

CSCI 1810 — Shortest Paths: Dijkstra's Algorithm and Topological Sorting on DAGs

Lecture notes, October 20, 2026

1 Motivation: longest and shortest paths

- **Longest paths in directed graphs.** Scoring of max score paths $\equiv$ alignments.
 - In general: NP-complete problems, intractable $\equiv$ only brute force algs. are available $\equiv$ exponential time algorithms.
- **Shortest path algorithms:** polynomial time algorithms.
- Shortest paths on edit-like graphs can be solved optimally in *linear time*.

Edit-like graphs, **DAG** = **d**irected **a**cyclic **g**raphs.

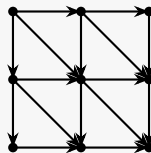
On DAGs we solve

$\left. \begin{array}{l} \text{longest path} \\ \text{shortest path} \end{array} \right\} \text{ in linear time.}$

Key concept: Topological Order (Section 4).

Added note

An edit graph for aligning two sequences is a grid in which every edge goes right, down, or diagonally down-right. No edge ever goes back up or left, so the graph has no cycles: it is a DAG. That is why alignment, a longest (max-score) path problem, can be solved in linear time in the size of the graph, even though longest path in a general directed graph is NP-complete.



2 Dijkstra's algorithm

Dijkstra's algorithm finds the shortest path from a fixed node, called the *source*, to every other node.

2.1 Setup

- $G = \langle N, A \rangle$ directed graph.
- N = the set of nodes/vertices.
- A = set of edges; each edge has a **non-negative cost**.
- $N = \{1, 2, \dots, n\}$, 1 = source node.

Problem. GIVEN: $G = (N, A)$, $L[i, j]$ = cost of edge (i, j) .

COMPUTE: The cost of the shortest paths from the source to each of the other nodes in G .

The cost matrix is

$$L[i, j] \geq 0 \text{ if } (i, j) \in A, \quad L[i, j] = \infty \text{ if } (i, j) \notin A.$$

2.2 The two sets S and C , and the array D

C, S : two sets of nodes.

- S = “**special**” nodes = nodes already chosen.
- C = the set of **candidate** nodes.

At any step S contains all the nodes whose *minimal* cost from the source node is already known (computed).

C = the rest of the nodes.

At any step we choose a candidate node in C whose cost to the source node is the smallest cost, and add it to S .

Definition 1. A path from the source to a node v is **special** if all intermediary nodes are in S .

Array D contains the cost of the shortest special path to each node in G .

Key fact (from the board). When we add a new node v from C to S , then the shortest special path to v is also the shortest path in the graph.

This is the **greedy choice step** of the algorithm: we commit to v and never revise $D[v]$ afterwards.

2.3 The algorithm

Algorithm 1 DIJKSTRA'S ALGORITHM

INITIALIZATION

```
1:  $S \leftarrow \{1\}$ 
2:  $C \leftarrow \{2, 3, \dots, n\}$ 
3: for  $i = 2$  to  $n$  do
4:    $D[i] \leftarrow L[1, i]$ 
5:    $P[i] \leftarrow 1$  ▷ predecessor array
6: end for
```

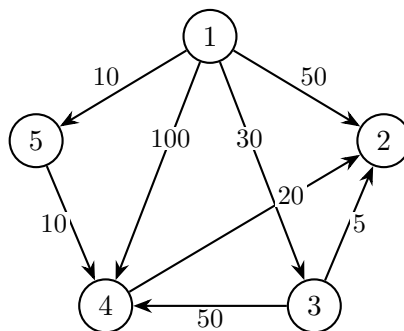
GREEDY LOOP

```
7: repeat  $n - 2$  times
8:    $v \leftarrow$  node in  $C$  with minimum value  $D[v]$ 
9:    $S \leftarrow S \cup \{v\}$ 
10:   $C \leftarrow C \setminus \{v\}$ 
11:  for each  $w \in C$  do
12:    if  $D[w] > D[v] + L[v, w]$  then
13:       $D[w] \leftarrow D[v] + L[v, w]$ 
14:       $P[w] \leftarrow v$ 
15:    end if
16:  end for
17: end repeat
```

Added note

- **Why $n - 2$ times?** C starts with $n - 1$ nodes. After $n - 2$ iterations only one node is left in C . Its D value is already the shortest special path through all other nodes, so it is final and nothing more needs to be done.
- **The array P .** $P[w]$ is the node just before w on the shortest path found so far. At the end, the path to any node w is read backwards: $w, P[w], P[P[w]], \dots, 1$.
- **Running time** of this array version: the loop runs $O(n)$ times and each iteration scans C , so $O(n^2)$ overall. With a priority queue (heap) it is $O((|N| + |A|) \log |N|)$.
- **Non-negative costs are required.** With a negative edge, a node already in S could later be reached more cheaply, which breaks the key fact above.

2.4 Worked example



Edge directions are drawn so that they match the table below (the direction of a few arrows was hard to read in the photos).

Step	v	C	$D = [D[2], D[3], D[4], D[5]]$
initialization	—	$\{2, 3, 4, 5\}$	$[50, 30, 100, 10]$
1	5	$\{2, 3, 4\}$	$[50, 30, 20, 10]$
2	4	$\{2, 3\}$	$[40, 30, 20, 10]$
3	3	$\{2\}$	$[35, 30, 20, 10]$

Added note

- Step 1: $v = 5$ (smallest, $D[5] = 10$). Via 5, node 4 costs $10 + 10 = 20 < 100$, so $D[4] = 20$, $P[4] = 5$.
- Step 2: $v = 4$ ($D[4] = 20$). Via 4, node 2 costs $20 + 20 = 40 < 50$, so $D[2] = 40$, $P[2] = 4$.
- Step 3: $v = 3$ ($D[3] = 30$). Via 3, node 2 costs $30 + 5 = 35 < 40$, so $D[2] = 35$, $P[2] = 3$.
- Stop after $n - 2 = 3$ iterations. Final shortest paths: $1 \rightarrow 3 \rightarrow 2$ (35), $1 \rightarrow 3$ (30), $1 \rightarrow 5 \rightarrow 4$ (20), $1 \rightarrow 5$ (10).

3 Proving correctness of algorithms

How do you prove correctness of programs/algorithms?

- **Invariants:** a statement that is true before the loop starts and stays true after every iteration.
- **Termination criteria:** the program must always terminate, and terminate with the right result.

There are games where, according to game theory, always the first or always the second player wins; the winning strategy is found by keeping an invariant (see Dijkstra's coin game in the reading, linked below).

Invariant (holds after every iteration of the greedy loop):

- (a) for every node in S , D is the cost of the shortest path from the source (shortest overall);
- (b) for every node in C , D is the cost of the shortest *special* path.

It holds at initialization: $S = \{1\}$, and the only special paths are single edges $1 \rightarrow i$, with cost $L[1, i]$.

Why (a) is kept (the key fact). Let v be the node in C with minimum $D[v]$. Suppose some path to v were shorter than $D[v]$. It must leave S somewhere; let x be the first node on it that is in C . The part up to x is a special path, so it costs at least $D[x] \geq D[v]$. The rest of the path has non-negative cost, so the whole path costs at least $D[v]$, a contradiction.

Why (b) is kept. Once v joins S , a new special path to $w \in C$ can use v as its last intermediate node. The update $D[w] \leftarrow \min(D[w], D[v] + L[v, w])$ checks exactly this.

Termination. The loop runs a fixed $n - 2$ times, so the algorithm always stops. At that point every node is in S (or is the last node of C , whose shortest special path is the shortest path), so D holds the answer.

Reading: S. Istrail, “Storytelling About Lighthouses: When Professor Dijkstra Slapped Me in the Quest for Beautiful Code,” *Conduit* (Brown CS), 2010. https://brown-cs181.github.io/resources/eDavidsonLectures/conduit_v19n1%202.pdf

4 Topological sorting

Definition 2. A **topological sort** or **topological ordering** of a directed graph is a linear ordering of the vertices such that for every directed edge from vertex u to vertex v , vertex u comes before v .

- A topological ordering is possible **if and only if** the graph is a DAG, i.e. it has no directed cycles.
- A topological ordering, if it exists, can be found in **linear time**.

Example. For the graph of the Dijkstra example, one topological order is

1, 3, 5, 4, 2,

and every edge points left-to-right in this list.

4.1 Topological sorting algorithm for a DAG $G = (V, E)$

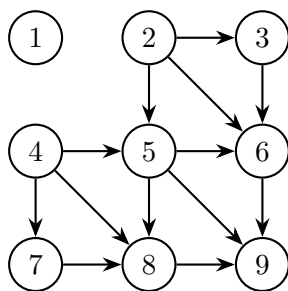
Algorithm 2 TOPOLOGICAL SORTING ALGORITHM for DAG $G = (V, E)$

```
1:  $L \leftarrow$  the list of topologically sorted vertices of  $V$   $\triangleright$  starts empty
2:  $S \leftarrow$  set of vertices with no incoming edges
3: while  $S \neq \emptyset$  do
4:   remove node  $u$  from  $S$ 
5:   add  $u$  to  $L$  at the tail
6:   for each node  $v$  with edge  $e = (u, v) \in E$  do
7:     remove edge  $e$  from graph
8:     if  $v$  has no incoming edges then
9:       put  $v$  into  $S$ 
10:    end if
11:  end for
12: end while
```

Added note

- **Linear time.** Each vertex enters S once and each edge is removed once, so the running time is $O(|V| + |E|)$. In practice, edges are not physically deleted; each vertex keeps a counter of its remaining incoming edges, decreased by one when an edge is “removed.”
- **Detecting cycles.** If the loop ends while edges are still left in the graph, the graph had a directed cycle and no topological order exists.

4.2 Example



The edge set is reconstructed from the photos so that the algorithm produces the same L as on the board.

Initially $S = \{1, 2, 4\}$ (no incoming edges). Running the algorithm:

	remove u	new vertices put into S	L
1	1	—	1
2	2	3	1, 2
3	4	5, 7	1, 2, 4
4	3	—	1, 2, 4, 3
5	5	6	1, 2, 4, 3, 5
6	7	8	1, 2, 4, 3, 5, 7
7	6	—	1, 2, 4, 3, 5, 7, 6
8	8	9	1, 2, 4, 3, 5, 7, 6, 8
9	9	—	1, 2, 4, 3, 5, 7, 6, 8, 9

$$L = \{1, 2, 4, 3, 5, 7, 6, 8, 9\}.$$

4.3 Shortest and longest paths on a DAG in linear time

Added note

Once the vertices are in topological order, process them left to right and relax each outgoing edge once:

- 1: $D[s] \leftarrow 0$; $D[v] \leftarrow \infty$ for all $v \neq s$
- 2: **for** each u in topological order **do**
- 3: **for** each edge $(u, v) \in E$ **do**
- 4: **if** $D[v] > D[u] + L[u, v]$ **then**
- 5: $D[v] \leftarrow D[u] + L[u, v]$; $P[v] \leftarrow u$
- 6: **end if**
- 7: **end for**
- 8: **end for**

When u is processed, every edge into u comes from an earlier vertex, so $D[u]$ is already final. Every edge is looked at once, giving $O(|V| + |E|)$ time. Unlike Dijkstra, this works with *negative* costs too.

For the **longest** (max-score) path, use $-\infty$ instead of ∞ and $<$ instead of $>$ (or negate all costs and find the shortest path). On an edit graph this is exactly the dynamic program for alignment.