# MAXIMUM FLOW

By: Berend Termeer

Transportation networks are networks where commodities are transported from their production center to the marketplace.[[1]](#footnote-1) An example of a transport network is the transportation of water through pipes. There is a network with multiple type of pipes and each of them have a maximum capacity they can transport. The goal is to transport as much from the production center to the marketplace.

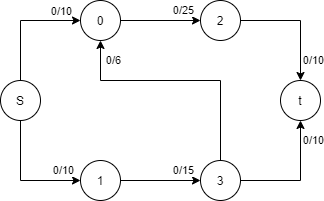
A network can be displayed in a graph with vertices with at least two vertices and . The vertex is the starting point, the production center, and does not have any incoming edges. Vertex is the endpoint of the network, the marketplace, and does not have any outgoing edges. Between the vertices there are edges. An example of a network is given in figure 1.

Figure : A maximum flow diagram in the initial state

Each edge in the flow capacity has a certain flow and capacity specified. Initially, the flow capacity through each edge is 0 and the capacity is a non-negative value.

## Folk-Fulkerson method

The goal is to find the maximum flow from *s* to *t*. This is done by using the Ford-Fulkerson method. This method repeatedly finds augmenting paths through the graph and augments the flow until no more augmented paths can be found. An augmented path is a path of edges from *s* to *t* with unused capacity greater than zero. The way that an augmented path can be found using this method is not specified for flexibility. Normally the depth-first search is used and is the default for folk-Fulkerson method. However, in this paper the Edmonds-Karp algorithm is used. This is a breadth-first algorithm specified for this method. Before going deeper into this algorithm, the Folk-Fulkerson method is defined.

In figure 2 is an augmented path displayed. An augmented path always has a bottleneck which is the “smallest” edge on the path. In this case, the bottleneck is:

The bottleneck value is used to augment the flow of the path. This means that the flow values along the path are increased by the bottleneck value. Besides this increase, the backward edges of the flow are decreased. These edges are called the residual edges and exist to “undo” bad augmenting paths which did not lead to the maximum flow

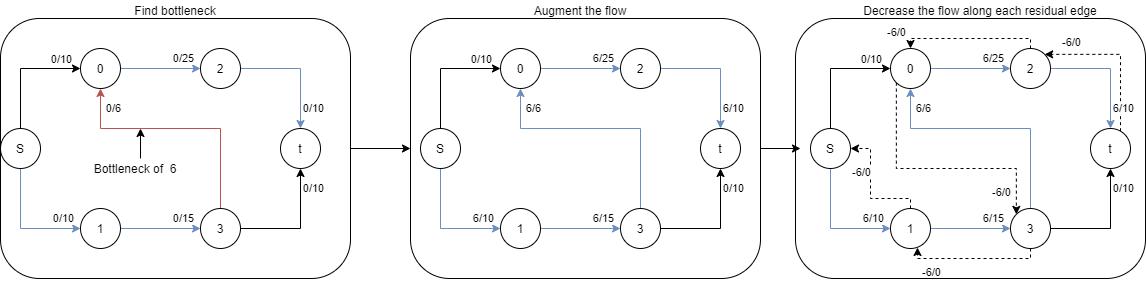


Figure : Augmented Path

The Folk-Fulkerson method continues to find augmenting paths until all of them are found between and.[[2]](#footnote-2)

## Edmonds-Karp Algorithm

As stated previously, the Folk-Fulkerson method does not specify the used algorithm to find augmented paths. By default, a depth-first search is used but can easily replaced by a different algorithm. A great candidate for optimization of finding the maximum flow is the Edmonds-Karp algorithm is used in this method. This algorithm looks for the shortest available paths by using the breadth-first search technique with a weight of 1 to each edge.[[3]](#footnote-3) How this exactly works can be seen in the table below. In this table *c* stands for the capacity and *f* for flow. The residual capacity of *vertex1(u)* and *vertex2(v)* is:

|  |  |  |
| --- | --- | --- |
| Capacity | Path | Resulting Network |
|  | s,d,e,t |  |
|  | s,a,b,t |  |
|  | s,d,c,b,t |  |
|  | s,a,c,b,t |  |

### Algorithm

Based on the previously stated rules for the Edmonds-Karp Algorithm the following pseudo-code of the algorithm can be made:

### Theoretical complexity

By analyzing this algorithm, it can be concluded that the path in with the minimum amount of edges can be found in time with the use of breadth-firsts search. Normally, a breadth-first search used in a graph with number of vertices and amount of edges requires time. However, we can assume that every vertex in the graph has at least one incident edge which implies that . We can conclude that . After the discovery of path , it takes time to augment using and time to update . In the best-case scenario, it only takes one iteration of the while loop and thus time to find the solution. However, the worst-case scenario must be found.

To find out the maximum iterations of the while loop, an indirect approach based on the breadth-first search tree (BFS tree) of starting from the endpoint is used. The BFS tree can be organized into levels: where and consist of all the vertices such that the path from to in the BFS has edges.

One of the useful properties of the BFS tree is that for all , every shortest path consists of exactly one vertex from each level (from low to high) and no other vertices. More specifically, every augmenting path that is chosen, that path consists of with for .

The following considerations can be made when graph when using the augmented using

* When contains a forward edge , then the may be deleted from and the backward can be added to . This is only possible if the backward is not in before the augmentation.
* When contains a backward edge , then this may be deleted from and the forward can be added to . This is only possible when the has been saturated before the augmentation.
* No other edges are added or deleted.
* So, every new edge that is created by augmenting using is the reverse of an edge that already belonged to .

We previously stated that every edge of goes from level to and this is the case of every newly created edge in after an augmentation. This means that for any vertex that the distance between to never decreases in this algorithm.

When choosing an augmenting path in there is always an edge that causes a bottleneck. Let us say that is the bottleneck edge of when . If is the bottleneck for then it is eliminated from after augmenting using . Suppose that and if this happens. To get back in later, most occupy a higher level than . (Remember that edges are only added to when they from one level to one preceding level.) As stated before, the distance between and does not increase. Thus, remains in the level or higher, and must rise to level to get back into

A BFS tree does not have levels above . This means that the total amount that can occur as a bottleneck is at most times. There are at maximum edges that can potentially appear in the graph and each of them can serve as a bottleneck at most times. There are at most bottleneck edges in total. An augmenting path is identified for every iteration inside the while loop and each path must have an edge with a bottleneck. This means that there are at most while loop iterations in total. In the first part of this part, it was stated that every iteration of the loop took times. The total of the Edmonds-Karp algorithm is .[[4]](#footnote-4)[[5]](#footnote-5)

### Practical complexity

Based on the previously stated pseudocode a script is written in python code. This mean that the complexity of the theoretically and the practical is the same. However, the maximum complexity is almost impossible to reach due to the fact this nearly impossible to reach. In the table below, the average of the amount of iterations in 100 tests are displayed with several different amount of edges and nodes. The created python script is found in the appendix 1.

|  |  |  |
| --- | --- | --- |
|  | Average iterations of 100 tests | Theoretical maximum complexity |
| M = 5, N = 8 | 12.92 | 320 |
| M = 5, N = 12 | 39 | 720 |
| M = 10, N = 18 | 48.76 | 3240 |
| M = 10, N = 27 | 484.39 | 7290 |
| M = 15, N = 28 | 154.41 | 11760 |
| M = 20, N = 38 | 542.84 | 28880 |
| M = 20, N = 57 | 47494.66 | 64980 |
| M = 30, N = 58 | 3730.58 | 100920 |

We can see that the theoretical maximum complexity increases when the amount of edges and/or vertices are increased. However, test results of the iterations show that the amount of iterations drastically increases when a vertices (*m*) have more edges (*n)*. This is visible in the last two tests. With m = 20 and n = 57 each vertex has 3 edges that flows to another edge. The amount if iterations are higher then with m = 30 and n = 58 where each vertex has only 2 edges that flow to another edge. The cause of this increase is that there are more possible paths to follow from each edge.

### How to use the script

The script uses a random generated system that is influenced by the input of the user. The user can change the following variables to gain an inside of the algorithm. These variables are:

1. #variables for generating a system
2. amountOfNodes = 20 #Must be higher than 2
3. minAmountOfedged = 2 #Must be higher than 1
4. maxAmountOfedged = 4 #Must be lower than the maximum amount of nodes
5. minAmountOfFlow = 100 #Must be higher than 0
6. maxAmountOfFlow = 105 # must be higher than 0
8. # choose which node is the source node and end node.
9. sourceNode = 1# must be inside the number of nodes -1 because of list starts a 0
10. endNode = 2 # must be inside the number of nodes -1 because of list starts a 0

How the system is generated, can be found in appendix 1 in the function *generateSystem().*

# References

* Bondy, J.A., Murty, U.S.R*.,* 1982: *Graph Theory with Applications*, Amsterdam.
* Cormen, T.H., Leiserson, E.E., Rivest, R.L.,2009: *Introduction to Algorithms*, London.
* Jiang, Z., Hu, X., Gao, S., 2013: *A Parallel Ford-Fulkerson Algorithm for Maximum Flow Problem,* Beijing.
* Kleinberg, B. 2012: Introduction to algorithms, New York.
* Wilf, H.S., 1994*: Algorithms and Complexity,* Philadelphia.

## Appendix 1: Python script of the Edmonds-Karp Algorithm

1. **import** sys
2. **import** os
3. **import** prettytable
4. **import** random
6. #generate system
7. **def** generateSystem():
8. **global** amountOfNodes
9. **global** minAmountOfedged
10. **global** maxAmountOfedged
11. **global** minAmountOfFlow
12. **global** maxAmountOfFlow
13. **global** sourceNode
14. **global** endNode
15. **global** totalAmountOfEdges
17. system = []
18. i = 0
19. **while** i < amountOfNodes:
20. newNode = []
21. g = 0
22. **while** g < amountOfNodes:
23. newNode.append(0)
24. g = g +1
26. #add edges if not end node
27. **if** i **is** **not** endNode:
28. amountOfEdges = random.randint(minAmountOfedged,maxAmountOfedged)
29. totalAmountOfEdges = totalAmountOfEdges + amountOfEdges
30. k=0
31. randomPoints = random.sample([x **for** x **in** range(amountOfNodes) **if** x **not** **in** [i, sourceNode]],amountOfEdges)
32. **while** k < amountOfEdges:
33. pipeValue = random.randint(minAmountOfFlow,maxAmountOfFlow)
34. newNode[randomPoints[k]] = pipeValue
35. k = k + 1
36. system.append(newNode)
37. i = i +1
38. **return** system
40. #Edmonds-Karp Algorithm
41. **def** printPartOne(wave):
42. **global** sourceNode
43. **global** endNode
44. **global** iteration
45. file.write("PART 1. Data\n")
46. file.write("1.1 System\n")
48. printSystem()
50. file.write("1.2 Begin and endnode\n Beginnode = " + str(sourceNode) + " \n Endnode   = " + str(endNode) + "\n\n")
51. file.write("Part 2. Trace\n\n")
52. file.write("wave " + str(iteration) + " . Current paths:\n")
53. file.write("\t 1) " + str1.join(str(v) **for** v **in** wave[0]) + ". Continue.")
55. **def** printPartThree(maximumFlow):
56. **global** amountOfNodes
57. **global** totalAmountOfEdges
59. file.write("\nPART 3. Result\n")
60. file.write("3.1 maximum complexity\n")
61. file.write("complexity is: O((m\*m)\*n) where m is total amount of edges and n is amount of vertices/nodes.\n")
62. calculatedMaxComplexity = (totalAmountOfEdges\*totalAmountOfEdges)\*amountOfNodes
63. file.write("("+str(totalAmountOfEdges) + "\*" + str(totalAmountOfEdges) + ") \* " + str(amountOfNodes) + "= " + str(calculatedMaxComplexity) + "\n\n")
64. file.write("3.2 Amount of iterations\n")
65. file.write("In total there were " + str(countIterations) + " iterations\n\n")
66. file.write("3.3 MaximumFlow\n The maximum flows is " + str(maximumFlow)+".\n\n")
67. file.write("3.4 FollowedPaths\n")
68. i = 0
69. **while** i < len(usedPaths):
70. file.write("\t"+str(i)+") " + str1.join(str(v) **for** v **in** usedPaths[i]) + ".\n")
71. i = i+ 1
72. file.write("\n3.5 System after subtractions\n")
73. printSystem()

76. **def** printSystem():
77. **global** nodes
78. header = ["Y/X",""]
79. **for** i **in** range(len(nodes[0])):
80. header.append(i)
81. table = prettytable.PrettyTable(header)
83. **for** i **in** range(len(nodes[1])):
84. row = [i,""]
85. **for** n **in** nodes[i]:
86. row.append(n)
88. table.add\_row(row)
89. file.write(str(table) + "\n")
91. **def** getAllAdjecentNodes(indexOfNode):
92. **global** nodes
93. allAdjecentNodes = []
94. i = 0
95. **while** i < len(nodes):
96. node = nodes[i]
97. **if**(node[indexOfNode]> 0):
98. allAdjecentNodes.append(i)
99. i = i + 1
100. **return** allAdjecentNodes
102. **def** alterNodes(wave):
103. **global** nodes
104. **global** sourceNode
105. **global** usedPaths
106. **global** countIterations
108. deletedPaths = []
110. g = 0
112. **while** g < len(wave):
113. countIterations = countIterations + 1
114. index = g +1
115. path = wave[g]
116. **if** path[-1] **in** path[:-1]:
117. file.write("\t "+str(index)+") " + str1.join(str(v) **for** v **in** path) + ".Error. Loop Found. Delete this path\n")
118. deletedPaths.append(path)
119. **elif** path[-1] == sourceNode:
120. file.write("\t "+str(index)+") " + str1.join(str(v) **for** v **in** path) + ". Endnode Found. Subtract path from system.\n")
121. path.reverse()
123. #get all values
124. i = 0
125. values = []
126. **while** i < len(path) -1:
127. values.append(nodes[path[i]][path[i+1]])
128. i = i +1
129. #find bottleneck
130. minValue = min(values)
132. #change values
133. i = 0
134. **while** i < len(path) - 1:
135. nodes[path[i]][path[i+1]] = nodes[path[i]][path[i+1]]-minValue
136. i = i+1
138. usedPaths.append(path)
139. deletedPaths.append(path)
140. **else**:
141. path.reverse()
142. i = 0
143. values = []
144. **while** i < len(path) -1:
145. values.append(nodes[path[i]][path[i+1]])
146. i = i +1
147. #find out if there is a value set on 0
148. minValue = min(values)
149. path.reverse()
150. **if** minValue == 0:
151. file.write("\t "+str(index)+") " + str1.join(str(v) **for** v **in** path) + ".Error. Value of zero in path. subtract path from system.\n")
152. **else**:
153. file.write("\t "+str(index)+") " + str1.join(str(v) **for** v **in** path) + ". Continue.\n")
154. g = g+1
156. **for** path **in** deletedPaths:
157. wave.remove(path)
158. **return** wave

161. **def** bfs(wave):
162. **global** nodes
163. **global** iteration
165. iteration = iteration+1
166. newWave = []
167. **for** path **in** wave:
168. lastIndex = path[-1]
169. allPossibleMoves = getAllAdjecentNodes(lastIndex)
170. **for** possibleMove **in** allPossibleMoves:
171. addMove = []
172. **for** x **in** path:
173. addMove.append(x)
174. addMove.append(possibleMove)
175. newWave.append(addMove)
176. **if** len(newWave) > 0:
177. file.write("\n\nwave " + str(iteration) + " . Current paths:\n")
178. newWave = alterNodes(newWave)
179. **if** len(newWave) > 0:
180. bfs(newWave)
181. **else**:
182. file.write("\n\n Search finished \n")
184. """file name and delete already exsisting data in file"""
185. fileName = "maxFlowAlgotirhmResult.txt"
186. file = open(fileName,"a")
187. file.truncate(0)

190. #variables for generating a system
191. amountOfNodes = 20 #Must be higher then 2
192. minAmountOfedged = 2 #Must be higher then 1
193. maxAmountOfedged = 4 #Must be lower then the maximum amount of nodes
194. minAmountOfFlow = 100 #Must be higher then 0
195. maxAmountOfFlow = 105 # must be higher then 0
197. # choose which node is the sourceNode and end node.
198. sourceNode = 1# must be inside the amount of nodes -1 because of list starts a 0
199. endNode = 2 # must be inside the amount of nodes -1 because of list starts a 0
201. #non-changable variables
202. totalAmountOfEdges = 0
203. maximumFlow = 0
204. leftOverlowAfterAlgorithm = 0
205. usedPaths = []
206. wave = [[endNode]]
207. iteration = 0
208. str1 = ", "
209. countIterations = 0

212. # make a capacity graph
213. nodes = generateSystem()
215. **for** node **in** nodes:
216. maximumFlow = maximumFlow + node[endNode]
218. #print starting data
219. printPartOne(wave)
220. #start algorithm
221. bfs(wave)

224. **for** node **in** nodes:
225. leftOverlowAfterAlgorithm = leftOverlowAfterAlgorithm + node[endNode]
226. maximumFlow = maximumFlow - leftOverlowAfterAlgorithm
228. #print results
229. printPartThree(maximumFlow)
231. file.close()
232. os.startfile(r"C:\Users\berend\Documents\Erasmus\algorithm analysis\maxFlowAlgotirhmResult.txt")

1. Bondy, J.A., Murty, U.S.R., *p. 191* (1982) [↑](#footnote-ref-1)
2. Wilf, H.S., p. 63-72 (1994). [↑](#footnote-ref-2)
3. Jiang, Z., Hu, X., Gao, S., p. 1-3 (2013). [↑](#footnote-ref-3)
4. Cormen, T.H., Leiserson, E.E., Rivest, R.L., p. 547-566 (2009). [↑](#footnote-ref-4)
5. Kleinberg, B. p 1-3 (2012). [↑](#footnote-ref-5)