UNIVERSITY OF EXETER

The Edmonds-Karp Algorithm

Abstract

The Edmonds-Karp algorithm is explained and its complexity analysed in this report. We look at how the algorithm was arrived at by detailing flow networks, the maximal flow problem, augmenting paths, residual graphs and the Ford-Fulkerson method on which it is based. The time complexity is looked at in detail. We then move onto looking at limitations and improvements for this algorithm, before seeing how it can be applied to real world problems.

670026071

1 THE MAIN PRINCIPLES OF THE EDMOND-KARP ALGORITHM

The Edmonds-Karp algorithm [1] solves the maximum flow problem. This is where you want to arrange traffic in a network to make the most efficient use of the available capacity [2] [3]. This can represented with a type of directed graph called a $flow\ network$. Material is produced at a source node, and it travels through the network to a sink, where it is consumed.

We can denote this network as a directed graph $G = (V,E)$ [3] [2] with the following features:

1. There is a single source node s belonging to V which generates traffic.
2. There is a single sink node t belonging to V which *absorbs* traffic as it arrives.
3. Each edge e transmits traffic within its capacity, which is a non-negative number we denote c_e.

A *cut* is a partitioning of the network G, into two disjoint sets of vertices, S and T. S includes the source and T includes the sink. The source and sink cannot be in the same set. The $cut-set$ is the set of edges that start in S and end in T. The capacity is the sum of the weights of the edges in the $cut-set$. The goal of a $max-flow\ min\ cut$ algorithm is to find the *cut* with the least capacity.

The Ford-Fulkerson method [4] is a general approach to solving this problem. The method works by sending flow along a path with available capacity from the source to the sink. These are known as *augmenting paths*. This is repeated until there are no more *augmenting paths*. This is maintained with the following. We start with $f(u,v) = 0$ for all u,v belonging to V, setting an initial flow value of 0. For each iteration, the flow value is increased in G by finding an *augmentingpath* in an associated *residual graph* G_f. Given a flow network G, and a flow f on G, we define the residual graph G_f of G with respect to f as follows:

1. The node set of G_f is the same as that of G
2. For each edge $e = (u,v)$ of G on which $f(e) < c_e$, there are $c_e - f(e)\ leftover$ units of capacity on which flow can be pushed forward hence the edge $e = (u,v)$ in G_f, with a capacity of $c_e - f(e)$ is included. These are forward edges.
3. For each edge $e = (u,v)$ of G on which $f(e) > 0$, there are $f(e)$ units of flow that we can undo if needed, by pushing flow backward, thus we include the edge $e = (v,u)$ in G_f, with a capacity of $f(e)$. Here the ends of e are the same, but its direction is reversed, thus edges included in this way are backward edges.

Once the edges of an augmenting path in G_f are known, we can easily identify specific edges in G for which we can increase the value of the flow. Each iteration of the method increases the value of the flow, but the flow on any particular edge of G may increase or decrease. In order to send more flow from the source to the sink, it might be necessary to decrease the flow on some edges. The flow is repeatedly augmented until our residual network has no more augmenting paths. The max-flow min-cut theorem shows that upon termination, this process gives the maximum flow [2] [5].

The Edmonds-Karp algorithm [1] is an implementation of the above Fords-Fulkerson method, except the augmented path is chosen as the shortest path from the source to the sink in the residual network using breadth-first search (BFS). Using BFS means the path structure contains a queue of vertices during its search. The potential augmenting path is expanded by removing a vertex u from the head of the queue, and by appending adjacent unvisited vertices, through which the augmented path may exist. Either the sink vertex t will be visited, or the path becomes empty which happens when no augmenting path is possible [5]. The steps for this process are laid out in Algorithm 1.

2 THE COMPLEXITY OF THE EDMOND-KARP ALGORITHM

The Edmonds-Karp algorithm makes some significant improvements over the Ford-Fulkerson algorithm to achieve a runtime of $O(|V|.|E^2|)$. This is because Edmonds-Karp removes the dependency on maximum flow for complexity, making it much better for graphs that have a larger maximum flow. The running time is based on the capacities of the edges, not on the problem size (i.e., the number of vertices or edges) [5].

BFS finds the shortest augmented path in $O(E)$ time. The number of vertices is smaller than the number of edges in the connected flow graph as every vertex has at least one incident edge. This improves on the standard time $O(E+V)$. Once a path P is discovered, it takes $O(E)$ time to augment f using P and $O(E)$ time to update G_f therefore one iteration of the while loop in Edmonds-Karp requires $O(E)$ time. To find the maximum number of iterations of the while loop, we examine the BFS tree. Every new edge created when augmenting f using P is the reverse of an edge that belongs to P. Every edge of P goes from level i to $i+1$, therefore every new edge created in G_f after the augmentation must go from level $i+1$ to level i. This

Algorithm 1 Edmond-Karp

Require: *graph, s, t*

```
1:  flow ← 0
2:  repeat
3:      q ← queue()
4:      q.push(s)
5:      pred ← array(graph.length)
6:      while not empty(q) do
7:          q ← q.push(s)
8:          for Edge e in graph[cur] do
9:              if pred[e.t] = null and e.t ≠ s and e.cap > e.flow then
10:                 pred[e.t] ← e
11:                 q.push(e.t)
12:     if not pred[t] = null then
13:         df ← ∞
14:         for e ← pred[t]; e ≠ null; e ← pred[e.s] do
15:             df ← min(df, e.cap − e.flow)
16:         for e ← pred[t]; e ≠ null; e ← pred[e.s] do
17:             e.flow ← e.flow + df
18:             e.rev.flow ← e.rev.flow − df
19:         flow ← flow + df
20: until pred[t] = null
```

means, for any vertex v, the distance from s to v never decreases as we run the algorithm. If an edge is found to be a bottleneck it is discarded from G_f after augmenting f using P. Suppose that u belongs to L_i and v belongs to L_i when this happens. In order for e to be added back into G_f later on, u must occupy a higher numbered level than v. As the distance from s to v never decreases, this means that v remains in level $L_(i+1)$ or higher, and u must rise to level $L_(i+2)$ or higher, before e is added back into G_f. The BFS tree has no levels numbered above V thus the total number of times that e can occur as a bottleneck edge during the algorithm is at most $V/2$. There are $2E$ edges that can potentially appear in the residual graph, and each of them serves as a bottleneck edge at most $E/2$ times so there are at most EV bottleneck edges. Each run of the while loop identifies an augmenting path which contains a bottleneck edge, therefore there are at most EV while loop iterations. As each iteration takes $O(E)$ time the overall running time of the algorithm is $O(|V|.|E^2|)$. There is a straight forward linear space complexity of $O(E+V)$ as only the graph needs storing.

3 THE LIMITATIONS AND CONSTRAINTS OF THE EDMONDS-KARP ALGORITHM

The main limitation of the Edmonds-Karp implementation is its high computational complexity. This has seen to be easily reduced to $O(EV^2)$ by Dinic's algorithm [6]. In this approach, BFS is used to check if flow can be increased. A level graph is constructed where the fewest number of edges from the source to the node, is its level. Multiple flows can then be sent using this level graph, which is better than Edmond-Karp, as that only sends flow across the path found by the BFS [7]. Some limitations might be encountered when applying the Edmonds-Karp to real world problems. Firstly, vertex capacities are where the flow network places a maximum capacity $k(v)$ flowing through a vertex v in the graph. This can be overcome by constructing a modified flow network G_m as follows. For each vertex v in G, create two vertices v^a and v^b. Create edge (v^a, v^b) with a flow capacity of $k(v)$. For each incoming edge (u,v) in G with capacity $c(u,v)$, create a new edge (u,v^a) with capacity $c(u,v)$. For each outgoing edge (v,w) in G, create edge (v^b,w) in G_m with capacity $k(v)$. A solution in G_m determines the solution to G. Secondly, the issue of undirected edges can be solved by also constructing a modified flow network G_m with the same set of vertices. For each edge (u,v) in G with capacity $c(u,v)$, construct a pair of edges (u,v) and (v,u) each with the same capacity $c(u,v)$. A solution in G_m determines the solution to G [5].

4 Applications of the Edmonds-Karp algorithm

4.1 Video Surveillance

Leading countries employ video surveillance along road networks as a means of managing traffic and incidents. It is important to decide upon how many sensors are needed, and where they should be placed. The road network can be modelled as a flow graph with the minimum cut set giving the optimal number of cameras to be placed. An artificial super source and super sink node will need to be added to the graph. The road segments are the edges, and junction points between them are vertices of the graph. All the edge capacities are set to one. It was found that when tested on a real road network with approximately 1.2 million edges and 10,000 vertices the algorithm found a min-cut set containing 54 edges, thus showing its effectiveness [8].

4.2 Tanker Scheduling

The Tanker scheduling problem is where perishable goods need to be delivered between several locations. There are exact delivery dates for when items should arrive. The aim is to find the minimum number of ships needed to meet the delivery dates of the shiploads. The network will contain a node for each shipment, and an arc from node i to node j if it is possible to deliver the shipment j after shipment i. A directed path corresponds to a feasible sequence of shipment pickups and deliveries. A source node s is added and connected to the origin of each shipment, and we add a sink node t and connect each destination node to it. All the edge capacities are set to one. We first establish a feasible flow in the network by solving a maximum flow problem. We then send flow from node t to node s until no more flow can be sent. The solution at this point is an optimal solution of the minimum flow problem [9].

REFERENCES

[1] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972. [Online]. Available: http://doi.acm.org/10.1145/321694.321699

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[3] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[4] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399âĂŞ–404, 1956.

[5] G. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2008.

[6] E. A. DINIC, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Doklady*, vol. 11, pp. 1277–1280, 1970.

[7] Y. Dinitz, *Dinitz' Algorithm: The Original Version and Even's Version*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 218–240.

[8] V. Ramesh, S. Nagarajan, J. J. Jung, and S. Mukherjee, "Max-flow min-cut algorithm with application to road networks," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 11, p. e4099, 2017, e4099 cpe.4099.

[9] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and M. Reddy, "Chapter 1 applications of network optimization," in *Network Models*, ser. Handbooks in Operations Research and Management Science. Elsevier, 1995, vol. 7, pp. 1 – 83.