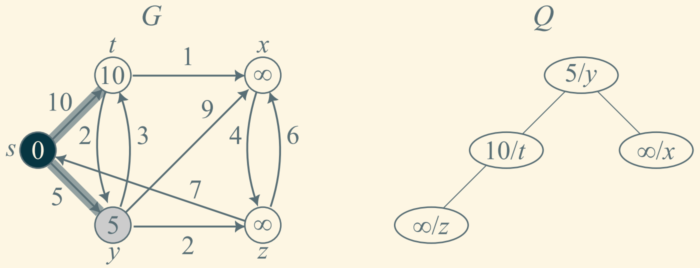
Lesson 05: Dijkstra's Shortest-Paths Algorithm

Michael T. Gastner (14 March 2023)



Disclaimer: These slides are based on and occasionally quote from 'Introduction to Algorithms' (3rd ed.) by Cormen et al. (2009), MIT Press.

Review of Previous Lesson

Heaps and Prim's Minimum-Spanning-Tree Algorithm

In the last session, we explored Prim's algorithm, which helps to find the minimum spanning tree by adding one vertex at a time. We start from a source vertex, which has no parent, and estimate the distance between the current tree and the vertices outside the tree. The algorithm then selects the vertex with the smallest estimate and adds it to the tree, along with the edge to its parent (i.e. the vertex from which the smallest estimate was discovered).

To implement Prim's algorithm efficiently, we used a heap-based min-priority queue, which helps maintain the elements of a dynamic set in a sorted order. We analysed the max-heap operations, including BUILD-MAX-HEAP, HEAP-EXTRACT-MAX and HEAP-INCREASE-KEY. Prim's algorithm uses the min-heap counterparts of these operations, which ensure a worst-case running time of $O(E \log V)$ for a connected graph.

Learning Objectives

By the End of This Lesson, You Should Be Able to . . .

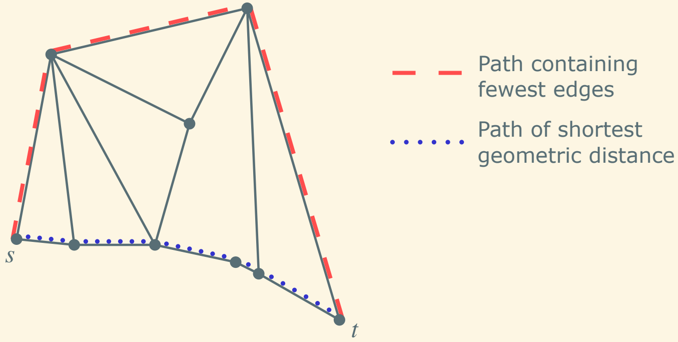
- Define the concept of a shortest path in a weighted graph.
- Apply Dijkstra's algorithm to a given weighted graph to find a shortest path.
- Argue that Dijkstra's algorithm is correct when none of the edge weights is negative.
- State the growth of the running time of Dijkstra's algorithm as a function of the input size.

Different Notions of Distance

Number of Edges, Geometric Distance, Time and Monetary Cost

In a previous lesson, we used breadth-first search to determine the minimum number of edges separating two vertices in a graph. However, the distance in a graph is not always meaningfully measured by the number of edges. Alternative measures include geometric distance, travel time and monetary cost.

In these cases, a shortest path may not necessarily minimise the number of traversed edges.


s
t
Path containing fewest edges
Path of shortest geometric distance

Shortest-Paths Problem

Mimimising the Sum of Weights Along All Edges in the Path

In a (weighted) **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

The **weight** of a path $p = \langle v_0, v_1, \dots, v_k \rangle$ is defined as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

We define the **shortest-path weight** (also known as **distance**) $\delta(u, v)$ from vertex u to vertex v as

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

Assumptions

Single-Source Shortest-Paths Problem and Nonnegative Edge Weights

In this lesson, we will focus on solving the **single-source shortest-paths problem**. Given a graph $G = (V, E)$, the objective is to find a shortest path from a given **source** vertex $s \in V$ to each vertex $v \in V$.

If none of the weights in the graph are negative, we can use **Dijkstra's algorithm** to find the solutions.

Which Complication Arises When Some Weights Are Negative?

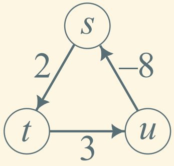
Negative-Weight Cycles

If weights are permitted to be negative, then shortest paths may not be well-defined.

For instance, in the graph on the right, the paths have the following weights:

- $\langle s, t \rangle$ has weight 2,
- $\langle s, t, u, s, t \rangle$ has weight -1,
- $\langle s, t, u, s, t, u, s, t \rangle$ has weight -4

and so on.


$$\begin{tikzcd} & s \arrow[dl, "2"'] & \\ t \arrow[rr, "3"'] & & u \arrow[ul, "-8"'] \end{tikzcd}$$

Whenever we complete the negative-weight cycle $s \to t \to u \to s$, the path becomes shorter. As a result, there is no shortest path because, for any given path from s to t, we can lower the weight by traversing the negative-weight cycle once more.

Dijkstra's Algorithm

Applicable if None of the Weights Are Negative

In Dijkstra's algorithm, each vertex v has two attributes:

- $v.d$ is an upper bound on the distance from s to v.
- $v.\pi$ is the predecessor of v on a shortest path from s to v.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate $u.d$, adds u to S, and updates the estimates of shortest-path weights for each neighbour of u.

The pseudocode on the next slide uses a min-priority queue of vertices, keyed by their d values.

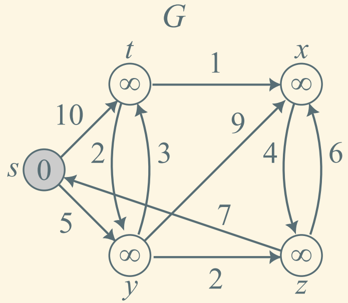
Dijkstra's Algorithm

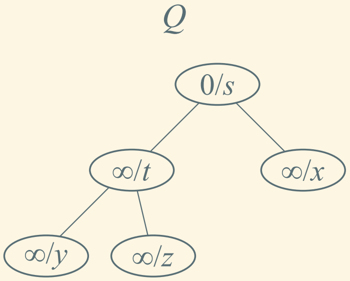
Pseudocode

```
DIJKSTRA($G, w, s$)
for each vertex $v \in G.V$ // Initialise vertex attributes
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$ // The source has distance 0 from itself
$S = \emptyset$
$Q = G.V$ // Use BUILD-MIN-HEAP
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$ // Use HEAP-EXTRACT-MIN
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        if $v.d > u.d + w(u, v)$
            $v.d = u.d + w(u, v)$ // Use HEAP-DECREASE-KEY
            $v.\pi = u$
```

Dijkstra's Algorithm

Illustration

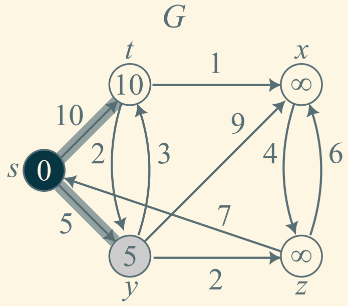


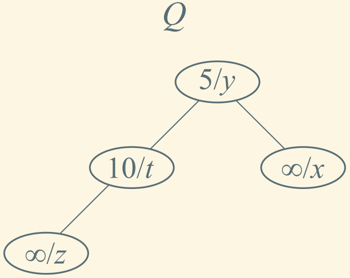


The state just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5.

Dijkstra's Algorithm

Illustration

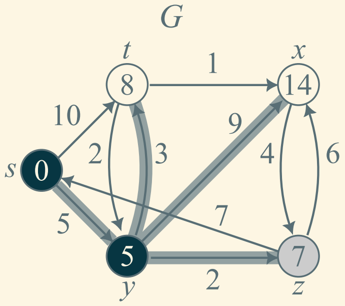


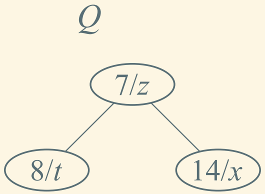


The situation after 1 iteration of the **while** loop. Shaded edges indicate predecessor values. The shaded vertex is vertex u in line 5 of the next iteration.

Dijkstra's Algorithm

Illustration

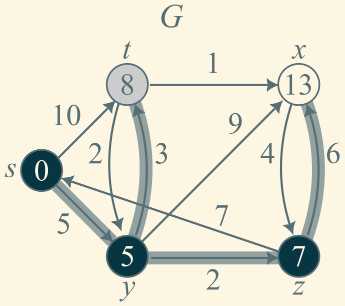


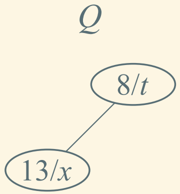


The situation after 2 iterations of the **while** loop. Shaded edges indicate predecessor values. The shaded vertex is vertex u in line 5 of the next iteration.

Dijkstra's Algorithm

Illustration

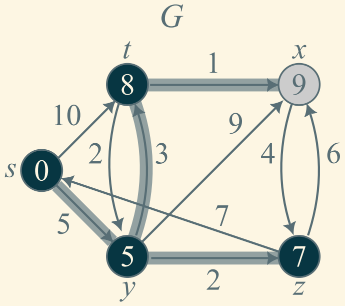


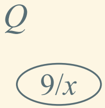


The situation after 3 iterations of the **while** loop. Shaded edges indicate predecessor values. The shaded vertex is vertex u in line 5 of the next iteration.

Dijkstra's Algorithm

Illustration

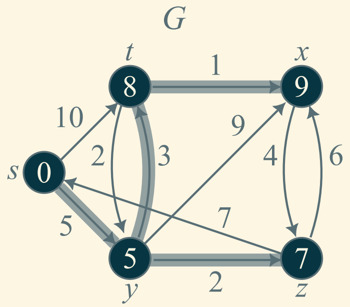




The situation after 4 iterations of the **while** loop. Shaded edges indicate predecessor values. The shaded vertex is vertex u in line 5 of the next iteration.

Dijkstra's Algorithm

Illustration



Q empty

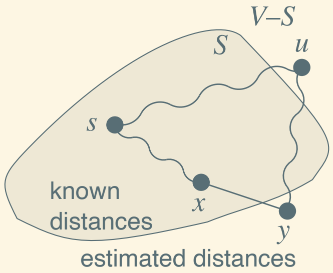
The d values and predecessors are the final values.

Dijkstra's Algorithm

Why Is It Guaranteed to Find Shortest Paths?

Suppose that u is the vertex with the smallest estimated (but not yet certain) distance from s.

If u's estimated distance is not equal to the shortest-path distance, then there must be a shorter path $s, \ldots, x, y, \ldots, u$, where y is the first vertex with an unknown distance and x is its predecessor. Because all distances are nonnegative, the sub-path from s to y via x must be shorter than the path along which we have first discovered u (the upper path in the figure).



However, this deduction contradicts the fact that u's estimated distance is smaller than y's. Therefore, u's estimated distance must be exact.

Worst-Case Running Time of Dijkstra's Algorithm $O(E \log V)$

- EXTRACT-MIN on line 8 is called at most $|V|$ times because, once a vertex is removed from Q, it is never inserted again.
- Using a binary heap implementation, each call to EXTRACT-MIN finishes in $O(\log V)$ time.
- Because the number of all adjacency list elements in a directed graph is E, there are $O(E)$ calls to HEAP-DECREASE-KEY on line 12 during the execution of DIJKSTRA.
- Each call of HEAP-DECREASE-KEY finishes in $O(\log V)$ time if a binary heap is used.

Therefore, the worst-case running time of DIJKSTRA using a binary heap is $O((V + E) \log V)$. If the graph is connected, we have $|V| = O(E)$. In that case, the running time of DIJKSTRA is $O(E \log V)$.

Implementation Achieving a Better Worst-Case Running Time

Using Fibonacci Heaps Instead of Binary Heaps

A Fibonacci heap reduces the time to $O(V \log V + E)$. However, for sparse graphs, both types of heaps have a running time of $O(V \log V)$. Because most real-world graphs are assumed to be sparse, the computational overhead and programming complexity of Fibonacci heaps usually make binary heaps the better choice in practice.

Outlook and Conclusion

Dijkstra's Algorithm and Further Graph Algorithms

In this lesson, we learnt that Dijkstra's algorithm solves the single-source shortest-paths problem in $O(E \log V)$ time if the graph $G = (V, E)$ does not contain negative-weight edges.

When there are negative-weight edges, Dijkstra's algorithm may give wrong answers. In that case, the Bellman-Ford algorithm should be used, which returns a d value of $-\infty$ for destination vertices that can be reached via a negative-weight cycle. However, the Bellman-Ford algorithm has a worst-case running time of $O(VE)$; thus, Dijkstra's algorithm is preferable if all weights are nonnegative.

There are many other graph algorithms of practical interest (e.g. the Floyd-Warshall algorithm for the all-pairs shortest-path problem and the Ford-Fulkerson method for the maximum-flow problem). However, these algorithms are beyond the scope of this course.