DSA Edmonds-Karp Algorithm

‹ Previous Next ›

The Edmonds-Karp algorithm solves the maximum flow problem.

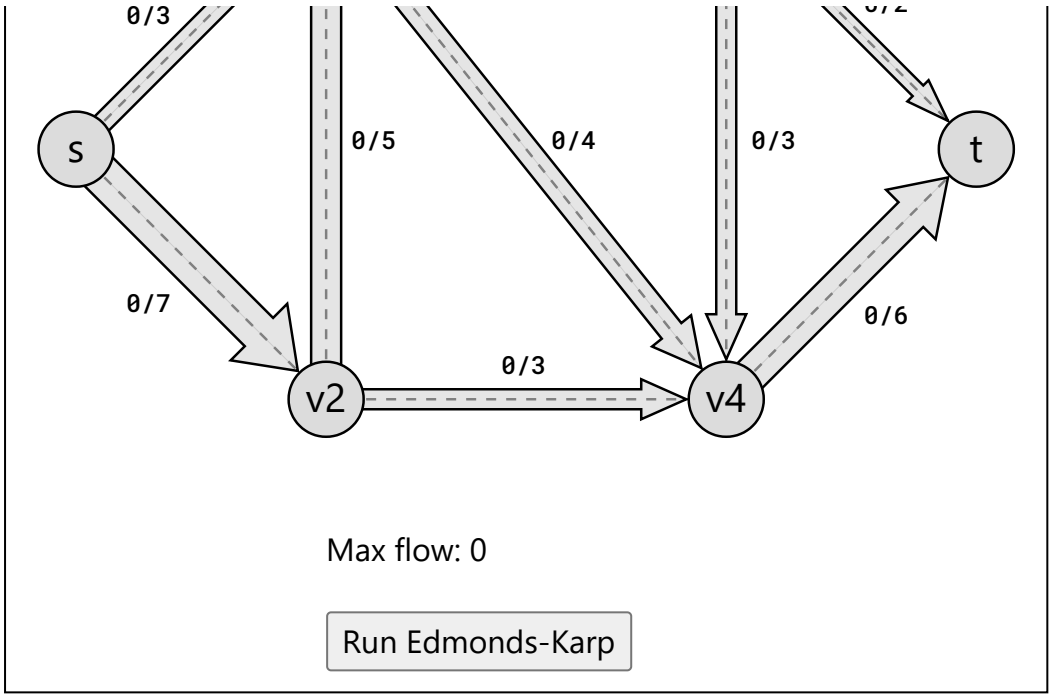
Finding the maximum flow can be helpful in many areas: for optimizing network traffic, for manufacturing, for supply chain and logistics, or for airline scheduling.

The Edmonds-Karp Algorithm

The Edmonds-Karp 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 Edmonds-Karp algorithm is very similar to the Ford-Fulkerson algorithm, except the Edmonds-Karp algorithm uses Breadth First Search (BFS) to find augmented paths to increase flow.



The Edmonds-Karp algorithm works by using Breadth-First Search (BFS) to find 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 Edmonds-Karp algorithm continues to find new paths to send more flow through until the maximum flow is reached.

In the simulation above, the Edmonds-Karp 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.

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

How it works:

1. Start with zero flow on all edges.
2. Use BFS to find an *augmented path* where more flow can be sent.

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 Edmonds-Karp

The Edmonds-Karp algorithm 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 Edmonds-Karp

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

To send flow back, in the opposite direction of the edge, a reverse edge is created for each original edge in the network. The Edmonds-Karp algorithm can then use these reverse edges to send flow in the reverse direction.

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 $v_1 \rightarrow v_3$ has a flow of 2, which means there is a residual capacity of 2 on the corresponding reversed edge $v_3 \rightarrow v_1$.

This just means that when there is a flow of 2 on the original edge $v_1 \rightarrow v_3$, 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.

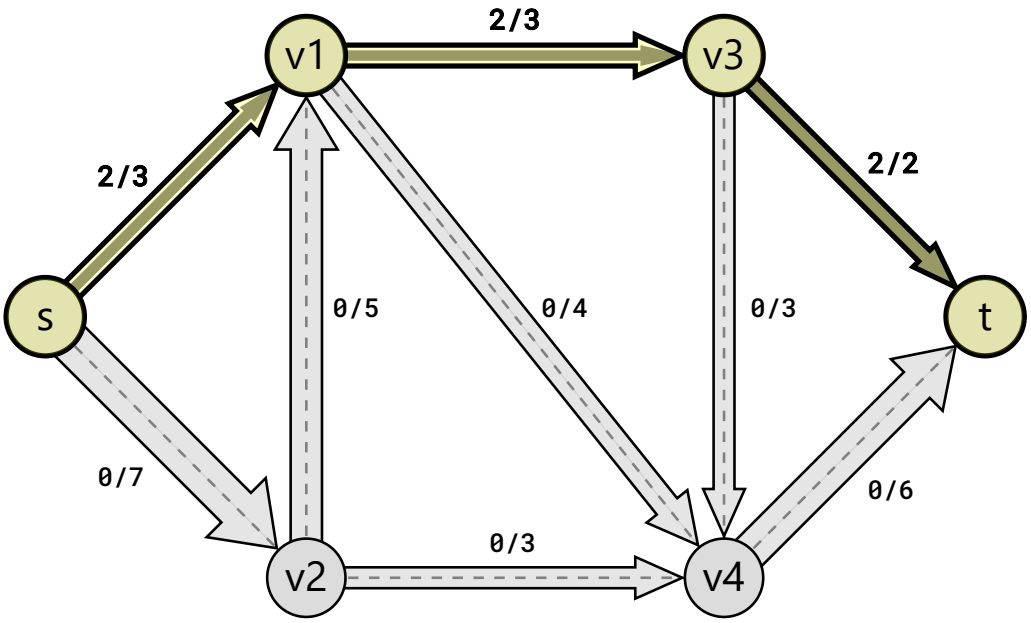
Manual Run Through

There is no flow in the graph to start with.

The Edmonds-Karp algorithm starts with using Breadth-First Search to find an augmented path where flow can be increased, which is $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$.

After finding the augmented path, a bottleneck calculation is done to find how much flow can be sent through that path, and that flow is: 2.

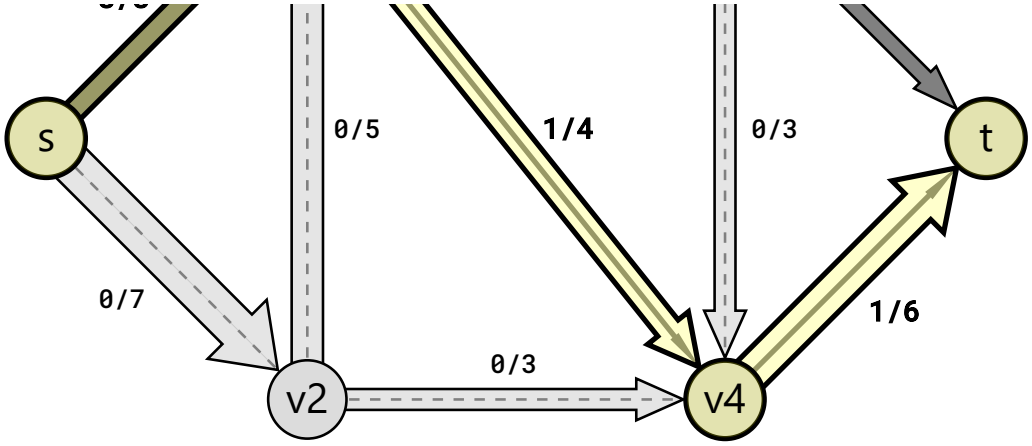
So a flow of 2 is sent over each edge in the augmented path.



The next iteration of the Edmonds-Karp algorithm is to do these steps again: Find a new augmented path, find how much the flow in that path can be increased, and increase the flow along the edges in that path accordingly.

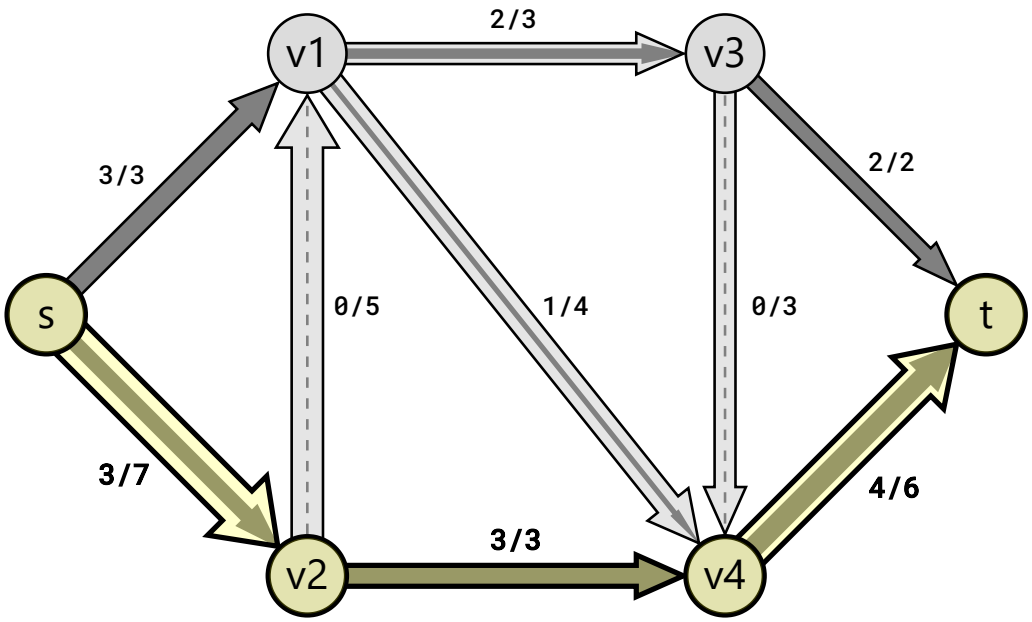
The next augmented path is found to be $s \rightarrow v_1 \rightarrow v_4 \rightarrow t$.

The flow can only be increased by 1 in this path because there is only room for one more unit of flow in the $s \rightarrow v_1$ edge.



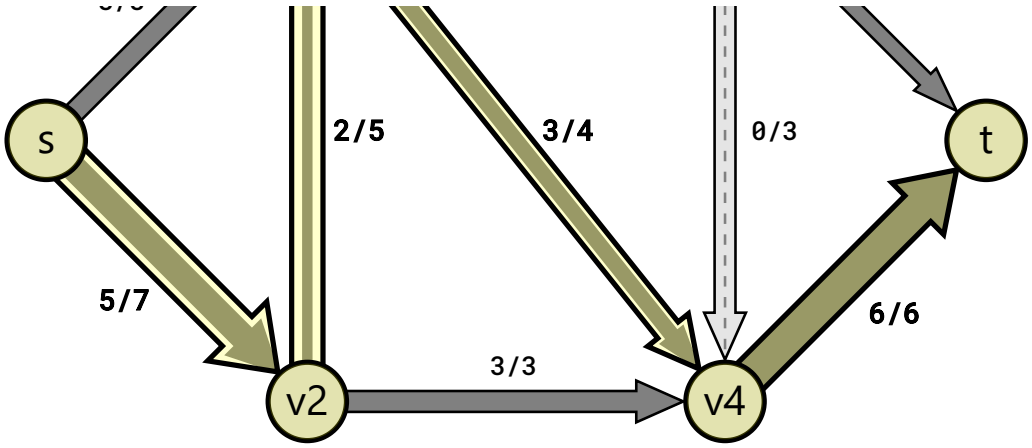
The next augmented path is found to be $s \rightarrow v_2 \rightarrow v_4 \rightarrow t$.

The flow can be increased by 3 in this path. The bottleneck (limiting edge) is $v_2 \rightarrow v_4$ because the capacity is 3.



The last augmented path found is $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$.

The flow can only be increased by 2 in this path because of edge $v_4 \rightarrow t$ being the bottleneck in this path with only space for 2 more units 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 Edmonds-Karp 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 Edmonds-Karp Algorithm

To implement the Edmonds-Karp 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 `bfs` method to find augmented paths, using Breadth-First-Search:

```
14      def bfs(self, s, t, parent):
19
22
28
```

Line 20-21: As long as there are vertices to be explored in the `queue`, take the first vertex out of the `queue` so that a path can be found from there to the next vertex.

Line 23: For every adjacent vertex to the current vertex.

Line 24-27: If the adjacent vertex is not visited yet, and there is a residual capacity on the edge to that vertex: add it to the queue of vertices that needs to be explored, mark it as visited, and set the `parent` of the adjacent vertex to be the current vertex `u`.

The `parent` array holds the parent of a vertex, creating a path from the sink vertex, backwards to the source vertex. The `parent` is used later in the Edmonds-Karp algorithm, outside the `bfs` method, to increase flow in the augmented path.

Line 29: The last line returns `visited[t]`, which is `true` if the augmented path ends in the sink node `t`. Returning `true` means that an augmenting path has been found.

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

```
        def edmonds_karp(self, source, sink):
            parent = [-1] * self.size
            max_flow = 0

            while self.bfs(source, sink, parent):
                path_flow = float("Inf")
                s = sink
                while(s != source):
                    path_flow = min(path_flow, self.adj_matrix[parent[s
                    s = parent[s]

                max_flow += path_flow
                v = sink
                while(v != source):
                    u = parent[v]
                    self.adj_matrix[u][v] -= path_flow
                    self.adj_matrix[v][u] += path_flow
                    v = parent[v]

                path = []
```

```
55
56                path.append(source)
57                path.reverse()
58                path_names = [self.vertex_data[node] for node in path]
59                print("Path:", " -> ".join(path_names), ", Flow:", path
60
            return max_flow
```

Initially, the `parent` array holds invalid index values, because there is no augmented path to begin with, and 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 35: The outer `while` loop makes sure the Edmonds-Karp algorithm keeps increasing flow as long as there are augmented paths to increase flow along.

Line 36-37: The initial flow along an augmented path is infinite, and the possible flow increase will be calculated starting with the sink vertex.

Line 38-40: The value for `path_flow` is found by going backwards from the sink vertex towards the source vertex. The lowest value of residual capacity along the path is what decides how much flow can be sent on the path.

Line 42: `path_flow` is increased by the `path_flow`.

Line 44-48: Stepping through the augmented path, going backwards from sink to source, the residual capacity is decreased with the `path_flow` on the forward edges, and the residual capacity is increased with the `path_flow` on the reversed edges.

Line 50-58: This part of the code is just for printing so that we are able to track each time an augmented path is found, and how much flow is sent through that path.

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

Example

Python:

```
        self.size = size
        self.vertex_data = [''] * size

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

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def bfs(self, s, t, parent):
        visited = [False] * self.size
```

Try it Yourself »

Time Complexity for The Edmonds-Karp Algorithm

The difference between Edmonds-Karp and Ford-Fulkerson is that Edmonds-Karp uses Breadth-First Search (BFS) to find augmented paths, while Ford-Fulkerson uses Depth-First Search (DFS).

This means that the time it takes to run Edmonds-Karp is easier to predict than Ford-Fulkerson, because Edmonds-Karp is not affected by the maximum flow value.

With the number of vertices V, the number of edges E, the time complexity for the Edmonds-Karp algorithm is

$$O(V \cdot E^2)$$

This means Edmonds-Karp does not depend on the maximum flow, like Ford-Fulkerson does, but on how many vertices and edges we have.

The reason we get this time complexity for Edmonds-Karp is that it runs BFS which has time complexity $O(E + V)$.

BFS must run one time for every augmented path, and there can actually be found close to $V \cdot E$ augmented paths during running of the Edmonds-Karp algorithm.

So, BFS with time complexity $O(E)$ can run close to $V \cdot E$ times in the worst case, which means we get a total time complexity for Edmonds-Karp: $O(V \cdot E \cdot E) = O(V \cdot E^2)$.

‹ Previous Sign in to track progress Next ›