

Edmonds–Karp algorithm

From Wikipedia, the free encyclopedia

In **computer science**, the **Edmonds–Karp algorithm** is an implementation of the **Ford–Fulkerson method** for computing the **maximum flow** in a **flow network** in *O*(*V* *E*²) time. The algorithm was first published by Yefim (Chaim) Dinic in 1970^[1] and independently published by Jack Edmonds and Richard Karp in 1972.^[2] Dinic's **algorithm** includes additional techniques that reduce the running time to *O*(*V*²*E*).

Contents [hide]

- 1

Algorithm
- 2

Pseudocode
- 3

Example
- 4

Notes
- 5


References

Algorithm [edit]

The algorithm is identical to the **Ford–Fulkerson algorithm**, except that the search order when finding the **augmenting path** is defined. The path found must be a shortest path that has available capacity. This can be found by a **breadth-first search**, as we let edges have unit length. The running time of *O*(*V* *E*²) is found by showing that each augmenting path can be found in *O*(*E*) time, that every time at least one of the *E* edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most *V*. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically. There is an accessible proof in *Introduction to Algorithms*.^[3]

Pseudocode [edit]

For a more high level description, see *Ford–Fulkerson algorithm*.



The Wikibook *Algorithm implementation* has a page on the topic of: **Edmonds–Karp**

```
algorithm EdmondsKarp
input:
    C[1..n, 1..n] (Capacity matrix)
    E[1..n, 1..?] (Neighbour lists)
    s (Source)
    t (Sink)
output:
    f (Value of maximum flow)
    F (A matrix giving a legal flow with the maximum value)
f := 0 (Initial flow is zero)
F := array(1..n, 1..n) (Residual capacity from u to v is C[u,v] - F[u,v])
forever
    m, P := BreadthFirstSearch(C, E, s, t, F)
    if m = 0
        break
    f := f + m
    (Backtrack search, and write flow)
    v := t
    while v ≠ s
        u := P[v]
        F[u,v] := F[u,v] + m
        F[v,u] := F[v,u] - m
        v := u
return (f, F)
```

```
algorithm BreadthFirstSearch
input:
    C, E, s, t, F
output:
    M[t] (Capacity of path found)
    P (Parent table)
P := array(1..n)
for u in 1..n
    P[u] := -1
P[s] := -2 (make sure source is not rediscovered)
M := array(1..n) (Capacity of found path to node)
M[s] := ∞
Q := queue()
Q.offer(s)
while Q.size() > 0
    u := Q.poll()
    for v in E[u]
        (If there is available capacity, and v is not seen before in search)
```

```

    if C[u,v] - F[u,v] > 0 and P[v] = -1
        P[v] := u
        M[v] := min(M[u], C[u,v] - F[u,v])
        if v ≠ t
            Q.offer(v)
        else
            return M[t], P
return 0, P

```

EdmondsKarp pseudo code using Adjacency nodes.

```

algorithm EdmondsKarp
  input:
    graph  (graph[v] should be the list of edges coming out of vertex v.
            Each edge should have a capacity, flow, source and sink as parameters,
            as well as a pointer to the reverse edge.)
    s      (Source vertex)
    t      (Sink vertex)
  output:
    flow   (Value of maximum flow)

  flow := 0  (Initial flow to zero)
  repeat
    (Run a bfs to find the shortest s-t path.
     We use 'pred' to store the edge taken to get to each vertex,
     so we can recover the path afterwards)
    q := queue()
    q.push(s)
    pred := array(graph.length)
    while not empty(q)
      cur := q.poll()
      for Edge e in graph[cur]
        if pred[e.t] = null and e.t ≠ s and e.cap > e.flow
          pred[e.t] := e
          q.push(e.t)

    (Stop if we weren't able to find a path from s to t)
    if pred[t] = null
      break

    (Otherwise see how much flow we can send)
    df := ∞
    for (e := pred[t]; e ≠ null; e := pred[e.s])
      df := min(df, e.cap - e.flow)

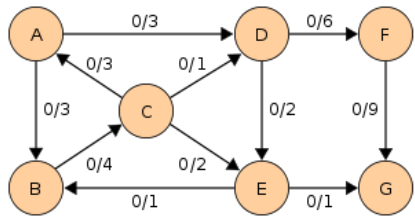
    (And update edges by that amount)
    for (e := pred[t]; e ≠ null; e := pred[e.s])
      e.flow := e.flow + df
      e.rev.flow := e.rev.flow - df

    flow := flow + df
  return flow

```

Example [\[edit\]](#)

Given a network of seven nodes, source A, sink G, and capacities as shown below:



In the pairs f/c written on the edges, f is the current flow, and c is the capacity. The residual capacity from u to v is $c_f(u, v) = c(u, v) - f(u, v)$, the total capacity, minus the flow that is already used. If the net flow from u to v is negative, it contributes to the residual capacity.

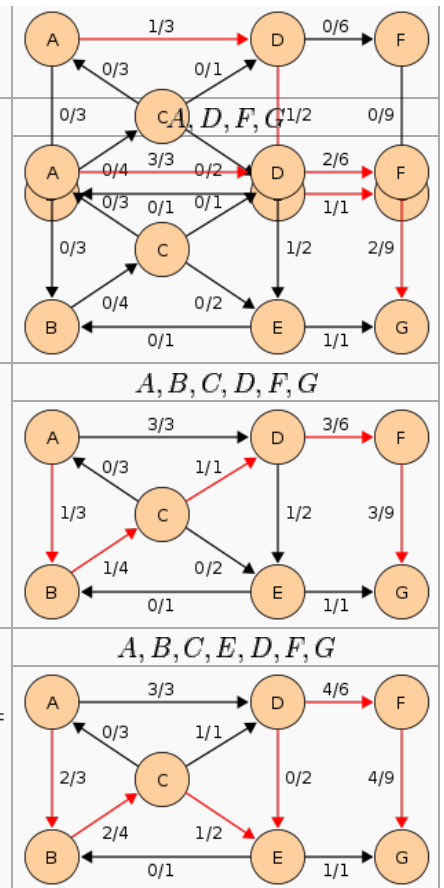
| Capacity | Path |
|---|-------------------|
| | Resulting network |
| $\min(c_f(A, D), c_f(D, E), c_f(E, G)) =$ $\min(3 - 0, 2 - 0, 1 - 0) =$ | A, D, E, G |
| | |

$$\min(3, 2, 1) = 1$$

$$\begin{aligned} \min(c_f(A, D), c_f(D, F), c_f(F, G)) &= \\ \min(3 - 1, 6 - 0, 9 - 0) &= \\ \min(2, 6, 9) &= 2 \end{aligned}$$

$$\begin{aligned} \min(c_f(A, B), c_f(B, C), c_f(C, D), c_f(D, F), c_f(F, G)) &= \\ \min(3 - 0, 4 - 0, 1 - 0, 6 - 2, 9 - 2) &= \\ \min(3, 4, 1, 4, 7) &= 1 \end{aligned}$$

$$\begin{aligned} \min(c_f(A, B), c_f(B, C), c_f(C, E), c_f(E, D), c_f(D, F), c_f(F, G)) &= \\ \min(3 - 1, 4 - 1, 2 - 0, 0 - (-1), 6 - 3, 9 - 3) &= \\ \min(2, 3, 2, 1, 3, 6) &= 1 \end{aligned}$$



Notice how the length of the [augmenting path](#) found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the [minimum cut](#) in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets $\{A, B, C, E\}$ and $\{D, F, G\}$, with the capacity

$$c(A, D) + c(C, D) + c(E, G) = 3 + 1 + 1 = 5.$$

Notes [\[edit\]](#)

- [▲] Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". *Soviet Math. Doklady* (Doklady) **11**: 1277–1280.
- [▲] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM* (Association for Computing Machinery) **19** (2): 248–264. doi:10.1145/321694.321699 [↗](#).
- [▲] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2009). "26.2". *Introduction to Algorithms* (third ed.). MIT Press. pp. 727–730. ISBN 978-0-262-03384-8.

References [\[edit\]](#)

1. Algorithms and Complexity (see pages 63–69). <http://www.cis.upenn.edu/~wlf/AlgComp3.html> [↗](#)

Categories: [Network flow](#) | [Graph algorithms](#)

This page was last modified on 7 May 2015, at 19:03.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#).
Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

