**Time Complexity Analysis of Ford-Fulkerson and Network Flow**

Seth Brenneman

Eastern Mennonite University

1. **Abstract**

The Ford-Fulkerson algorithm is a very famous algorithm that can determine the maximum flow through a given network. A network can be many different things and can be applied in many different problems. To find the maximum flow through a network, the Ford-Fulkerson algorithm uses a breadth-first search or a depth-first search to find paths from a source node (starting node) to a sink node (ending node). This is called augmenting a path and as long as the algorithm is able to augment a path from the source to the sink it continues to run and adds the bottleneck of each path to the maximum flow. Assuming that all edges in the graph are integers, the Ford-Fulkerson algorithm can be implemented in O(*mC*) time [1]. This specific implementation of the Ford-Fulkerson algorithm was used in determining if a certain set of lights and switches were ergonomic. The lights and switches in this problem were points (x, y) within a set of walls. In order for the lights and switches to be ergonomic, each light and switch had to have a line segment between them that was not intersected by a wall. A visual representation of the problem can be seen below.

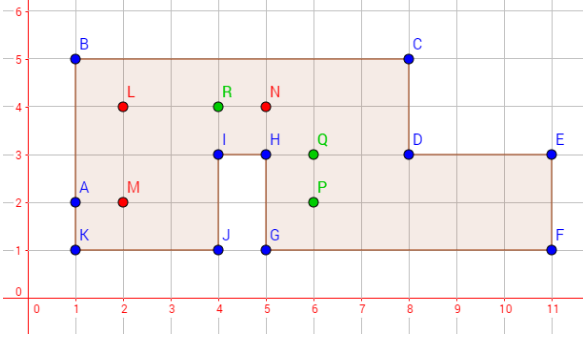




Figure 1: Visual representation of an ergonomic set of lights and switches.

1. **Background**

According to a scientific paper where David K. Smith reviews the book *Network Flows: Theory, Algorithms, and Applications*, he states, “The study of network flows can be traced to the early days of linear programming, and, like that subject, many of the mathematical foundations were laid in the generation from the mid-1940's to the mid-1970's” [2]. The development of the Ford-Fulkerson algorithm, therefore, fits perfectly within this time period as the mathematical problem of maximum flow through a network was first proposed to Ford and Fulkerson in the year 1955. The book *Flows in Networks* was then published by Ford and Fulkerson in the year 1962 where they outlined the specifics of the algorithm. They believed that study of maximal flow arose “naturally in the study of transportation or communication networks” [3].

1. **Methods**

The lights and switches problem originated from *Algorithm Design* by Jon Kleinberg and Eva Tardos. Specifically, chapter 7 exercise 6. Case sizes for this implementation of the Ford-Fulkerson algorithm were made by hand and were of size 3, 6, 9, 12, and 15 respectively. These sizes were for both the list of lights and the list of switches. The original area of the wall structure was too small for the larger amounts of lights and switches and was therefore adjusted to fit the larger case sizes. The algorithm was written in Python version 3.8.2 on a laptop with an Intel Core i5-8250U CPU alongside 8 GB of memory. To time the speed of the algorithm, the Python timeit module was used. A representation of the bipartite graph used to determine if a set of three lights and three switches was ergonomic can be seen below.

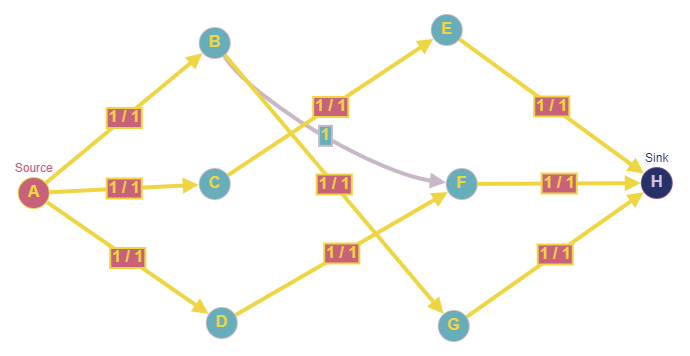


Figure 2: Maximum flow through a bipartite graph of three lights and three switches.

1. **Complexity**

This implementation of the Ford-Fulkerson algorithm demonstrated polynomial time. This was no surprise as the book *Algorithm Design* detailed how the time complexity of Ford-Fulkerson is O(*mC*). The *m* being the number of edges of the graph and *C* being the value of the maximum flow. Below is a scatter plot of the data collected.

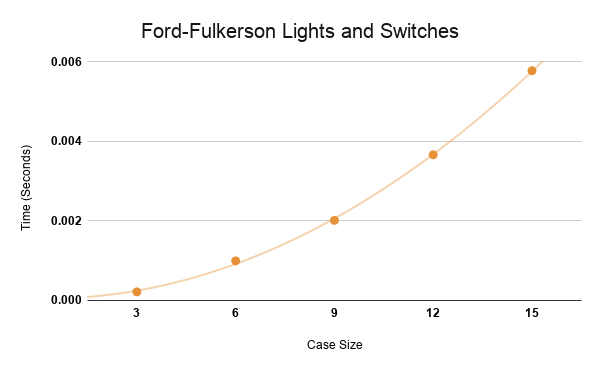


Figure 3: Scatter plot of data.

The best fit line for this scatter plot was polynomial and had an *R2* = 1. This algorithm was timed using Python’s timeit module. Larger case sizes would have been preferred but creating them by hand was a difficult task.

1. **Experiments**

In order to run a bipartite graph through the Ford-Fulkerson algorithm and the breadth-first search, an adjacency matrix was required. Below is a representation of an adjacency matrix that the Ford-Fulkerson algorithm and breadth-first search were looking for.

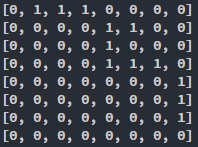


Figure 4: Example of an adjacency matrix used in the algorithm.

This matrix was generated by first setting up the source edges. The source edges are represented by the three 1’s along the top of the matrix. The sink edges are represented by the three 1’s closest to the bottom corner. Then each light and switch were checked against each other to see whether a light was visible from a switch or the line segment between them did not intersect with a wall. A light that is visible from a switch is represented by the 1’s more in the center of the matrix. A zero indicates that a light is not visible from a particular switch.

1. **Conclusion**

When implementing the Ford-Fulkerson algorithm it is important to keep in mind that there is a lot more that goes into it than just finding the maximum flow of a particular graph. First the graph must be generated and then a traversal such as a breadth-first or depth-first search must be implemented. Every individual problem that requires the maximum flow may be slightly different in small ways but the Ford-Fulkerson algorithm itself still stand as an efficient and important algorithm in the world of computer science.

**References**

[1] Jon Kleinberg, Eva Tardos. (2006). Algorithm Design. Pearson Education, Inc.

[2] David K. Smith. (1994). Book review: Network Flows: Theory, Algorithms, and Applications. Journal of the Operational Research Society Vol. 45, No. 11.

[3] LR Ford Jr, DR Fulkerson. (1962). “Flows in Networks”. Princeton Legacy Library.

**Appendix – Code**

"""

Seth Brenneman

--------------

Analysis of Algorithms

----------------------

April 20, 2021

--------------

For-Fulkerson

-------------

Credit to Daniel Showalter for providing the function to check whether two lines intersect for the lights and switches problem.

The max\_flow\_graph function was adapted from Daniel Showalter's maxFlow function.

Credit to Quinston Pimenta (https://www.youtube.com/watch?v=GoVjOT30xwo) for detailing and showing how the Ford Fulkerson algorithm

works and translates into code. He is responsible for providing the breadth first search function and the Fulkerson function to get my code running.

"""

from timeit import timeit

*#Walls = [(1,2),(1,6),(8,6),(8,4),(11,4),(11,0),(5,0),(5,4),(4,4),(4,0),(1,0),(1,2)]*

Walls = [(1,2),(1,5),(8,5),(8,3),(11,3),(11,1),(5,1),(5,3),(4,3),(4,1),(1,1),(1,2)]

*#\* n = 3*

Lights = [(2,4),(2,2),(5,4)]

Switches = [(4,4),(6,3),(6,2)]

*#\* n = 6*

*#Lights = [(2,2),(3,3),(6,5),(7,1),(10,3),(8,3)]*

*#Switches = [(2,5),(4,5),(5,5),(9,2),(8,1),(7,4)]*

*#\* n = 9*

*#Lights = [(2,2),(3,3),(6,5),(7,1),(10,3),(8,3),(3,2),(6,1),(7,5)]*

*#Switches = [(2,5),(4,5),(5,5),(9,2),(8,1),(7,4),(3,4),(9,1),(3,1)]*

*#\* n = 12*

*#Lights = [(2,2),(3,3),(6,5),(7,1),(10,3),(8,3),(3,2),(6,1),(7,5),(2,1),(6,2),(7,2)]*

*#Switches = [(2,5),(4,5),(5,5),(9,2),(8,1),(7,4),(3,4),(9,1),(3,1),(10,1),(8,2),(2,3)]*

*#\* n = 15*

*#Lights = [(2,2),(3,3),(6,5),(7,1),(10,3),(8,3),(3,2),(6,1),(7,5),(2,1),(6,2),(7,2),(6,3),(9,3),(2,4)]*

*#Switches = [(2,5),(4,5),(5,5),(9,2),(8,1),(7,4),(3,4),(9,1),(3,1),(10,1),(8,2),(2,3),(7,3),(3,5),(10,2)]*

*#\* Case 1: Ergonomic*

*#Lights = [(2,4),(2,2),(5,4)]*

*#Switches = [(4,4),(6,3),(6,2)]*

*#\* Case 2: Not ergonomic*

*#Lights = [(2,4),(2,2),(5,4)]*

*#Switches = [(6,2),(7,4),(6,3)]*

*#\* Case 3: Not ergonomic*

*#Lights = [(2,4),(2,2),(5,4)]*

*#Switches = [(6,2),(7,2),(4,4)]*

*#\* Return true if line segments AB and CD intersect*

*# Source: http://bryceboe.com/2006/10/23/line-segment-intersection-algorithm/*

def ccw(A,B,C):

    return (C[1]-A[1]) \* (B[0]-A[0]) > (B[1]-A[1]) \* (C[0]-A[0])

def intersect(A,B,C,D):

    return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)

def visible(pt1,pt2,Walls):

    x1,y1 = pt1

    x2,y2 = pt2

    for i,wall in enumerate(Walls[:-1]):

        x3,y3 = wall

        x4,y4 = Walls[i+1]

        if intersect((x1,y1),(x2,y2),(x3,y3),(x4,y4)):

            return False

    return True

def breadth\_first\_search(graph, source, sink, parentTracker):

    discovered = [False] \* len(graph) *#Create a list that stores whether a node has been visited*

    discovered[source] = True *#Set the source node to discovered*

    queue = [] *#Queue to store the nodes that will soon be traversed*

    queue.append(source) *#Append the source node to the queue*

    parentTracker[source] = -1 *#This parent tracker list keeps track of the parent nodes for each node*

    while len(queue) > 0: *#While the queue still has nodes to be traversed*

        u = queue.pop(0) *#Current node (u) is the next node from the queue*

        for v in range(0, len(graph)): *#For all the other nodes in the graph*

            if discovered[v] == False and graph[u][v] > 0: *#If there exists a path between u and v and v has not been discovered and the residual graph is greater than 0*

                queue.append(v) *#Add v to the queue*

                discovered[v] = True *#Set the v node to visited*

                parentTracker[v] = u *#Make u the parent of node v*

    if discovered[sink]: *#If the sink was visited return True and look for another traversal*

        return True

    else:

        return False

def max\_flow\_graph(walls, lights, switches):

    numLights = len(lights)

    numSwitches = len(switches)

    vertices = 1 + numLights + numSwitches + 1

    adjMatrix = [[]] \* vertices

    i = 0

    while i < vertices:

        adjMatrix[i] = list(0 for j in range(0, vertices))

        i += 1

    for i in range(1): *#\* Setup source edges*

        for j in range(1, numLights + 1):

            adjMatrix[i][j] = 1

    for i in range(numSwitches + 1, vertices - 1): *#\* Setup sink edges*

        for j in range(1):

            adjMatrix[i][-1] = 1

    for i, light in enumerate(lights): *#\* Populate the adjacency matrix by checking each light and switch combination*

        for j, switch in enumerate(switches):

            if visible(light, switch, walls):

                adjMatrix[i + 1][j + numLights + 1] = 1

    for i in adjMatrix:

        print(i)

    print('')

    return adjMatrix

def ford\_fulkerson(graph, source, sink):

    parentTracker = [0] \* len(graph)

    u, v = 0, 0

    residualGraph = graph

    maxFlow = 0

    while breadth\_first\_search(residualGraph, source, sink, parentTracker):

        pathFlow = float('inf')

        v = sink

        while not v == source:

            u = parentTracker[v]

            pathFlow = min(pathFlow, residualGraph[u][v])

            v = parentTracker[v]

        v = sink

        while not v == source:

            u = parentTracker[v]

            residualGraph[u][v] -= pathFlow

            residualGraph[v][u] += pathFlow

            v = parentTracker[v]

        maxFlow += pathFlow

    return maxFlow

def main():

    if len(Lights) != len(Switches):

        print("The number of lights and the number of switches must be the same.")

        exit(0)

    else:

        graph = max\_flow\_graph(Walls, Lights, Switches)

        maxFlow = ford\_fulkerson(graph, 0, len(graph) - 1)

    if maxFlow == len(Lights):

        print("These lights and switches are ergonomic.")

    else:

        print("These lights and switches are not ergonomic.")

main()

*#\* Credit to Brandon Chupp for figuring out the timeit wrapper function*

*#\* --------------------------------------------------------------------*

def wrapper(func, \*args):

    def wrapped():

        return func(\*args)

    return wrapped

Ford\_Fulkerson = wrapper(main)

N = len(Lights)

*#print("size:", N, "Ford Fulkerson Time:", timeit(Ford\_Fulkerson, number = 10000)/10000)*