Algorithms. Lecture Notes 12

Shortest Paths Again

We have already solved the Single Source Shortest Path problem in the case when all edge lengths are equal, as a byproduct of BFS. For the case of general positive edge lengths we need to extend the idea of BFS appropriately.

Let s denote the source node. First of all, it is clear that $d(s, s) = 0$. Now let $x \neq s$ be a node closest to s, that is, $l(s, x)$ is minimal. Then we have $d(s, x) = l(s, x)$, since no path from s to x other than the single edge (s, x) can be shorter. In an attempt to generalize this observation, we may want to look for an algorithm that determines the kth closest node, for $k = 1, 2, 3, \ldots$. The first two steps were simple. The key question is: Can we efficiently obtain the kth closest node z, provided that we already know the set S of the $k-1$ closest nodes, along with their correct distances $d(s, x)$ for all $x \in S$?

To analyze the situation, consider a shortest path P from s to z. The crucial point is that all nodes on P except z are already in S. If this were not true, some node $y \neq z$ on P would be outside S, but the subpath of P from s to y would be a shortest path from s to y, hence $d(s, y) < d(s, z)$, contradicting the choice of z as the kth closest node.

Hence, in order to identify z, it suffices to extend a shortest path from s to some $x \in S$ by one more edge. But how do we find the correct x and z? We may simply try all candidate pairs: We compute the minimum of all $d(s, x) + l(x, z)$, where $x \in S$, $z \notin S$. Then we move to S a node z that minimizes this sum. At this moment we also know that $d(s, z) = d(s, x) + l(x, z)$, and we have extended S correctly. The resulting algorithm is named **Dijkstra's algorithm**.

A naive implementation would compute, in each of the $n - 1$ steps, all these sums from scratch. This would cost $O(nm)$ time. It is better to exploit monotonicity properties: Dijkstra's algorithm computes iteratively better and better paths from s to all other nodes. For $x \in S$, the computed distances $d(s, x)$ are already correct, as the above inductive argument has

shown. The implementation trick is now to maintain preliminary distances $d(s, z)$ also to all nodes $z \notin S$. They may be larger than the correct distances. We define $d(s, z)$ to be the length of a shortest path consisting of a path inside S, plus one last edge to z. Whenever some $z \notin S$ is moved to S, we must update all preliminary distances $d(s, y)$, $y \notin S$, since we might get a shorter path from s to y, with z as the second last node.

To avoid confusion it is wise to write down the whole algorithm again: Initially we set $S := \emptyset$, $d(s, s) := 0$ and $d(s, y) := \infty$ (or a huge constant) for all $y \neq s$. Then, as long as $S \neq V$, we put a node $z \notin S$ with smallest $d(s, z)$ in S (knowing that this is the correct distance), and for all $y \notin S$ we update $d(s, y)$ as follows: $d(s, y) := \min(d(s, y), d(s, z) + l(z, y))$. Since these are less than n update operations, the time complexity is $O(n^2)$.

Dijkstra's algorithm chooses the next node according to a greedy rule, but it also exhibits elements of dynamic programming. We see that these techniques should not be understood as a rigid classification of efficient algorithms, rather, they are just general design principles, and many algorithms combine several techniques.

For sparse graphs we can further improve the time for Dijkstra's algorithm to $O(m \log n)$. Observe that, in each iteration, only the smallest $d(s, z)$, $z \notin S$, is needed. A weakness of the above implementation is that we still search for a minimum in each iteration. By storing all the preliminary distances in a priority queue we can obtain each minimum in $O(\log n)$ time. Now each update operation needs $O(\log n)$ time, too, instead of $O(1)$. (The old value must be removed and the new value inserted in the data structure.) But since every edge (z, y) is involved in only update operation, the claimed time bound follows.

Now we have computed the distances $d(s, y)$. In order to get some shortest path from s to any y, we may proceed as in every dynamic programming algorithm and trace the minimum computations backwards.

We remark without details that the Shortest Paths problem becomes much easier in DAGs. There one can take advantage of a topological order and give a pretty straightforward dynamic programming algorithm that runs in $O(m)$ time.

Network Flow Algorithms

Let $G = (V, E)$ be a directed graph where every edge e has an integer capacity $c_e > 0$. Two special nodes $s, t \in V$ are called **source and sink**, all other nodes are called internal. We suppose that no edge enters s or leaves t.

A **flow** is a function f on the edges such that $0 \leq f(e) \leq c_e$ holds for all edges e (capacity constraints), and $f^+(v) = f^-(v)$ holds for all internal nodes v (conservation constraints), where we define $f^-(v) := \sum_{e=(u,v)\in E} f(e)$ and $f^+(v) := \sum_{e=(v,u)\in E} f(e)$. (As a menominic aid: $f^-(v)$ is consumed by node v, and $f^+(v)$ is generated by node v.) The value of the flow f is defined as $val(f) := f^+(s)$. The Maximum Flow problem is to compute a flow with maximum value.

For any flow f in G (not necessarily maximum), we define the **residual graph** G_f as follows. G_f has the same nodes as G. For every edge e of G with $f(e) < c_e$, G_f has the same edge with capacity $c_e - f(e)$, called a **forward edge**. The difference is obviously the remaining capacity available on e. For every edge e of G with $f(e) > 0$, G_f has the opposite edge with capacity $f(e)$, called a **backward edge**. By virtue of backward edges we can "undo" any amount of flow up to $f(e)$ on e by sending it back in the opposite direction. The **residual capacity** is defined as $c_e - f(e)$ on forward edges and $f(e)$ on backward edges

Now let P be any simple directed $s-t$ path in G_f, and let b be the smallest residual capacity of all edges in P. For every forward edge e in P, we may increase $f(e)$ in G by b, and for every backward edge e in P, we may decrease $f(e)$ in G by b. It is not hard to check that the resulting function f' on the edges is still a flow in G. We call f' an **augmented** flow, obtained by these changes. Note that $val(f') = val(f) + b > val(f)$.

Now the basic **Ford-Fulkerson algorithm** works as follows: Initially let $f := 0$. As long as a directed $s-t$ path in G_f exists, augment the flow f (as described above).

To prove that Ford-Fulkerson outputs a maximum flow, we must show: If no $s-t$ path in G_f exists, then f is a maximum flow.

The proof is done via another concept of independent interest: An $s-t$ **cut** in $G = (V, E)$ is a partition of V into sets A, B with $s \in A, t \in B$. The capacity of a cut is defined as $c(A, B) := \sum_{e=(u,v):u\in A, v\in B} c_e$.

For subsets $S \subset V$ we define $f^+(S) := \sum_{e=(u,v):u\in S, v\notin S} f(e)$ and $f^-(S) := \sum_{e=(u,v):u\notin S, v\in S} f(e)$. Remember that $val(f) = f^+(s) - f^-(s)$ by definition. (Actually we have $f^-(s) = 0$ if no edge goes into s.) We can generalize this equation to any cut: $val(f) = \sum_{u\in A}(f^+(u) - f^-(u))$, which follows easily from the conservation constraints. When we rewrite the last expression for $val(f)$ as a sum of flows on edges, then, for edges e with both nodes in A, terms $+f(e)$ and $-f(e)$ cancel out in the sum. It remains $val(f) = f^+(A) - f^-(A)$. It follows $val(f) \leq f^+(A) = \sum_{e=(u,v):u\in A, v\notin A} f(e) \leq \sum_{e=(u,v):u\in A, v\notin A} c_e = c(A, B)$. In words: The flow value $val(f)$ is bounded

by the capacity of any cut (which is also intuitive).

Next we show that, for the flow f returned by Ford-Fulkerson, there exists a cut with $val(f) = c(A, B)$. This implies that the algorithm in fact computes a maximum flow.

Clearly, when the Ford-Fulkerson algorithm stops, no directed $s-t$ path exists in G_f. Now we specify a cut as desired: Let A be the set of nodes v such that some directed $s-v$ path is in G_f, and $B = V \setminus A$. Since $s \in A$ and $t \in B$, this is actually a cut. For every edge (u, v) with $u \in A$, $v \in B$ we have $f(e) = c_e$ (or v should be in A). For every edge (u, v) with $u \in B$, $v \in A$ we have $f(e) = 0$ (or u should be in A because of the backward edge (v, u) in G_f). Altogether we obtain $val(f) = f^+(A) - f^-(A) = f^+(A) = c(A, B)$. In words: The flow value $val(f)$ equals the capacity of a minimum cut (which is still intuitive). The last statement is the famous **Max-Flow Min-Cut Theorem.**

It is important to notice that Ford-Fulkerson in the basic version does not guarantee polynomial time; this depends on the capacities. But if we always choose a shortest augmenting path, we get polynomial time – we have to skip the proof of the time bound.

Bipartite Matching

Here is one of the simplest but also most important examples of a reduction of another graph problem to Maximum Flow.

In a bipartite graph $G = (X, Y, E)$, the node set is split into sets X, Y, and edges exist only between X and Y. A **matching** is a set of pairwise node-disjoint edges. The Bipartite Matching problem asks to find a matching of maximum size in a bipartite graph. Typical applications are job assignment problems: Nodes in X are jobs to be done, nodes in Y are workers or machines, and an edge means that the worker/machine is able to do the job. A matching is then a set of jobs that can be executed in parallel.

Bipartite Matching is reduced to Maximum Flow as follows: Add a source s and a sink t, insert edges from s to all nodes in X, and from all nodes in Y to t, orient the edges of E from X to Y, and set all edge capacities 1. Then the maximum matchings correspond exactly to the maximum flows with integer values (0 or 1) on the edges. (This equivalence needs a proof, however this is simple enough.) Now we can use Ford-Fulkerson to solve the problem in $O(mn)$ time. (Do you see why the time bound holds?)

One can also find maximum matchings in general graphs is polynomial time, but this is much more tricky.