A Closer Look at the Ford-Fulkerson Network Flow Algorithm:

The Case of Dinner Scheduling

Finnian N.G. Wengerd

Eastern Mennonite University

I. Abstract

Using the Ford-Fulkerson Algorithm, the Dinner Scheduling problem was solved to find the optimum schedule for residents of an apartment to prepare dinner given the dates each of them are unable to cook. The algorithm was then tested to find that the time complexity was linear as expected given the time complexity of the Ford-Fulkerson Algorithm has a time complexity of $O(Ct)$.

II. Background

Network Flow is a concept that is used in many aspects of business today. From setting up actual digital telecommunications network to analogue circuits to divvying up tasks to workers for the most optimal scheduling with the capabilities at hand. A network is essentially a series of connections that overlap at certain nodes. These connections (or arcs) are given a capacity that states the amount of flow that can go through that connection[1]. When these networks can be represented by bipartite graph, say matching up students with internships, it can be beneficial to employ the Ford-Fulkerson Maximum Flow Labeling Algorithm.

In problem fifteen at the end of chapter seven of Kleinberg and Tardos's *Algorithm Design* it challenges the reader to find a path through a network visiting each node once. The problem is set up focusing on n number of individuals living in a house where they take turns cooking. However, with busy schedules, they do not have an easy round about way to divvy up the work. Given a list of people and nights:

$$p = \{p_1, p_2, p_3, ... p_n\} \text{ and } d = \{d_1, d_2, d_3, ... d_n\}$$

and for each person, there is a set of nights $S_i \subset \{d_1, d_2, d_3, ... d_n\}$ where the person is *unable* to cook, find a *feasible dinner schedule* if one exists such that each person cooks once, each day is covered by one person, and no person p_i is cooking on a night in their set S_i [2].

III. Methods

The Ford-Fulkerson Algorithm, introduced in the mid-1950's splits a network into a bipartite graph using some two distinguishable assets of the nodes. The theorem states that "A flow f has maximum value if, and only if, there is no flow augmenting path with respect to f"[3]. In other words, the algorithm will end if the graph no longer has an augmented path. This will guarantee the graph with respect to f contains a max flow.

In this case, given a number n of both people and nights, and given a set of the available nights for each person, a bipartite graph can be made. For illustrative purposes, an example will be run through the Ford-Fulkerson Algorithm.

$$p = \{p_0, p_1, p_2, p_3\} \text{ and } d = \{d_0, d_1, d_2, d_3\}$$

$S_0 \subset \{d_1, d_2, d_3\}$

$S_1 \subset \{d_0, d_2, d_3\}$

$S_2 \subset \{\}$

$S_3 \subset \{d_2\}$

Given these sets, a network can be set up by finding the inverse of each set to find the days that they are available.

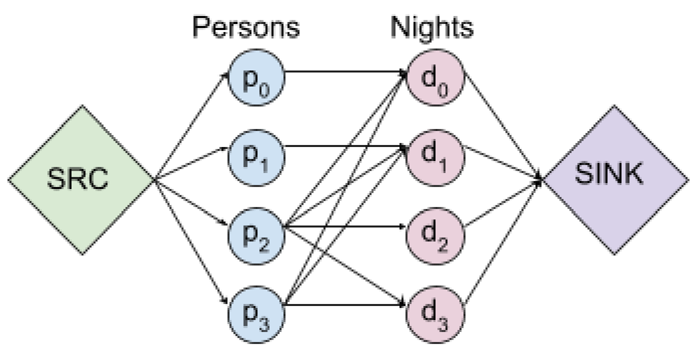
$S_0 \subset \{d_0\}$

$S_1 \subset \{d_1\}$

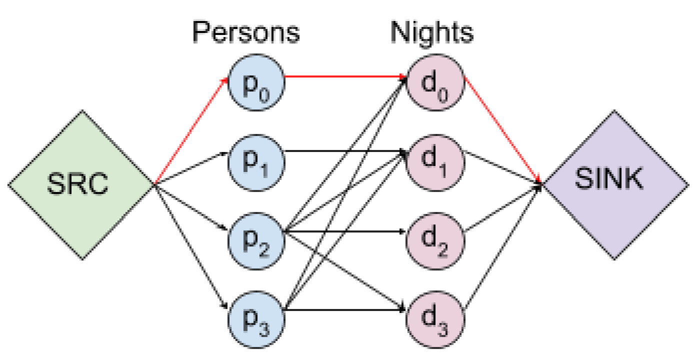
$S_2 \subset \{d_0, d_1, d_2, d_3\}$

$S_3 \subset \{d_0, d_1, d_3\}$

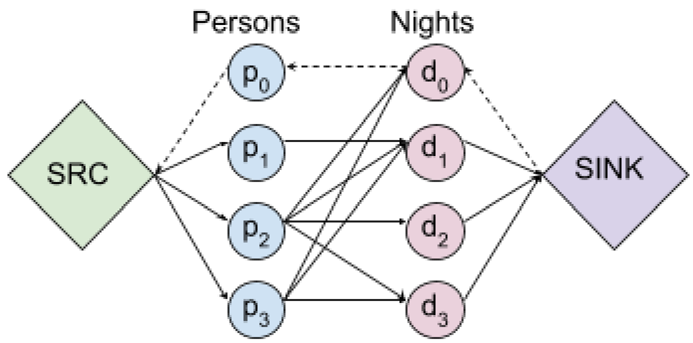
A physical network can be drawn out attaching a source and sink as distinguished nodes. Because there is no foreseen capacities, they shall remain at one.



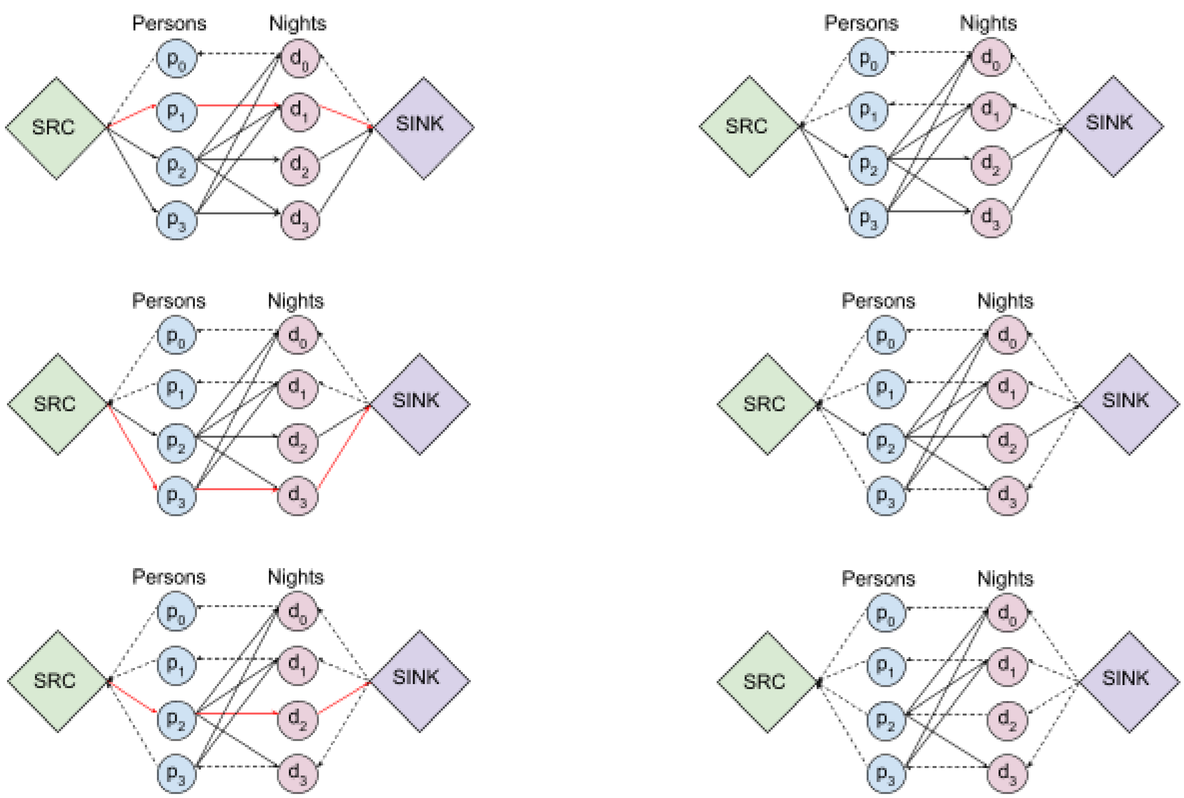
To begin, a random path is chosen



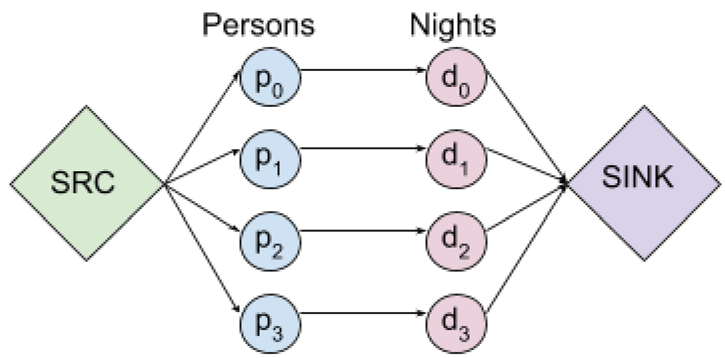
Now a residual graph is created.



The flow is calculated (1) and the process is repeated until there are no more augmented paths.

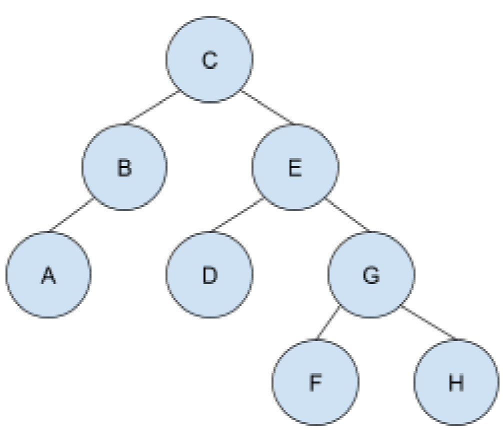


In the last residual graph, there are no connections going into the sink. This is proof that there are no more augmented paths available. Now, the final graph is created.



IV. Complexity

Before running any algorithm, it is pertinent to know if it is truly worth delving into one by knowing its time complexity, Big-O. When running through the Ford-Fulkerson Algorithm, a pointer looks at each of the nodes. So if a graph has a range of R, the time complexity would be $O(R * our\ searching\ algorithm)$. The code in the appendix runs a Breadth First Search.



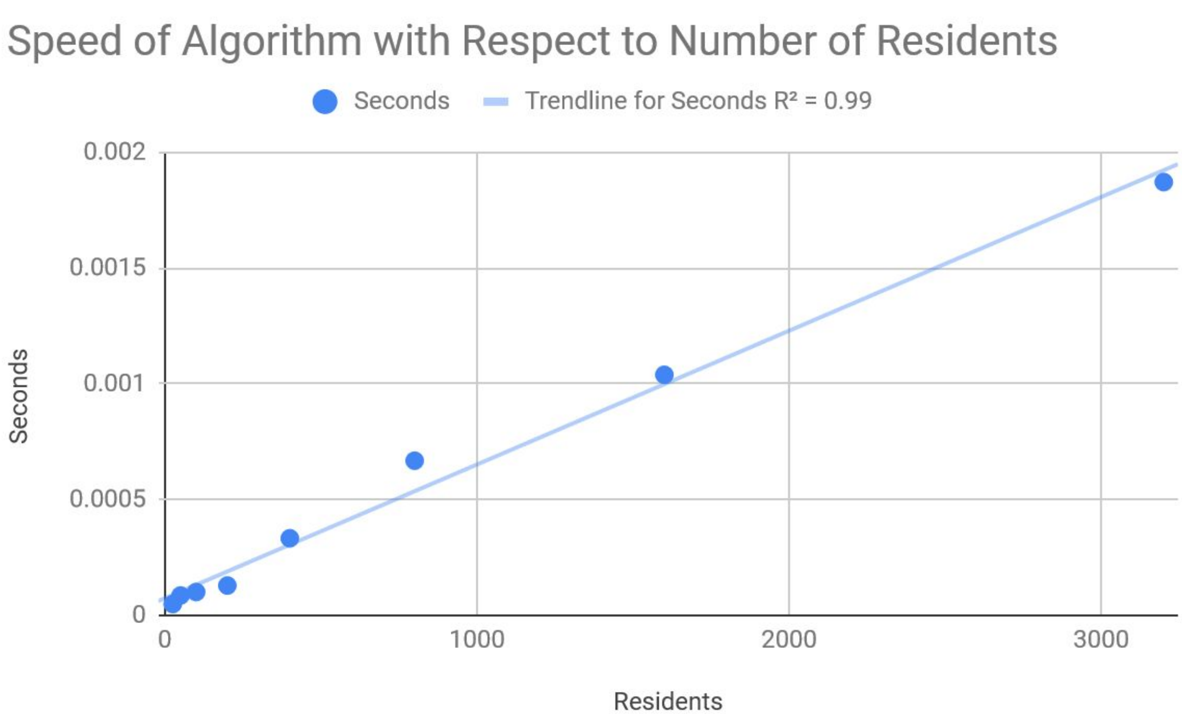
Given the tree to the left, a Breadth First Search would look at the first node at the root, C, and look to what is adjacent to it, B and E. C would then be added to the visited list and the pointer would move to B. If there are new nodes that are not found in the visited list, it is added to the list of items in that row.

Floor	Nodes	Discovered
F_0	[C]	[C]
F_1	[B,E]	[C,B,E]
F_2	[A,D,G]	[C,B,E,A,D,G]
F_3	[F,H]	[C,B,E,A,D,G,F,H]

If t represents the number of edges and n represents the number of vertices, then in the worst case scenario, the run time of a Breadth First Algorithm is $O(e+n)$ so the Ford-Fulkerson Algorithm used here is $O(C*(t+n))$. Given that every vertex has at least one edge, it can be said that $2t \geq n$ so $O(C(3t))$ which for all intents and purposes is $O(Ct)$.

V. Experiments

The speed of the algorithm grew linearly as the size of the number of residents and days increased as shown in the graph below.



VI. Conclusion

The algorithm performed as expected based on the theoretical complexity of the Ford-Fulkerson Algorithm. The time complexity of the Ford-Fulkerson Algorithm, $O(Ct)$, is a linear progression which is reflected in the graph above.

The algorithm took much less time than originally anticipated. In fact, making the test cases took about what felt like 99% of the overall test speed from the time the shell was started to the completion of the code.

VII. References

Ford, L. R. *Network Flow Theory*. Defense Documentation Center, 14 Aug. 1956, apps.dtic.mil/dtic/tr/fulltext/u2/422842.pdf.

Kleinberg, Jon, and Tardos Éva. *Algorithm Design*. Pearson, 2014.

Greenberg, Harvey J. *Ford-Fulkerson Max Flow Labeling Algorithm*. University of Colorado at Denver, 22 Dec. 1998, pdfs.semanticscholar.org/1cd0/fe3146ef431db5e515b4fe168c4475a7ac9f.pdf.

VIII. Appendix A-Code

```
'''
Finnian Wengerd
Algorithms: Network Flow
Paper 2
Professor Daniel Showalter
Problem 15
04/10/2019

Given a set number of people (p) and days (d), where p == d, and sets of days(S)
where persons {p1,...,pn} are unavailable to cook dinner,
find a Bipartite Graph (G) that represents the perfect matching
if and only if there exists a feasible schedule

'''
from datetime import datetime
import random
#test case:

def start(p,d,S):
    p = p
    d = d
    S = S
    Available = {}
    allEdges = {}
    src = 'src'            #initialize src
    sink = 'sink'          #initialize sink
    timeframeDict = {}          #initialize timeframe as a dictionary


          '''This function takes the days unavailable and
          creates a dict of available days for each person'''

    def createAvailable():
        global timeframe
        timeframe = ['d'+str(day) for day in range(1,d+1)]
        for person, days in S.items():
            x = str(days)
            x = x[1:-1]
            Available[person] = [possible for possible in timeframe if possible not in days]
```

```
'''This function adds the src to the dictionary Available and the sink to the timeframeDict
(see Lights and Switches start())'''
    def createSrcSink():
        Available[src] = []         #Adds the key src to the Available dictionary
        Available[sink] = []
        for person in Available:  #for each person including src listed in dictionary
            if person is not src and person is not sink:   #removes src as an option to iterate through
                Available[src].append(person)   #add person as a connection to src
        for days in timeframe:    #for each day listed in timeframe (for each day of the week)
            timeframeDict[days] = [sink] #add the key day and connect sink as the connection in
timeframeDict
            '''
            This function first combines the dictionaries Available and timeframeDict to show all of
the possible edges from src to sink.
            Then it lookes through the available edges and finds a path
            through the graph that allows each
            p to have one d and vice versa (See Lights and Switches tryPath())'''

    def tryPath():
        allEdges = {**Available, **timeframeDict}
        pointer = src                                   #pointer starts at the source
        visited = []                              #A list of visited nodes
        counter = 0                               #Starts the counter for times the pointer has been at
sink
        while allEdges[pointer] != [] and pathExists(allEdges, pointer): #While the pointer has no
children and a path from the
                                                  #pointer to the sink still exists
            next_node = allEdges[pointer][0]                  #the next node is set to the first/left
child

            allEdges[pointer].remove(next_node)               #removes the path taken from parent
to child from allEdges
            allEdges[next_node].append(pointer)               #adds a new opposite edge from
child to parent in allEdges
            if next_node not in visited:                #if the child of the current node has not been
visited
                visited.append(next_node)               #go to the child and record the path in
visited
                pointer = next_node
            else:
                visited.remove(next_node)               #if not, remove the child from the visited list
                pointer = next_node
            if next_node == "sink":                     #If the visited node is the sink
              counter +=1
              pointer = src
```

```
    if p == counter:                    #if there are the same number of edges leaving the src as
there are entering the sink
      #print("True")                            #There is a solution
      return True
    else:
      #print("False")                           #There is no solution
       return False


      #This function checks to see if a path from a starting place to the sink still exists
  def pathExists(allEdges,start):
    examined = []
    queue = [start]
    while queue != []:
      pointer = queue.pop(0)
      examined.append(pointer)
      for child in allEdges[pointer]:
        if child == sink:
          return True
        if child not in queue and child not in examined:
          queue.append(child)
    return False
  createAvailable()
  createSrcSink()
  tryPath()
  endtime = datetime.now()
  return endtime
#Begin


p = 3200
d = 3200
S = {}
def testcase():
  S = {}
  for i in range(1,p+1):
    randomDayJ = random.randint(1,d)
    for j in range(randomDayJ):
      mynumday = 'd'+str(random.randint(1,d))
      if 'p'+str(i) not in S:
        S['p'+str(i)] = [mynumday]
      else:
        if mynumday not in S['p'+str(i)]:
          S['p'+str(i)].append(mynumday)
```

```
totalTime = []
deltaT = 0
totalseconds = 0
averageTime = 0


for i in range(100):
    testcase()
    starttime = datetime.now()
    endtime = start(p,d,S)
    deltaT = endtime-starttime
    totalTime.append(deltaT.total_seconds())
for i in totalTime:
    totalseconds +=i
averageTime = (totalseconds/len(totalTime))

print("Average runtime: %f seconds." %(averageTime))
```

IX. Appendix B-Representative Sample of Timed Results

Residents	**Average Time of 100 Runs (s)**
25	0.000049
50	0.000086
100	0.000101
200	0.000129
400	0.000333
800	0.000668
1600	0.001039
3200	0.001871