Algorithms for Data Processing
Lecture IV: Graph Algorithms – Shortest Path and Dijkstra's Algorithm

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science
http://www.inf.unibz.it/˜artale
artale@inf.unibz.it

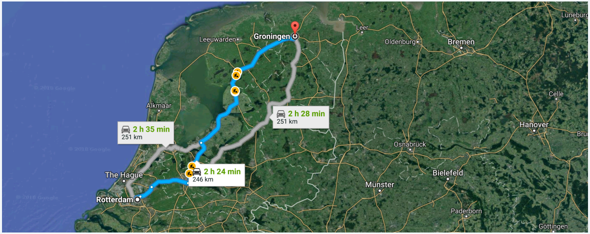
2019/20 – First Semester
MSc in Computational Data Science — UNIBZ

Shortest Path

- While BFS is able to find the shortest s-t path when edges have uniform cost, we need different algorithms when edges have a distinct traversal cost.
- We consider here the following scenario:
  - A directed graph $G = (V, E)$ with a designated start node s.
  - We assume that s has a path to every other node in G.
  - Each edge e has an associated *traversal cost* $\ell_e \geq 0$.
  - For a path P, the *length* of P–denoted $\ell(P)$–is the sum of the lengths of all edges $e \in P$.
- In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the shortest-paths problem.

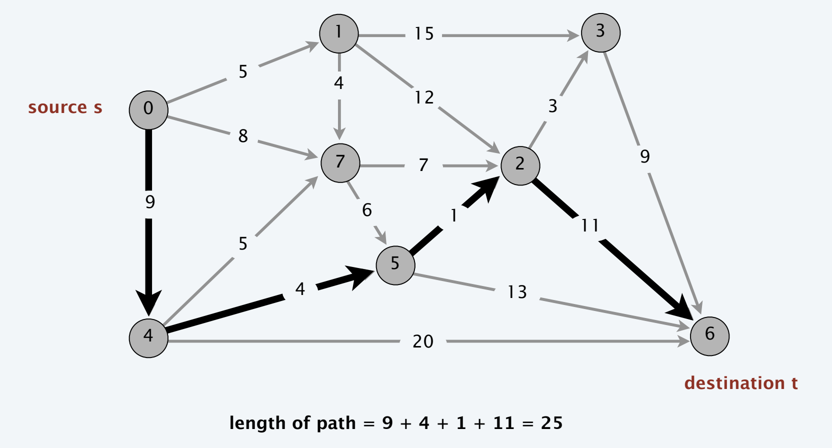
Edsger Dijkstra

“ *What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.* ” — Edsger Dijsktra



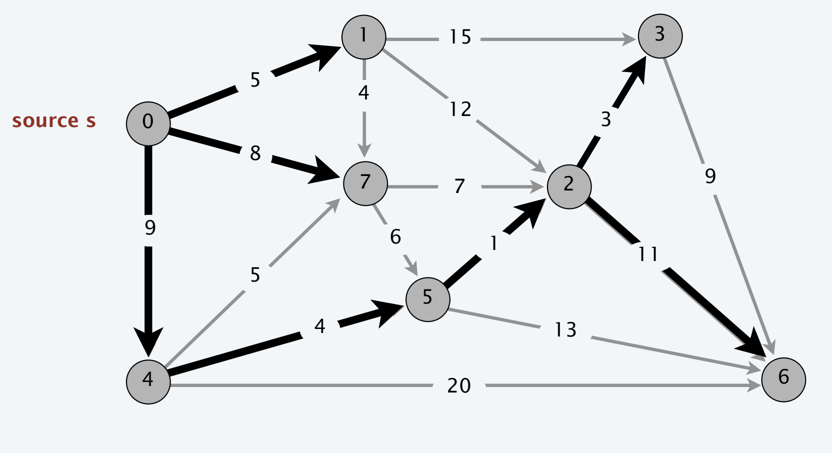
Single-pair shortest path problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.



Single-source shortest path problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.



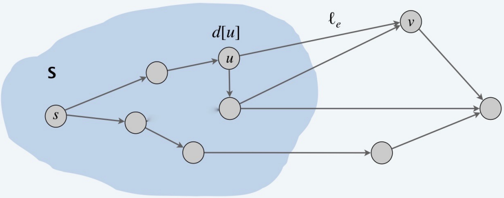
Dijkstra's algorithm (for single-source shortest paths problem)

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightarrow u$ path.

- Initialize $S \leftarrow \{ s \}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e = (u,v) \,:\, u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$



Dijkstra's algorithm (for single-source shortest paths problem)

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

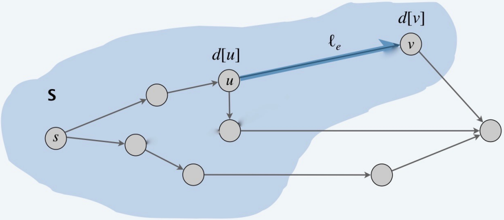
- Initialize $S \leftarrow \{ s \}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e = (u,v) \,:\, u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S, and set $d[v] \leftarrow \pi(v)$.

- To recover path, set $pred[v] \leftarrow e$ that achieves min.



Shortest Path: An Example from the book Algorithm Design

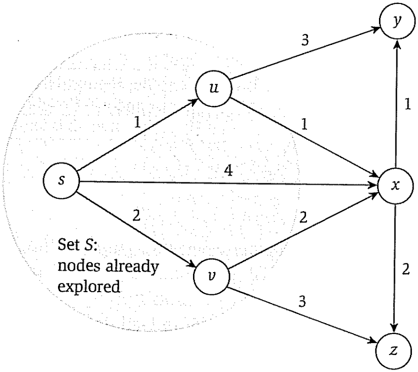


Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

Dijkstra's algorithm: proof of correctness

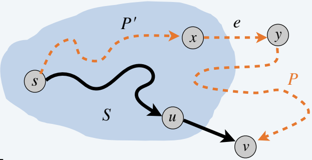
Invariant. For each node $u \in S$: $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

Pf. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d[s] = 0$.

Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any other $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the subpath from s to x.
- The length of P is already $\geq \pi(v)$ as soon as it reaches y:



$$\ell(P) \underbrace{\geq}_{\text{non-negative lengths}} \ell(P') + \ell_e \underbrace{\geq}_{\text{inductive hypothesis}} d[x] + \ell_e \underbrace{\geq}_{\text{definition of } \pi(y)} \pi(y) \underbrace{\geq}_{\text{Dijkstra chose } v \text{ instead of } y} \pi(v) \quad \blacksquare$$

Dijkstra Algorithm

We assume that $G = (V, E, L)$, where L contains the cost for each edge.

```
Dijkstra(G,s)
```

Let S be the set of explored nodes, and;

π an array of n elements such that $\pi[u]$ is the shortest s–u path;

$S = \{s\}$; $\pi[s] = 0$; pred$[s] = null$;

while $S \neq V$ **do**

  min-dist $= \infty$;

  for *each* $v \in V \setminus S$ *with* $(u, v) \in E$ *and* $u \in S$ **do**

    for *each* $(u, v) \in E$ *with* $u \in S$ **do**

      if $(\pi[u] + \ell(u, v) <$ min-dist$)$ **then**

        min-dist $= \pi[u] + \ell(u, v)$;

        node = v; pred$[node]$ = u

  $S = S \cup \{node\}$; $\pi[node] =$ min-dist.

Dijkstra Algorithm – Complexity

- Each While iteration adds a new node v to the set S.
- Then, there are $n - 1$ iterations of the While-loop for a graph with n nodes.
- Each iteration considers each node $v \notin S$, and goes through all the edges between S and v to determine the minimum distance vertex. This takes $O(m)$.
- Thus, the Algorithm runs in $O(m \cdot n)$.

Dijkstra's algorithm: efficient implementation

Critical optimization 1. For each unexplored node $v \notin S$:
explicitly maintain $\pi[v]$ insted of recomputing them in each iteration

$$\pi(v) \;=\; \min_{e\,=\,(u,v)\;:\;u\in S} \; d[u] + \ell_e$$

- For each $v \notin S$: $\pi(v)$ can only decrease (because set S increases).

- More specifically, suppose u is added to S and there is an edge $e = (u, v)$ leaving u. Then, it suffices to update:

$$\pi[v] \leftarrow \min \{ \pi[v],\ \pi[u] + \ell_e) \}$$

recall: for each $u \in S$,
$\pi[u] = d[u]$ = length of shortest $s \rightarrow u$ path

Critical optimization 2. Use a priority queue (PQ)
to choose an unexplored node that minimizes $\pi[v]$.

Brief Introduction to Priority Queues

- Priority Queue (PQ). Data Structure where elements have a priority value, or key, and we access just the element with *highest priority*.
- Data Structure. Balanced Binary Tree, where the root contains the element with highest priority.
  - Heap Order. The key of any element is at least as large as the key of its parent node.
- Cost of managing a PQ.
  - Extraction. $O(1)$, since we can access only the root element, which has the highest priority.
  - Addition and Deletion: $O(\log n)$.

From the Book: Chapter 2

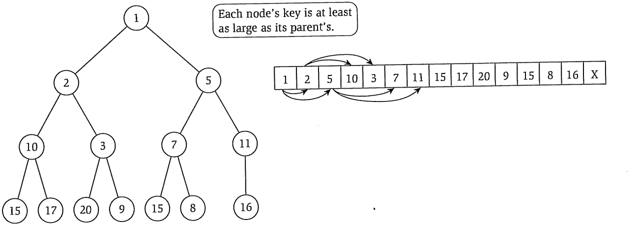


Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.

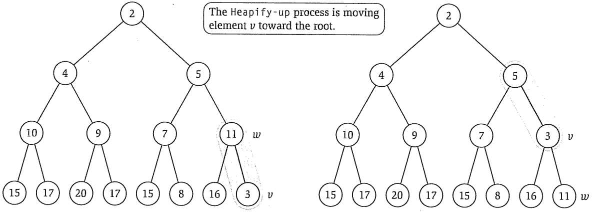


Figure 2.4 The Heapify-up process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).

Dijkstra's algorithm: efficient implementation

Implementation.

- Algorithm maintains $\pi[v]$ for each node v.
- Priority queue stores unexplored nodes, using $\pi[\cdot]$ as priorities.
- Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightarrow u$ path.

DIJKSTRA (V, E, ℓ, s)

pred[s] $\leftarrow$ null; π[s] $\leftarrow$ 0

FOREACH $v \neq s$: $\pi[v] \leftarrow \infty$, $pred[v] \leftarrow null$;

Create an empty priority queue pq.

FOREACH $v \in V$: INSERT(pq, v, $\pi[v]$).

WHILE (IS-NOT-EMPTY(pq))
  $u \leftarrow$ DEL-MIN(pq).
  FOREACH edge $e = (u, v) \in E$ leaving u:
    IF ($\pi[v] > \pi[u] + \ell_e$)
      $\pi[v] \leftarrow \pi[u] + l_e$;
      DECREASE-KEY(pq, v, $\pi[v]$).
      $pred[v] \leftarrow u$.

Dijkstra's algorithm: which priority queue?

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs. ← $\Theta(n^2)$ edges
- Binary heap much faster for sparse graphs. ← $\Theta(n)$ edges
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman–Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

† amortized

Thank You!