Dinic's Algorithm

Ian Orzel, Grant Butler, Josh Temel

April 2022

Contents

0.1 Abstract

Dinic's algorithm solves the maximum flow problem, which is a problem that has applications in computer networks and in electrical circuit theory. It also runs significantly faster than other common algorithms, such as Ford-Fulkerson and Edmonds-Karp. This paper features an implementation of the algorithm that is shown to be both correct and have the right time complexity. The correctness and the time complexity of the algorithm is also analyzed theoretically. We find that the algorithm runs in $O(|V|^2|E|)$ time, where $|V|$ is the number of vertices in the network and $|E|$ is the number of edges in the network.

Chapter 1

The Max Flow Problem

A graph is a common tool utilized by computer scientists and mathematicians to study many different phenomenon. There are many different types of graphs, but we examine a unique case of them in this paper, which we call a network. These graphs are directed, weighted, and have a defined source and sink node.

We will try to build some intuition on what is implied by a network. The simplest analogy is to visualize a network of pipes. The edges in the network are pipes, and the vertices are junction points of pipes. Rather than visualizing the "weights" of edges as weights, we call these capacities. These represent the maximum flow that can be sent down a pipe. Then, in the network, we define the source as an infinite source of water and the sink as an infinite reserve for water.

Then, one can imagine flow being sent through the network. Flow is sent from the source and ends up in the sink. At each node other than the source and the sink, the amount of flow input into the node equals the amount of flow that is output. One can imagine the flow as water being sent through the network of pipes. Hence, the flow value is the amount of flow that has been sent through the network.

Now, we can define the problem of interest that we wish to solve through this project: what is the maximum amount of flow that we can send through this network from the source to the sink? The algorithm, Dinic's algorithm, attempts to find the max flow of an arbitrary network.

Chapter 2

Historical Solutions

Although this paper will focus on an analysis of Dinic's algorithm, there are other proposed algorithms that solve the max flow problem. Two of them that will be discussed in this section are the Ford-Fulkerson algorithm and the Edmonds-Karp algorithm. Although these also solve the max flow problem, their time complexities are worse when compared to Dinic's algorithm. With that being said, a lot of the ideas of Dinic's algorithm are reflected in the other two algorithms. Becuase these algorithms are much simplier to understand than Dinic's algorithm, we will spend some time discussing them in order to help bridge into the algorithm.

2.1 Ford-Fulkerson

The Ford-Fulkerson algorithm is a very basic algorithm to solve the max flow problem. It operates on the simple idea of picking an arbitrary path from the source to the sink, sending flow down the path, and updating the graph repeatedly until the source became disconnected from the sink. Some these steps seem a little ambiguous, so we will provide some minor description of how they operate.

The idea of picking an arbitrary path from the source to the sink is obvious, so we first focus on sending flow down the path. This is done by finding the edge on the path with the smallest capacity. This value will be the amount of flow that is sent down the path. This flow value is stored, as the max flow of the graph will be the sum of all of these flow values.

Next, the graph will need to be updated along this path. First, each edge along the path will have its capacity decreased by the amount of flow that was just sent along the path. Then, we need to define a back edge of an edge in a network, which is a corresponding edge that points in the opposite direction. During the update process, the back edges of every edge in the path will be increased by the amount of flow sent along the path.

This process continues until the source is disconnected from the sink. In this case, a blocking flow has been reached, which means that the corresponding flow is the max flow of the network. There is more information about this fact in the mathematical analysis.

The algorithm has a time complexity of $O(|E| * f)$, where $|E|$ is the number of edges in the network and f is the max flow of the network. We can clearly see that this algorithm is not ideal, as we generally do not want the time complexity of an algorithm to be a function of the value being computed. It is for this reason that the Edmonds-Karp algorithm improves upon the shortcomings of Ford-Fulkerson.

2.2 Edmonds-Karp

The Edmonds-Karp algorithm operates similarly to the Ford-Fulkerson algorithm, but there is one major difference between the two. While Ford-Fulkerson chooses an arbitrary path to send the flow down for each iteration, Edmonds-Karp instead chooses the shortest path as found by breadth-first search (BFS). One must be careful when stating "shortest" path, as, although the graph is weighted, this shortest path does not take into account the weights of the edges. Instead, this shortest path is defined by the number of edges included in the path between the two nodes. Hence, Edmonds-Karp finds a shortest path between the source and sink, and it sends flow down this path.

Beyond this fact, Ford-Fulkerson and Edmonds-Karp are identical. So why does there exist a separate name for this seemingly very similar algorithm. This differnce can be seen in the time complexity, as Edmonds-Karp has a complexity of $O(|V||E|^2)$, were $|V|$ is the number of vertices in the network and $|E|$ is the number of edges in the network. Now, we posit that Edmonds-Karp has a "better" time complexity than Ford-Fulkerson, but this is not the most obvious statement. Ford-Fulkerson is unique in the fact that its time complexity is a function of the value it is computing. So, if this value is very small, Ford-Fulkerson runs much faster than Edmonds-Karp. But, in a more general case where the max flow can be arbitrary large (which it is often assumed to be), Edmonds-Karp runs in much faster time (as it is a special case of Ford-Fulkerson).

Chapter 3

Dinic's Algorithm

3.1 Description of Dinic's

The purpose of Dinic's Algorithm is to solve the max flow problem in a much faster time complexity than other solutions. It performs in $O\left(|V|^2|E|\right)$ time, which is much faster than the Ford-Fulkerson or Edmonds-Karp algorithms. Dinic's starts in a similar way to Edmonds-Karp and uses BFS to find the level of each node in the graph. Then, multiple DFSs are used to find paths from the source to the sink, where the only valid paths are ones that go from level L to $L+1$. The capacity of the edges are then updated based on the bottleneck value through the path. This is repeated until a blocking flow is reached, where no more flow can be sent from the source to the sink. Adding up all bottleneck value values renders the maximum flow through the graph.

3.2 Visualization of Dinic's

The following section details a step by step example of the algorithm running. [1]

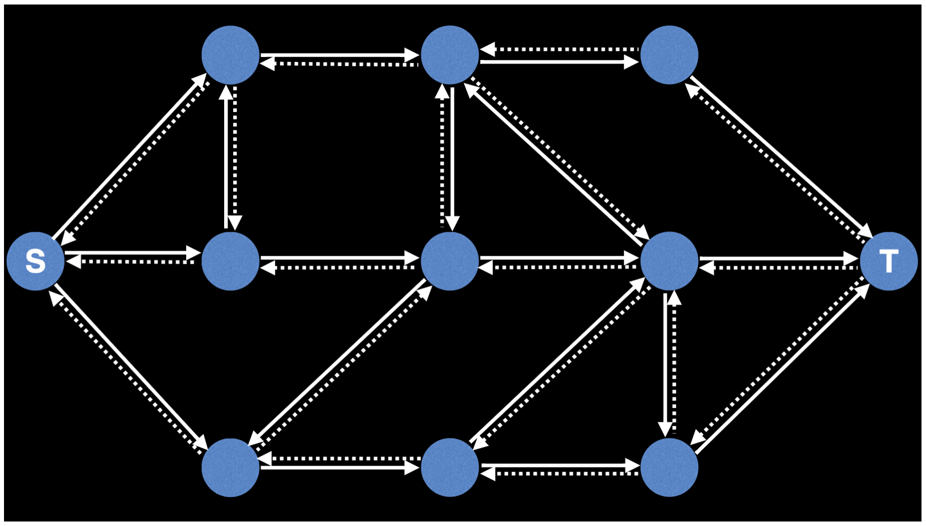


Figure 3.1: Graph G with nodes s (source) and t (sink).

[1] All figures from: `https://github.com/williamfiset/Algorithms/blob/master/slides/graphtheory/network_flow_algorithms.pdf`

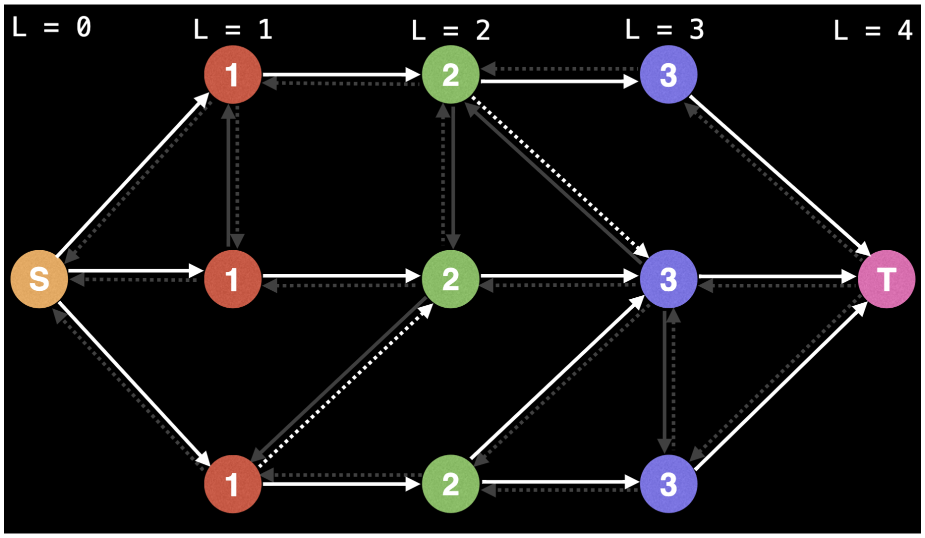


Figure 3.2: Marked levels after running *BFS(G, s)*

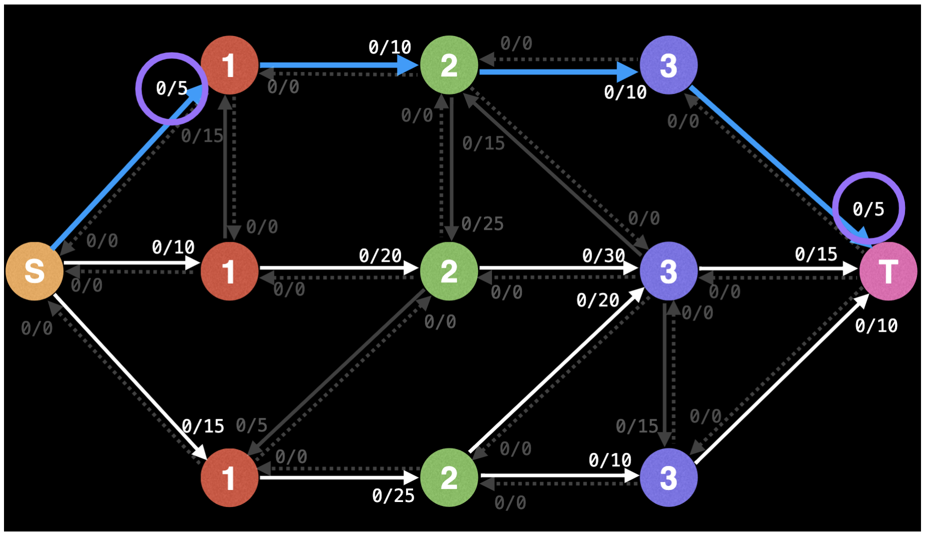


Figure 3.3: Path found using *DFS(G, s, t, ∞)*,
marking **bottleneck value** of the path.

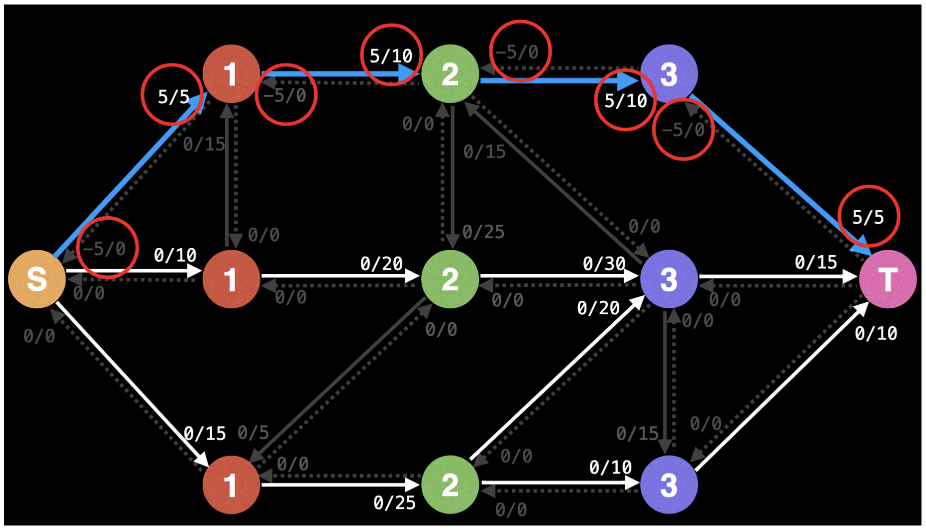


Figure 3.4: Augmenting the flow values on the path with **bottleneck value**.

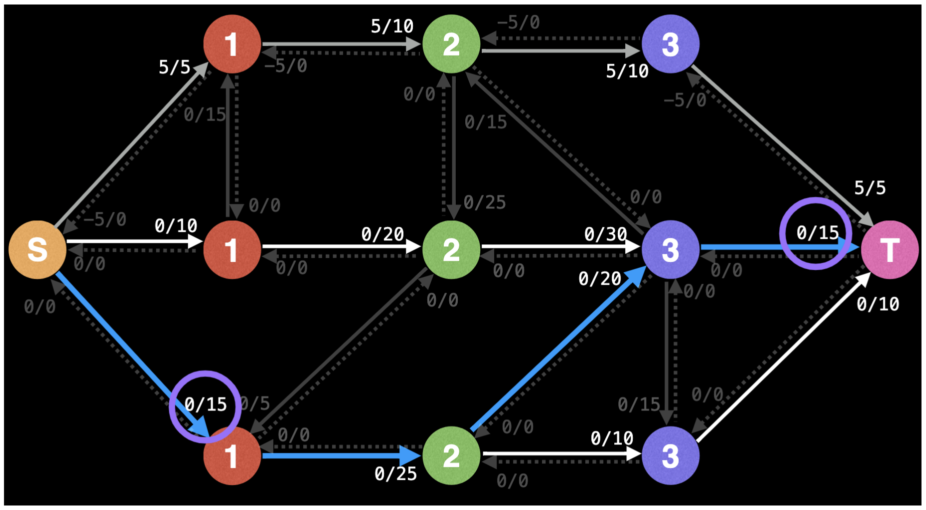


Figure 3.5: Another path found using *DFS(G, s, t, ∞)*.

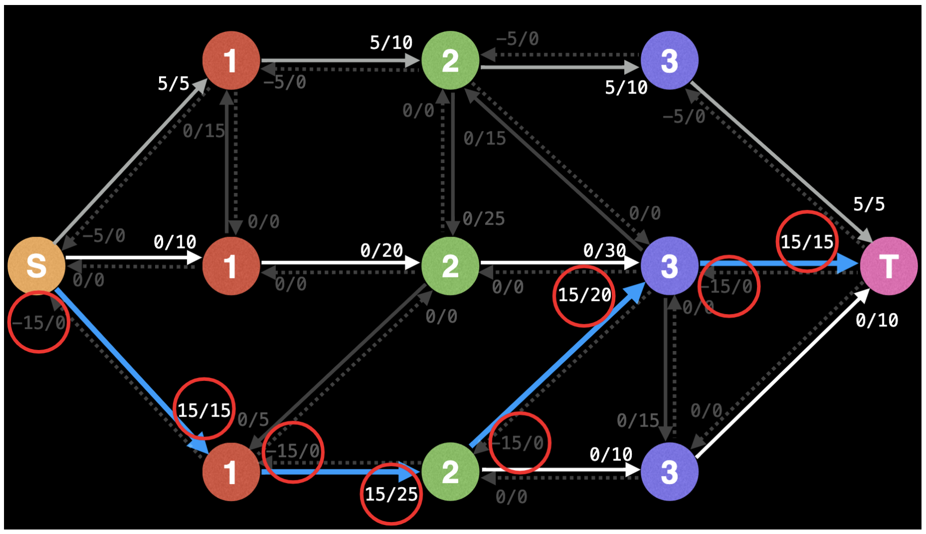


Figure 3.6: Augmented flow values on the path.

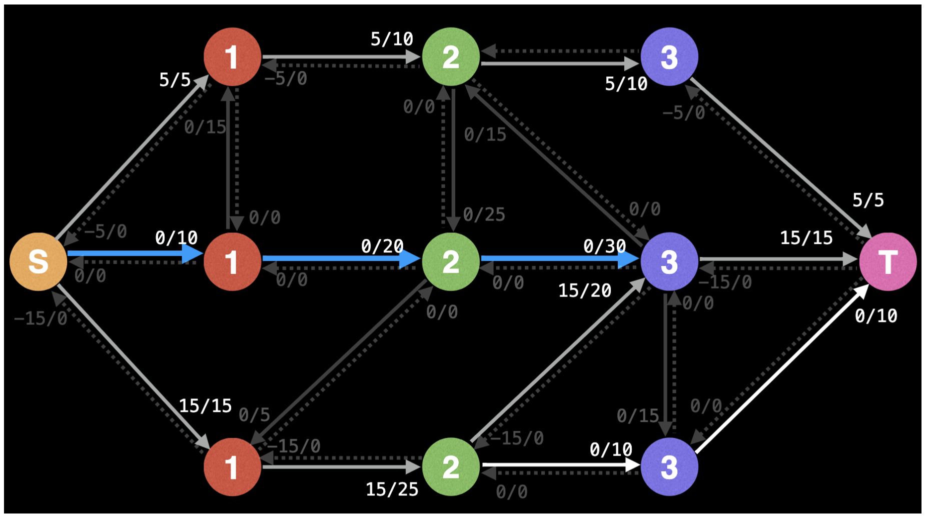


Figure 3.7: Blocking flow reached, so the algorithm has reached its end.

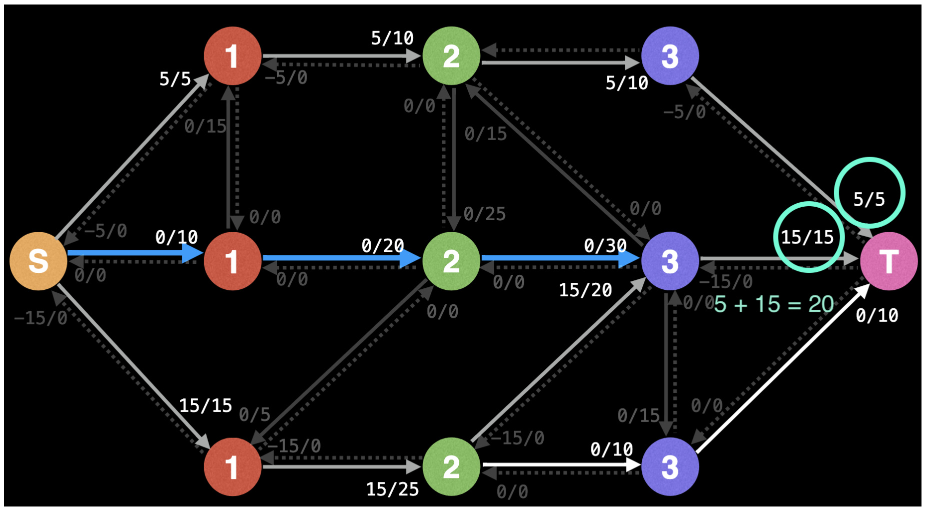


Figure 3.8: Maximum flow is the sum of the bottleneck values, so $5 + 15 = 20$.

3.3 Pseudo Code for Dinic's

The following section contains a pseudo code example of how an implementation of the algorithm can be accomplished.

Algorithm 1 Dinic's Algorithm

```
procedure BFS(G, s) G → graph, s → source
    G.queue.push(s) insert s in queue

    global lvl = 1
    s.lvl = lvl
    s.visited = true

while G.queue is not empty:
    lvl += 1
    v = G.queue.pop()

    for each w ∈ G.adj[s]
    if w is not visited then
        G.queue.push(w)
        mark w as visited
        w.lvl = lvl
    end if

end procedure BFS

```

```
procedure DFS(G, v, t, flow)
    if v == sink or not flow then
        return flow
    end if

for i in (v. . .sizeof(G.adj)):
    edge = G.adj[v][i]
    if edge[0].lvl == v.lvl + 1 then
        path = DFS(G, edge[0], t, min(flow, edge[2] - edge[3])
        if path then
            G.adj[v][i][3] += path
            G.adj[edge[0]][edge[1]][3] -= path
            return path
        end if
    end if
    return 0
end procedure DFS

procedure MAX_FLOW(G, s, t)
    BFS(G, s)
    path_flow = DFS(G, s, t, $\infty$) send infinite 'water' down the pipes
while path_flow:
    flow += path_flow
    path_flow = DFS(G, s, t, $\infty$)
    if not t.lvl then
        break
    end if
    return flow
end procedure MAX_FLOW
```

Chapter 4

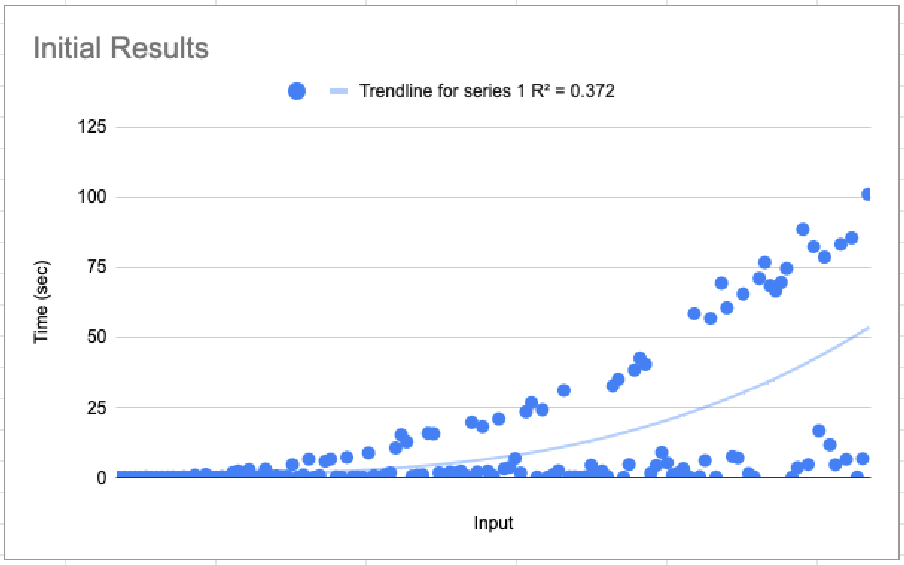
Code Analysis

This section of the document discusses the analysis of our algorithm including assessing both the time complexity and correctness of the algorithm. In the sections below the processes that were employed are described in detail. Additionally, the results will be provided in full.

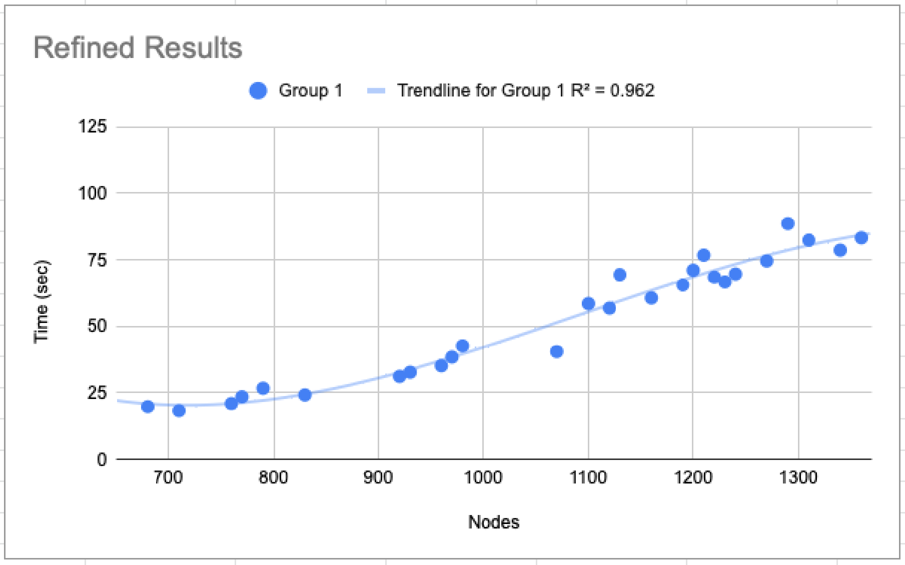
4.1 Time Complexity

As previously mentioned in the report, the time complexity of this algorithm is $O(|V|^2|E|)$. In order to confirm this, our team created many random test cases where we controlled the number of nodes and edges in a weighted graph. However, it should be noted that we did not control whether these test cases connected the source and sink. This later became an issue when trying to analyze the results, but this will be discussed later. After generating each test case, we ran the algorithm and recorded the number of nodes, edges, max flow (if any), and execution time of the algorithm in seconds and stored these results in a text file. This was repeated a number of times while increasing the number of nodes and edges each time.

After enough iterations have been recorded, the text file was imported into an excel sheet, and the execution times were plotted on a scatter chart. An example of this chart can be seen below.



What rapidly became apparent was that the data appeared to follow two different trends as the input was increased. We found that in these cases the max flow was consistently zero. The reason for this is because in these test cases the sink was not connected to the source which is the best case scenario for this algorithm so the execution time was consistently fast no matter the size of the input. In order to prevent this from skewing our r^2 value, we decided to exclude these results and simply plot only the execution times for the test cases where the sink and source were connected. This resulted in the following scatter plot:

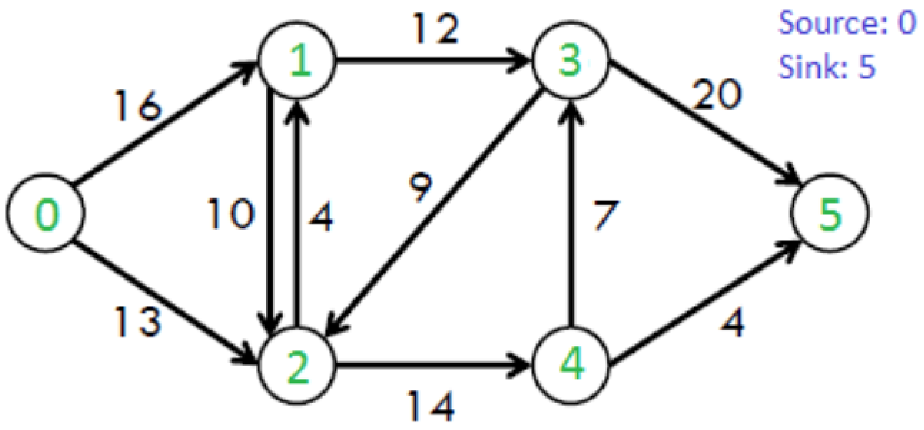


Here, we saw that we got an excellent r^2 value. From this, we concluded that the time complexity of the algorithm was as we expected it to be.

4.2 Correctness

In addition to time complexity, we also assessed our algorithm for correctness. Our test for correctness is relatively simple and of great importance to supporting our results for time complexity. If the algorithm is not correct, what does the time complexity matter?

To test correctness, we found known test cases online and ran them through our algorithm to see if they produced the expected max flow. Below is an example of one test case that we found on GeeksforGeeks.



Here you can see that the source is indeed connected to the sink and from the website we knew that the maxflow for this particular graph should be 23. This is what we obtained after running it through the algorithm. Therefore, our practical analysis is complete.