The shortest path problem (section 15.3)

We study a basic combinatorial problem. But first:

- A (directed) walk in a directed graph $D = (V, E)$ is a sequence
$$P = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k)$$
where $k \geq 0$, $v_i \in V$ $(0 \leq i \leq k)$ and $e_i = (v_{i-1}, v_i)$ $(i \leq k)$. We say that P goes from v_0 to v_k, and call P a v_0v_k-walk.
- A (directed) path is walk P where $v_0, v_1, \ldots, v_k$ are distinct; it is called a v_0v_k-path.
- The difference is that a walk may contain cycles.

The shortest path problem: given a directed graph $D = (V, E)$ with a nonnegative number (length, weight) c_{ij} for each edge (i, j), and two nodes s and r, find a shortest path P from s to r. Here the length of a path is the sum of the c_{ij}'s for its edges.

- The problem has immediate applications in most types of networks, for instance, road networks (shortest route to drive), flight networks, ..., or in dynamic optimization problems that arise e.g. in economics/finance.
- The shortest path problem is also a subproblem in many different, more complicated problems.
- Often we wish to find a shortest path between several node-pairs. Several algorithms actually find a shortest path from the initial node s to all other nodes, so we may use such an algorithm from each initial node.

Example: dynamic programming (dynamical systems with sequencial decisions): this important example will be given at the end!

Network flow formulation

The shortest path problem is a special case of the minimum cost network flow problem:

$$\min\{c^T x : Ax = -b,\ x \geq O\}.$$

- Here A is the node-edge incidence matrix of the graph, c is the cost vector (the edge lengths), and $b = (b_v : v \in V)$ is the vector given by $b_s = 1$, $b_r = -1$ and $b_v = 0$ otherwise.
- This approach works because there is an integer optimal solution, and the edges with positive flow must contain a path from s to r: $x_{ij} = 1$ for all edges in the path, and $x_{ij} = 0$ otherwise. (If there are edges with zero length, one may get cycles in addition to the path.)
- So one may solve the shortest path problem as a min. cost network flow problem using the network simplex algorithm.
- However, simpler and faster algorithms also exist! We shall discuss two such methods.

The Bellman-Ford algorithm

- For $v \in V$ og $k \geq 0$ (integer), we define $d_k(v)$ as the minimum length of an *sv*-walk with *at most k edges*. If there is no such walk, define $d_k(v) = \infty$.

How can we compute these these distance functions?

The Bellman-Ford's algorithm: *let* $d_0(s) = 0$ *and* $d_0(v) = \infty$ *for each* $v \neq s$. *Compute the functions* $d_1, d_2, \ldots, d_n$ *by*

$$d_{k+1}(v) = \min\{d_k(v), \min_{u:(u,v)\in E}(d_k(u) + c_{uv})\} \tag{1}$$

for all $v \in V$.

Theorem: *The Bellman-Ford algorithm finds the correct distances, i.e.,* $d_k(v)$ *becomes the minimum length of an sv-walk with at most k edges. In particular,* $d_{n-1}(v)$ *is the length of a shortest sv-path (here n is the number of nodes in the graph).*

Proof: A shortest sv-path with at most $k+1$ edges has either (i) at most k edges or (ii) it has $k+1$ edges and contains an edge (u, v) as its final edge. But in case (ii) the subpath to u must be a shortest su-path with at most k edges (for otherwise we could find another shorter su-path and thereby improve the sv-path). □

- The equation (1) for computing d_{k+1} based on d_k is called Bellman's equation. It is also used in similar problems called (discrete) *dynamic programmering* or *optimal control* (continuous version); the equation is then called the *Hamilton-Jacobi-Bellman (HJB) equation*.
- The BF-algorithm has complexity (number of arithmetic computations) $O(nm)$ where the graph has n nodes and m edges.
- The algorithm has another important property: it can also be used if there are negative lengths on the edges. The BF algorithm will then decide if there exists a cycle reachable from s with total length which is negative; then $d_n(v) < d_{n-1}(v)$. If this does not happen, the BF algorithm finds a shortest sv-path.

Dijkstra's algorithm

- This is also an algorithm for the shortest path problem.
- It only works for nonnegative edge lengths (which is most common in applications!)
- Dijkstra's algorithm is faster than the Bellman-Ford algorithm.
- Note: our description is slightly different than the one in the book: we start at s and move forward along edges while Vanderbei goes backwards!
- A usual n is the number of nodes.

- The algorithm performs n iterations, in each iteration one node is added to a certain set $\mathcal{F}$ and certain computations are done. At the start $\mathcal{F} = \emptyset$.
- One has a value (a label) d_i, for each node i: d_i is an upper bound on the (shortest) distance from s to i. Initially: $d_s = 0$, and $d_i = \infty$ otherwise. $\mathcal{F}$ consist of the nodes to which one already has found a shortest path, for these nodes d_i is *equal* to the distance from s to i.
- In each iteration:
  1. choose an $i \notin \mathcal{F}$ with d_i smallest possible ("a closest node"), and update $\mathcal{F} := \mathcal{F} \cup \{i\}$.
  2. for each edge $(i, j) \in E$ where $j \notin \mathcal{F}$, set

$$d_j = \min\{d_j, d_i + c_{ij}\}$$

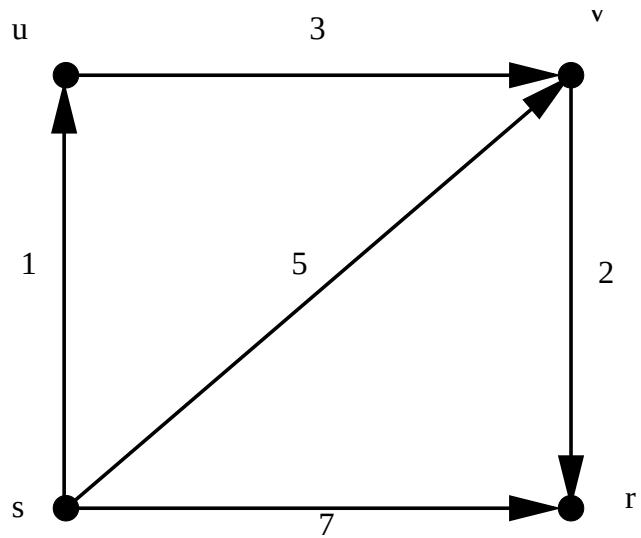
     and, if d_j was reduced, set a pointer $prev(j) = i$.

- This means that, at the start of each iteration, $d(v)$ is equal to the length of a *shortest sv-path that only uses nodes in* $\mathcal{F}$.

We have (without giving the proof, which is a rather simple induction proof, by the way):

Theorem: *Dijkstra's algorithm finds a shortest path, and corresponding distances* d_v, *from* s *to each node* v. *The complexity is* $O(n^2)$.

Example: use Dijkstra (and Bellman-Ford) here:



Application: Dynamic programming

In dynamic programming (DP) one has a discrete dynamic process which one wants to control best possible by a decision at each discrete time step t_i ($i \leq m$). The process is described by functions f_i where

$$s_{i+1} = f_i(s_i, x_i) \quad (i \leq m) \tag{2}$$

and s_i is the state at time t_i and x_i is a variable called the *control* at time t_i. We assume that everything here is discrete (actually finite), so the control $x_i \in \mathcal{X}$ and the state $s_i \in \mathcal{S}$ where $\mathcal{X}$ and $\mathcal{S}$ are finite sets. Further, $f_i : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$. There are also given *cost functions* $g_i(s_i, x_i)$ giving the "cost" by choosing the control x_i at state s_i in time t_i ($i \leq m$). The problem in DP is to find a *control* $x_1, x_2, \ldots, x_m$ that minimizes the total cost

$$\sum_{i=1}^{m} g_i(s_i, x_i)$$

subject to the constraints (2) and $s_i = s^*$, $s_{m+1} = s^{**}$ where s^* og s^{**} are the given initial and terminal state of the system.

Construct a directed graph D with nodes (t_i, s_j) for each time t_i $(i \leq m)$ and each state $s_j \in \mathcal{S}$. Furthermore, introduce an edge from (t_i, s_j) to (t_{i+1}, s_k) if there is a control x_i such that $s_k = f_i(s_i, x_i)$, and let this edge have cost equal to $g_i(s_i, x_i)$ (the smallest such cost, if there are several controls that give this state).

Note that the graph D gets nodes as in a grid, where edges are directed "towards the right" bewteen nodes $(t_i, \cdot)$ and nodes $(t_{i+1}, \cdot)$. Therefore D has no directed cycle.

We can solve DP by finding a shortest path in D from the node (s^*, t_1) to the node (s^{**}, t_{m+1}) !!!

- This is simple and powerful!! Still, DP is a *very general model*, with lots of application in science, engineering and economics. So such problems may be solved using e.g. The Bellman-Ford algorithm as described above, very efficiently. (Well, at least is the number of states is "reasonable").
- A variant with variable terminal state can also be treated in a similar way.
- There are more complicated extensions of DP: stochastic state transitions, continuous time etc. This requires more sophisticated methods.