Course notes

Shortest Paths and Dijkstra's Algorithm

Author Name

Author	**Author Name**
Course	**Algorithms and Data Structures**
Last updated	**November 2, 2026**

How to read this document

These notes are a chapter of the course Algorithms and Data Structures. They are based on the lecture slides on shortest paths and on my handwritten notes of the lecture of 12 March. The structure follows the slides; the notes taken in class are reported in boxes marked “From my notes”. Anything added by the writer is marked as a proposed solution or as a remark.

The boxes have fixed meanings:

DEFINITION

Definition

A formal definition of a concept, such as the shortest-path distance.

EXAMPLE

Example

A worked example, step by step.

REMARK

Remark

A clarification, or a correction of an error in the sources.

FROM MY NOTES

A note taken in class, reported faithfully.

KEY POINT

What must be clear for the exam.

EXERCISE

Exercise

A question from the slides, always followed by a proposed solution.

Proposed solution. The answer is written here.

Contents

How to read this document iii

4 Shortest Paths: Dijkstra's Algorithm 1

4.1 The problem 1
4.2 The idea 1
4.3 A worked example 2
4.4 The algorithm 3
4.5 Cost 4
4.6 Negative weights 5
4.7 Summary 5

Chapter 4

Shortest Paths: Dijkstra's Algorithm

Given a weighted graph, we often want to know how far each vertex is from a starting vertex, for instance the cheapest route between two places. Dijkstra's algorithm solves this problem when all the weights are nonnegative: it repeatedly settles the closest vertex that is not yet final.

Slides 04 · Handwritten notes, lecture of 12 March

4.1 The problem

Let $G = (V, E)$ be a directed graph in which every edge (u, v) has a weight $w(u, v) \geq 0$, and let $s \in V$ be a distinguished vertex, the **source**. We want, for every vertex v, the distance from s to v.

DEFINITION

Shortest-path distance

The *shortest-path distance* from s to v is

$$d(s, v) = \min\{\, w(p) : p \text{ is a path from } s \text{ to } v \,\},$$

where $w(p)$ is the sum of the weights of the edges of the path p. If no path exists, $d(s, v) = \infty$.

4.2 The idea

The algorithm keeps an **estimate** $d[v]$ for every vertex. At the beginning $d[s] = 0$ and $d[v] = \infty$ for every other vertex. The estimate is then improved step by step: at each step we

extract the vertex with the smallest estimate among those not yet extracted, and we **relax** its outgoing edges.

DEFINITION

Relaxation

Relaxing an edge (u, v) means checking whether the path that reaches v through u is better than the current estimate of v:

$$\text{if } d[u] + w(u,v) < d[v] \text{ then } d[v] \leftarrow d[u] + w(u,v) \text{ and } pred[v] \leftarrow u.$$

Here $pred[v]$ records the previous vertex on the best path found so far.

Figure 4.1 shows the small graph that we use as an example.

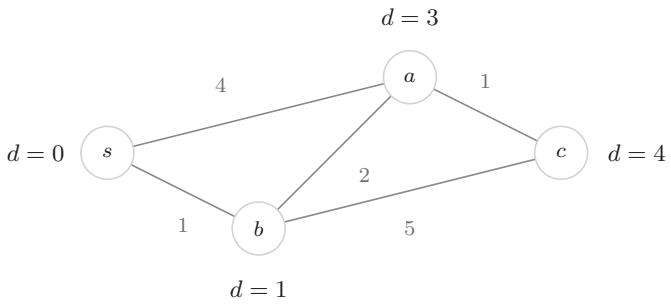


Figure 4.1. The example graph of the slides (redrawn). The numbers on the edges are the weights, and the values next to the vertices are the final distances from s.

4.3 A worked example

EXAMPLE

Trace on the example graph

We run the algorithm on the graph of fig. 4.1. At each step the vertex with the smallest estimate is extracted, and its outgoing edges are relaxed. The last column shows the estimates after the step.

Step	Extracted	Relaxed edges	$(d[s], d[a], d[b], d[c])$
1	s	$s \to a$ gives 4; $s \to b$ gives 1	$(0, 4, 1, \infty)$
2	b	$b \to a$ gives $3 < 4$, so $d[a]$ drops to 3; $b \to c$ gives 6	$(0, 3, 1, 6)$
3	a	$a \to c$ gives $4 < 6$, so $d[c]$ drops to 4	$(0, 3, 1, 4)$
4	c	no outgoing edges	$(0, 3, 1, 4)$

At the end, $d[s] = 0$, $d[b] = 1$, $d[a] = 3$ and $d[c] = 4$.

FROM MY NOTES

Once u has been extracted, $d[u]$ never changes again!! It is final.

The professor says to think of $d[v]$ as an upper bound on the distance from s to v, an upper bound that goes down as we find better paths.

Counterexample seen in class: $s \to a$ with weight 2, $s \to b$ with weight 3, $b \to a$ with weight -2 ...

4.4 The algorithm

ALGORITHM

Dijkstra's algorithm

```
1  function Dijkstra(G, s) returns the distances d and the predecessors pred
2      for each v ∈ V do
3          d[v] ← ∞, pred[v] ← nil
4      d[s] ← 0
5      Q ← priority queue with all the vertices of V, keyed by d
6      while Q ≠ ∅ do
7          u ← Extract-Min(Q)                          // d[u] is now final
8          for each edge (u, v) in the outgoing edges of u do
9              Relax(u, v)
10     return d, pred

11 function Relax(u, v) returns nothing
12     if d[u] + w(u, v) < d[v] then
13         d[v] ← d[u] + w(u, v), pred[v] ← u
```

The same algorithm in Python is shown below. The graph is a dictionary that maps each vertex to the list of pairs (neighbour, weight).

PYTHON

Dijkstra's algorithm

```python
import heapq

def dijkstra(graph, s):
    dist = {v: float("inf") for v in graph}
    pred = {v: None for v in graph}
    dist[s] = 0
    heap = [(0, s)]
    while heap:
        du, u = heapq.heappop(heap)
        if du > dist[u]:
            continue  # outdated entry, u is already final
        for v, w in graph[u]:
            if du + w < dist[v]:
                dist[v] = du + w
                pred[v] = u
                heapq.heappush(heap, (dist[v], v))
    return dist, pred
```

On the example graph of fig. 4.1, with source s, the function returns $d[a] = 3$, $d[b] = 1$ and $d[c] = 4$.

REMARK

Priority queue in Python

The heapq module has no operation that decreases the key of an entry. Instead, the code pushes a new entry each time a distance decreases, and it skips the outdated entries when they are popped. The heap thus plays the role of Q in the pseudocode.

4.5 Cost

The cost depends on the data structure used for the priority queue Q.

- With a binary heap: $O((V + E) \log V)$.
- With an array: $O(V^2)$.

REMARK

Correction of the sources

The slides' comments mention an $O(E \log E)$ bound for the array version: this is a

typo. The correct bound is $O(V^2)$. The V extractions each scan an array of V entries, and the E relaxations cost $O(1)$ each; since $E \leq V^2$ in a simple graph, the total is $O(V^2)$.

KEY POINT

Which data structure should we use? A binary heap is better than an array when the graph is sparse, that is, when E is much smaller than V^2.

4.6 Negative weights

The assumption $w(u, v) \geq 0$ is essential. The algorithm relies on the fact that an extracted vertex is never improved again. With a negative edge this fails: a path found later can reach an already extracted vertex with a smaller distance. The slides therefore suggest Bellman-Ford when some weight is negative.

FROM MY NOTES

With negative weights, an edge we process later could lower a distance that was already "closed".

KEY POINT

Can Dijkstra's algorithm handle negative weights? No: once a vertex is extracted its distance is final, and a negative edge can make that distance wrong. For graphs with negative weights, use Bellman-Ford.

4.7 Summary

Technique	Idea	Cost
Dijkstra with a binary heap	Extract the vertex with the smallest estimate and relax its edges	$O((V + E) \log V)$
Dijkstra with an array	Find the minimum by scanning an array	$O(V^2)$

KEY POINT

Dijkstra's algorithm computes the distances from the source when all the weights are nonnegative. It repeatedly extracts the vertex with the smallest estimate, which is then final, and relaxes its outgoing edges. With negative weights it can fail.

EXERCISE

Exercise from the slides

Why can Dijkstra's algorithm fail with negative weights? Give a counterexample.

Proposed solution. Take the edges $s \to a$ with weight 2, $s \to b$ with weight 3 and $b \to a$ with weight -2. Dijkstra extracts s, then a with $d[a] = 2$, and only then b. When b is extracted, the relaxation of $b \to a$ would give $3 + (-2) = 1 < 2$, but a has already been extracted and its estimate is never updated again. The algorithm returns $d[a] = 2$, while the true distance is 1, achieved by the path $s \to b \to a$.