Discrete Optimization – Lecture Notes handout 3

Instructor: Gabriel Nivasch

May 31, 2012

These lecture notes are about finding shortest paths in directed graphs. These notes are generally based on Chapter 2.2 of the book *Combinatorial Optimization* by Cook, Cunningham, Pulleyblank, and Schrijver, Wiley-Interscience, 1998.

Suppose we are given a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. Each edge $e \in E$ is assigned a *length* c_e. We do allow negative lengths (it sounds strange, but they sometimes do come up in applications). However, we assume there are no negative cycles (a negative cycle is a directed cycle whose total length is negative), since negative cycles create problems.

Sometimes we are given two vertices $s, t \in V$ and we want to find the shortest path from s to t, and sometimes we are just given a vertex $s \in V$ and we want to find the shortest path from s to all the other vertices in G. (Note that if G were to contain a negative cycle, then there might be an arbitrarily short path from s to t, by taking the cycle arbitrarily many times on the way from s to t.)

1 Finding the shortest s-t path using linear programming

Suppose we want to find the shortest path from s to t. To represent a path from s to t, we introduce a variable x_e for each edge $e \in E$, which should equal 1 if edge e is used in the path, and 0 otherwise.

Then we can formulate our problem as the following program:

Minimize $\sum_{e \in E} x_e c_e$ subject to:

- $x_e \in \{0, 1\}$ for every $e \in E$;
- The vertex s has exactly one outgoing edge with $x_e = 1$;
- The vertex t has exactly one incoming edge with $x_e = 1$;
- Every other vertex has either 0 incoming and 0 outgoing, or 1 incoming and 1 outgoing, edges with $x_e = 1$.

This program, unfortunately, is not a linear program. Let us call it the *desired program*.

Instead, we formulate the following linear program:

Minimize $\sum_{e \in E} x_e c_e$ subject to:

- $x_e \geq 0$ for every $e \in E$;

- For every vertex v we have

$$\sum_{e \text{ into } v} x_e - \sum_{e \text{ out of } v} x_e = \begin{cases} -1, & \text{for } v = s; \\ 1, & \text{for } v = t; \\ 0, & \text{for all other } v. \end{cases}$$

We claim that the above linear program has an optimal solution which satisfies the desired program. Here is a sketch of the proof:

Let $\mathbf{x}^*$ be an optimal solution for the linear program, and let $E' \subseteq E$ be the subset of edges for which $x_e \neq 0$.

Without loss of generality we can assume that E' does not contain any cycle: If E' contains a cycle, its total length cannot be negative (because G has no negative cycles), nor can it be positive, because then we could lower the x_e's on the cycle in sync, thus decreasing the objective function without violating any constraint.

Therefore, if E' contains any cycle, it must have total length 0. Then we can lower the x_e's on this cycle, as above, until one of the x_e's becomes 0. We can repeat this procecure until E' has no cycles.

Once E' has no cycles, it consists of paths from s to t with total weight 1 (the paths might overlap). All these paths must have the same total length, or else we could improve the objective function by subtracting from a longer path and adding to a shorter path. But if all the paths have the same length, then by assigning $x_e = 1$ just to the edges of this path and $x_e = 0$ otherwise we obtain a solution with exactly the same objective function.

1.1 Feasible potentials

Given the graph G and a vertex s, for each $v \in V$ let y_v^* be the length of the shortest path from s to v. The variables y_v^* satisfy the following two important properties:

- $y_s^* = 0$;
- For every edge $e = vw$ we have $y_w^* \leq y_v^* + c_{vw}$.

Based on these properties, we say that an assignment $y = \{y_v\}$ to the vertices $v \in V$ is a *feasible potential* if $y_s = 0$ and $y_w \leq y_v + c_{vw}$ for every $vw \in E$. (We showed in an exercise that if G has a negative cycle then G does not have any feasible potential.)

We already showed in an exercise that if y is a feasible potential for G, then, for every $v \in V$, y_v is a *lower bound* for the length of the shortest path from s to v.

We also proved, using LP duality, that the maximum feasible potential in fact *equals* the shortest path; meaning, for every $v \in V \setminus \{s\}$ there exists a feasible potential y for G in which y_v *equals* the length of the shortest path from s to v.

2 Ford's algorithm for shortest paths

We now describe a general framework for shortest-path algorithms. This general framework is known as *Ford's algorithm*.

First we need the following result:

Lemma 1. *Let $G = (V, E)$ be a graph without negative cycles, let $s \in V$ be a vertex, and for every other vertex $v \in V \setminus \{s\}$ let p_v be the (or a) shortest path from s to v (i.e., p_v is a set of edges connecting s to v). Let $P = \bigcup p_v$ be the union of all these shortest paths. Then, we can assume without loss of generality that every vertex $v \in V$ has at most one incoming edge in P.*

Proof. Suppose that some vertex v has two incoming edges in P. That means that there are two vertices $w_1, w_2 \in V$ whose corresponding shortest paths p_{w_1} and p_{w_2} both go through v but take different routes from s to v. These two routes must have the same length, so we can assume without loss of generality that we always take one of the routes. □

Lemma 1 implies that, in order to represent all shortest paths from s, all we need is a pointer $p(v)$ for each vertex v (the "parent of v"), which tells us which vertex comes before v in the shortest path from s to v. Then, in order to reconstruct this path, we repeatedly follow the parent pointers until we reach s.

For every vertex $v \in V$ we will also keep a variable $y(v)$, which stores the length of the shortest path from s to v we have found so far.

We define the following initialization routine:

`initialize`(y, p): $y(s) \leftarrow 0$,
$y(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$,
$p(v) \leftarrow$ NULL for every $v \in V$.

We say that an edge $uv \in E$ is *incorrect* if $y(w) > y(v) + c_{vw}$. The following routine checks whether an edge is incorrect, and if so, corrects it:

`do_edge`(vw): if $y(w) > y(v) + c_{vw}$:
$y(w) \leftarrow y(v) + c_{vw}$,
$p(w) \leftarrow v$.

Now we can formulate Ford's algorithm, our general scheme:

```
Ford: initialize(y, p),
      while there exists an incorrect edge:
          let e be an incorrect edge,
          do_edge(e).
```

Why does the algorithm always terminate? Because, first of all, the algorithm is always making progress: At each step it lowers some $y(v)$. Furthermore, there are only a finite number of states: A state is entirely specified by the $p(v)$ pointers (since the $y(v)$ values can be deduced from these pointers), so there are at most n^n states. Therefore, Ford's algorithm can take at most a finite number of steps.

Ford's algorithm provides an alternative proof that shortest-path equals max-feasible-potential: Throughout the execution of Ford's algorithm, every $y(v)$ holds an *upper bound* on the length of the shortest path from s to v. Furthermore, when the algorithm stops there are no incorrect edges, so the final y is a feasible potential, so it is a

lower bound on that length. Therefore, the final y is a feasible potential that *equals* the shortest path from s to each vertex.

2.1 Worst-case running time

Ford's algorithm always terminates, but its running time depends on the choice of the incorrect edge to correct at each step. In the exercises we saw an example of a graph with $O(n)$ vertices and edges, for which there exists a "bad" sequence of choices that make Ford's algorithm take roughly 2^n steps. Thus, Ford's algorithm's worst-case running time is at least exponential in n.

3 The Bellman–Ford algorithm

The *Bellman–Ford algorithm* is a specific instance of Ford's algorithm in which the edges are corrected in a suitable order and we achieve a reasonably good running time:

```
Bellmann-Ford:  initialize(y, p),
                (let e_1, e_2, ..., e_m be an ordering of the edges of E)
                repeat the following n times ("stages"):
                    for i = 1, 2, ..., m:
                        do_edge(e_i).
```

We must prove that this algorithm is correct, meaning, that when it terminates it holds the shortest paths to all the vertices.

We claim that, for every $v \in V$, if the (or a) shortest path from s to v has i edges, then after the i-th "stage" the algorithm already holds the shortest path from s to v.

Indeed, let $e'_1, e'_2, \ldots, e'_i$ be the edges in this shortest path, and label the intermediate vertices in the path $v_1, v_2, \ldots, v_{i-1}$. Note that $e'_1 e'_2 \ldots e'_j$ is the shortest path to v_j for every $j < i$. In the first stage we call `do_edge` on e'_1, so at the end of the first stage $y(v_1)$ is correct. Then, in the second stage we call `do_edge` on e'_2, so that at the end of the second stage $y(v_2)$ is correct; and so on.

The running time of Bellman-Ford is $O(mn)$. This running time is not so good, but surprisingly, no asymptotically better algorithm is known for this problem.

Here is an alternative formulation of the Bellman–Ford algorithm. Define the following routine:

```
do_vertex(v):   (let e_1, e_2, ..., e_k be the outgoing edges of v)
                for i = 1, 2, ..., k:
                    do_edge(e_i).
```

Then:

```
Bellmann-Ford2:  initialize(y, p),
                 (let v_1, v_2, ..., v_n be an ordering of the vertices of V)
                 repeat the following n times ("stages"):
                     for i = 1, 2, ..., n:
                         do_vertex(v_i).
```

This is clearly the same as before, because at every stage we are calling `do_edge` to every edge exactly once.

4 A faster algorithm for acyclic graphs

If G has no directed cycles then we can compute all shortest paths from s in time $O(m+n)$, by first computing a topological sort of G and then executing a *single stage* of `Bellman-Ford2` using this order for the vertices. The details have been given as an exercise.

5 A faster algorithm for graphs without negative lengths

If all the lengths c_e are nonnegative then there exists a faster algorithm, called *Dijkstra's algorithm*. The idea is to keep a set S of "unvisited" vertices, which at the beginning contains all of V. At each step we remove from S the vertex v for which $y(v)$ is smallest, and we call `do_vertex` on it:

`Dijkstra`: `initialize`(y,p),
$S \leftarrow V$,
while $S \neq \emptyset$:
    let v be the vertes in S with smallest $y(v)$,
    `do_vertex`(v),
    $S \leftarrow S \setminus \{v\}$.

We now prove that Dijstra's algorithm is correct. We need to show that when the algorithm terminates no edge is incorrect. The proof goes through several steps.

Let $v_1, v_2, \ldots, v_n$ be the vertices of G in the order in which Dijkstra calls `do_vertex` on them. For each v_i, denote by y_i^* the value of $y(v_i)$ at the time in which v_i is visited.

Claim 2. *We have* $y_1^* \leq y_2^* \leq \cdots \leq y_n^*$.

Proof. The call `do_vertex`(v_i) does not lower any $y(w)$ from being larger-egual to $y(v_i)$ to being smaller than $y(v_i)$. This is because the lengths c_e of the outgoing edges of v_i are nonnegative.

In particular, before visiting v_i we had $y(v_{i+1}) \geq y(v_i)$, and this will still be true after v_i is visited. □

Claim 3. *For each vertex* v_i, y_i^* *is the* final *value of* $y(v_i)$.

Proof. After we visit v_i we visit vertices v_j that have $y(v_j) \geq y(v_i)$, and they cannot help lower $y(v_i)$, since all their outgoing edges have nonnegative length. □

Claim 4. *When the algorithm terminates, no edge is incorrect.*

Proof. Suppose for a contradiction that when the algorihtm terminates we have an incorrect edge $v_iv_j \in E$ with $y(v_j) > y(v_i) + c_{v_iv_j}$. By Claim 3, when v_i was visited, $y(v_i)$ had already achieved its final value. Therefore, during the visit of v_i, $y(v_j)$ should have been lowered to $y(v_i) + c_{v_iv_j}$; contradiction. □

5.1 Implementing Dijkstra's algorithm efficiently

In order to implement Dijsktra's algorithm efficiently we need a data structure for S that supports an initial insertion of elements with keys, extraction of the element with the smallest key, and decreasing keys of elements. The indicated data structure for this is a *Fibonacci heap*. If S is implemented as a Fibonacci heap then Dijkstra's algorithm has running time $O(m + n \log n)$.