

Lecture 23: Shortest Paths

CS 2110

November 12, 2026

Today's Learning Outcomes

80. Identify substructures in graphs such as neighborhoods, paths, and cycles both visually and programmatically.

83. Visualize the state of a graph traversal at a given point of its execution, describing each node as either undiscovered, discovered, visited, and/or settled.

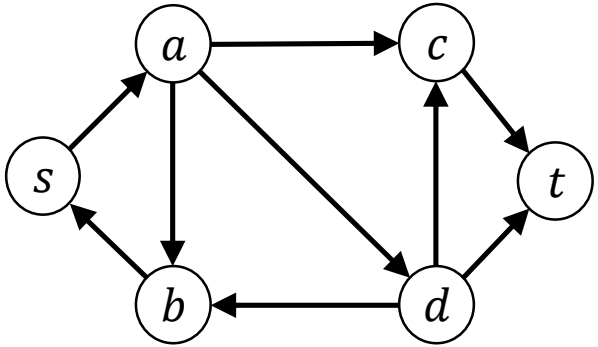
84. Describe Dijkstra's shortest path invariant and use it to establish properties on the unvisited portions of a graph at a given point in the execution of Dijkstra's algorithm.

85. Implement Dijkstra's shortest path algorithm and analyze its time/space complexities.

Traversal Levels

The **level** of a vertex v, $\ell(v)$, in a graph traversal starting at s is the length of the shortest $s \rightsquigarrow v$ path.

*BFS visits vertices in increasing level order.





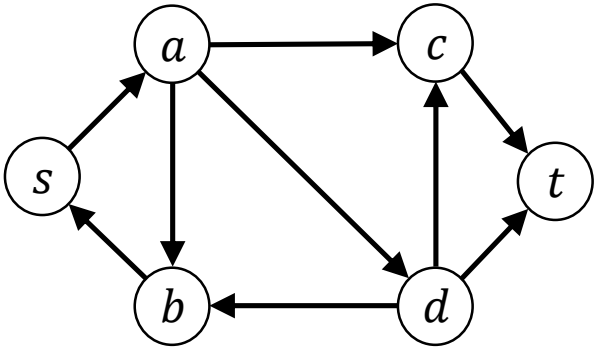
Coding Demo: Level-Augmented BFS



Key idea: Replace discovered Set with Map whose values keep track of levels

Shortest Paths in Unweighted Graphs

For a vertex v at level ℓ, the shortest $s \rightsquigarrow v$ path includes ℓ edges (by the definition of ℓ).





Coding Demo: BFS with PathInfo



Key ideas:

- PathInfo record class combines level + prev edge
- Update discovered Map to have PathInfo values

Reconstructing Shortest Paths

Exercise: Use a loop to compute `path` in this method definition.

```
record PathInfo<V>(int level, V prev) {}

/** Reconstructs and returns the shortest `src` -> `dst` path
 *  using the given `info` map produced by BFS. */
static <V> List<V> reconstructPath(Map<V,PathInfo<V>> info, V src, V dst) {
  List<V> path = new LinkedList<>();



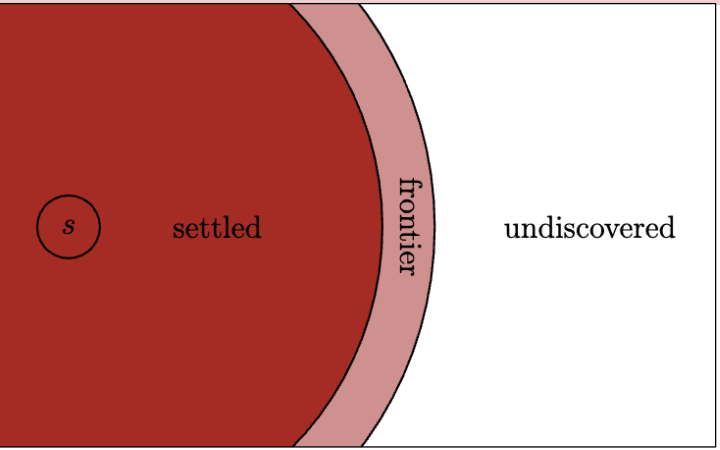






  return path;
}
```

Unweighted Shortest Paths (BFS) Invariants



1. `discovered` map records shortest (known) path distance to every discovered vertex

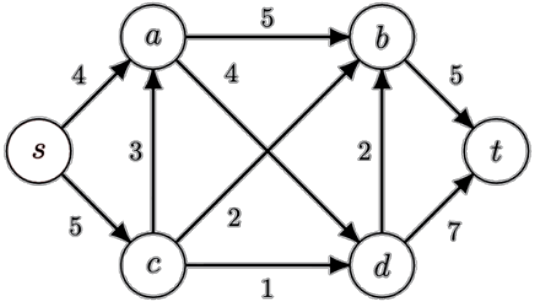
2. Vertex v `dequeue()`d from `frontier` is *unvisited* vertex with min level

3. If a vertex v is settled, we've computed the shortest $s \rightsquigarrow v$ path, which contains only settled vertices

Adapting to Weighted Graphs (*non-negative weights)

High-Level Structure is the same as BFS

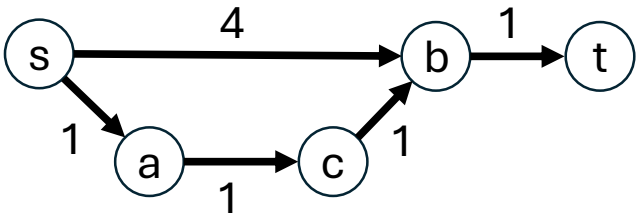
- Track `discovered` and `frontier` vertices



- Visit/settle one `frontier` vertex in each main loop iteration
- During visit, use outgoing edges to discover new vertices and update `discovered` map and `frontier`

Adapting to Weighted Graphs (*non-negative weights)

Issue 1: Distance order may differ from discovery order



discovered

vertex					
dist					
prev					

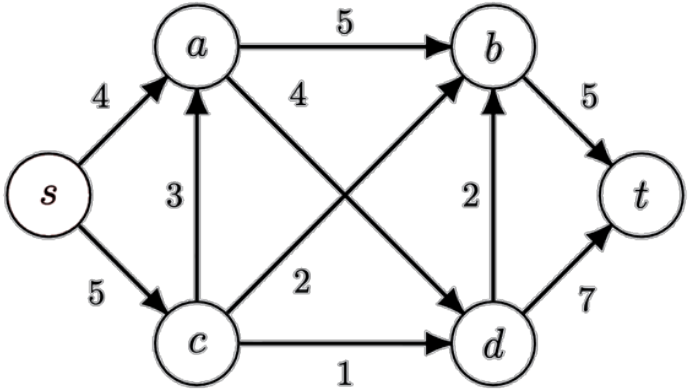
Issue 2: May locate shorter path to vertex later in the traversal



Coding Demo: Dijkstra's Algorithm



Dijkstra's Algorithm Walkthrough



discovered

vertex	s	a	b	c	d	t
distance						
prev						

frontier

* The lecture notes have an animation of this example.

Dijkstra's Invariant

At the start of each main loop iteration:

1. We've located the shortest path to all *settled* vertices.
2. The shortest *known* path to the *closest* `frontier` vertex v^* is guaranteed to be the true shortest $s \rightsquigarrow v^*$ path.

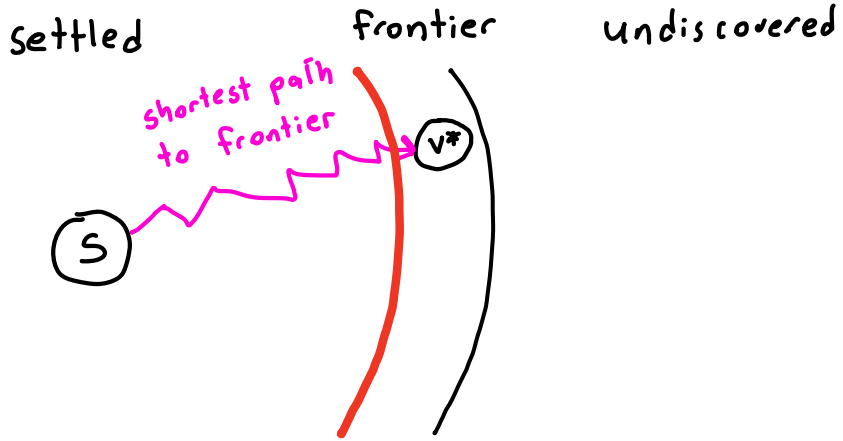
Proof Sketch:

Dijkstra's Invariant

At the start of each main loop iteration:

1. We've located the shortest path to all *settled* vertices.
2. The shortest *known* path to the *closest* `frontier` vertex v^* is guaranteed to be the true shortest $s \rightsquigarrow v^*$ path.

Proof Sketch:



Properties of Dijkstra's Algorithm

Vertices are visited/settled in increasing *distance* order (just like BFS).

The best *known* distance to a vertex v can only decrease as the algorithm runs (and we locate new paths to v)

For any discovered but unsettled vertex v,

$\leq$ length of shortest $s \rightsquigarrow v$ path $\leq$

Dijkstra's Algorithm Runtime Analysis

```
while(!frontier.isEmpty()) {
  V v = frontier.remove();
  for (WeightedEdge<V> edge : v.outgoingEdges()) {
    V neighbor = edge.head();
    double dist = discovered.get(v).distance() + edge.weight();
    if (!discovered.containsKey(neighbor)) {
      discovered.put(neighbor, new PathInfo<>(dist, v));
      frontier.add(neighbor, dist);
    } else if (discovered.get(neighbor).distance() > dist)  {
      discovered.put(neighbor, new PathInfo<>(dist, v));
      frontier.updatePriority(neighbor, dist);
}}}
```