AA Final Project

Audrey, Jack, Shirin

May 2021

1 Introduction

In this project, we are working on creating a fair carpool sharing system via the use of network flows. The goal is to use the Edmonds-Karp algorithm in order to maximize the flow while making sure all people are driving a fair amount. In this documentation, we will define fairness, setup a general network flow, go through a by hand example that explains the Edmonds-Karp algorithm with the network flow diagram, explain our code, and any potential next steps.

For our first step, we are defining the driving fairness as

$$r_i = \sum_{x}^{d} \frac{1}{k_x} \tag{1}$$

where r_i is the share of responsibility of driving in the carpool for each person. This equation will sum k_x which is k number of people who share the responsibility on day x up to the last day of the carpool d.

Figure 1 is an outline of the network flow for the fair carpool sharing problem.

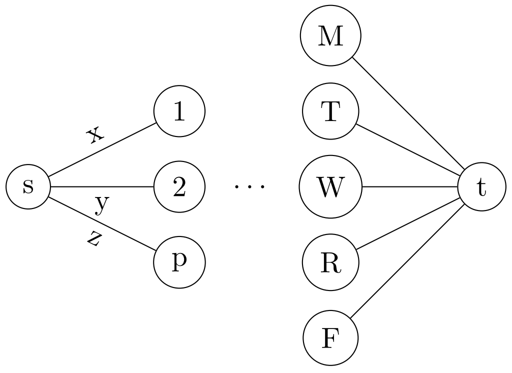
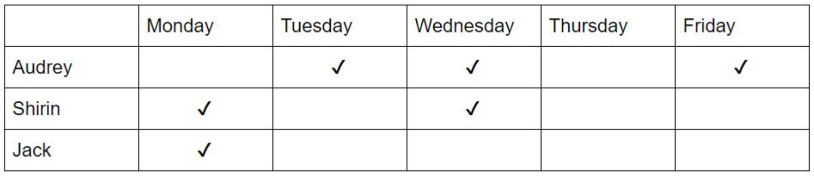


Figure 1: Network Flow for Fair Carpool Sharing Problem

In figure 1, the people are represented in nodes 1,2 through p where p is the number of people in the schedule. The edge weights x, y, and z represent the r_i, or the fairness driving responsibility that person has. The ellipses between the people and the days of the week depend on the schedule since the edges represent which person goes in the carpool on which day.

2 By Hand Example

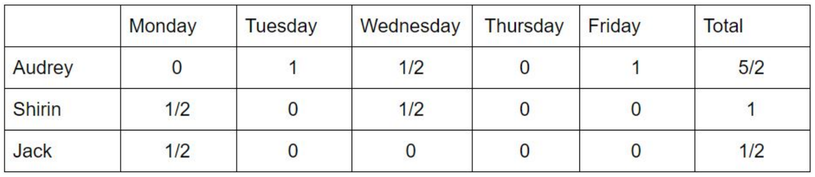
Figure 2 displays a carpool schedule where a check mark means that person needs a ride on each given day of the week. For the remainder of this problem, we will use this schedule to determine who will drive on what day. For the purpose of this problem, we'll ignore the fact that none of us really drive.



	Monday	Tuesday	Wednesday	Thursday	Friday
Audrey		✓	✓		✓
Shirin	✓		✓		
Jack	✓				

Figure 2: Driving Schedule Example

From here, we can represent this as a network flow graph using the graph and fairness function introduced in Section 1. The table in Figure 3 displays the fairness value for each individual for each day, and then sums them to get the total fairness value for each individual for the week. If we sum the fairness value for Audrey, Jack, and Shirin, we also note that it sums to 4 - which is equivalent to the number of days of the week people need rides as expected.



	Monday	Tuesday	Wednesday	Thursday	Friday	Total
Audrey	0	1	1/2	0	1	5/2
Shirin	1/2	0	1/2	0	0	1
Jack	1/2	0	0	0	0	1/2

Figure 3: Fairness values for the given driving schedule

Now, we can represent it using the network flow template displayed in Figure 1 as shown in Figure 4. The fairness is the value from the source to each of the people as that is the capacity a given person can drive and still have the carpool sharing be fair. Although we may normally see integer values as capacities, we can also have fractional capacities and flows. All the unlabeled edges have a capacity of one. This is because only one person can drive on a given day.

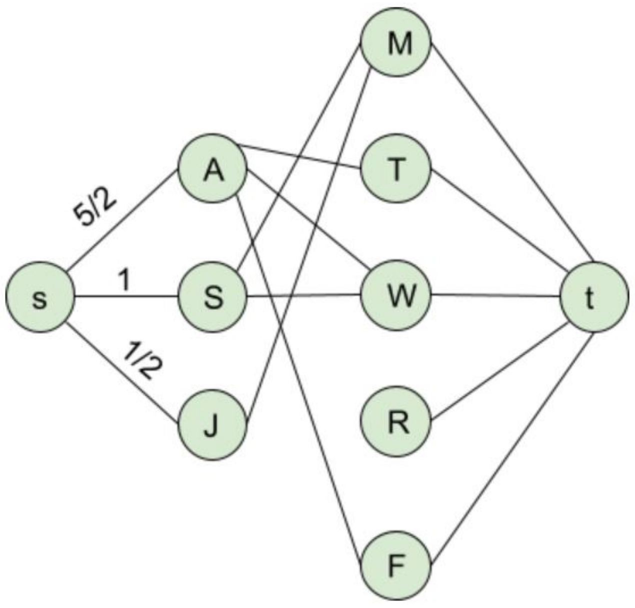


Figure 4: Network flow graph for given driving schedule

Now given the graph in Figure 4, we can carry out the Edmonds-Karp algorithm. Since this takes a while to do by hand, we will only walk through a one iteration to explain the overall concept:

1. Create the residual graph as shown in Figure 5. The numbers across the edges represent the amount of available flow remaining. The unlabeled edges have value 1.

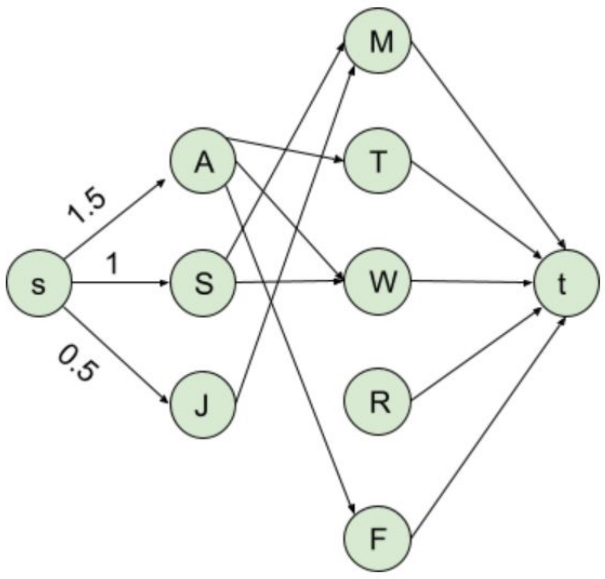


Figure 5: Residual graph for given driving schedule

2. Identify the shortest augmented path using BFS.

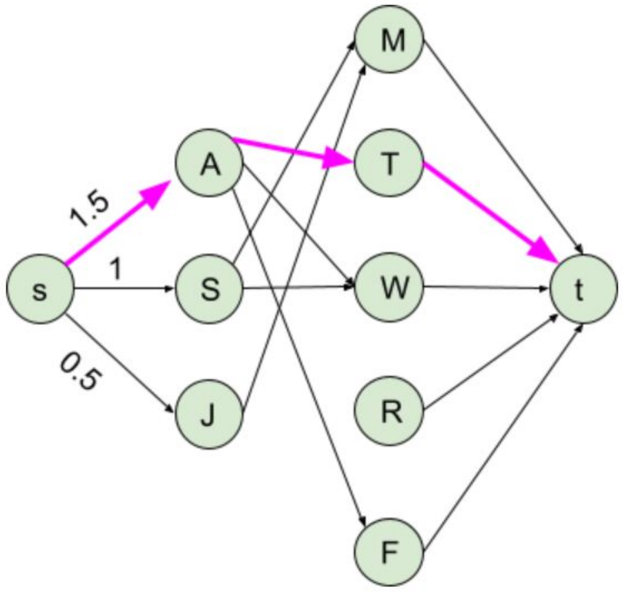


Figure 6: Residual graph with chosen augmented path in pink

3. Find the constraining flow across the shortest augmented path.

   In graph in Figure 6, since the capacity across A-T and T-t is 1, which is less than the capacity across s-A of 1.5, our constraining flow is 1.

4. Add this flow to the original graph and subtract it from the residual graph (and change the direction of the arrow if capacity remaining is zero). Figure 7 displays the updated original network diagram for this iteration (flow in numerator in black, capacity in denominator in red), and Figure 8 displays the new residual diagram for this iteration.

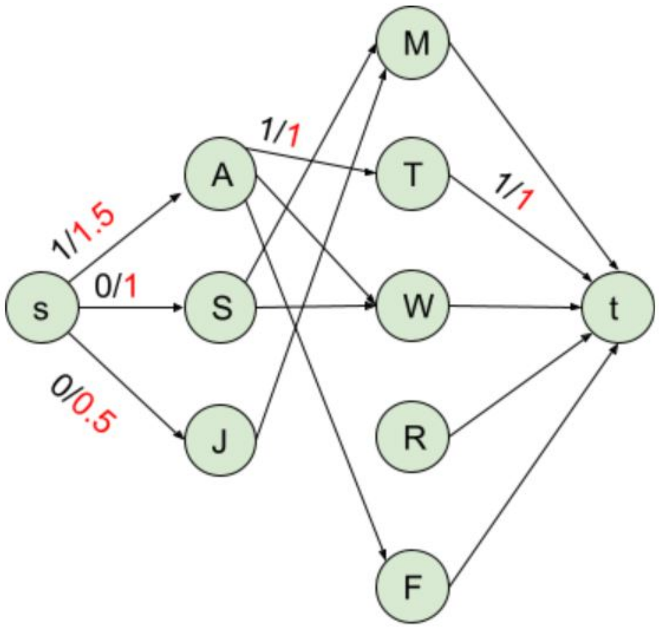


Figure 7: Updated original network

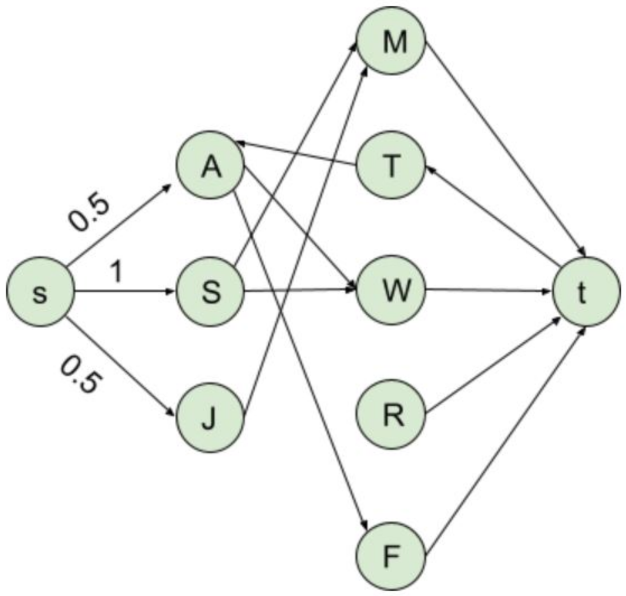


Figure 8: Updated residual graph

5. Now using this revised version of the residual, we continue this process until there are no paths remaining

3 Code Overview

In this section, we will go over how we solved the carpool problem with the Edmonds-Karp algorithm.

First, we define our graphs and made a schedule generator in generate_schedule.py. The way we represent our network flow is through a 2D adjacency matrix. The adjacency matrix is a square matrix, so the column and row indices are numbered the same way. The adjacency matrix always stores the source index at 0 and the sink at the last index of the list. Index 1 to p, where p is the number of people in the carpool, represent the number of people. Index p+1 to p+d represent the days that the people are using the carpool. The values are all initialized as 0, and if there is a connection between two nodes on the network flow diagram, then we edit adjacency matrix at their respective indices of the nodes accordingly.

The structure of overview of our Edmonds-Karp algorithm is as follows:

```
def edmondkarp(graph, node):
    while there is a path from source to sink:
        retrace the path back from sink to source
        find the minimum residual of the path
        add the min residual to the max flow
        update the graph with the residual
        continue till no more paths are left
    return max flow
```

There are several helper function that we used in carpool.py file. The first one is called *adjacent*, which takes in an adjacency matrix and node returns a list of nodes that are adjacent each other. This is used in our *bfs*, the breadth first search function. Our *bfs* function works as follows:

```
def bfs(source,sink,parent,graph):
    visited = a list of false with length = # of nodes
    create a queue with our source
    while queue is not empty:
        u = queue.pop(0)
        for index in adjacent(graph, u):
            parent[index] = u
            if index is sink, return True and parent
            else append index in queue and visited[index] = True
    return False, parent
```

Note that parent is the way we track our path back from the sink to source. When we pass parent in, it initially should be a list with the length equal to the number of nodes and initialized with the value of -1 to show that the node does not have a parent at the moment.

The functions *print_g* and *print_schedule* are used to make our outputs look more readable. *print_g* prints the adjacency matrix. *print_schedule* prints a 2D list with columns representing days and rows representing people. The value at each row-column intersection is the probability (or responsibility percent) of the person to drive on the given day.

4 Source Code

All of our work can be found on: https://github.com/JackMao981/Carpool

5 Conclusions and Next Steps

Overall, we had a good experience and were able to overcome some challenges in this project. Our algorithm is able to successfully generate a driving responsibility schedule that allows us to determine who should drive on which day by looking at the person with the highest percentage on that day. When we compare the original carpool schedule to the results, we notice that the number of days a person is using the carpool is usually proportional to how much a person would drive.

In the future, we would take into consideration what happens when there is a tie. Currently, we say that they should just flip a coin, but instead, we could assign the person who is driving for fewer days than the other person or the person who is using the carpool for more days than the other. We could also take into account fractional driving distances where if they are travelling long distances, the people in the carpool could take turns for that same carpool day. We could also look into rotation driving where we would look at the schedule for multiple weeks and determine who would drive based off of more days. We could also set up different ways to define fairness. We could determine it by having everyone just drive the same amount regardless or we could incorporate other factors such as difficulty driving (time of driving such as rush hour) or distance of the trip.

6 Sources

1. Network Flows Algorithms Textbook - Chapter 2.2
2. https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
3. https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm