DSA Ford-Fulkerson Algorithm

‹ Previous Next ›

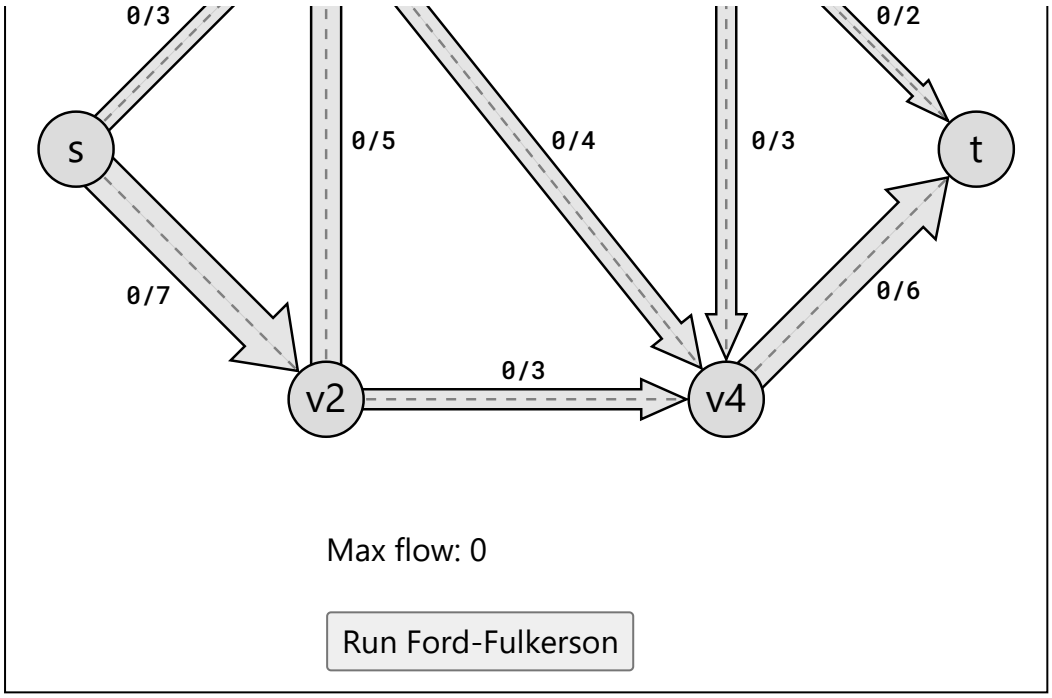
The Ford-Fulkerson algorithm solves the maximum flow problem.

Finding the maximum flow can be helpful in many areas: for optimizing network traffic, for manufacturing, for supply chain and logistics, or for airline scheduling.

The Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm solves the maximum flow problem for a directed graph.

The flow comes from a source vertex (s) and ends up in a sink vertex (t), and each edge in the graph allows a flow, limited by a capacity.



The Ford-Fulkerson algorithm works by looking for a path with available capacity from the source to the sink (called an *augmented path*), and then sends as much flow as possible through that path.

The Ford-Fulkerson algorithm continues to find new paths to send more flow through until the maximum flow is reached.

In the simulation above, the Ford-Fulkerson algorithm solves the maximum flow problem: It finds out how much flow can be sent from the source vertex s, to the sink vertex t, and that maximum flow is 8.

The numbers in the simulation above are written in fractions, where the first number is the flow, and the second number is the capacity (maximum possible flow in that edge). So for example, `0/7` on edge $s \rightarrow v_2$, means there is `0` flow, with a capacity of `7` on that edge.

Note: The Ford-Fulkerson algorithm is often described as a *method* instead of as an *algorithm*, because it does not specify how to find a path where flow can be increased. This means it can be implemented in different ways, resulting in different time complexities. But for this tutorial we will call it an algorithm, and use Depth-First-Search to find the paths.

You can see the basic step-by-step description of how the Ford-Fulkerson algorithm works below, but we need to go into more detail later to actually understand it.

1. Start with zero flow on all edges.
2. Find an *augmented path* where more flow can be sent.
3. Do a *bottleneck calculation* to find out how much flow can be sent through that augmented path.
4. Increase the flow found from the bottleneck calculation for each edge in the augmented path.
5. Repeat steps 2-4 until max flow is found. This happens when a new augmented path can no longer be found.

Residual Network in Ford-Fulkerson

The Ford-Fulkerson algorithm actually works by creating and using something called a *residual network*, which is a representation of the original graph.

In the residual network, every edge has a *residual capacity*, which is the original capacity of the edge, minus the the flow in that edge. The residual capacity can be seen as the leftover capacity in an edge with some flow.

For example, if there is a flow of 2 in the $v_3 \rightarrow v_4$ edge, and the capacity is 3, the residual flow is 1 in that edge, because there is room for sending 1 more unit of flow through that edge.

Reversed Edges in Ford-Fulkerson

The Ford-Fulkerson algorithm also uses something called *reversed edges* to send flow back. This is useful to increase the total flow.

For example, the last augmented path $s \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow t$ in the animation above and in the manual run through below shows how the total flow is increased by one more unit, by actually sending flow back on edge $v_4 \rightarrow v_3$, sending the flow in the reverse direction.

Sending flow back in the reverse direction on edge $v_3 \rightarrow v_4$ in our example meas that this 1 unit of flow going out of vertex v_3, now leaves v_3 on edge $v_3 \rightarrow t$ instead of $v_3 \rightarrow v_4$.

A reversed edge has no flow or capacity, just residual capacity. The residual capacity for a reversed edge is always the same as the flow in the corresponding original edge. In our example, the edge $v_3 \rightarrow v_4$ has a flow of 2, which means there is a residual capacity of 2 on the corresponding reversed edge $v_4 \rightarrow v_3$.

This just means that when there is a flow of 2 on the original edge $v_3 \rightarrow v_4$, there is a possibility of sending that same amount of flow back on that edge, but in the reversed direction. Using a reversed edge to push back flow can also be seen as undoing a part of the flow that is already created.

The idea of a residual network with residual capacity on edges, and the idea of reversed edges, are central to how the Ford-Fulkerson algorithm works, and we will go into more detail about this when we implement the algorithm further down on this page.

Manual Run Through

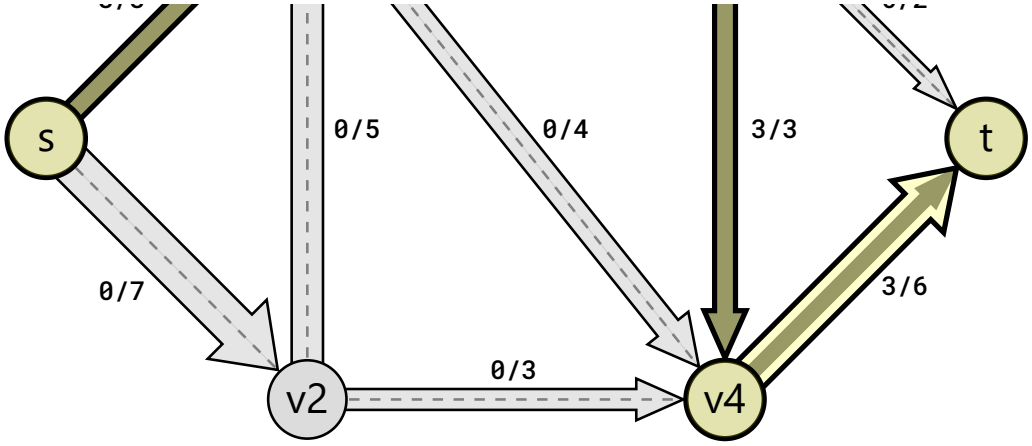
There is no flow in the graph to start with.

To find the maximum flow, the Ford-Fulkerson algorithm must increase flow, but first it needs to find out where the flow can be increased: it must find an augmented path.

The Ford-Fulkerson algorithm actually does not specify how such an augmented path is found (that is why it is often described as a method instead of an algorithm), but we will use Depth First Search (DFS) to find the augmented paths for the Ford-Fulkerson algorithm in this tutorial.

The first augmented path Ford-Fulkerson finds using DFS is $s \rightarrow v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow t$.

And using the bottleneck calculation, Ford-Fulkerson finds that 3 is the highest flow that can be sent through the augmented path, so the flow is increased by 3 for all the edges in this path.



The next iteration of the Ford-Fulkerson algorithm is to do these steps again:

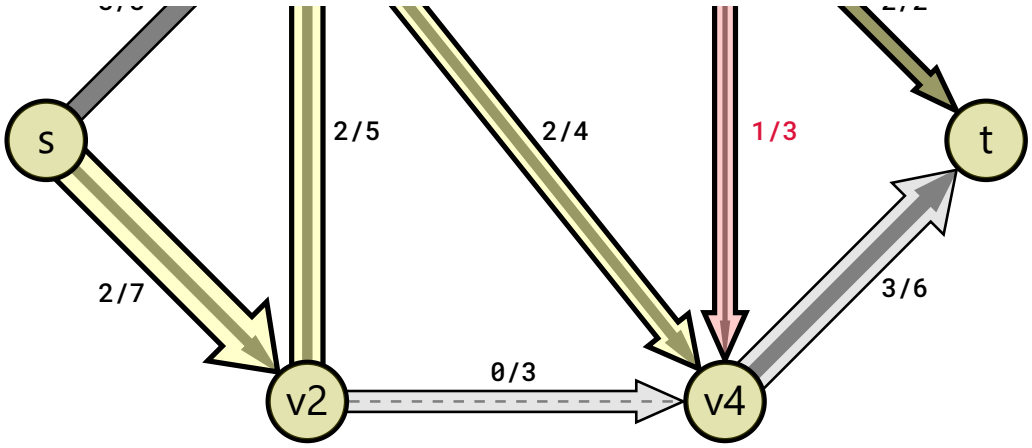
2. Find a new augmented path
3. Find how much the flow in that path can be increased
4. Increase the flow along the edges in that path accordingly

The next augmented path is found to be $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow t$, which includes the reversed edge $v_4 \rightarrow v_3$, where flow is sent back.

The Ford-Fulkerson concept of reversed edges comes in handy because it allows the path finding part of the algorithm to find an augmented path where reversed edges can also be included.

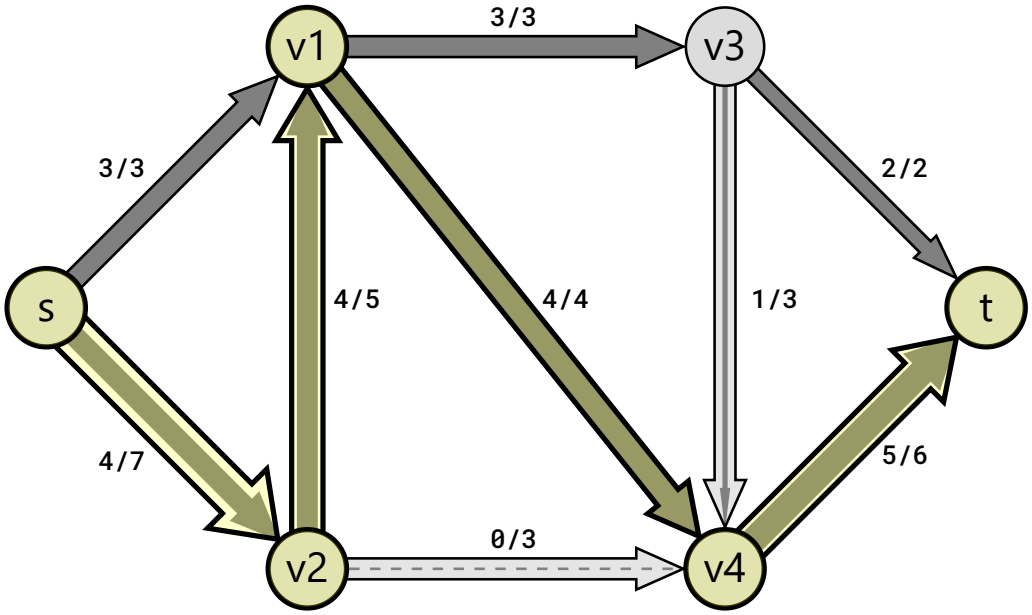
In this specific case that means that a flow of 2 can be sent back on edge $v_3 \rightarrow v_4$, going into $v_3 \rightarrow t$ instead.

The flow can only be increased by 2 in this path because that is the capacity in the $v_3 \rightarrow t$ edge.



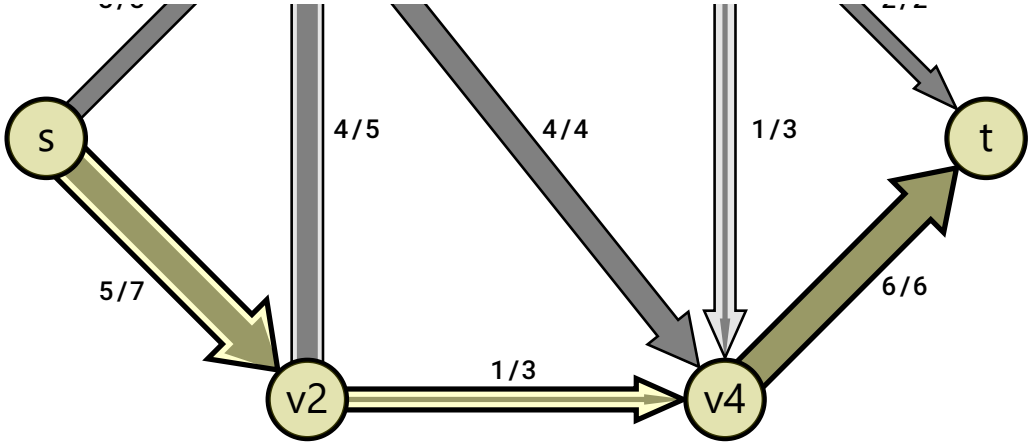
The next augmented path is found to be $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$.

The flow can be increased by 2 in this path. The bottleneck (limiting edge) is $v_1 \rightarrow v_4$ because there is only room for sending two more units of flow in that edge.



The next and last augmented path is $s \rightarrow v_2 \rightarrow v_4 \rightarrow t$.

The flow can only be increased by 1 in this path because of edge $v_4 \rightarrow t$ being the bottleneck in this path with only space for one more unit of flow ($capacity - flow = 1$).



At this point, a new augmenting path cannot be found (it is not possible to find a path where more flow can be sent through from s to t), which means the max flow has been found, and the Ford-Fulkerson algorithm is finished.

The maximum flow is 8. As you can see in the image above, the flow (8) is the same going out of the source vertex s, as the flow going into the sink vertex t.

Also, if you take any other vertex than s or t, you can see that the amount of flow going into a vertex, is the same as the flow going out of it. This is what we call *conservation of flow*, and this must hold for all such flow networks (directed graphs where each edge has a flow and a capacity).

Implementation of The Ford-Fulkerson Algorithm

To implement the Ford-Fulkerson algorithm, we create a `Graph` class. The `Graph` represents the graph with its vertices and edges:

```
1 class Graph:
2     def __init__(self, size):



6
```

Line 3: We create the `adj_matrix` to hold all the edges and edge capacities. Initial values are set to `0`.

Line 4: `size` is the number of vertices in the graph.

Line 5: The `vertex_data` holds the names of all the vertices.

Line 7-8: The `add_edge` method is used to add an edge from vertex `u` to vertex `v`, with capacity `c`.

Line 10-12: The `add_vertex_data` method is used to add a vertex name to the graph. The index of the vertex is given with the `vertex` argument, and `data` is the name of the vertex.

The `Graph` class also contains the `dfs` method to find augmented paths, using Depth-First-Search:

```
    def dfs(self, s, t, visited=None, path=None):
        if visited is None:
            visited = [False] * self.size
        if path is None:
            path = []

        visited[s] = True
        path.append(s)

        if s == t:
            return path

        for ind, val in enumerate(self.adj_matrix[s]):
            if not visited[ind] and val > 0:
                result_path = self.dfs(ind, t, visited, path.copy()
                if result_path:
                    return result_path
```

Line 15-18: The `visited` array helps to avoid revisiting the same vertices during the search for an augmented path. Vertices that belong to the augmented path are stored in the `path` array.

Line 20-21: The current vertex is marked as visited, and then added to the path.

Line 23-24: If the current vertex is the sink node, we have found an augmented path from the source vertex to the sink vertex, so that path can be returned.

Line 26-30: Looping through all edges in the adjacency matrix starting from the current vertex `s`, `ind` represents an adjacent node, and `val` is the residual capacity on the edge to that vertex. If the adjacent vertex is not visited, and has residual capacity on the edge to it, go to that node and continue searching for a path from that vertex.

Line 32: `None` is returned if no path is found.

The `fordFulkerson` method is the last method we add to the `Graph` class:

```
        def fordFulkerson(self, source, sink):
            max_flow = 0

            path = self.dfs(source, sink)
            while path:
                path_flow = float("Inf")
                for i in range(len(path) - 1):
                    u, v = path[i], path[i + 1]
                    path_flow = min(path_flow, self.adj_matrix[u][v])

                for i in range(len(path) - 1):
                    u, v = path[i], path[i + 1]
                    self.adj_matrix[u][v] -= path_flow
                    self.adj_matrix[v][u] += path_flow

                max_flow += path_flow

                path_names = [self.vertex_data[node] for node in path]
                print("Path:", " -> ".join(path_names), ", Flow:", path
```

Initially, the `max_flow` is `0`, and the `while` loop keeps increasing the `max_flow` as long as there is an augmented path to increase flow in.

Line 37: The augmented path is found.

Line 39-42: Every edge in the augmented path is checked to find out how much flow can be sent through that path.

Line 44-46: The residual capacity (capacity minus flow) for every forward edge is reduced as a result of increased flow.

Line 47: This represents the reversed edge, used by the Ford-Fulkerson algorithm so that flow can be sent back (undone) on the the original forward edges. It is important to understand that these reversed edges are not in the original graph, they are fictitious edges introduced by Ford-Fulkerson to make the algorithm work.

Line 49: Every time flow is increased over an augmented path, `max_flow` is increased by the same value.

iteration.

After defining the `Graph` class, the vertices and edges must be defined to initialize the specific graph, and the complete code for the Ford-Fulkerson algorithm example looks like this:

Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, c):
        self.adj_matrix[u][v] = c
```

```
13
14      def dfs(self, s, t, visited=None, path=None):
15          if visited is None:
```

Try it Yourself »

Time Complexity for The Ford-Fulkerson Algorithm

The time complexity for the Ford-Fulkerson varies with the number of vertices V, the number of edges E, and it actually varies with the maximum flow f in the graph as well.

The reason why the time complexity varies with the maximum flow f in the graph, is because in a graph with a high throughput, there will be more augmented paths that increase flow, and that means the DFS method that finds these augmented paths will have to run more times.

Depth-first search (DFS) has time complexity $O(V + E)$.

DFS runs once for every new augmented path. If we assume that each augmented graph increase flow by 1 unit, DFS must run f times, as many times as the value of maximum flow.

This means that time complexity for the Ford-Fulkerson algorithm, using DFS, is

$$O((V + E) \cdot f)$$

For *dense graphs*, where $E > V$, time complexity for DFS can be simplified to $O(E)$, which means that the time complexity for the Ford-Fulkerson algorithm also can be simplified to

$$O(E \cdot f)$$

A dense graph does not have an accurate definition, but it is a graph with many edges.

The next algorithm we will describe that finds maximum flow is the Edmonds-Karp algorithm.

```
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

Previous Sign in to track progress Next

COLOR PICKER

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate