∞
1	0	8_S	4_S	∞	∞	∞	∞
2	0	8_S	4_S	13_B	7_B	∞	∞
3	0	8_S	4_S	12_D	7_B	∞	∞
4	0	8_S	4_S	10_A	7_B	15_A	∞
5	0	8_S	4_S	10_A	7_B	15_A	16_C
6	0	8_S	4_S	10_A	7_B	15_A	16_C

Table 1: Final Results from Dijkstra's algorithm in table form

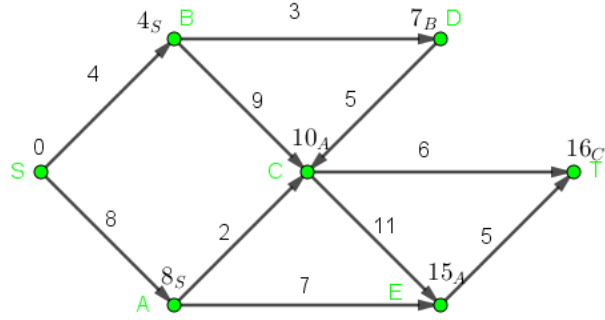


Figure 3: Results of Dijkstra's Algorithm in Graph Form

3.1.2 Dial's Implementation

The computational bottleneck of Dijkstra's algorithm is in finding the lowest temporary distance label. As such, if we can reduce the time it takes to find the lowest temporary distance label, then we can reduce the total run time of Dijkstra's algorithm.

Dial's implementation attempts to do just that by sorting the temporary distance labels into buckets of size 1. In this instance, the size of the bucket relates to the distance labels rather than the number of nodes in the bucket [10]. That is to say, that a bucket will hold all of the temporary distance labels that share the same value. These buckets can contain more than one node. There can be up to $nC + 1$ buckets where C is the longest edge distance and n is the number of nodes in the network. The buckets are stored in a linked list and, by moving through the linked list, we can find the first non-empty bucket. Then we only have to find the minimum distance label from the nodes in that bucket. There is also a circular way to store the buckets where the lowest bucket has the same value as the lowest temporary distance label. The circular way to store buckets has C buckets because that all of the distance labels will be within C of the most recent permanent distance label.

If Dial's implementation had been used for the example of Dijkstra's algorithm above, after the first iteration the only buckets that would contain nodes are buckets 4 and 8. Then bucket 4 is the first bucket to contain a node and the temporary label 4_S becomes a permanent node. Now the only bucket that contains a temporary label is bucket 8 until the next iteration.

This implementation runs in $\mathcal{O}(m + nC)$ time where m is the number of buckets, n is the number of nodes, and C is the longest edge distance. The lower run time of this implementation is due to using buckets to break up the nodes. Directly finding the minimum distance from a network of hundreds or thousands of nodes can be computationally expensive, so using an implementation that reduces the complexity will reduce the time it takes to complete the algorithm.

3.1.3 Radix Heap Implementation

Another implementation of Dijkstra's algorithm for reducing run time is the radix heap implementation. This implementation is part of a family of other heap implementations, named as such because the nodes and arcs are arranged into heaps.

This implementation uses the same buckets as Dial's implementation, but they are cleverly designed so that the number of nodes that the buckets contain are 1, 1, 2, 4, 8... where the distance labels inside the buckets are monotonously increasing [10]. These buckets are not

necessarily of length 1 and their sizes vary due to the spread of the distance labels. The nodes are placed into the proper bucket as before, but this time the first bucket only contains one node. As such, it is much easier to find the lowest temporary node as it is in a bucket by itself. This implementation has a worst case run time of $\mathcal{O}(m + n \log(nC))$ [10].

3.2 Label Correcting: Generic Algorithms

From the name it may seem like label correcting algorithms are almost the same as label setting algorithms, such as Dijkstra’s algorithm. However, in label correcting algorithms, each distance label on a node is considered temporary until the algorithm is complete.

Label correcting algorithms find the shortest path by reaching an optimality condition

$$d(j) > d(i) + c_{ij} \tag{1}$$

where i, j are nodes in the set of nodes N , $d(i)$ represents the distance to node i from the source and c_{ij} represents the distance between i and j . When the equation in (1) is true, then the distance label on node j is updated so that the condition is false.

A generic version of this algorithm begins by setting the distance on the source to 0 and the distances on every other node to ∞ . The algorithm continues until the optimality condition in (1) is not satisfied by any arc (i, j) where $i, j \in N$. If the distance to j is updated, then the predecessor node, $pred(j)$ is updated. As a result of this algorithm, we have a predecessor tree that shows us how the nodes relate to one another based on the preceding node.

Since the arcs are arbitrarily chosen, this algorithm requires $\mathcal{O}(m)$ time to determine if the optimality condition is violated where m is the number of arcs and each node is updated at most $2nC$ times, such that n is the number of nodes and C is the length of the longest arc. As such this algorithm runs in $\mathcal{O}(mn^2C)$ time.

The generic label correcting algorithm allows for negative arc lengths. However, if there are negative arc lengths the algorithm will never converge to the answer as some nodes will be constantly updated with lower and lower distances. Since we know the length of the longest arc C and the number of nodes n , we know that the upper bound on any distance from the source has an upper bound of nC . Similarly, we are able to find that the lower bound of the distance is $-nC$. As such, if the algorithm finds a distance label less than $-nC$ the algorithm will terminate so that it does not run forever.

3.2.1 Modified Generic Algorithm

The generic label correcting algorithm has to reevaluate each arc every iteration, which is not very effective. In order to reduce computation time, a modified version of the generic algorithm was developed. This algorithm uses a list to keep track of nodes that may not yet be optimized.

The list begins with only the source node, then the source node is removed from the list and the nodes at the end of arcs leaving the source are added. After this, each of these nodes are evaluated to see if they violate the optimality condition from Equation (1). If an arc branching off from a node violates the optimality condition then the ending node is added to the list if it is not already there. The algorithm continues until the list of nodes is empty.

This algorithm has a run time of $\mathcal{O}(nmC)$ which is less than the original generic algorithm because we no longer have to check every arc to see if it violates the optimality condition.

3.2.2 Implementations

A few implementations of the modified label correcting algorithm further reduce the run time. One such implementation is known as the FIFO or First In First Out implementation. All nodes that need to be evaluated or reevaluated are appended to the end of the list. The list will then take the first node on the list, remove it from the list, and evaluate the arcs leaving the node. This implementation runs in $\mathcal{O}(mn)$ time due to the way that it treats the list [10].

Another implementation of the modified label correcting algorithm is the Dequeue implementation. This implementation adds nodes to either the beginning or the end of the list. If a node has been evaluated before, it is added to the beginning of the list; and if it has not been evaluated yet, the node is added to the end of the list. For example, if node A has already been evaluated, it is added to the beginning of the list. However, if A has not yet been evaluated it will be added to the end of the list. By using the dequeue implementation, there is a higher chance that nodes towards the end of the list will only have to be evaluated once, which helps to reduce the chance of evaluating the same few nodes over and over again. Empirically, this implementation has been more efficient than the FIFO implementation, but there is no mathematical proof to show this result.

3.3 Label Correcting: Floyd-Warshall Algorithm

Just as there is a way to apply Dijkstra's algorithm to find the distance between two specific nodes, there is a label correcting algorithm for finding the shortest path between **any** two nodes. This family of algorithms is known as all pairs shortest path algorithm and includes the Floyd-Warshall algorithm.

Before we can understand this algorithm, we must first understand the new optimality conditions for all pairs shortest path algorithms. In this optimality condition, let $d[i, j]$ represent the distance between nodes i and j where $i, j \in N$. These distances represent the all-pairs shortest path distances if and only if they satisfy

$$d[i, j] \leq d[i, k] + d[k, j] \tag{2}$$

for all nodes i, j , and k [10].

In the generic version of the all-pairs shortest path algorithm, the algorithm continues until the optimality condition in Equation (2) is satisfied. However, the generic algorithm does not include a methodical way to iterate through all of the combinations of nodes. The Floyd-Warshall algorithm provides a cleverly simple way to evaluate all possible node combinations.

The algorithm begins by setting the distances between all nodes to ∞ and the distance between the node and itself, $d[i, i]$, to 0. Setting $d[i, i] = 0$ may seem silly, but this will come in handy in the future. Set-up also includes setting the arc distance between two points that are directly connected and setting the predecessor node for each arc.

The meat of the Floyd-Warshall algorithm is a loop that iterates through each node $k \in N$. Then for each pair $[i, j]$ we check the optimality condition. If $d[i, j] > d[i, k] + d[k, j]$ then $d[i, j]$ is updated and the predecessor node is updated to that of $d[k, j]$. That is $pred[i, j] = pred[k, j]$.

This algorithm performs n operations in the loop through each node k . Each of these operations occurs in $\mathcal{O}(1)$ time which gives this algorithm an over run time of $\mathcal{O}(n^3)$ [10].

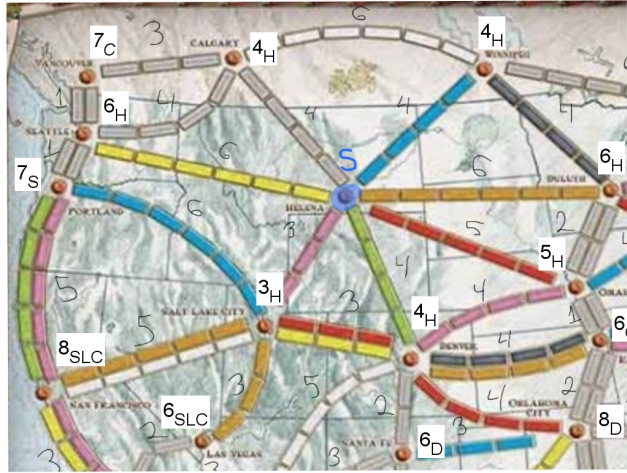


Figure 5: Results of Dijkstra’s algorithm on *Ticket to Ride*

3.5 Summary

The two large categories of algorithms that solve the shortest path problem are label setting and label correcting. Label setting algorithms have two sets of nodes, those with permanent distance labels and those with temporary labels. The best known label setting algorithm is Dijkstra’s algorithm that runs in $\mathcal{O}(n^2)$ time where n is the number of nodes. We discussed two implementations for Dijkstra’s algorithm, Dial and Radix Heap, which run in $\mathcal{O}(m + nC)$ and $\mathcal{O}(m + n \log(nC))$ time respectively where m is the number of arcs and C is the largest/longest arc. These implementations reduce the worst case runtime by organizing the distance labels into buckets.

For label correcting algorithms, we explored a generic algorithm and the Floyd-Warshall algorithm. The generic algorithm found the shortest distances by using the optimality condition in Equation (1). This algorithm has a worst case run time of $\mathcal{O}(nmC)$. The implementations of the generic algorithm, FIFO and dequeue, have worst case run times of $\mathcal{O}(nm)$ as a result of keeping possibly wrong nodes in a list.

4 Maximum Flow

In addition to the shortest path problem, the maximum flow problem can be solved using network flows. The maximum flow problem relates to how much flow can go through a network. Networks for maximum flow look slightly different from those used for shortest paths. The arcs of maximum flow networks have capacities, flows, and residual flows instead of distances.

We will discuss three different maximum flow algorithms: Ford-Fulkerson, Push/Relabel, and Blocking Flow. Each of these algorithms approaches the maximum flow problem in a slightly different way. The Ford-Fulkerson algorithm uses augmenting paths from the source to the sink. By comparison the Push/Relabel algorithm uses the idea of flow excess and the Blocking Flow algorithm moves through the network one arc at a time.

These algorithms will achieve the same maximum flow, with varying complexity and worst case run times.

4.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson method was developed by L.R. Ford and D.R. Fulkerson in 1956 to compute the maximum flow through a flow network. To accomplish this goal, this algorithm finds augmenting paths through a network from the source of the network to the sink of the network. Every network has a unique source node and a unique sink node. If the source and sink nodes are the same, then there is no flow through the network.

Each edge in a flow network is associated with a direction, a flow, and a maximum capacity. These characteristics of the edges are then used to find the maximum flow through a network.

Ford-Fulkerson can be broken down into 3 easy steps [5]:

1. Find an Augmenting Path, a , through the network from source to sink.
2. Find the maximum flow, f , through a .
3. Increase the flow through each edge by the flow f .

The first step of the Ford-Fulkerson algorithm is to find an augmenting path from the source to the sink. This path can contain both forward and backward edges and is arbitrarily chosen from the edges that are not yet at their maximum capacity.

The next step of the Ford-Fulkerson algorithm is to find the maximum flow through the chosen augmenting path. We can find the maximum flow by finding the minimum difference between each edge's flow and capacity.

Finally, the flow of each forward edge is increased by the maximum flow through the path and the flow of each backward edge is decreased by the maximum flow through the path. It is important to note here that edges must have a flow that is between 0 and the capacity of the edge.

4.1.1 Example of Ford-Fulkerson Algorithm

In order to better understand the mechanics of the Ford-Fulkerson algorithm, we can work through an example using the graph in Figure 6. The source of our starting graph is the vertex labeled S while the sink vertex is labeled with a T . Each arc of this network is labeled with the flow and the capacity in the form "flow, capacity".

From Figure 6 we can find an augmenting path from the source S to A , then D and completing at the sink T . This path is highlighted in Figure 7 and was arbitrarily chosen. This algorithm does not specify an order for finding augmenting paths, so they are arbitrarily chosen.

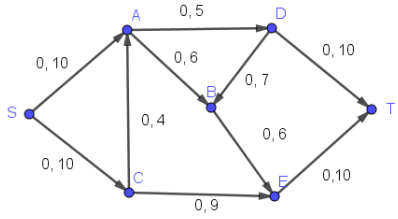


Figure 6: Starting Graph for the Ford-Fulkerson Example

Now that we have a path, we have to find the maximum flow through this augmenting path. We can see that the capacity from S to A is 10, A to D is 5, and D to T is 10. Since the maximum capacity through our augmenting path is 5, we then increase the flow of each of our edges by 5 as can be seen in the left network of Figure 7.

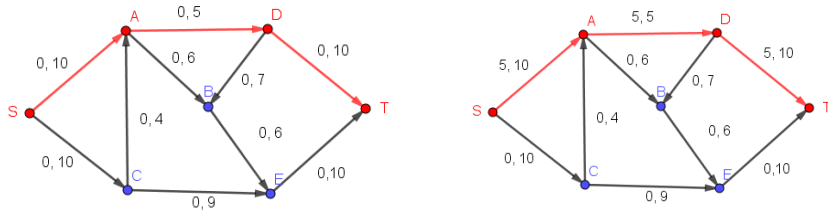


Figure 7: First Augmenting Path in the Ford-Fulkerson Example

Since the edge between S and A is not at capacity yet, we will continue to use that edge. However, the edge from A to D is at capacity so we can no longer use that edge in our augmenting path. Then a new augmenting path, as seen in Figure 8, goes from the source S to A , then B , E , and finally the sink T . Notice that every augmenting path begins at the source and ends at the sink of the network.

Now that there is flow through some of the edges in this network, we have to find the remaining flow through each of the edges. We can find that the remaining flow through the edge SA is 5 by subtracting the current flow through SA , 5, from the capacity of SA , 10. Similarly, the edge AB has a capacity of 6 and a current flow of 0, so the maximum flow that could go through AB is 6. We can also find the maximum flow through BE to be 6 and the maximum flow through ET is 10.

Since the minimum of the maximum flows is 5, we add 5 to the flow through each of the edges in this augmenting path. Our updated graph can be seen in the right graph of Figure 8.

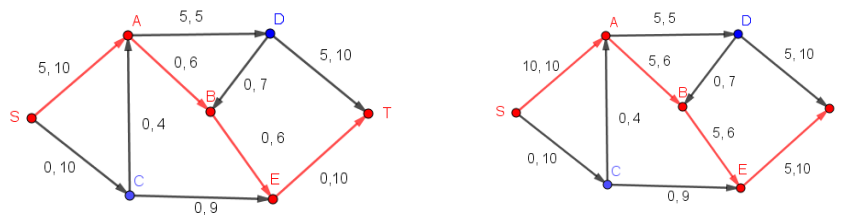


Figure 8: Second Augmenting path of the Ford-Fulkerson Example

For the next augmenting path, the edge SA is at capacity, so it cannot be used anymore. Now

we must build our augmenting path from the edge SC . The next arbitrarily chosen augmenting path is $SCABET$ as seen highlighted in Figure 9. From the graph in Figure 9, we can see that the maximum flow through this augmenting path is 1. This comes from the difference between the capacity and the flow of the edges AB and BE .

Each edge is then augmented by 1 as seen in the right hand graph in Figure 9.

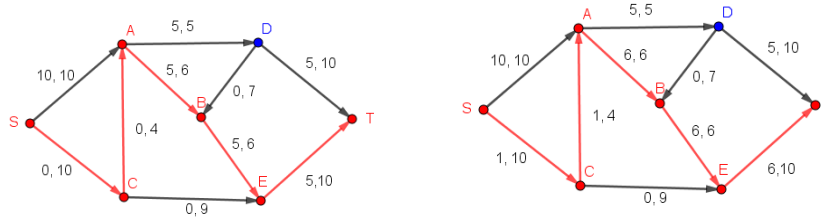


Figure 9: Third Augmenting path from the Ford-Fulkerson Example

Once more, we can find an augmenting path from S to T through the network. The augmenting path can no longer go through the edge CA , because each of the edges leaving from A is at maximum capacity. So the augmenting path must use the edge CE . Thus the fourth augmenting path from S to T is $SCET$, as seen in the left graph of Figure 10.

The maximum flow through the graph is limited by the edge ET , which has a remaining flow of 4. We can then augment each of the edges in this augmenting path by 4, the results of which can be seen in the right graph of Figure 10.

From the right graph of Figure 10, we can see that 5 of the 10 edges are at capacity. Additionally, we can see that no more augmenting paths can be made through this network. The edge DT is not yet at maximum capacity, but every edge flowing into D is at maximum capacity. The edge ET is also at maximum capacity. Therefore, there are no more augmenting paths through the network. The final result of the Ford-Fulkerson example is in Figure 11. This graph has a maximum flow of 15 units of flow from the source S to the sink T .

4.1.2 Edmonds-Karp Implementation

In the Ford-Fulkerson method, augmenting paths are arbitrarily chosen. While humans are capable of arbitrarily choosing paths, computers are not. As such, Edmonds and Karp developed a way to choose paths in an orderly fashion. This implementation chooses the augmenting path by the number of edges. It begins with the shortest augmenting path and works towards the longer augmenting paths. That is, this algorithm uses the augmenting paths with the fewest number of edges first, then moves onto paths with more edges. By starting with the shortest augmenting path, this implementation is less computationally heavy with a worst case runtime

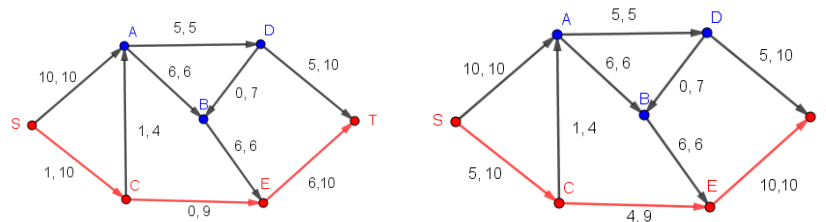


Figure 10: Fourth Augmenting Path from the Ford-Fulkerson Example

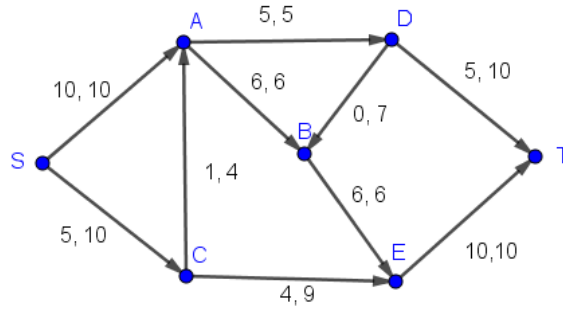


Figure 11: Final Graph from the Ford-Fulkerson Example

of $\mathcal{O}(n^2m)$ [7].

4.2 Push/Relabel Algorithm

The Push/Relabel algorithm for determining the maximum flow through a network was developed by Goldberg and Tarjan in 1988. This algorithm, in addition to finding the max flow through a network, uses distance labeling to determine the distance of a node from the sink. This type of algorithm is also referred to as Preflow-Push [2].

Push/Relabel algorithms do not use augmenting paths as the Ford Fulkerson algorithm does. Instead push/relabel algorithms use the excess flow at each node. The excess at each node is determined by the difference between the flow into each node and the flow out of each node. The source will always have a negative excess because there is no flow into the source. Similarly, the sink will always have a positive excess because no flow leaves the sink. Nodes with positive excess are considered active nodes. The source and sink nodes are not considered to be active by convention.

The algorithm begins by setting the distance labels on each node with the distance from the sink and by finding the excess on each node $i \in N \setminus \{s, t\}$ where the arc (s, i) exists in the network. Then, while there is an active node the algorithm chooses an active node i . If the network contains an admissible arc (i, j) then a flow of δ is pushed through arc (i, j) where δ is the minimum between the excess on i and the residual capacity between i and j . If there is no admissible arc then the distance on i is updated to $\min\{d(j) + 1 : (i, j) \in A\}$. When the excess of node i is pushed, this is considered to be a non-saturating push because there is still residual flow on arc (i, j) . A non-saturating push reduces the excess on i to 0 and i becomes an inactive node. This algorithm continues until there are no more active nodes. That is, there have been n non-saturating pushes made during the algorithm [11].

This algorithm runs in $\mathcal{O}(n^2m)$ time when a list is used to store the active nodes [10].

4.2.1 Implementations

As with the Ford-Fulkerson algorithm, specific implementations have been developed to reduce the run time of the algorithm.

One such implementation is the FIFO implementation, or First In First Out implementation, which runs in $\mathcal{O}(n^3)$ time by updating nodes in a first in first out order [10]. That is, the first node to be added to the list of active nodes is the first node to be evaluated. By using this

specification, the algorithm does not have to choose which active node to use.

Another implementation is known as the Highest-Label implementation. This implementation looks first at the active nodes with the largest distance labels and runs in $\mathcal{O}(n^2m^{1/2})$ time [10].

A final implementation is the Excess Scaling algorithm. This implementation has a run time of $\mathcal{O}(nm + n^2 \log(U))$ and pushes flow from a node with sufficiently large excess to a node with sufficiently small excess [10]. With the lowest worst-case run time, this implementation is the fastest implementation of the Push/Relabel algorithm.

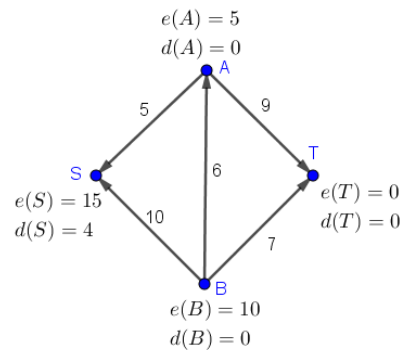
4.2.2 Example of Push/Relabel Algorithm

In order to better understand the Push/Relabel algorithm we will walk through an example using the Push/Relabel algorithm. This is the same network that we used for the Ford-Fulkerson example and as a sanity check we should find the same maximum flow as when we used the Ford-Fulkerson algorithm.

For the sake of this example, we will use the residual graph of the network. The residual network keeps track of the residual flow through each arc. Later in the example when we send flow through an arc we will “reverse” that arc for that amount of flow. For example if we send a flow of 5 through an arc with a capacity of 9 then there will be 4 forward flow remaining and 5 backwards flow remaining on that arc.

At the beginning of the algorithm we set the distance from the sink to the source to be the number of nodes in the network $d(s) = 4$. By setting the distance on the source to the number of nodes we ensure that flow will be moving towards the sink. We also send out flow from the source along the two arcs leaving the source. As such, we push a flow of 5 along the arc (S, A) and a flow of 10 along the arc (S, B) . Since we have pushed a maximum flow along the arcs (S, A) and (S, B) , then we reverse the direction of flow along those arcs.

When we initialized the algorithm the distances from each node to the sink were set as 0 and they will remain that way unless incremented by a relabel action.



On our first iteration through the algorithm, both nodes A and B are active nodes. Then we can arbitrarily choose to examine node A for this iteration. Then we check to see if there is an arc (A, j) that satisfies the inequality

$$d(A) \geq d(j) + 1,$$

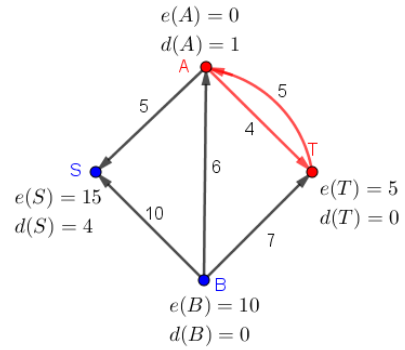
where j is a node in the network. Since there are no arcs that satisfy this inequality then $d(A)$ is updated to

$$\begin{aligned} \min\{d(j) + 1 : j \in N \text{ such that } (i, j) \text{ exists}\} \\ = \min\{d(T) + 1\} = 1. \end{aligned}$$

Now that $d(A) = 1$, we can reevaluate the inequality $d(A) \geq d(j) + 1$, and find that we can push flow along the arc (A, T) . The amount of flow that we can push is

$$\min\{e(A), c_{AT}\} = 5.$$

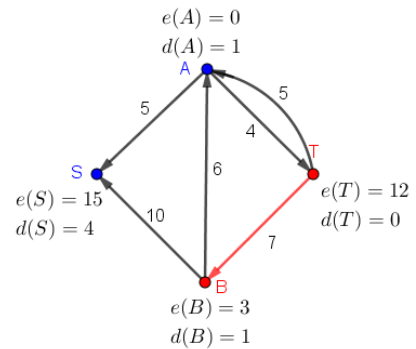
Thus, we push a flow of 5 along the arc (A, T) . This push is shown in red in the graph to the right. Notice that when we push a flow of 5, the residual graph shows a residual forward flow of 4 and a residual backward flow of 5.



On the second iteration through, the only active node is B . Now, we check to see if there are any nodes that satisfy the inequality $d(B) \geq d(j) + 1$ where $j \in \{A, T\}$. Since $d(B) = 0$, there are no nodes that satisfy this inequality. So we must update the distance label on B to

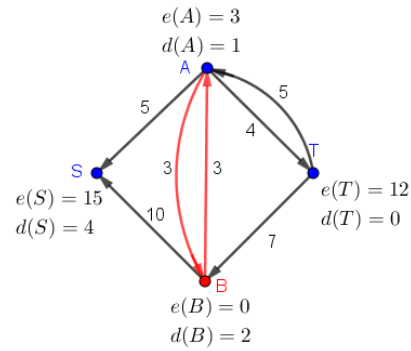
$$d(B) = \min\{d(A) + 1, d(T) + 1\} = 1.$$

With the updated distance label on node B we can now satisfy the inequality $d(B) \geq d(T) + 1$ and can augment along the arc (B, T) with a flow of 7. The flow is restricted by the capacity of the arc (B, T) . Then we update the residual graph with the new flow and update the excess on T to 12.

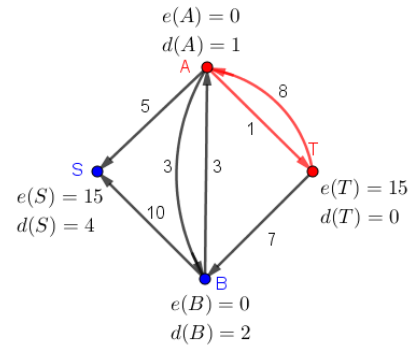


In the third iteration through the push/relabel algorithm, B is still the only active node. We once again check to see if B can push flow to other nodes. Node B can no longer push flow to T since that arc is at capacity. Thus we must check if $d(B) \geq d(A) + 1$ to see if B can push flow to A . Since this inequality is not satisfied then the distance on B is updated to $d(B) = 2$.

Now that the distance on B has been updated, we can push a flow of 3 along the arc (B, A) since $d(B) \geq d(A) + 1$. Then, we update the excess on node A to $e(A) = 3$ and the residual graph.



On our fourth, and final, iteration, node A is the only active node in the network. Node A could push flow to nodes S, B , and T since there are arcs extending out from A to those nodes. However, we must still satisfy the inequality $d(A) \geq d(j) + 1$ where $j \in \{S, B, T\}$. The only node that satisfies this is node T . Since $d(A) \geq d(T) + 1$ then we can push a flow of 3 through the arc (A, T) .



With this push, node A is no longer an active node. Thus we conclude the algorithm because there are no active nodes remaining. This network has a maximum flow of 15 units as seen in Figure 12.

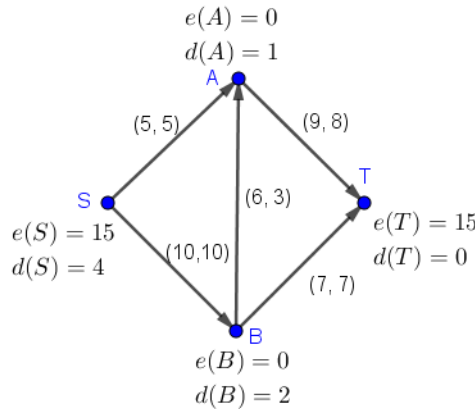


Figure 12: Result network of the Push/Relabel Example

4.3 Blocking Flow Algorithm

The blocking flow algorithm is similar to the Ford-Fulkerson algorithm but has a few key differences. These differences allow for a lower run time and lower memory usage.

This algorithm begins at the source node s and moves through the network one arc at a time by performing two actions: advancing and retreating. We can think of this like solving a maze.

Think of turns in the maze as nodes and straight paths in the maze as arcs. To solve a maze, we would advance forward from the start, or source node, until we hit a dead end. Then we would retreat back the way we came and “delete” the path that led to a dead end.

The pattern of advancing and retreating would continue until we reached the end of the maze, the sink node. However, a network can have multiple paths from the source to the sink, unlike a maze which usually has 1. So once we make it out of the maze, we delete all of the paths that we took through the maze and start again. Once there are no more paths leaving the source, the algorithm is complete and the maximum flow through the network is the number of paths created.

In the algorithm described above, the key assumption is that all arcs have a capacity of 1. This assumption may work in binary cases, either x happens or x doesn’t happen, but it is not helpful for real world applications.

As such, the blocking flow algorithm was updated to account for networks with capacities greater than 1. It works in a similar way using the two actions, advance and retreat. The difference comes at the end when, instead of deleting all of the arcs used, they are increased by the amount of flow that can go through the path. Then the arcs in the path that are now at capacity are deleted from the network and we begin again.

This algorithm has a maximum running time of $\mathcal{O}(n^2m)$ for finding the maximum flow through a capacitated network [6].

4.4 Application: *Ticket To Ride*

Once again we can apply these techniques to the game *Ticket to Ride*. However, this time, instead of finding the shortest path, we will be finding the number of paths that can connect two points. As such, arcs on the network need to be updated with new labels that show the capacity, or number of paths, that can go through that arc. The game board with capacities is in Figure 13.



Figure 13: Capacities on *Ticket to Ride*

Now that we have a capacitated network, we can choose a source city and a sink city. Since Carroll College is in Helena, we will make Helena our source and arbitrarily choose our sink to be Salt Lake City.

It is clear right off the bat that the shortest path between Helena and Salt Lake City is the pink path of length 3. However, this path only has a capacity of one. So if multiple players need a connection between Helena and Salt Lake City, they must find other paths.

After completing 7 iterations through the Ford-Fulkerson algorithm using the Edmonds-Karp implementation, we can find that it is possible for 7 players to make a connection between Helena and Salt Lake City. However, it is only possible for 7 players to connect Helena and Salt Lake City if each player has only one train leaving Helena and one entering Salt Lake City. If one player has 3 trains leaving Helena, then it will only be possible for 5 players to connect Helena to Salt Lake City.

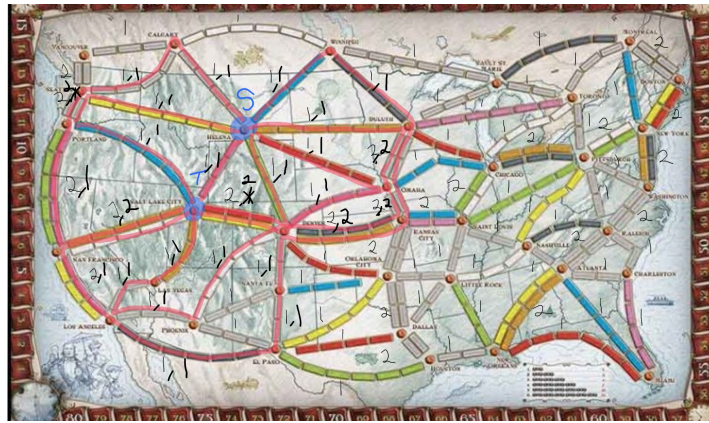


Figure 14: Final flow from Helena to Salt Lake City using Ford-Fulkerson

4.5 Summary

The maximum flow problem finds the maximum amount of flow that can go through a network from a source to a sink. One of the best known algorithms for finding the maximum flow is the Ford-Fulkerson algorithm which takes advantage of augmenting paths. This algorithm does not have a worst case run time due to the arbitrarily chosen paths. Using the Edmonds-Karp implementation, the worst case run time is bounded by $\mathcal{O}(n^2m)$ as a result of augmenting along the shortest paths first.

The next algorithm that we explored was the Push/Relabel or Preflow-Push algorithm. This algorithm utilizes the excess flow at each node to determine if the node is active and the distance from each node i to the source. This algorithm has a worst case run time of $\mathcal{O}(n^2m)$ when a list is used to store active nodes. Each of the implementations for this algorithm revolve around different ways of evaluating the list and have lower maximum run times.

The final algorithm that we explored was the Blocking Flow algorithm. This algorithm works for both capacitated and uncapacitated networks by advancing and retreating through the network. The Blocking Flow algorithm has a maximum run time of $\mathcal{O}(n^2m)$ to find the maximum flow through a network.

5 Minimum Cost

The minimum cost problem aims to determine the cheapest way to transport flow from one place to another. Minimum cost algorithms work to strike the delicate balance between minimizing cost and maximizing flow. Thus, the shortest path problem and the maximum flow problem are considered to be specified versions of the minimum cost problem. Consider a company that has to figure out the cheapest way to move their product from factories to distribution warehouses and then to stores. The company will want to find the cheapest way to send the amount of product necessary.

5.1 Minimum Cost Concepts

Before we can discuss algorithms that solve the minimum cost problem, we will explain a few key concepts.

Pseudo-flow: The pseudo-flow of a network, denoted as x , is the amount of flow that is flowing forwards through the network. This comes in handy when keeping track of flow through a residual network.

Residual Network: The residual network $G(x)$ keeps track of the residual flow through a network rather than the capacities and current flows through an arc.

Supply/Demand: The supply or demand on node i is represented by $b(i)$. When $b(i) > 0$ we say that i is a supply node. When $b(i) < 0$, we say that i is a demand node and when $b(i) = 0$ we say that i is a trans-shipment node.

Imbalance: The imbalance of a node i is represented as

$$e(i) = b(i) + \sum_{j:(j,i) \in A} x_{ji} - \sum_{j:(i,j) \in A} x_{ij},$$

where x_{ij} represents an arc from i to j , for all $i \in N$ where A is the set of arcs leaving or entering i . This is to say the imbalance on i is the supply/demand on i minus the inflow to i plus the outflow from i . As with the supply/demand labels, the imbalance on i can be classified into three categories: when $e(i) > 0$ then i is in excess, when $e(i) < 0$ then i is in deficit, and when $e(i) = 0$ then i is balanced.

Node Potentials: The node potential $\pi(i)$ of a node $i \in N$ is the negative distance of the shortest path from the source node s to node i with respect to the cost to traverse that path. That is $\pi(i) = -d(i)$ with respect to c_{ij} . The shortest path between these two nodes can be found using any of the algorithms explored in the shortest path section or any other shortest path algorithm [3].

Reduced Costs: The reduced cost of an arc is found using node potentials. The reduced cost is the actual cost minus the node potential at the start node for the arc plus the node potential for the node at the end of the arc. That is to say, $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$. Additionally, since $\pi(i) = -d(i)$, then we know that $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$.

5.2 Successive Shortest Path Algorithm

The successive shortest path algorithm finds the shortest path from a source to a sink, with respect to the node potentials, then augments the flow along that path. This algorithm uses pseudo-flows and the residual network to find the total flow on a network with possibly more than one source and sink [10].

This algorithm begins by creating the sets *Excess* and *Deficit*, where *Excess* is the set of nodes in excess and *Deficit* is the set of nodes in deficit. Then, we initialize the node potentials at each node to zero, $\pi(i) = 0$. Finally, we initialize the pseudo-flow x to zero since there is no flow yet through the residual network.

Once the variables are set up, we can get into the meat of the algorithm. On each iteration through the algorithm, we find the shortest path from a source node s in *Excess* to every other node $i \in N$ with respect to the reduced costs of the arcs. These distances are then used to update the node potentials, $\pi(i)$, for every node in the network and to find the shortest path P from s to a sink node t chosen from *Deficit*.

It may seem odd to be finding the shortest path with respect to the reduced cost as compared to the actual cost of using an arc. However, using reduced costs does not affect the optimal solution [10].

Then we augment along the path P with the minimum of the imbalance on s , the negative imbalance on t and the minimum residual capacity of arcs in P . In mathematical notation we are augmenting by

$$\delta = \min \{e(s), -e(t), \min \{r_{ij} : \text{arc}(i, j) \in P\}\}.$$

Now, we increase the pseudo-flow x by δ , update the residual network with the augmenting flow, and the reduced costs c_{ij}^{π} . Since the residual network has been updated, we check to see if the sets of excess and deficit nodes are still accurate. That is, we continue iterating through the algorithm until there are no more nodes with excess flow. That is continue until the set of excess nodes, *Excess*, is empty.

This algorithm does not have a worst case run time. However, its worst case number of iterations is $\mathcal{O}(nC)$ where n is the number of nodes and C is the length of the longest arc. Additionally, on each iteration the shortest path problem is solved for a specific source node. The time it takes to solve the shortest path problem is bounded by $S(n, m, nC)$, which represents the worst case time to find the shortest paths using an unspecified algorithm. Thus, the overall complexity of this algorithm is $\mathcal{O}(nC \times S(n, m, nC))$ [10].

5.2.1 Implementations

Unlike the other algorithms that solve the shortest path and maximum flow problems, clever data structures do not directly help to reduce the worst case run time of this algorithm. However, when Dijkstra's algorithm is used to determine the shortest path distances, we can stop the algorithm when the sink node has a permanent distance label. At this point we can then update the node potentials to

$$\pi(i) = \begin{cases} \pi(i) - d(i) + d(t) & \text{if } i \text{ is permanently labeled} \\ \pi(i) & \text{if } i \text{ is temporarily labeled} \end{cases}$$

where $d(t)$ is the distance to the selected sink node t from the selected source s and $d(i)$ is the distance from the source to node i . Using this schematic, temporarily labeled nodes do not have

to be updated. By not updating the temporary distance nodes, we reduce the number of nodes that have to be updated. Thus, we reduce the amount of time spent on the shortest path step. By reducing the time spent on this step, we reduce the time it takes to do each iteration and thus reduce the worst case time on the algorithm.

5.2.2 Example

To better understand this algorithm, it is very helpful to work through an example. For our example we will use the network on the top of Figure 15.

Before we begin, we must rewrite the network as the residual network by changing the notation on the arcs from “ $\$c_{ij} \text{ (} cap_{ij}, f_{ij}\text{)}$ ” to “ (c_{ij}^π, r_{ij}) ” where c_{ij} is the cost of using arc (i, j) , cap_{ij} is the capacity of the arc (i, j) , f_{ij} is the flow through arc (i, j) , c_{ij}^π is the reduced cost of arc (i, j) , and r_{ij} is the residual flow of arc (i, j) . The original network is the top graph of Figure 15 and the residual network is the lower graph.

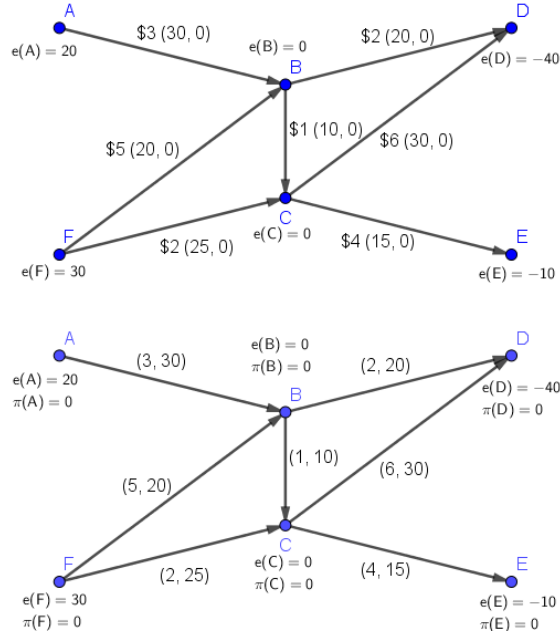


Figure 15: Shortest Successive Path Example - Starting and Residual Networks

Now that we have the residual network, we can define the set of excess nodes, $Excess = \{A, F\}$ and the set of deficient nodes, $Deficit = \{D, E\}$. On the first iteration through the residual network, let the source be the node $A \in Excess$ and let the sink be the node $D \in Deficit$. Then, by finding the shortest path to each node with respect to the reduced cost, the node potentials $\pi(i)$ are updated to $-d(i)$ for each node in the network. Thus, we find that the shortest path from source A to sink D is $A - B - D$ with a distance of 5 as seen in the top graph of Figure 16.

Next, we augment the flow by

$$\begin{aligned} \delta &= \min \{e(s), -e(t), \min \{r_{ij} : \text{arc}(i, j) \in P\}\} \\ &= \min \{20, -(-30), \min \{r_{AB} = 30, r_{BD} = 20\}\} = 20, \end{aligned}$$

so the flow along the path $A - B - D$ is augmented by 20 and the network is updated with new node potentials, reduced costs, and imbalances. Notice that when we send 20 units of flow from

A to D the imbalance on node A decreases to 0 and the imbalance on node D increases to -20 . Since the imbalance on A is reduced to 0, then A is removed from the set of excess nodes. We also update the pseudo-flow x by δ , such that $x = 20$. The updated residual network is seen in the bottom graph of Figure 16.

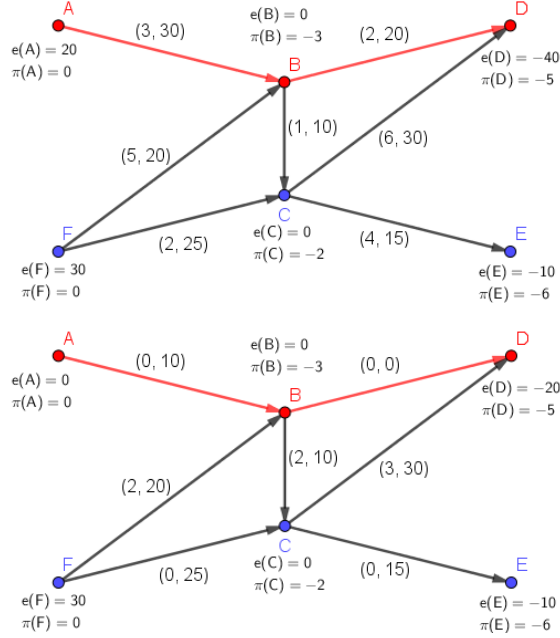


Figure 16: Shortest Successive Path Example - Iteration 1

In the second iteration through the successive shortest path algorithm, the set of excess nodes is $Excess = \{F\}$ since node A has no more excess. The set of deficient nodes remains $Deficit = \{D, E\}$ since nodes D and E have deficits of -20 and -10 respectively. Then we can choose $F \in Excess$ as the source and the node $D \in Deficit$ as the sink. Once again we find the shortest path to each other node from F with respect to the reduced costs. Thus, we find that the shortest path from F to D is $F - C - D$ with a reduced cost distance of 3. This path is shown in red in the left network of Figure 17.

Next we augment along this path with a flow of 20 from

$$\delta = \min \{30, -(-20), \min\{r_{FC} = 25, r_{CD} = 30\}\}$$

and update the residual flow on each arc along the path by reducing the residual flow by 20. We also update the pseudo-flow by another 20, so the pseudo flow x is 40. Then we update the imbalances to reflect the updated residual flow. Finally, we update the reduced costs using the new node potentials. The updated network can be seen on the lower half of Figure 17.

t

Now, on the third iteration through the algorithm, the set of excess nodes is $Excess = \{F\}$ and the deficit nodes are $Deficit = \{E\}$. So the augmenting path will have the source $F \in Excess$ and the sink $E \in Deficit$. Then we find the shortest paths from F to every other node in the network. Thus, we find that the shortest path from F to E is $F - C - E$ with a distance of 6. This shortest augmenting path can be seen in the top graph of Figure 18 in red.

Then we find the amount of flow that can be augmented along this path using the minimum

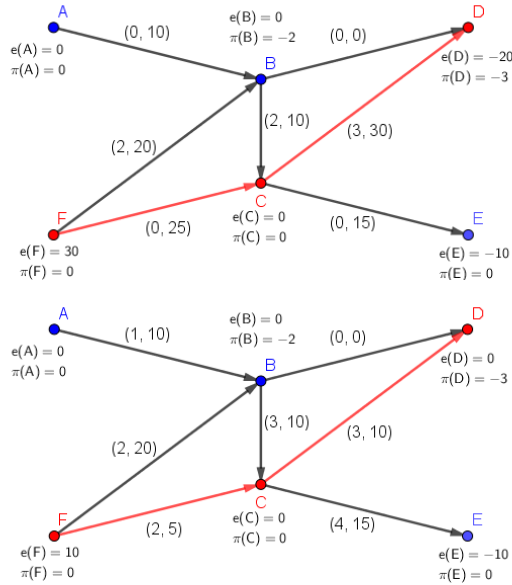


Figure 17: Shortest Successive Path Example - Iteration 2

of the node potentials and the residual capacities. Thus,

$$\delta = \min \{10, -(-10), \min\{r_{FC} = 5, r_{CE} = 15\}\} = 5$$

and we augment along the path $F - C - E$ with a flow of 5. Once the augmentation is complete, we update the node imbalances on F and E to be $e(F) = 5$ and $e(E) = -5$. We also increase the pseudo-flow by another 5 so that $x = 45$. Finally, we complete this iteration by updating the residual network. The updated residual network is in the bottom graph of Figure 18.

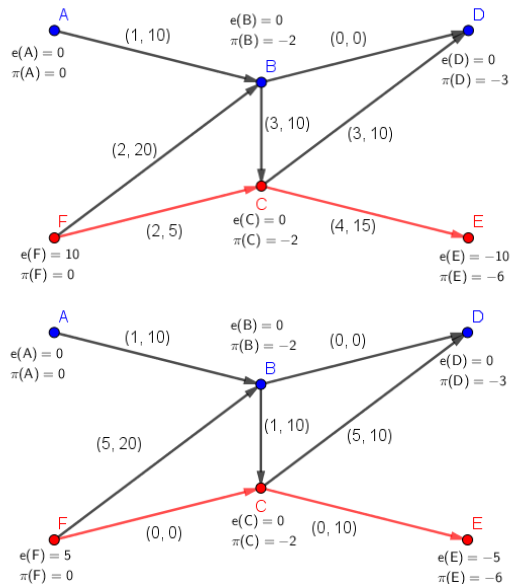


Figure 18: Shortest Successive Path Example - Iteration 3

On our fourth and final iteration, the sets of excess and deficit nodes remain the same. As such the source is once again F and the sink is E . We find the shortest path from F to every other node and update the node potentials, which leads to finding the shortest path from F to

E . The shortest path is now $F - B - C - E$ and can be seen in red on the upper network of Figure 19. Notice that the arc (F, C) is no longer used because its residual capacity has been reduced to 0. As a result, the shortest path chosen was chosen in part because it was the only augmenting path remaining between nodes F and E .

Now, we find that the flow through this path is 5 as constrained by the imbalances on F and E . Then the residual flows and node imbalances along the shortest path $F - B - C - E$ are updated to reflect the augmented flow. The pseudo-flow is also increased on last time to $x = 50$. The updated residual network, lower network of Figure 19, is then translated back to the original network to show the final flows through the network as shown in Figure 20. Since the algorithm has been completed the pseudo-flow is translated into an actual flow of 50 units. Using the final flows through the network, we are able to find that the lowest cost for sending 50 units through this network is \$340 using the flows indicated in Figure 20.

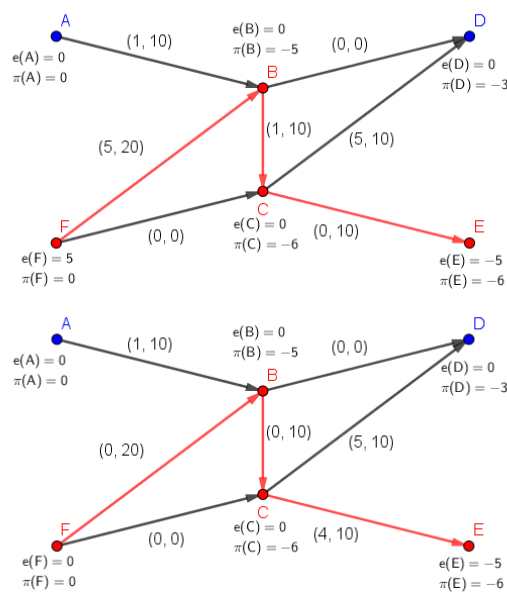


Figure 19: Shortest Successive Path Example - Iteration 4

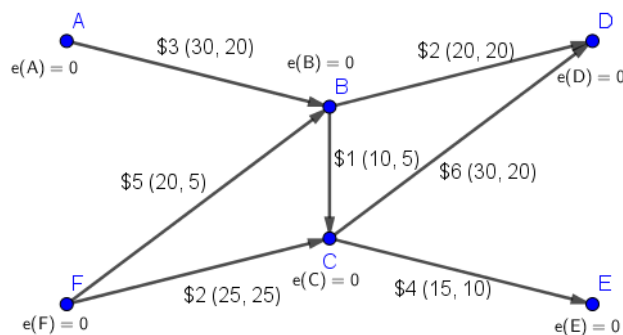


Figure 20: Shortest Successive Path Example - Final Network

5.3 Application to *Ticket to Ride*

As with the shortest path problem and the maximum flow problem, the minimum cost problem can be applied to *Ticket to Ride*. There are a few different ways that we could frame *Ticket to Ride* as a minimum cost problem. The first way to apply minimum cost to this game is by treating the number of cards it takes to build a route as the cost and the flow through an arc as the number of players that can build on that arc. This way of framing focuses on what is required to build the path and assumes that other players will kindly leave the path that you need alone. From personal experience, this never happens.

Another, less naive approach to using minimum cost is to make the cost associated with each arc the number of turns required to gather the required cards and lay the path. The flow through the arc will still represent the number of players that can place routes using that arc. This approach can be complicated by taking into account the probability that the player will be able to collect the correct cards.

6 Conclusion

Throughout this exploration of network flows, we have looked at three types of problems: the shortest path problem, the maximum flow problem, and the minimum cost problem.

Each of these problems relates to a specific optimization goal. Both shortest path and maximum flow algorithms focus on optimizing only one variable, distance or flow, whereas minimum cost algorithms seek to optimize one variable with constraints on another variable. Each of the algorithms explored is used for solving large scale optimization problems that would be illogical to do by hand.

The shortest path problem leads to algorithms to minimize the distance between two places. Distance could be physical distance, time, cost, or another factor to minimize. These algorithms are used by navigation services, such as Google Maps, to find the shortest path from one place to another with respect to both time and distance. When planning a road trip, even a short one to the grocery store, we often use shortest path algorithms to find the fastest way to get to our desired location.

In a similar fashion, the maximum flow problem tries to maximize the amount of flow through the network. Flow could be many things from water through pipes, cars along a road, or even text messages pinging off cell phone towers. Maximum flow algorithms are able to find the maximum amount of flow through a network. These networks can be small, such as a local network of water pipes, or huge, like a country wide network of roads.

Finally, we combined the shortest path and maximum flow problems together for the minimum cost problem. The goal of the minimum cost problem is to minimize the cost of sending a set amount of flow. Cost could be a variety of factors such as money, time, or distance. Minimizing the cost requires some version of a shortest path algorithm to minimize the cost, but still has to fulfill the flow requirements. To ensure that the required amount of flow travels through the network, a maximum flow algorithm is used. By tweaking the shortest path and maximum flow algorithms we build minimum cost algorithms. Minimum cost algorithms are used for solving a variety of real world problems, such as sending a product from factories to storefronts.

Network flows help to solve complex real world problems that would be very difficult, if not impossible, to do by hand. They are easily scalable and are used to optimize networks of all sizes. In a world that is progressing quickly with the growth of the internet and technology, network flows provide a way to keep up with ever growing networks.

References

- [1] Dijkstra's algorithm — https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=827353023, February 2018. Page Version ID: 827353023.
- [2] PushRelabel Maximum Flow Algorithm—https://en.wikipedia.org/w/index.php?title=Push%E2%80%93relabel_maximum_flow_algorithm&oldid=826817636, February 2018. Page Version ID: 826817636.
- [3] Stephen Billups. Lecture 20: Min-Cost Flow Problems, 2010.
- [4] Frederick S. Hillier, Gerald J. Lieberman, Bodhibrata Nag, and Preetam Basu. *Introduction to Operations Research*. McGraw-Hill, ninth edition, 2012.
- [5] Jonathan Gross and Jay Yellen. *Graph Theory and Its Applications*. CRC Press, 1999.
- [6] David Karger. Lecture 11: Blocking Flows. October 2006.
- [7] Bobby Kleinberg. Edmonds-Karp Max-Flow Algorithm—<https://www.cs.cornell.edu/courses/cs4820/2012sp/handouts/edmondskarp.pdf>, 2010.
- [8] Kristina Lundqvist. Big O notation—http://web.mit.edu/16.070/www/lecture/big_o.pdf, 2003.
- [9] Alan R. Moon. Ticket to Ride—<https://boardgamegeek.com/boardgame/9209/ticket-ride>, 2004.
- [10] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [11] University of California, Davis. Continuation of the Preflow-Push Algorithm—<https://www.youtube.com/watch?v=w8MKEo8mJDE>, 2013.

Appendix

Important Vocabulary

Node (or Vertex): A node or vertex is a point on a graph. Nodes are typically notated by either letters or numbers.

Edge: Arc on a graph that represents the relationship between two nodes. In general it is possible for an edge to begin and end at the same point, however that case will not arise in this paper. Edges are notated in a few ways including ij , e_{ij} , and $\text{arc}(i, j)$ where i and j are nodes in a network N [5].

Arc: Directed edge typically notated as $\text{arc}(i, j)$ where i, j are nodes in a network N . The arc begins at node i and ends at node j . The set of arcs in the network N is often referred to as A .

Flow in a network: The flow through a network. Common examples are a liquid traveling through pipes or energy, phone calls, commodities or other types of flow traveling through a network [5].

Capacity: Maximum possible flow through an arc. Typical notation is c_{ij} or cap_{ij} where i and j are nodes in the network N [5].

Residual Capacity: Remaining possible flow through an arc. Notated as r_{ij} where $r_{ij} = c_{ij} - f_{ij}$ on the $\text{arc}(i, j)$

Source: The starting node of a flow network. Typically notated with a lower case s .

Sink: The ending node of a flow network. Typically notated by a lower case t .

Flow Network: A connected graph with a distinguished source node and a distinguished sink node [5].

Augmenting Path: A path through the network from the source to the sink that is not at capacity.

Ticket to Ride

The board game *Ticket to Ride*, developed by Alan R. Moon, has become an international sensation. This game is relatively simple to play but requires strategy to win. The goal of the game is to get as many points as possible by placing routes and completing route tickets. On each turn, the player has the option to place a route, draw train cards (needed to complete routes) or draw more route tickets. As such during game play there is a balance between “greed - adding more cards to your hand, and fear - losing a critical route to a competitor.” according Mr. Moon [9].

Algorithm Pseudo-Code

This section contains some Java-esque pseudo-code for four of the algorithms described above. Throughout the pseudo-code there are a few recurring ideas. Every algorithm has a node labeled s that denotes the source node. Similarly, every algorithm except Dijkstra’s algorithm also has a sink node labeled with a t . All other lower case letters, such as i and j , denote generic nodes in the network. Other common notation includes:

- $d(i)$, the distance from the source s to node i
- f_{ij} , the flow from node i to node j along the arc (i, j)
- c_{ij} , the capacity of arc (i, j)
- r_{ij} , the residual capacity of arc (i, j) computed as $r_{ij} = c_{ij} - f_{ij}$

Algorithm specific notation is introduced directly before the algorithm.

Dijkstra's Algorithm

Dijkstra's Algorithm is one of the many algorithms that solves the shortest path problem. By using temporary and permanent distance labels, Dijkstra's algorithm iterates through the network finding the shortest path from the source node to every other node.

Notation

l_{ij}	Length of arc (i, j)
T	Set of temporarily labeled nodes
P	Set of permanently labeled nodes
$pred(j)$	The node that the lowest distance on j came from

Algorithm

```

Dijkstra's Algorithm {
  // Set up variables
  d(i) = ∞, d(s) = 0
  T = {n ∈ N} \ {s}
  P = {s}

  // Algorithm
  while T is not empty {
    choose i where d(i) = min{d(j) : j ∈ T}
    add i to P, remove i from T
    for each arc(i, j) ∈ A(i) {
      if d(j) > d(i) + lij {
        d(j) = d(i) + lij
        pred(j) = i
      }
    }
  }
}

```

Ford-Fulkerson

The Ford-Fulkerson algorithm solves the maximum flow problem by using augmenting paths. Each augmenting path stretches from the source s to the sink t . Once an augmenting path is found then the flow along that path is increased so that at least one arc is at capacity.

Algorithm

```
Ford-Fulkerson Algorithm {
  // Set up Variables
   $r_{ij} = c_{ij}$ 

  //Algorithm
  While there is an augmenting path  $P$  from  $s$  to  $t$  {
     $\delta = \min \{r_{ij} : \text{arc}(i, j) \in P\}$ 
    If  $\text{arc}(i, j)$  is a forward arc{
       $f_{ij} = f_{ij} + \delta \forall \text{arc}(i, j) \in P$ 
    }
    If  $\text{arc}(i, j)$  is a backward arc {
       $f_{ij} = f_{ij} - \delta \forall \text{arc}(i, j) \in P$ 
    }

     $r_{ij} = c_{ij} - f_{ij} \forall \text{arc}(i, j) \in P$ 
  }
}
```

Push/Relabel

The Push/Relabel or Preflow-Push algorithm solves the maximum flow problem by utilizing the idea of active nodes and excess flow. By “flooding” the network with flow then scaling back the flow so that every interior node has no excess flow.

Notation

$d(i)$		Distance from source s to node i
A		Set of arcs in the network
N		Set of nodes in the network
$e(i)$		Excess on node i

Algorithm

```
Push/Relabel Algorithm {
  // Set up Variables
   $d(s) = n, d(i) \forall i \in N \setminus \{s\}$ 
   $e(i) = f_{si} \forall \text{arc}(s, i) \in A$ 

  // Algorithm
  while there is an active node  $i \in N \setminus \{s, t\}$  {
    If there exists node  $j$  such that  $r_{ij} > 0$  and  $d(i) \geq d(j) + 1$ {
      Push  $\delta = \min\{e(i), r_{ij}\}$ 
    } else {
      Relabel  $d(i) = \min\{d(j) + 1 : \text{arc}(i, j) \in A \text{ where } r_{ij} > 0\}$ 
    }
  }
}
```

Successive Shortest Paths

The successive shortest paths algorithm is one of the solutions to the minimum cost problem. It utilizes reduced costs, node potentials, and node imbalances to ensure that the required amount of flow is sent through the network. Additionally, this algorithm uses distance with respect to reduced costs to find the cheapest path on each iteration.

Notation

x	Pseudo-flow through the network
$d(i)$	Distance from source s to node i with respect to the reduced costs c_{ij}^π
$e(i)$	Imbalance on node i . Computed as $e(i) = b(i) + \sum_{j:(j,i) \in A} x_{ji} - \sum_{j:(i,j) \in A} x_{ij}$
$\pi(i)$	Node potential on node i . Computed using $\pi(i) = -d(i)$
c_{ij}^π	Reduced cost on arc (i, j) . Computed using $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$

Algorithm

Successive Shortest Path Algorithm {

// Set up Variables

$x = 0$

$\pi(i) = 0, e(i) = b(i) \forall i \in N$

$Excess = \{i : e(i) > 0\}, Deficit = \{i : e(i) < 0\}$

// Algorithm

while $Excess$ is not empty {

 choose $s \in E$ and $t \in D$

 find $d(j)$ from s to j with respect to c_{ij}^π

 let P be the shortest path from s to j

$\delta = \min\{e(s), -e(t), \min\{r_{ij} : (i, j) \in P\}\}$

$r_{ij} = r_{ij} - \delta$ for all arc $(i, j) \in P$

$\pi(i) = -d(i), x = x + \delta$

 Update $Excess, Deficit, c_{ij}^\pi$

}

}