(M. Dietzfelbinger, June 24, 2022)

4.4 Dijkstra's algorithm

Input: (Directed or undirected) Graph $G = (V, E)$ with edge weights $\ell(u, v) \geq 0$, for $(u, v) \in E$, and a starting node s.

Notation: If $p = (v_0, v_1, v_2, \ldots, v_k)$ is a path from v_0 to v_k, the length $\ell(p)$ of p is $\ell(v_0, v_1) + \ell(v_1, v_2) + \cdots + \ell(v_{k-1}, v_k)$.

The empty path (v_0) (no edge) has length 0.

$d(s, v) :=$ length of a shortest path from s to v.
(This could be ∞, if v is not reachable from s.)

The following data structure is used:

`dist`$[1..n]$ is an array of reals, one entry for each node $v \in V = \{1, \ldots, n\}$.
Underway, `dist`$[v]$ is a tentative/preliminary distance from s to v.
Initially, `dist`$[s] = 0$, and `dist`$[v] = \infty$ for all $v \neq s$.
Always: `dist`$[v]$ is the key for entry v, if v is in the priority queue H.

`prev`$[1..n]$ is an array of nodes, `prev`$[v]$ is the node from where the currently estimated shortest path reaches v.

Available is also a data structure H, which is a *priority queue* (PQ), with operations

- *create* (an empty PQ).
- *insert*(H, v, t): Inserts node v with "priority" (alarm clock time) t.
- *decreaseKey*(H, v, t): reduces priority (alarm clock time) of node v to t.
- *ejectMin*(H) (in the book: "*eject*"): outputs some node stored in H with minimum priority (alarm clock time), removes this node from H.

A possible implementation of PQs are *binary heaps* (see Section 4.5.2 in the book). With this, an operation takes time $O(\log n)$, where n is the number of nodes in the graph. There are other implementations, like a simple array or d-ary heaps, see book.

Procedure needed in Dijkstra's algorithm, in the main loop:

```
procedure update(u, v)
    if dist[u] + ℓ(u, v) < dist[v] then
        prev[v] ← u;
        dist[v] ← dist[u] + ℓ(u, v).
```

This corresponds to adjusting the alarm clock if v can be reached faster via u than before. Note that now the status of v with respect to the priority queue has to be adjusted, if dist$[v]$ has changed.

Dijkstra's algorithm

```
procedure Dijkstra(G, ℓ, s)
  Input: Directed or undirected graph G = (V, E)
      with edge weights ℓ(u, v) ≥ 0, (u, v) ∈ E, starting node s ∈ V
  Output: dist[v] = d(s, v) for all nodes v ∈ V;
      if v not reachable from s, dist[v] = ∞.
      prev[v] is predecessor of v in a shortest-path tree with root s.

  for all u ∈ V:
    dist[u] ← ∞
    prev[u] ← nil

  H ← empty priority queue
    dist[s] ← 0
    insert(H, s)    //    Object s, Priority dist[s] ← 0

  while not isempty(H) do
    u ← ejectMin(H)
    for all edges (u, v) ∈ E do
        old ← dist[v]
        if dist[u] + ℓ(u, v) < old then    //    update(u, v)
            prev[v] ← u;
            dist[v] ← dist[u] + ℓ(u, v)
            if old = ∞
                then insert(H, v, dist[v])    //    Object v, new priority dist[v]
                else decreaseKey(H, v, dist[v])    //    new priority: new dist[v]

  return arrays dist[1..n] and prev[1..n].
```

Note: In contrast to the algorithm in the book we do not put all nodes into the priority queue at the beginning. A node v enters the priority queue only when dist$[v]$ is assigned a value $< \infty$ ("v is found").

Running time: Case 1: H is represented by the $\texttt{dist}[1..n]$ array. Finding the minimum entry takes time $O(n)$, inserting or deleting an entry takes constant time (mark it as "*scanned*", 1 bit). The total running time is $O(n^2 + m)$.
Case 2: H ist respresented as a binary heap, see discussion group. The operations *ejectMin*, *insert*, *decreaseKey* all take time $O(\log n)$. The overall time is $O((n + m) \log n)$.

Proof of correctness:

We say a node v is ***found*** when $\texttt{dist}[v]$ gets a value $< \infty$. We say a node u is ***scanned*** when it is taken out of the priority queue and *update* is applied to all outgoing edges (u, v). It is clear that the start node s is found in the initialization, and that s is the first node to be scanned. For $v \neq s$: node v is found when for the first time $\mathit{update}(u, v)$ is carried out in the course of scanning u, for some node u found earlier. If $\mathit{update}(u, v)$ is carried out, we are sure that $\texttt{dist}[u] < \infty$ before that and $\texttt{dist}[v] < \infty$ afterwards.

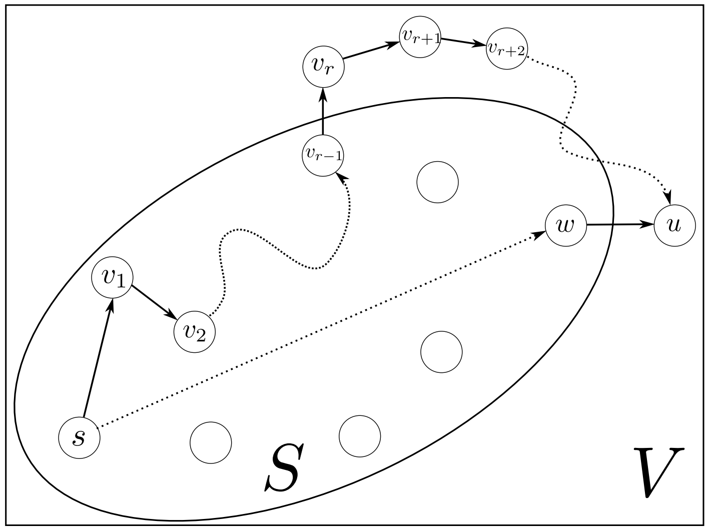
Claim 1: Assume $\texttt{dist}[v] < \infty$, i.e., has been found before, at any time in the algorithm. Then there is a path of length $\texttt{dist}[v]$ from s to v.
(In particular: $d(s, v) \leq \texttt{dist}[v]$. Note the claim is true even though $\texttt{dist}[v]$ may attain first larger and then smaller and smaller values in the course of the algorithm.)

Proof: Indirect. Assume there is some time t and some node v such that $\texttt{dist}[v]$ is set to some value $< \infty$ at time t in the algorithm, but there is no s-v path of length $\texttt{dist}[v]$. Choose t as small as possible with this property. Note first that $v \neq s$, since $\texttt{dist}[s]$ is set to $d(s, s) = 0$ at the beginning (and never changed). So at time t the value $\texttt{dist}[v]$ is set to some value $\texttt{dist}[u] + \ell(u, v)$, in the course of an operation $\mathit{update}(u, v)$. We have $\texttt{dist}[u] < \infty$, so u has been found before, and by the choice of t, there is a path from s to u of length $\texttt{dist}[u]$. Together with the edge (u, v) we get a path from s via u to v of length $\texttt{dist}[v] = \texttt{dist}[u] + \ell(u, v)$, a contradiction. □

Remark: When v is found, $\texttt{dist}[v]$ gets a value $< \infty$. Since nodes are taken out of H until no node with $\texttt{dist}$-value $< \infty$ is left, node v will be scanned at some time.

Claim 2: If v is reachable from s, then v will be found at some time.

Proof: Consider an arbitrary path $p = (s = v_0, v_1, v_2, \ldots, v_k = v)$ from s to v. Assume for a contradiction that v is never found. Choose i minimal such that v_i is never found. Then $i > 0$, since s is found during initialization. By choice of i we know that v_{i-1} is found at some time. By the Remark, it is then also scanned at some (later) time, and $\mathit{update}(v_{i-1}, v_i)$ is carried out. Since v_i is never found, we then have $\texttt{dist}[v_i] = \infty$, and $\mathit{update}(v_{i-1}, v_i)$ changes $\texttt{dist}[v_i]$ to $\texttt{dist}[v_{i-1}] + \ell(v_{i-1}, v_i) < \infty$, contradiction. □



Claim 3: When u is scanned, we have $\texttt{dist}[u] = d(s,u)$.

(We have seen that Claim 1 implies that $\texttt{dist}[u] \geq d(s,u)$ is always true.)

Proof: We prove this indirectly. Assume for a contradiction that there is some point t in time at which some node u with $d(s,u) < \texttt{dist}[u]$ is scanned. Choose t minimal with this property, and let S be the set of all nodes scanned strictly before time t. The algorithm sets $\texttt{dist}[s] = 0 = d(s,u)$ in the initialization, and node s is scanned first, so $s \in S$ and $t > 0$. Choose a path $p = (s = v_0, v_1, \ldots, v_k = u)$ of length $\ell(p) = d(s,u)$ from s to u. Let $r \leq k$ be minimal such that $v_r \notin S$. (The situation is given in the picture above.) We observe:

(a) By the algorithm and since $v_{r-1} \in S$: After v_{r-1} has been scanned, we have

$$\texttt{dist}[v_r] \leq \texttt{dist}[v_{r-1}] + \ell(v_{r-1}, v_r) = d(s, v_{r-1}) + \ell(v_{r-1}, v_r).$$

Afterwards, $\texttt{dist}[v_{r-1}]$ cannot change anymore (by Claim 1 it has the minimal possible value), and $\texttt{dist}[v_r]$ may only decrease; so the inequality still holds at round t.

(b) By the definition of $d(s,\cdot)$ we have $d(s, v_{r-1}) \leq \ell((v_0, \ldots, v_{r-1}))$.

(c) By the assumption, and since all edge costs are nonnegative:
At time t: $\ell((v_0, \ldots, v_{r-1})) + \ell(v_{r-1}, v_r) \leq c(p) = d(s,u) < \texttt{dist}[u]$.

From (a)–(c) we get $\texttt{dist}[v_r] < \texttt{dist}[u]$, in round t. This means that in round t a node with a smaller `dist` value than u is available for scanning, and hence the algorithm will not choose u. This is the desired contradiction. $\square$