lem-in

Alexander Mayorov

I. Introduction

This document aims to document my work on a student project at School 42. For official documentation of the code please see pdf.

In short, the goal of the project is to move a given number of ants from a start node to an end node through an node-disjoint undirected network with all capacities equal to 1 using the least number of steps. A step is defined as a simultaneous movement of ants between nodes. No collisions between ants can ever occur.

The document is structured as follows. I first search for an effective solution to a simpler problem of edge-disjoint minimum cost flow in Sections II. and III.. I then adapt the algorithm in Section IV. for the problem node-disjoint minimum cost flow that is more closely linked to my actual problem. I then put everything together in Sections VI. and V..

II. Edge-disjoint minimum cost flow (basic)

Let's start with the problem of edge-disjoint minimum cost flow.

Definition 1. Flow constraints are constraints stipulating that at every node other than `start` an `end`, the amount of incoming flow equals the amount of outgoing flow.

Note that in our particular case, any path from `start` to `end` or a closed path, such as cycle, automatically satisfies the flow constraints.

Definition 2. A residual network G_r is formed from a network G by replacing all edges along paths involved in a flow f with edges directed in the opposite direction to the flow, and negating their cost.[2]

Observation 2.1. A cycle in the residual graph G_r doesn't change the value of the flow in the graph G. Moreover, the cost of the flow changes in relation to the cost of the cycle. [3]

Observation 2.2. A given flow f in the network has minimum cost in the network G if there exist no negative cost cycles in the residual network G_r.[3]

Observation 2.3. The shortest path from `start` to `end` in the graph G doesn't introduce any negative cycles in the corresponding residual graph G_r.

This is true, because in the residual graph the negative cost of going from `end` to `start` along the shortest path is less than the positive cost of going from `start` to `end` along any of the remaining paths.

Observation 2.4. A minimum cost flow of any given amount (less than the overall max-flow of the network) can be obtained iteratively searching for the shortest path and replacing the network with its residual.

Observation 2.5. Time complexity for getting a minimum cost flow of a value n is $\mathcal{O}(nVE)$, where V is the number of vertices in the graph and E is the number of edges.

Due to the presence of negative cost edges in the residual network, we cannot use the more efficient shortest path search algorithms, such a Dijkstra's algorithm. Therefore, we must run Bellman-Ford algorithm, whose complexity is $\mathcal{O}(VE)$, of which there are n.

III. Edge-disjoint minimum cost flow (efficient)

The time complexity of the algorithm from Section II. is prohibitive when then the number of nodes exceeds ≈ 1000. Let's see if we can do better.

Definition 3. A price function is a function that assigns a numerical value to every node.

Using it, we can replace the original network G with a corresponding network G', in which the *effective* cost of an edge $c^*_{v\to w}$ is equal to

$$c^*_{v\to w} = c_{v\to w} + p(w) - p(v)\,,$$

where $c_{v\to w}$ is the original cost and $p(x)$ is the price of the node x.

Observation 3.1. Introducing a price function doesn't affect the presence or absence of negative cost cycles in a network since the price terms around the cycle cancel out.[3]

Observation 3.2. If we use the distance to each node from `start` as its price, we can get rid of negative cost terms in the residual graph. [3, 4]

This is true, because, along the shortest path in the graph, the effective cost of an edge $v \to w$ is 0. Since we're using the shortest path to augment the flow, the reversed edges in the residual graph also have an effective cost of 0.

Observation 3.3. Finding a minimum cost flow of a value n using distance as a

price function has a time complexity $\mathcal{O}(nE \log V)$, since in the absence of negative cost edges, we can use Dijkstra's algorithm to get the shortest path.

IV. Node-disjoint minimum cost flow

The algorithm described above finds edge-disjoint paths yielding minimum cost flow of a given value n. We could find modify it to get node-disjoint paths with a trick.

Observation 4.1. We can split each node along the shortest path, excluding `start` and `end`, into (1) an *IN* node, to which edges point to, and (2) an *OUT* node, from which edges originate. If we do so, we can enforce that as soon as the search branches onto one of these nodes, the only way out is via one of the inverse edges created in the residual graph. [1, 4]

Please note that, as well as "splitting" nodes that lie on paths, we should "unsplit" a node, once there is no path in the flow that passes through it.

V. Putting it all together

The lem-in problem largely reduces to node-disjoint minimum cost flow. However, the two problems are not equivalent.

First, the number of ants is fixed. Therefore, the maximum flow in the graph may not give us the best possible solution. For example, in the case of a single ant, a single path of length 5 (flow = 1) is better than a hundred paths (flow = 100) of which the shortest has length of 80. Therefore, we must consider all possible flows less than or equal to the number of ants.

Is this really optimal?

Second, we need to optimally assign our ants to the paths in the network, such as the number of steps from `start` to `end` is minimal. How this is done is discussed in Section VI..

VI. Assigning ants to paths

Observation 6.1. If there are n ants travelling along a single path of length a, then it would take them $a + n - 1$ steps to reach the *end*.

Because all nodes have a capacity of 1, after k steps only k ants may have left `start` and entered the path. The first ant reaches the `end` at the ath step, whereas the last one does so at the $a + n - 1$th step.

Observation 6.2. If we have two paths of length a and b, respectively, and n ants,

then, in order for all ants to get to the `end` in the least number of steps, we should assign $a - b$ more ants to the first path than to the second.

As a result $\frac{n+(a-b)}{2}$ ants should follow the first path and at most $\frac{n-(a-b)}{2}$ the second. Furthermore, it will take at most $a + \frac{n+(a-b)}{2} - 1$ to reach the `end`.

This observation also holds in general. If we have M paths, of which the longest has length z, and n ants, then a jth path of length a_j should be followed by

$$n_j = z - a_j + \left[n - \sum_{i=1}^{M}(z - a_i)\right]/M\,,$$

ants, and the number of steps to each the `end` is

$$\begin{aligned} N_{\text{steps}} &= a_j + z - a_j + \left[n - \sum_{i=1}^{M}(z - a_i)\right]/M - 1 \\ &= z - 1 + \left[n - \sum_{i=1}^{M}(z - a_i)\right]/M\,. \end{aligned}$$

Observation 6.3. If we consider paths in order of increasing length, we can stop including more paths once the length of the next path is greater than the currently computed number of steps.

Note that this does **not** suggest that we can stop increasing flow in the network as soon as the above condition is satisfied, because the paths may change as the flow changes. All possible flow values must be considered, as is stipulated in Section V.It only means that, for a **given** flow, we can shortcut when calculating the number of steps and the optimal assignment.

References

[1] http://www.macfreek.nl/memory/Disjoint_Path_Finding

[2] https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

[3] https://courses.csail.mit.edu/6.854/06/scribe/s12-minCostFlowAlg.pdf

[4] https://en.wikipedia.org/wiki/Suurballe%27s_algorithm