Week 10: BFS & Shortest Paths

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

November 24, 2025

Agenda

1. Mini-Quiz
2. Assignment
3. Breadth-First Search
4. Weighted Graphs
5. Shortest Paths in DAGs
6. Dijkstra's Algorithm
7. Additional Practice

Mini-Quiz

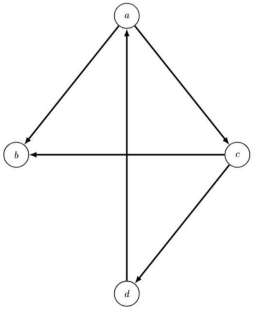
Q1

If a directed graph has no directed cycles then there is no back edge in any DFS tree of it.

- (●) True
- ( ) False

Q2

Consider the graph below.



What is a valid adjacency list for vertex c?

- a. $[a]$
- b. $[b, d]$

Q3

Decide whether the following are true or false.

True	False	
○	◉	If a directed graph has a sink, then it has no directed cycle.
◉	○	If a directed graph has no directed cycle, then it has a sink.

Scoring method: **Subpoints** ?

Q4

Let u, v be two vertices connected via an edge in an **undirected graph**. Suppose we compute a DFS tree of this graph. It is possible that $pre(u) < post(u) < pre(v) < post(v)$.

- ○ True
- ◉ False

Q5

Suppose we have a directed graph with 2 different directed cycles. We need to remove at least 2 edges for a topological order to exist in the graph.

- ○ True
- ◉ False

Q6

Recall the notation S_i^u used to refer to the set of vertices at distance i from some vertex u, in some directed graph D. Assume that there exists a vertex w with $w \in S_2^u$ and $w \in S_3^v$. Then, the graph distance from u to v in D is at most 5.

- ○ True
- ◉ False

Q7

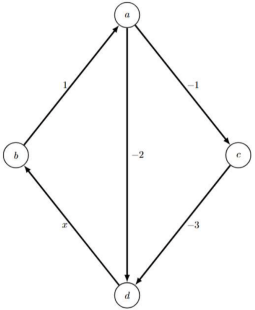
Suppose that we run a BFS on a directed graph and after sorting by the enter-time the vertices are d, e, b, a, c, f. What would the order be if we sorted by leave-time instead?

- a. f, c, a, b, e, d.
- b. d, e, b, a, c, f.
- c. a, b, c, d, e, f.
- d. f, e, d, c, b, a.
- e. Impossible to specify with the current information.

Clear my choice

Q8

Consider the graph below.

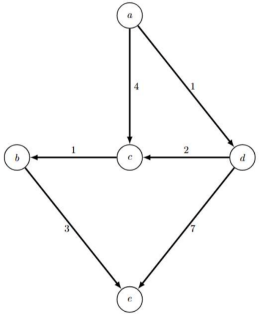


What is the minimum value of x for which there is no negative directed cycle?

Answer: 3

Q9

Consider the graph below.



What is the length of the shortest path from a to e?

Answer: 7

Q10

Consider the queue $Q = [3, 5, 2, 4, 1]$. Suppose we carry out the following operations:

1. dequeue
2. dequeue
3. enqueue(1)
4. enqueue(4)

What is the final state of the queue? The enqueue operation adds an element to the right/end of the queue.

- a. $Q = [3, 5, 2, 4, 1]$
- b. $Q = [3, 5, 2, 1, 4]$
- c. $Q = [2, 4, 1, 1, 4]$
- d. $Q = [2, 4, 1, 4, 1]$

Clear my choice

Assignment

Feedback Assignments 8

Well done overall! Some common points:

- **Ex. 8.1 (Handshake Lemma)**: Well done! Just make sure that you first **explicitly model the situation as a graph** in order to apply graph theory. Otherwise, you are dealing with undefined quantities.
- **Ex. 8.3**
  - **Be precise**, e.g., define any specific neighbors you talk about and make your reasoning explicit.
  - **Use the terminology** introduced in the lecture. For instance, when arguing about connectedness, you can argue in terms of connected components, which have a precise definition and for which you can use all the properties shown in the lecture.
  - Even though these simpler proofs lend themselves to intuitive argument, be rigorous and **practice formalizing** them. Be **careful with intuitive arguments** (when too vague) and **proofs by example** or visual sketches for existence proofs (for counterexample a visual example is sufficient).

Remember to check the detailed feedback on Moodle! Reach out if you have any questions regarding the corrections.

Ex. 9.4

Exercise 9.4 *Bipartite graphs, Eulerian graphs and painting rooms* **(2 points)**.

In this exercise, you can use Theorem 1 above.

(a)* Prove Theorem 1.

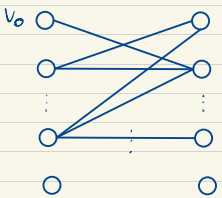
There is a much simpler proof than shown in the master solution, which uses BFS. We'll see this later!

Ex. 9.4

(b) Prove or disprove the following statements:

(1) Every graph G that is bipartite and Eulerian must have an even number of edges.

Proof: Let $G = (A \dot\cup B, E)$ be a bipartite graph with partition A, B.



Since G is Eulerian, there exists a closed Eulerian walk.
Let v_0 be the starting vertex and w.l.o.g. let $v_0 \in A$.

Now suppose for sake of contradiction that $|A \dot\cup B|$ was odd.

Notice first that $\forall e \in E\ (|e \cap A| = 1)$, i.e., every edge flows between A and B by def. of bipartiteness.

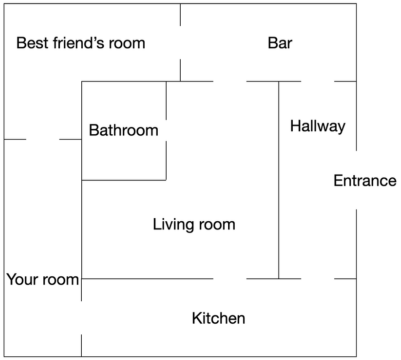
Then since $|A \dot\cup B|$ is odd and we follow a closed walk of length $|A \dot\cup B|$, we find $v_0 \in B$, a contradiction.

It follows that $|A \dot\cup B|$ is even. ∎

Ex. 9.4

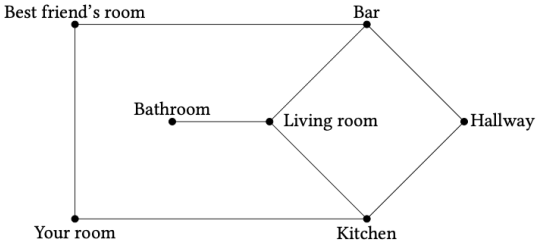
(c) You recently moved in with your best friend (see floor plan below) and you would like to repaint the room walls. Every room should be painted either in red or in purple (as these are your favorite colors), and you also would like that whenever you walk from a room to another room through a door, the color changes. Is that possible?

Note that there are 7 rooms (i.e. the Hallway, the Bathroom and the Kitchen are counted as rooms).



Ex. 9.4

We first model the above floor plan as the following graph, where the vertices represent the different rooms of the flat, and two vertices are conncted with an edge whenever there is a door between the two corresponding rooms.



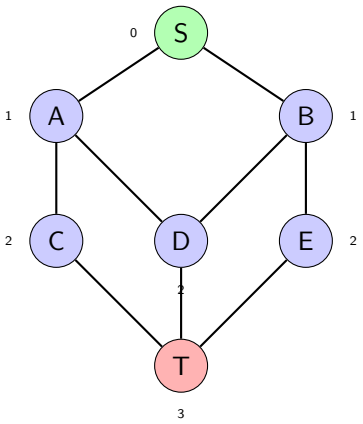
We check if the graph is 2-colorable, i.e., bipartite.

By Thm. 1, this graph cannot be bipartite since it contains an odd-length cycle. □

Breadth-First Search

Motivation: Finding Shortest Paths in Unweighted Graphs

Problem: Find the shortest path from a start vertex to all other vertices.



Numbers = distance from S

Key observation:

- In an **unweighted** graph, shortest path = path with **fewest edges**
- DFS doesn't guarantee shortest paths!
- We need to explore **level by level**

Solution: Breadth-First Search (BFS)

BFS: The Idea

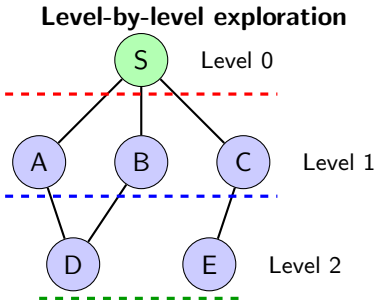
Strategy: Explore vertices in order of their distance from the start

Algorithm sketch:

1. Start at vertex s, mark distance 0
2. Explore all neighbors of s (distance 1)
3. Then explore all *their* neighbors (distance 2)
4. Continue level by level...

Data structure: Queue (FIFO)

- **Enqueue:** Add to back
- **Dequeue:** Remove from front



Key property: BFS visits vertices in order of increasing distance!

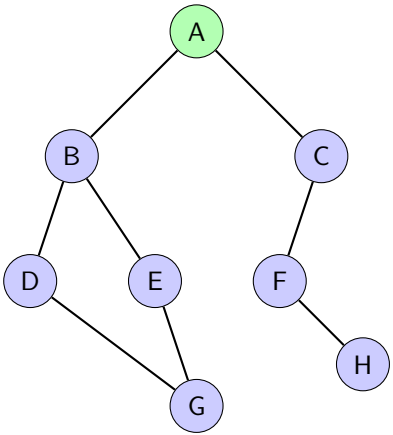
BFS: Pseudocode

```
1:  function BFS(G = (V, E), s ∈ V)
2:      Initialize dist[v] ← ∞ for all v ∈ V
3:      dist[s] ← 0
4:      Q ← {s}                                   ▷ Initialize queue with start vertex
5:      while Q ≠ ∅ do
6:          u ← dequeue(Q)
7:          for each edge (u, v) ∈ E do
8:              if v not yet visited (i.e., dist[v] = ∞) then
9:                  dist[v] ← dist[u] + 1
10:                 enqueue(Q, v)
```

Key observations

- Each vertex enters the queue **at most once**
- Vertices are dequeued in order of increasing distance
- **Runtime**: $O(|V| + |E|)$ with adjacency list (analogously to DFS)

BFS: Visual Example



Execution trace:

Step	Queue	Process
0	{A}	-
1	{B, C}	A
2	{C, D, E}	B
3	{D, E, F}	C
4	{E, F, G}	D
5	{F, G}	E
6	{G, H}	F
7	{H}	G
8	{}	H

Distances from A:

$\text{dist}[A] = 0$ $\quad \text{dist}[E] = 2$

$\text{dist}[B] = 1$ $\quad \text{dist}[F] = 2$

$\text{dist}[C] = 1$ $\quad \text{dist}[G] = 3$

$\text{dist}[D] = 2$ $\quad \text{dist}[H] = 3$

BFS vs DFS: Comparison

	BFS	**DFS**
Data Structure	Queue (FIFO)	Stack (LIFO) / Recursion
Exploration	Level-by-level	Deep-first, then backtrack
Shortest Paths	(unweighted)	
Space Complexity	$O(\|V\|)$	$O(\|V\|)$
Time Complexity	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
Applications	Shortest paths	Topological sort
	Connected components	Cycle detection
		Path existence

Rule of thumb: Use BFS when you care about *distance*, use DFS when you care about *structure* (cycles, connectivity, ordering).

Proof of Theorem 1

Theorem 1. A graph is bipartite if and only if it does not contain any cycle of odd length.

Proof.

Hint: Use BFS to color the vertices.

Proof of Theorem 1

Theorem 1. A graph $G = (A \uplus B, E)$ is bipartite if and only if it contains no odd cycles.

($\Rightarrow$) Odd cycle implies not bipartite

- Let C be an odd cycle $(v_1, v_2, \ldots, v_k, v_1)$ (where k is odd).
- In a valid bicoloring, adjacent vertices must flip colors.
- Flipping colors an odd number of times means v_1 and v_k have the same color.
- The edge (v_k, v_1) therefore connects two vertices of the same color. G cannot be bipartite.

($\Leftarrow$) No odd cycles implies bipartite

Assume G does not contain any odd cycles.

- Run BFS and color layer L_i based on parity of i.
- Suppose for contradiction that some edge $(u, v) \in E$ connects nodes of the same color.
- *Note:* Edges only link $L_i \leftrightarrow L_{i+1}$ or $L_i \leftrightarrow L_i$.
- Same color $\implies$ same parity $\implies$ u, v must be in the **same layer** L_i.
- Let w be the lowest common ancestor of u and v in the BFS tree and notice that the paths $w \rightsquigarrow u$ and $w \rightsquigarrow v$ have the same length d.
- Cycle $w \rightsquigarrow u \to v \rightsquigarrow w$ has odd length $d + 1 + d = 2d + 1$, a contradiction.

Hence, we never color two vertices with the same color and G is bipartite $\square$

Weighted Graphs

From Unweighted to Weighted Graphs

Real-world motivation: Not all edges are equal!

Examples:

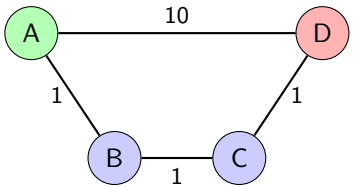
- Road networks: distances vary
- Flight routes: costs/times differ
- Networks: bandwidth varies

Definition

A **weighted graph** $G = (V, E, c)$ has a cost function:

$$c : E \to \mathbb{R}$$

assigning a cost $c(e)$ to each edge $e \in E$.



Unweighted: Shortest path is (A, D) (1 edge)
Weighted: Shortest path is (A, B, C, D) (cost 3)
Fewer edges doesn't mean lower cost!

Important: BFS no longer works! We need different algorithms.

Shortest Paths: Problem Definition

Definition: Path Cost

For a path $P = (v_0, v_1, \ldots, v_\ell)$, the **cost** is:

$$c(P) := c(v_0, v_1) + c(v_1, v_2) + \cdots + c(v_{\ell-1}, v_\ell) = \sum_{i=0}^{\ell-1} c(v_i, v_{i+1})$$

Definition: Distance

The **distance** from u to v is:

$$d(u, v) := \min\{c(P) \mid P \text{ is a path from } u \text{ to } v\}$$

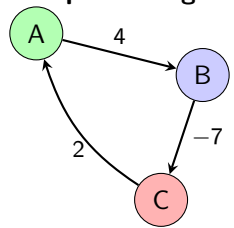
If no path exists, $d(u, v) = \infty$.

Key question: What if edge costs can be **negative**?

The Problem with Negative Edge Costs

Negative edges complicate things!

Example 1: Negative cycle

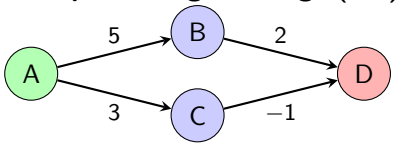


Cycle cost: $4 + (-7) + 2 = -1$

Going around the cycle decreases cost!

$\Rightarrow$ No shortest path exists

Example 2: Negative edge (OK)



Shortest path (A, D): (A, C, D) (cost 2)

This is fine—no negative cycles!

Key insight: Negative *edges* are OK, but negative *cycles* make shortest paths undefined

Assumption for Today: Non-Negative Edge Costs

For the rest of this lecture, we assume:

$$c(e) \geq 0 \quad \text{for all } e \in E$$

Why this assumption helps

- Guarantees shortest paths exist (if paths exist at all)
- Enables greedy algorithms like Dijkstra
- In many settings a reasonable assumption

What about graphs with negative edges but no negative cycles?

- Dijkstra's algorithm **fails**

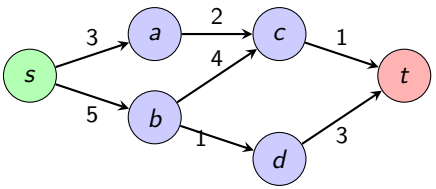
  $\Longrightarrow$ This week's lecture: Bellman-Ford

Shortest Paths in DAGs

Shortest Paths in DAGs

Recall from last week:

Question: What if our weighted graph is a DAG (Directed Acyclic Graph)?



Key advantage:

- No directed cycles $\Rightarrow$ we have a topological ordering!
- Remember: If $(v_i, v_j) \in E$, then $i < j$ in the ordering

Intuition:

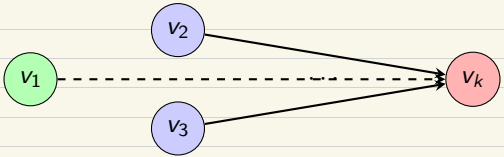
- Process vertices left-to-right
- When we process v_j, all predecessors are already done

Can we exploit this structure to find shortest paths efficiently?

Your Turn: Derive a Recurrence

Task: Suppose we have a topological ordering $v_1, v_2, \ldots, v_n$ of our DAG, with $v_1 = s$ (start vertex).

Question: How can we express $d(s, v_k)$ (shortest distance to v_k) in terms of distances to vertices that come *before* v_k in the ordering?



Hints:

- Any path to v_k must come from some predecessor
- In topological order, all predecessors of v_k have index $< k$
- What's the base case?

Time: 3-4 minutes

Solution: Recurrence for DAG Shortest Paths

Recurrence

For a DAG with topological ordering $v_1, v_2, \ldots, v_n$ where $v_1 = s$:

Base case:

$$d(s, v_1) = d(s, s) = 0$$

Recursion: For $k \geq 2$,

$$d(s, v_k) = \begin{cases} \min\limits_{\substack{(v_i, v_k) \in E \\ i < k}} \{d(s, v_i) + c(v_i, v_k)\} & \text{if } v_k \text{ has predecessors} \\ \infty & \text{if } v_k \text{ unreachable from } s \end{cases}$$

Correctness (informal):

- Any path to v_k must pass through some predecessor v_i (with $i < k$)
- The shortest such path is the minimum over all predecessors

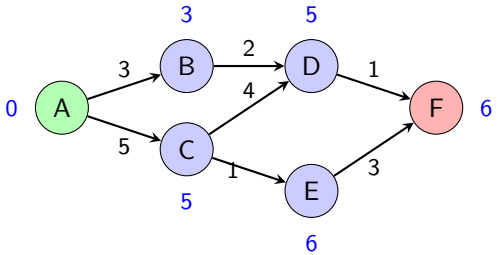
Algorithm: Shortest Paths in DAGs

```
1: function DAG-ShortestPath(G, s)
2:     Topologically sort V: v_1, ..., v_n
3:     Initialize d[v] ← ∞, d[s] ← 0
4:     for u in topological order do
5:         for each neighbor v of u do                ▷ Outgoing edges
6:             d[v] ← min(d[v], d[u] + c(u, v))
```

Runtime Analysis:

- Topological sort: $O(|V| + |E|)$
- Main loop: $\sum_{v \in V} 1 + \deg_{in}(v) = O(|V| + |E|)$
- **Total:** $O(|V| + |E|)$

Example: DAG Shortest Paths



Topological order: A, B, C, D, E, F

Process in order:

1. A: $d[A] = 0$
2. B: $d[B] = d[A] + 3 = 3$
3. C: $d[C] = d[A] + 5 = 5$
4. D:
   - Via B: $d[B] + 2 = 5$
   - Via C: $d[C] + 4 = 9$
   - $d[D] = 5$
5. E: $d[E] = d[C] + 1 = 6$
6. F:
   - Via D: $d[D] + 1 = 6$
   - Via E: $d[E] + 3 = 9$
   - $d[F] = 6$

Result: Shortest path to F is (A, B, D, F) with cost 6

Key Insight: Why DAGs are Easy

The crucial observation:

Subproblem Ordering

In a DAG with topological ordering, we have a **natural order** to process vertices:

- When computing $d(s, v_k)$, all dependencies $d(s, v_i)$ with $i < k$ are *already computed*
- Dynamic Programming!

What we need:

1. A recurrence relation
2. An ordering that respects dependencies

In DAGs:

1. We derived the recurrence
2. Topological order gives us this for free!

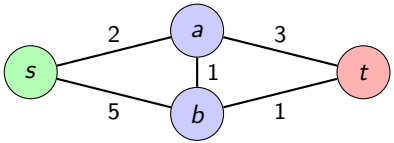
Question: What if the graph has cycles? Can we still use this approach?

Dijkstra's Algorithm

The Challenge: Undirected Graphs

So far: DAGs ⇒ topological ordering ensures all predecessors processed before each vertex

But what about undirected graphs (or directed graphs with cycles)?



No topological ordering possible

The problem:

- Every vertex can be reached from every other vertex (in cycles)
- No "natural" ordering that guarantees all predecessors are processed

What do we do?

- We need a *different* invariant
- Cannot rely on structure
- Must enforce correct processing order ourselves!

Challenge: How can we ensure we've found the shortest path to each vertex?

Dijkstra's Algorithm: The Idea

Solution: Enforce a *stronger* invariant — process vertices in order of **increasing distance**!

Dijkstra's Invariant

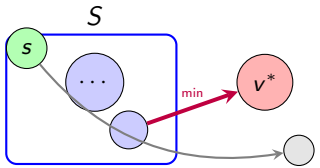
Maintain a set S of vertices for which we've determined the shortest distance.

Key property: Always add the *closest* unprocessed vertex to S next.

This ensures: $d(s, v_1) \leq d(s, v_2) \leq \cdots \leq d(s, v_n)$ for processing order.

Why this works:

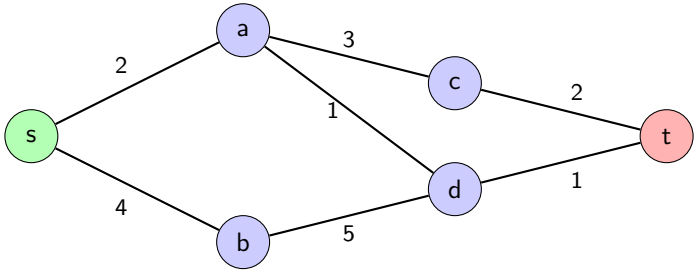
- When we add v to S with distance d
- Any other path to v must leave S at some point
- That path has length $\geq d$ (by non-negative weights and our greedy choice)
- So d is indeed the shortest distance!



Always pick closest unprocessed vertex

Your Turn: Run Dijkstra's Algorithm

Task: Run Dijkstra's algorithm on the graph below starting from s. Show the order in which vertices are added to S and the final distances.



Think about: How is this similar to the DAG approach? How is it different?

Time: 5-6 minutes

Comparison: DAG vs. Dijkstra

	DAG Algorithm	**Dijkstra's Algorithm**
Processing order	Topological order	Strictly increasing distance from s
Guarantee	All predecessors processed before each vertex	All closer vertices processed before each vertex
Why correct?	No "later" vertex can provide shortcut (DAG structure)	No "farther" vertex can provide shortcut (non-negative weights + greedy choice)
Finding order	Free! (topological sort)	Must build greedily
Recurrence	$d(s, v_k) = \min_{(v_i, v_k), i<k}\{d(s, v_i) + c(v_i, v_k)\}$	

Key insight: Both use the same recurrence, but Dijkstra needs a *stricter* processing order because it lacks the DAG structure!

Why Building the Ordering is Correct

Lemma (assuming non-negative costs)

Let S be the set of the $k-1$ closest vertices to s. Let $v^* \notin S$ be the vertex with the minimum tentative distance.

Claim: $d[v^*]$ is the shortest path to v^*, **and therefore v^* is the k-th closest vertex.**

Proof Sketch: Let P be **any** path from s to v^*.

1. Path P starts in S and ends outside, so it must cross the boundary.
2. Let (u, v) be the **first edge** in P where $u \in S$ and $v \notin S$.
3. We lower-bound the cost of P:

$$\begin{aligned} c(P) &\geq d(s,u) + c(u,v) \quad \text{(non-negative weights)} \\ &\geq d[v] \quad \text{(definition of tentative distance)} \\ &\geq d[v^*] \quad \text{(Greedy Choice: we picked } v^* \text{ over } v) \end{aligned}$$

4. **Conclusion:** No path to v^* is shorter than $d[v^*]$. Since v^* is the closest of the remaining vertices, it is the k-th closest overall. □

Correctness of The Priority Queue

Recall our recurrence:

$$d(s, v_k) = \min_{\substack{u \in S \\ u \to v_k}} \{d(s, u) + c(u, v_k)\}$$

This says: to compute shortest distance to v_k, we need to consider **all edges** from vertices in S to v_k.

In Dijkstra's algorithm, we do this efficiently using a priority queue (min-heap):

- We extract one vertex v^* at a time from the priority queue
- We only relax edges *from that one vertex*
- We never explicitly enumerate all edges from S to find the minimum

Question: How do we know that v^* is the vertex that achieves the minimum in the recurrence?
Why is extracting one vertex at a time sufficient?

Dijkstra: The Invariant

Key insight: The priority queue maintains tentative distances that already incorporate *all* edges from S seen so far.

Invariant (maintained throughout algorithm)

At any point, for each vertex $v \notin S$:

$$d[v] = \min_{\substack{u \in S \\ (u,v) \in E}} \{d[u] + c(u, v)\}$$

That is, $d[v]$ stores the **best distance to v using any edge from S**.

Why this is maintained:

1. Initially: $S = \emptyset$, $d[s] = 0$, all others $d[v] = \infty$
2. When we add vertex u to S:
   - We relax all edges (u, v): update $d[v] = \min\{d[v], d[u] + c(u, v)\}$
   - This incorporates the new edges from u into all tentative distances
3. So $d[v]$ always reflects the minimum over *all* $u \in S$

Why Extracting the Minimum is Correct

When we extract v^* with minimal $d[v^*]$:

Two crucial properties

1. $d[v^*] = \min_{u \in S, (u,v^*) \in E}\{d[u] + c(u, v^*)\}$ (by invariant) $\implies$ correct distance
2. $d[v^*] \leq d[v]$ for all $v \notin S$ (by extraction from min-heap)

What this means:

- Property (1): $d[v^*]$ already accounts for *all possible ways* to reach v^* from S
- Property (2): v^* is the closest unprocessed vertex
- Together: v^* is exactly the vertex that achieves the minimum in our recurrence!

Key point: By maintaining the invariant incrementally, the priority queue always gives us the next vertex in distance order, with its correct final distance.

Questions?

Additional Practice

Modified BFS

Exercise 10.4 *Number of minimal paths* **(1 point)**.

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Let $v, v' \in V$ be two distinct vertices and suppose that the distance between the two is k.

Describe an algorithm which counts the number of paths from v to v' of length k. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n, and the adjacency list Adj of G. Argue why your algorithm is correct and why it satisfies the runtime bound.

Hint: *Modify BFS.*

Modified BFS

Key Observation

The number of shortest paths to v_i must equal the sum of all paths of length $i-1$ ending at neighbors of v_i that are at distance $i-1$ from v.

Algorithm: We run a BFS starting from v. We maintain an array $paths[]$ where $paths[u]$ stores the number of minimal paths from v to u and an array $dist[]$, which stores $dist(v,u)$.

- Initialize: $paths[v] = 1$, all others 0; $dist[v] = 0$ all others -1.
- When exploring neighbor w of u:
  - If $dist[w] < 0$: Set $dist[w] = dist[u] + 1$, $paths[w] = paths[u]$ and enqueue w.
  - If $dist[w] = dist[u] + 1$: Update $paths[w] = paths[w] + paths[u]$.

Once BFS finishes, we return $paths[v']$.

Modified BFS - Solution

Recurrence:

Formally, for each vertex v_i at distance i from v, we compute:

$$paths[v_i] = \begin{cases} 1 & \text{if } i = 0 \text{ (i.e., } v_i = v) \\ \sum_{\substack{u \in S_{i-1} \\ (u, v_i) \in E}} paths[u] & \text{if } i \geq 1 \end{cases}$$

where $S_k = \{u \in V \mid \text{dist}(v, u) = k \in \mathbb{N}\}$

Proof of Correctness (1/2)

Proof. We prove that (i) the recurrence is correct and (ii) that BFS proceeds in correct order and terminates.

(1) **Correctness of Recurrence:**

Let $v' \in V$ be a vertex at distance $i \in \mathbb{N}$ from v.

- If $i = 0$, then $v = v'$ and trivially there exists exactly one path.
- For $i > 0$, suppose for sake of contradiction that we would need to consider vertices at distance $t < i - 1$. But then $dist(v, v') < i$, since we can construct a shorter path, a contradiction. Analogously, if we considered $t > i - 1$, we would consider non-minimal paths, a contradiction. Hence, we only need to consider the set S_{i-1}.

Since we can continue every path ending at a neighbor at distance $i - 1$ with exactly one edge to v', summing over the number of paths is correct.

Proof of Correctness (2/2)

(2) **Correctness of Computation Order**

It remains to show that BFS computes the recurrence in correct order and terminates. We proceed by induction on the distance $k \in \mathbb{N}$.

- **B.C.:** Let $k = 0$ and notice that the only vertex at distance 0 is the starting vertex. We correctly initialize this to 1. When BFS dequeues the starting vertex, the number of paths is correct.
- **I.H.:** Assume that for some $k \in \mathbb{N}$ and an arbitrary vertex $v' \in V$ at distance k from v, $paths[v']$ has been correctly computed, when BFS dequeues v'.

Proof of Correctness (2/2)

(2) **Correctness of Computation Order (ctd.)**

- **I.S.:** Now let v' be an arbitrary vertex at distance $k+1$ and assume it has just been dequeued.

  We know from the lecture that

  - **BFS processes** all nodes in S_i **before** processing any node in S_{i+1}, meaning all $v_k \in S_k$ have already been dequeued and processed.

    By I.H. $paths[v_k]$ is correct for all $v_k \in S_k$, in particular for the neighbors of v' at distance k.

  - Since each $v_k \in S_k$ is enqueued and dequeued exactly once, we have already computed $\sum_{\substack{u \in S_{i-1} \\ (u,v_i) \in E}} paths[u]$ correctly (also see Correctness of Recurrence).

  It follows that $paths[v']$ has been computed correctly when we dequeue v'.

Termination: Since the graph is finite and BFS visits each connected vertex exactly once, the algorithm terminates and returns $paths[v']$. □

Runtime

Runtime:

We inherit the runtime from BFS since we only add a constant number of operations per edge, and can extract the solution in constant time. Hence, our algorithm runs in $O(n + m)$

Bonus: Longest Paths in DAGs

Exercise 9.4 (2024)

Let $G = (V, E)$ be a directed graph without directed cycles (i.e., a **directed acyclic graph** or short DAG).

Assume that $V = \{v_1, \ldots, v_n\}$ (for $n = |V| \in \mathbb{N}$) and that the sorting $v_1, v_2, \ldots, v_n$ of the vertices is a **topological sorting**.

The **goal** of this exercise is to find the **longest path** in G.

Exercise Structure

- Part a): Properties of a path in G
- Part b): DP-algorithm for finding a longest path

Part (a) - Order of a Path in G

Part (a)

Let P be a path in G. Prove that if $P = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, then $i_1 < i_2 < \cdots < i_k$.

Part (a) - Order of a Path in G

Part (a)

Let P be a path in G. Prove that if $P = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, then $i_1 < i_2 < \cdots < i_k$.

Proof.

Recall: A topological sorting is a **linear order** of vertices that satisfies all dependencies, i.e., **for any edge** $(v, w) \in E$, we have that v **comes before** w in the sorting.

Notice:

- Given a path $P = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, its edges are $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}) \ldots, (v_{i_{k-1}}, v_{i_k})$.
- Hence, for any vertex v_{i_t}, $t \in [k-1]$, and successor $v_{i_{t+1}}$, we have $i_t < i_{t+1}$.

$\implies$ The sequence of indeces $(i_1, \ldots, i_k)$ is strictly increasing, i.e., $i_1 < i_2 < \cdots < i_k$.

□

Part (b) - Longest-Path Finding

Part (b)

Describe a **bottom-up DP** algorithm that **returns the length of the longest path** in G in $O(|V| + |E|)$ time.

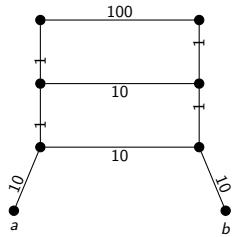
Assume that G is provided as a pair (n, Adj) of the integer $n = |V|$ and the adjacency lists Adj.

Your algorithm can **access** $\text{Adj}[u]$, which is a list of vertices to which u has a direct edge, **in constant time**. Formally, $\text{Adj}[u] := \{v \in V \mid (u, v) \in E\}$.

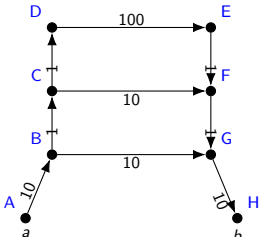
Shortest vs. Longest Paths: Some Intuition

- **Shortest Paths:** Efficiently solvable (e.g., BFS, Dijkstra).
  - Greedy (local optimum $\rightarrow$ global optimum) works.
- **Longest Paths: Much harder...**
  - **Undirected Graphs**:
    - Greedy (heaviest edge) often fails: can be a trap (see example).
    - No simple decision order; may need backtracking.
  - **DAGs**
    - Key: no cycles enable **topological sorting**
    - Topological order guarantees that we have all required information to make optimal decisions

Undirected Graph

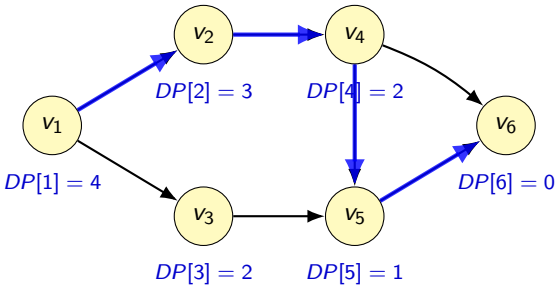


Directed Graph (DAG)



1. **Dimensions of the DP-table:**
   DP$[1 \ldots n]$, where $n = |V|$
2. **Meaning of each entry**
   DP$[i]$ = length of the longest path starting in v_i, $i \in [n]$

   (*Alternatively*: DP$[i]$ = length of the longest path ending in v_i, $i \in [n]$.
   Changes calc. order.)

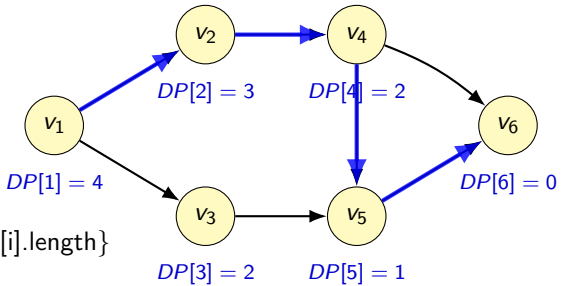


3. **Base Case**

   For sink v_n, we know $\deg_{out}(v_n) = 0$
   $\implies DP[n] = 0$

   Recursion:

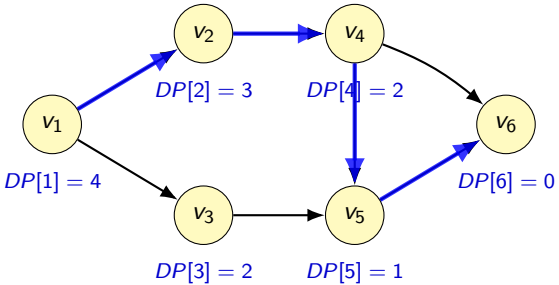
$$DP[i] = \begin{cases} 0, \text{ if Adj[i] == null} \\ 1 + \max\{DP[\text{Adj[i][k]}] \mid 1 \leq k \leq \text{Adj[i].length}\} \end{cases}$$



3. **Correctness:** For $i \in [n]$,

   If v_i is a sink, the longest path starting in v_i has length 0, since there are no outgoing edges.

   Else, the longest path starting in v_i must join a longest path among the successors of v_i, since assuming otherwise leads to a contradiction. □



4. **Calculation Order:**
   For $i = n-1, \ldots, 1$, compute $DP[i]$.

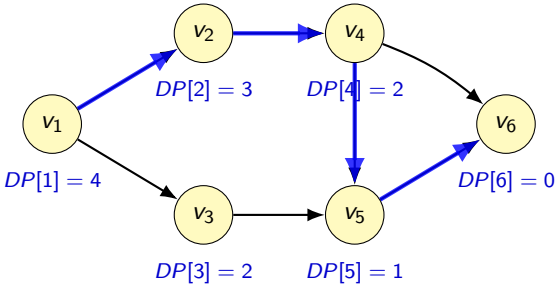
Correctness:

By Part (a), we know that all paths $P = (v_{j_0}, \ldots, v_{j_k})$, satisfy $j_0 < \ldots < j_k$.

Assume P is a longest path of length $>= 1$ starting in vertex v_i (i.e., $v_i = v_{j_0}$).

Notice that since $j_0 < j_1$, $DP[j_1]$ already contains the length of the longest path starting in v_{j_1}.

By the correctness of the recursion, we correctly compute $DP[j_0] = DP[i]$. □



5. **Extracting Solution:**
   Extract solution by iterating over the array and maintaining maximum, i.e., $\max_{i \in [n]} DP[i]$

6. **Runtime:**

Base Case:	$O(1)$	(1)
Recursion:	$O\Big(\sum_{i=1}^{n}(1+\deg_{\text{out}}(v_i))\Big) \overset{\text{Handsh. Lem.}}{\leq} O(\lvert V\rvert + \lvert E\rvert)$	(2)
Extr. Sol:	$O(\lvert V\rvert)$	(3)
Total:	$O(\lvert V\rvert + \lvert E\rvert)$	(4)

(1) Filling in B.C. requires a const. number of operations.

(2) Iterate over $\deg_{out}$ neighbors for each vertex exactly once + extra operations (+1).

(3) Linear pass over vertices to extract max.