Wikipedia
The Free Encyclopedia

Wikipedia

Ford–Fulkerson algorithm

The **Ford–Fulkerson method** or **Ford–Fulkerson algorithm** (**FFA**) is a greedy algorithm that computes the maximum flow in a flow network. It is sometimes called a "method" instead of an "algorithm" as the approach to finding augmenting paths in a residual graph is not fully specified[1] or it is specified in several implementations with different running times.[2] It was published in 1956 by L. R. Ford Jr. and D. R. Fulkerson.[3] The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a fully defined implementation of the Ford–Fulkerson method.

The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Algorithm

Let $G(V,E)$ be a graph, and for each edge from u to v, let $c(u,v)$ be the capacity and $f(u,v)$ be the flow. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

Capacity constraints	$\forall (u,v) \in E:\ f(u,v) \le c(u,v)$	The flow along an edge cannot exceed its capacity.
Skew symmetry	$\forall (u,v) \in E:\ f(u,v) = -f(v,u)$	The net flow from u to v must be the opposite of the net flow from v to u (see example).
Flow conservation	$\forall u \in V: u \neq s \text{ and } u \neq t \Rightarrow \sum_{w \in V} f(u,w) = 0$	The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.
Value(f)	$\sum_{(s,u) \in E} f(s,u) = \sum_{(v,t) \in E} f(v,t)$	The flow leaving from s must be equal to the flow arriving at t.

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V,E_f)$ to be the network with capacity $c_f(u,v) = c(u,v) - f(u,v)$ and no flow. Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u,v) > 0$ and $c(v,u) = 0$ then $c_f(v,u) = c(v,u) - f(v,u) = f(u,v) > 0$.

```
Algorithm Ford–Fulkerson

Inputs Given a Network $G = (V,E)$ with flow capacity $c$, a source node $s$, and a sink node $t$
Output Compute a flow $f$ from $s$ to $t$ of maximum value

  1. $f(u,v) \leftarrow 0$ for all edges $(u,v)$
  2. While there is a path $p$ from $s$ to $t$ in $G_f$, such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$:
      1. Find $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
      2. For each edge $(u,v) \in p$
          1. $f(u,v) \leftarrow f(u,v) + c_f(p)$ (Send flow along the path)
          2. $f(v,u) \leftarrow f(v,u) - c_f(p)$ (The flow might be "returned" later)
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

The path in step 2 can be found with, for example, a breadth-first search (BFS) or a depth-first search in $G_f(V,E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t, and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal. See also Max-flow Min-cut theorem.

If the graph $G(V,E)$ has multiple sources and sinks, we act as follows: Suppose that $T=\{t \mid t \text{ is a sink}\}$ and $S=\{s \mid s \text{ is a source}\}$. Add a new source s^* with an edge (s^*,s) from s^* to every node $s \in S$, with capacity $c(s^*,s)=d_s=\sum_{(s,u)\in E} c(s,u)$. And add a new sink t^* with an edge (t,t^*) from every node $t \in T$ to t^*, with capacity $c(t,t^*)=d_t=\sum_{(v,t)\in E} c(v,t)$. Then apply the Ford–Fulkerson algorithm.

Also, if a node u has capacity constraint d_u, we replace this node with two nodes $u_{\mathrm{in}}, u_{\mathrm{out}}$, and an edge $(u_{\mathrm{in}}, u_{\mathrm{out}})$, with capacity $c(u_{\mathrm{in}}, u_{\mathrm{out}})=d_u$. Then apply the Ford–Fulkerson algorithm.

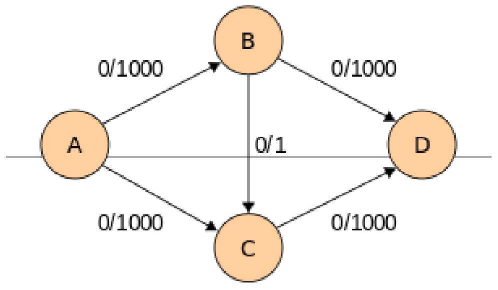
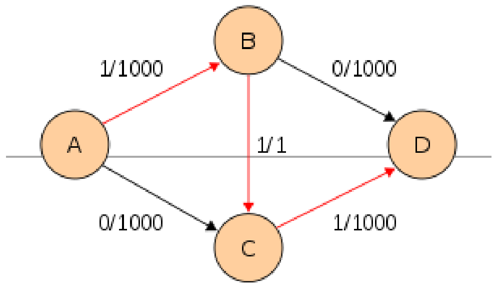
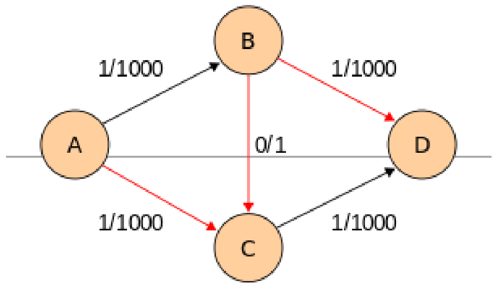
Complexity

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values.[4] When the capacities are integers, the runtime of Ford–Fulkerson is bounded by $O(Ef)$ (see big O notation), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in $O(E)$ time and increases the flow by an integer amount of at least 1, with the upper bound f.

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time.

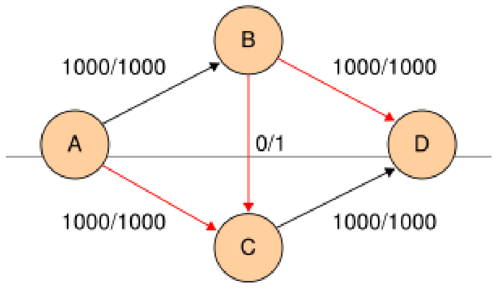
Integral example

The following example shows the first steps of Ford–Fulkerson in a flow network with 4 nodes, source A and sink D. This example shows the worst-case behaviour of the algorithm. In each step, only a flow of 1 is sent across the network. If breadth-first-search were used instead, only two steps would be needed.

Path	Capacity	Resulting flow network
	Initial flow network	 B A C D 0/1000 0/1000 0/1 0/1000 0/1000
A,B,C,D	$\begin{aligned} &\min(c_f(A,B),c_f(B,C),c_f(C,D)) \\ =\ &\min(c(A,B)-f(A,B),c(B,C)-f(B,C),c(C,D)-f(C,D)) \\ =\ &\min(1000-0,1-0,1000-0)=1 \end{aligned}$	 B A C D 1/1000 0/1000 1/1 0/1000 1/1000
A,C,B,D	$\begin{aligned} &\min(c_f(A,C),c_f(C,B),c_f(B,D)) \\ =\ &\min(c(A,C)-f(A,C),c(C,B)-f(C,B),c(B,D)-f(B,D)) \\ =\ &\min(1000-0,0-(-1),1000-0)=1 \end{aligned}$	 B A C D 1/1000 1/1000 0/1 1/1000 1/1000

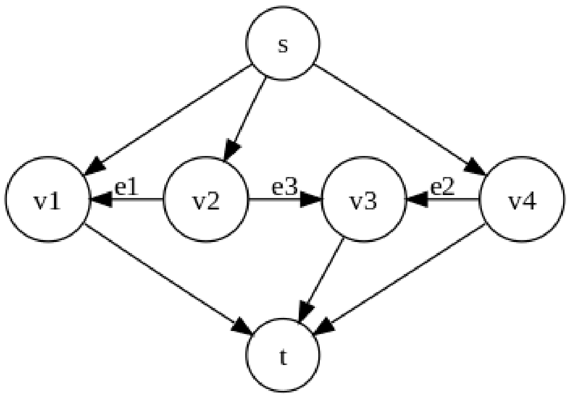
After 1998 more steps ...

Final flow network



Notice how flow is "pushed back" from C to B when finding the path A, C, B, D.

Non-terminating example



Consider the flow network shown on the right, with source s, sink t, capacities of edges e_1, e_2 and e_3 respectively 1, $r = (\sqrt{5} - 1)/2$ and 1 and the capacity of all other edges some integer $M \geq 2$. The constant r was chosen so, that $r^2 = 1 - r$. We use augmenting paths according to the following table, where $p_1 = \{s, v_4, v_3, v_2, v_1, t\}, p_2 = \{s, v_2, v_3, v_4, t\}$ and $p_3 = \{s, v_1, v_2, v_3, t\}$.

Step	Augmenting path	Sent flow	Residual capacities		
			e_1	e_2	e_3
0			$r^0 = 1$	r	1
1	$\{s, v_2, v_3, t\}$	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Note that after step 1 as well as after step 5, the residual capacities of edges e_1, e_2 and e_3 are in the form r^n, r^{n+1} and 0, respectively, for some $n \in \mathbb{N}$. This means that we can use augmenting paths p_1, p_2, p_1 and p_3 infinitely many times and residual capacities of these edges will always be in the same form. Total flow in the network after step 5 is $1 + 2(r^1 + r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1 + 2\sum_{i=1}^{\infty} r^i = 3 + 2r$. However, note that there is a flow of value $2M + 1$, by sending M units of flow along sv_1t, 1 unit of flow along sv_2v_3t, and M units of flow along sv_4t. Therefore, the algorithm never terminates and the flow does not even converge to the maximum flow.[5]

Another non-terminating example based on the Euclidean algorithm is given by Backman & Huynh (2018), where they also show that the worst case running-time of the Ford-Fulkerson algorithm on a network $G(V, E)$ in ordinal numbers is $\omega^{\Theta(|E|)}$.

Python implementation of Edmonds–Karp algorithm

```python
import collections


class Graph:
    """
    This class represents a directed graph using
    adjacency matrix representation.
    """

    def __init__(self, graph):
        self.graph = graph  # residual graph
        self.row = len(graph)

    def bfs(self, s, t, parent):
        """
        Returns true if there is a path from
        source 's' to sink 't' in residual graph.
        Also fills parent[] to store the path.
        """

        # Mark all the vertices as not visited
        visited = [False] * self.row

        # Create a queue for BFS
        queue = collections.deque()

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        # Standard BFS loop
        while queue:
            u = queue.popleft()

            # Get all adjacent vertices of the dequeued vertex u
            # If an adjacent has not been visited, then mark it
            # visited and enqueue it
            for ind, val in enumerate(self.graph[u]):
                if (visited[ind] == False) and (val > 0):
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        # If we reached sink in BFS starting from source, then return
        # true, else false
        return visited[t]

    # Returns the maximum flow from s to t in the given graph
    def edmonds_karp(self, source, sink):
        # This array is filled by BFS and to store path
        parent = [-1] * self.row

        max_flow = 0  # There is no flow initially

        # Augment the flow while there is path from source to sink
        while self.bfs(source, sink, parent):
            # Find minimum residual capacity of the edges along the
            # path filled by BFS. Or we can say find the maximum flow
            # through the path found.
            path_flow = float("Inf")
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Add path flow to overall flow
            max_flow += path_flow

            # update residual capacities of the edges and reverse edges
            # along the path
            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow
```

See also

- Berge's theorem
- Approximate max-flow min-cut theorem
- Turn restriction routing
- Dinic's algorithm

Notes

1. Laung-Terng Wang, Yao-Wen Chang, Kwang-Ting (Tim) Cheng (2009). *Electronic Design Automation: Synthesis, Verification, and Test* (https://archive.org/details/electronicdesign00wang). Morgan Kaufmann. pp. 204 (https://archive.org/details/electronicdesign00wang/page/n240). ISBN 978-0080922003.
2. Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein (2009). *Introduction to Algorithms* (https://archive.org/details/introductiontoal00corm_805). MIT Press. pp. 714 (https://archive.org/details/introductiontoal00corm_805/page/n734). ISBN 978-0262258104.
3. Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network" (http://www.cs.yale.edu/homes/lans/readings/routing/ford-max_flow-1956.pdf) (PDF). *Canadian Journal of Mathematics*. **8**: 399–404. doi:10.4153/CJM-1956-045-5 (https://doi.org/10.4153%2FCJM-1956-045-5). S2CID 16109790 (https://api.semanticscholar.org/CorpusID:16109790).
4. "Ford-Fulkerson Max Flow Labeling Algorithm". 1998. CiteSeerX 10.1.1.295.9049 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.295.9049).
5. Zwick, Uri (21 August 1995). "The smallest networks on which the Ford–Fulkerson maximum flow procedure may fail to terminate" (https://doi.org/10.1016%2F0304-3975%2895%2900022-O). *Theoretical Computer Science*. **148** (1): 165–170. doi:10.1016/0304-3975(95)00022-O (https://doi.org/10.1016%2F0304-3975%2895%2900022-O).

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 26.2: The Ford–Fulkerson method". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 651–664. ISBN 0-262-03293-7.
- George T. Heineman; Gary Pollice; Stanley Selkow (2008). "Chapter 8:Network Flow Algorithms". *Algorithms in a Nutshell*. Oreilly Media. pp. 226–250. ISBN 978-0-596-51624-6.

- Jon Kleinberg; Éva Tardos (2006). "Chapter 7:Extensions to the Maximum-Flow Problem" (https://archive.org/details/algorithmdesign0000klei/page/378). *Algorithm Design*. Pearson Education. pp. 378–384 (https://archive.org/details/algorithmdesign0000klei/page/378). ISBN 0-321-29535-8.
- Samuel Gutekunst (2019). *ENGRI 1101*. Cornell University.
- Backman, Spencer; Huynh, Tony (2018). "Transfinite Ford–Fulkerson on a finite network". *Computability*. **7** (4): 341–347. arXiv:1504.04363 (https://arxiv.org/abs/1504.04363). doi:10.3233/COM-180082 (https://doi.org/10.3233%2FCOM-180082). S2CID 15497138 (https://api.semanticscholar.org/CorpusID:15497138).

External links

- A tutorial explaining the Ford–Fulkerson method to solve the max-flow problem (http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=maxFlow)
- Another Java animation (http://www.cs.pitt.edu/~kirk/cs1501/animations/Network.html)
- Java Web Start application (http://rrusin.blogspot.com/2011/03/implementing-graph-editor-in-javafx.html)

Media related to Ford-Fulkerson's algorithm at Wikimedia Commons

-