Shortest Paths [BFS, Djikstra]

Sides based on material by Kani, Chekuri, Erickson et. al.

All mistakes are my own! - Ivan Abraham (Fall 2024)

Image by ChatGPT (probably collaborated with DALL-E)

Breadth first search (BFS)

Overview

- Breadth-first search (BFS) is an algorithm for traversing or searching a Tree or Graph data structure which returns the nodes of the graph level by level.
- BFS on a graph with n vertices and m edges takes $O(n+m)$ time (obtained from BasicSearch by processing edges using a queue data structure).
- It processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex)
- **DFS** good for exploring graph structure | **BFS** good for exploring distances

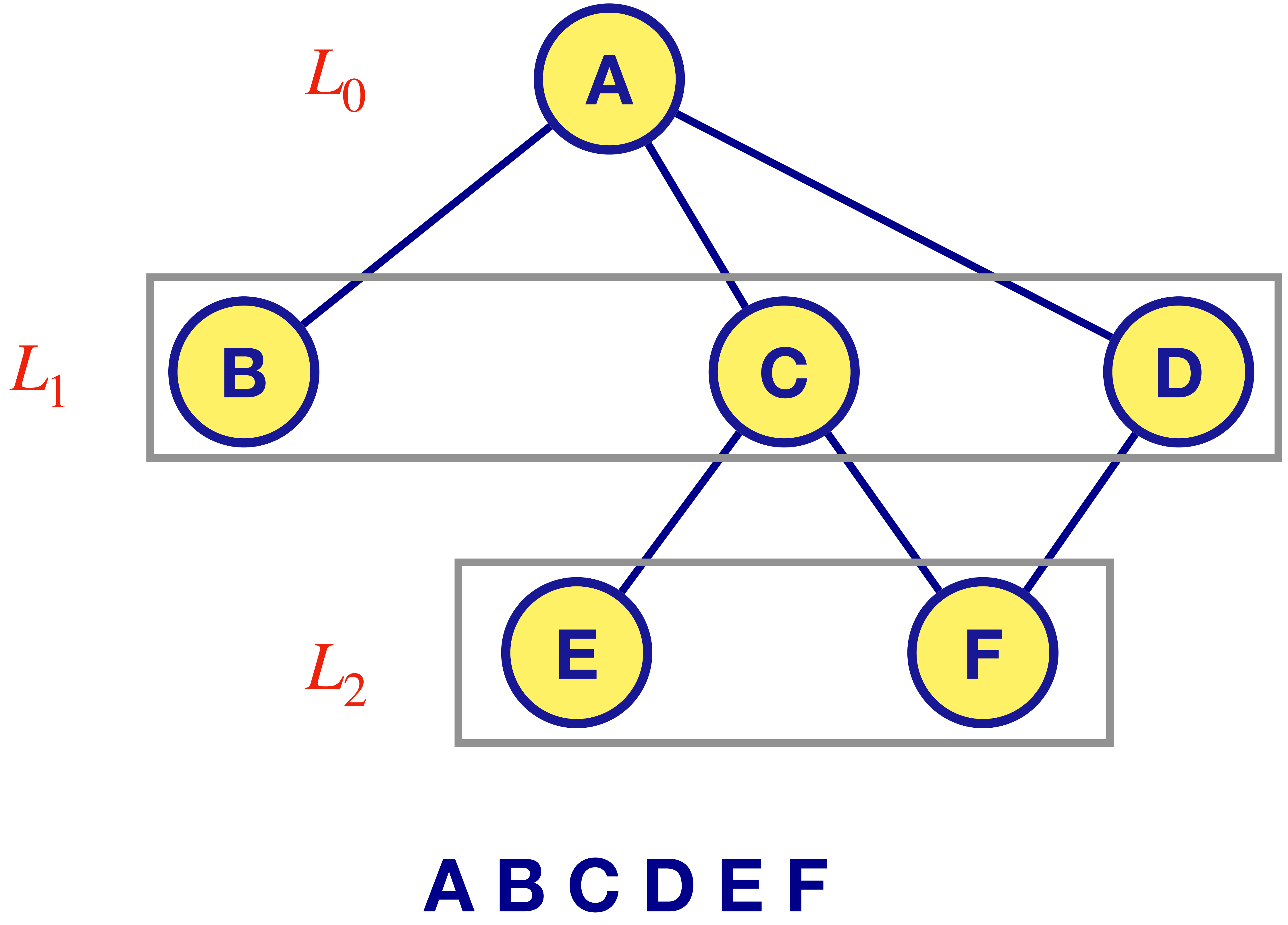
Breadth first search (BFS)

BFS traversal of a graph returns the nodes of the graph level by level.

The Idea of the BFS:

Visit the vertices as follows:

- Visit all vertices at distance 1
- Visit all vertices at distance 2
- Visit all vertices at distance 3 etc.



Queue data structure

Queues

A queue is a list of elements which supports the operations:

- **Enqueue**: Adds an element to the end of the list
- **Dequeue**: Removes an element from the front of the list
- Elements are extracted in first-in first-out (FIFO) order, i.e., elements are picked in the order in which they were inserted.
    - Contrast with LIFO (stacks)

BFS algorithm

Pseudocode

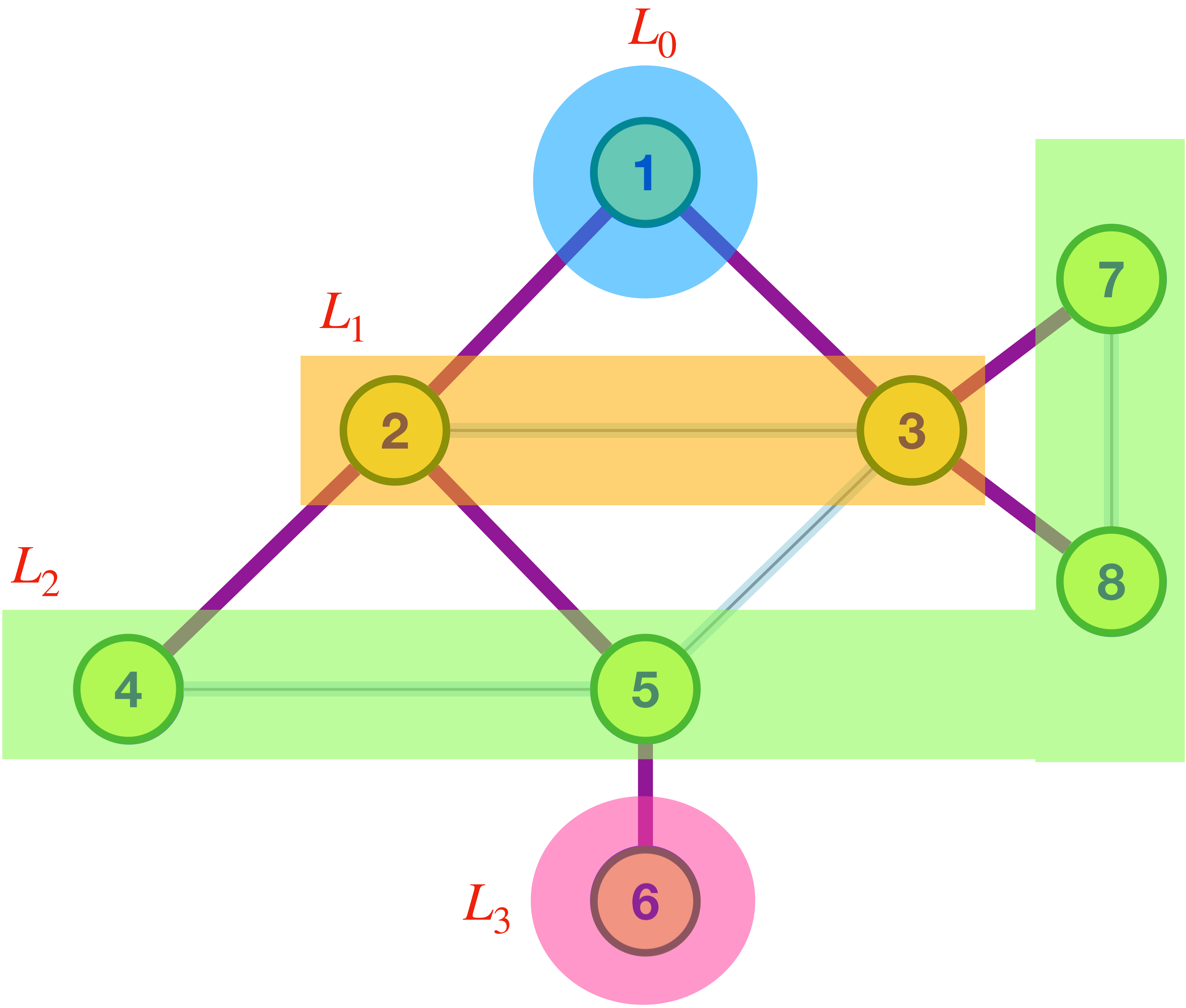
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
BFS(s):
    Mark all vertices as unvisited;
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enqueue(Q,s)
    while Q is non-empty do
          u = dequeue(Q)
          for each vertex v ∈ Adj(u)
              if v is not visited then
                add edge (u,v) to T
                Mark v as visited and enqueue(v)
```

Proposition

BFS(s) runs in $O(n + m)$ time

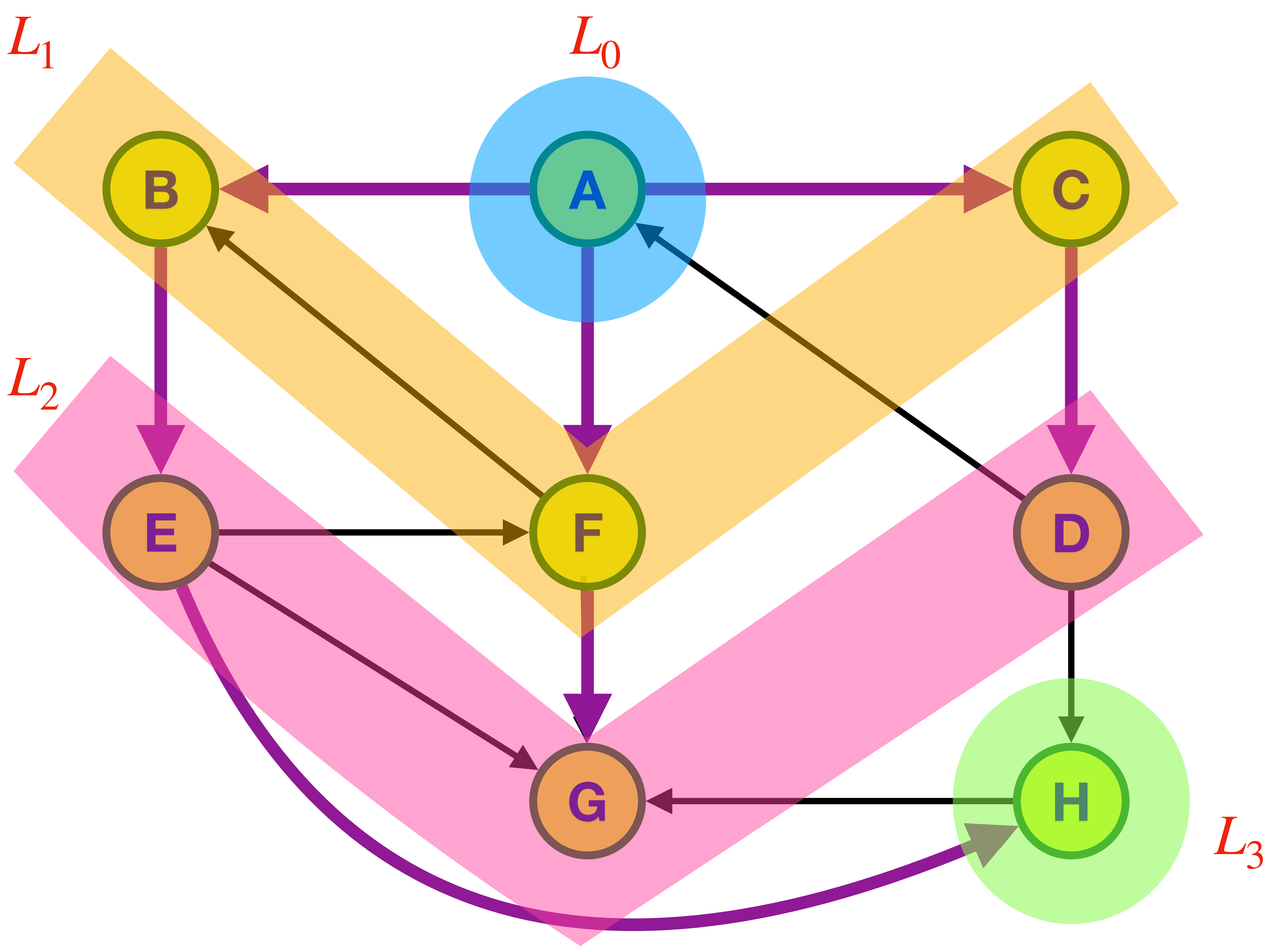
BFS: An example in undirected graphs



1	2	3	4	5	7	8	6		

BFS tree is the set of **purple edges**

BFS: An example in directed graphs



Q1: A

Q2: B C F

Q3: C F E

Q4: F E D

Q5: E D G

Q6: D G H

Q7: G H

Q8: H

Q9:

BFS with distances

```
BFS(s):
    Mark all vertices as unvisited; for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enqueue(s)
    while Q is non-empty do
          u = dequeue(Q)
          for each vertex v ∈ Adj(u) do
              if v is not visited do
                add edge (u,v) to T
                Mark v as visited, enqueue(v)
                and set dist(v) = dist(u) + 1
```

Properties of BFS

Undirected graphs

Theorem: *The following properties hold upon termination of BFS(s)*

- Search tree is the set of vertices in the connected component of s.
- If $\text{dist}(u) < \text{dist}(v)$ then u is visited before v.
- For every vertex u, $\text{dist}(u)$ is the length of a shortest path (in terms of number of edges) from s to u.
- If u, v are in connected component of s and $e = \{u, v\}$ is an edge of G, then $|\text{dist}(u) - \text{dist}(v)| \leq 1$.

Properties of BFS

Directed graphs

Theorem: *The following properties hold upon termination of BFS(s)*

- Search tree contains exactly the set of vertices reachable from s.
- If $\text{dist}(u) < \text{dist}(v)$ then u is visited before v.
- For every vertex u, $\text{dist}(u)$ is indeed the length of shortest path from s to u.
- If u is reachable from s and $e = (u, v)$ is an edge of G, then $\text{dist}(v) \leq 1 + \text{dist}(u)$.

BFS with layers

- BFS is a simple algorithm but proving its properties formally is not straight forward
- Since BFS explores graph in increasing order of distance from source s, there is a simpler variant that makes BFS exploration transparent and easier to understand.
  - Given G and $s \in V$, define $L_i = \{v \mid \mathrm{dist}(s, v) = i\}$.
  - Then $L_0 = \{s\}$
  - And L_k can be found from L_{k-1} for $k \geq 1$ inductively.

BFS with layers

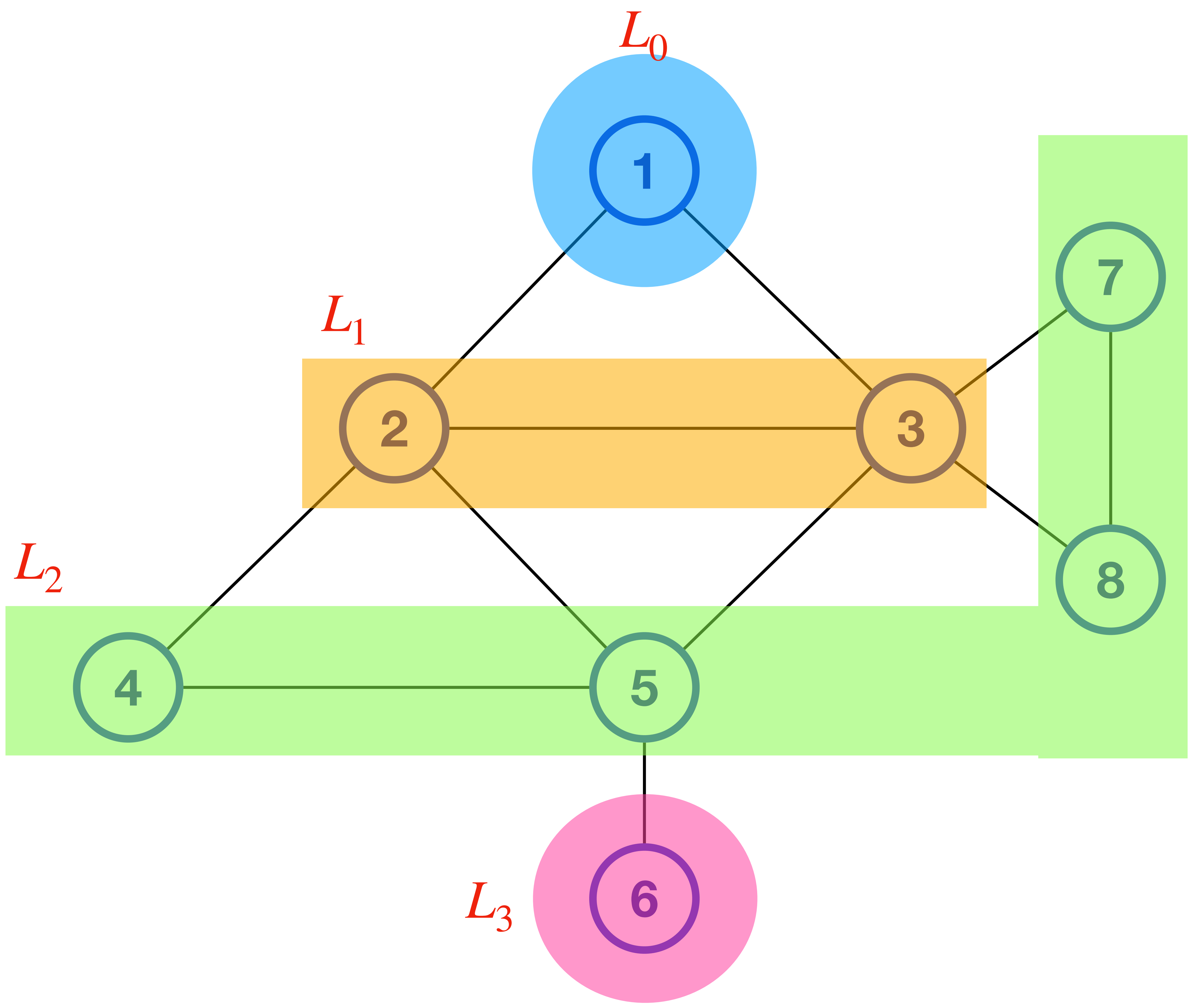
```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
          initialize L_{i+1} to be an empty list
          for each u in L_i do
              for each edge (u, v) ∈ Adj(u) do
              if v is not visited
                 mark v as visited
                 add (u, v) to tree T
                 add v to L_{i+1}
          i = i + 1
```

Running time: $O(n + m)$

BFS with layers

Example - undirected

- Layer 0: 1
- Layer 1: 2, 3
- Layer 2: 4, 5, 7, 8
- Layer 3: 6



BFS with layers: undirected graph

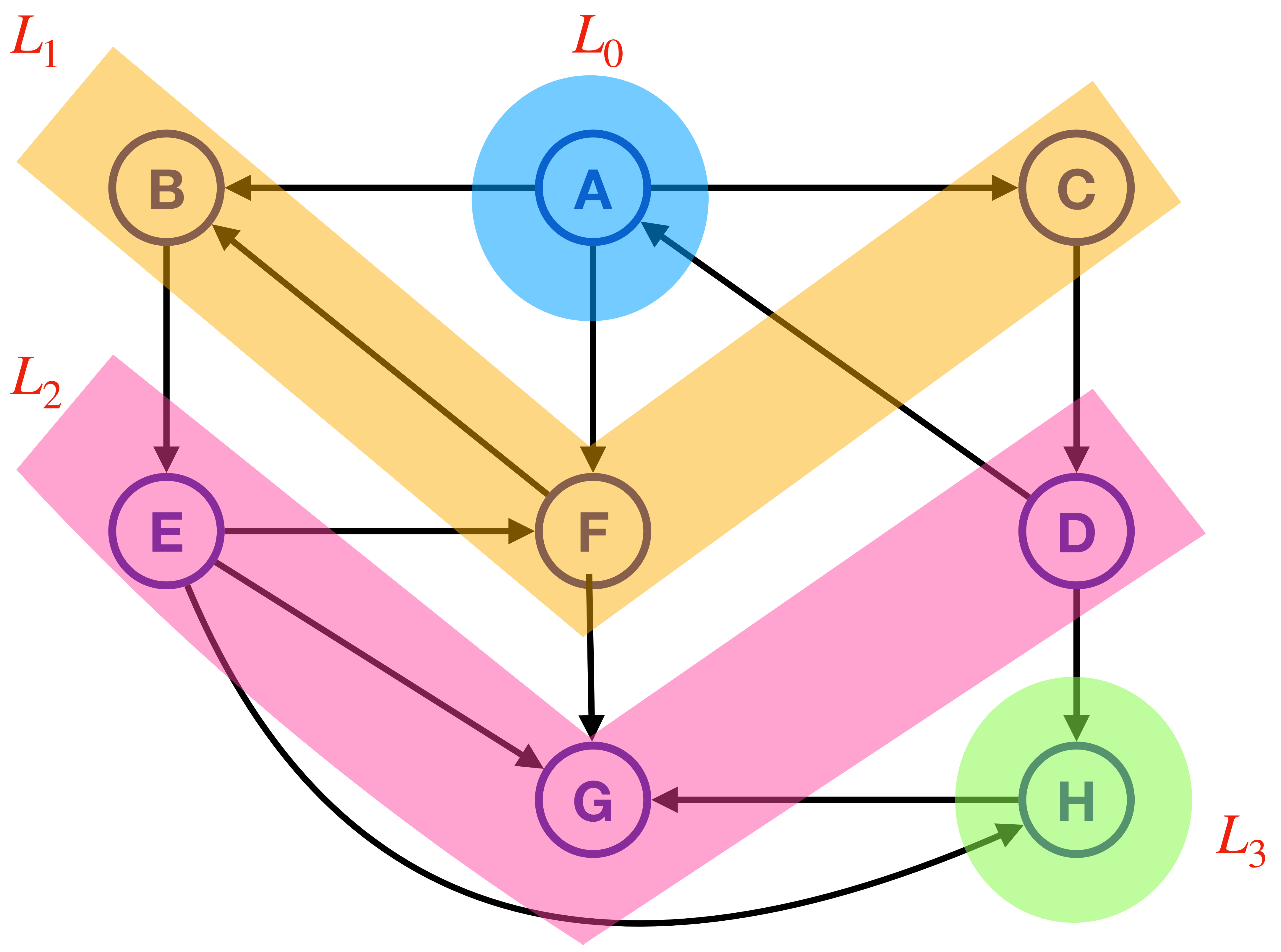
Properties

- BFSLayers(s) outputs a BFS tree
- L_i is the set of vertices at distance exactly i from s.
- If G is undirected, each edge $e = \{u, v\}$ is one of three types:
  - *tree* edge between two *consecutive* layers
  - non-tree *forward/backward* edge between two consecutive layers
  - non-tree *cross-edge* with both u, v in same layer
- Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers!

BFS with layers

Example - directed

- Layer 0: A
- Layer 1: B, F, C
- Layer 2: E, G, D
- Layer 3: H



BFS with layers: directed graph

Properties

Proposition: *The following properties hold on termination of BFS(s) if G is directed.*

- Each edge $e = \{u, v\}$ is one of four types:
  - A *tree* edge between consecutive layers, $u \in L_i,\ v \in L_{i+1}$ for some $i \geq 0$
  - A non-tree *forward* edge between consecutive layers
  - A non-tree *backward* edge
  - A *cross-edge* with both u, v in same layer

Shortest path problems

Description

Given graph $G = (V, E)$ with associated edge lengths (or costs), denote for an edge $e = uv$ the quantity $l(e) = l(uv)$ as its length or cost.

- Given nodes s, t find shortest path (in terms of summed lengths/costs) from s to t . (SSPP)
- Given node s find shortest path from s to all other nodes (SSSP)
- Find shortest paths between all pairs of nodes (APSP)

Shortest walks vs. paths

- A path is a sequence of ***distinct*** vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$.
- A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$.
- Finding walks is often easier than finding paths (concatenating two walks gives a walk, while concatenating two paths may not give a path).
- For edges with non-negative weights/lengths, finding the shortest walk is the same as finding the shortest $s \to t$ path.

Single-source shortest paths

Assumption: non-negative edge lengths

Single-source shortest path problems (SSSPs)

- Input: A (undirected or directed) graph $G = (V, E)$ with *non-negative edge lengths.* For edge $e = (u, v)$, $l(e) = l(u, v)$ is its length.

- Given nodes s, t find shortest path from s to t.

- Given node s find shortest path from s to all other nodes.

- Restrict attention to directed graphs

Single-source shortest paths

Assumption: non-negative edge lengths

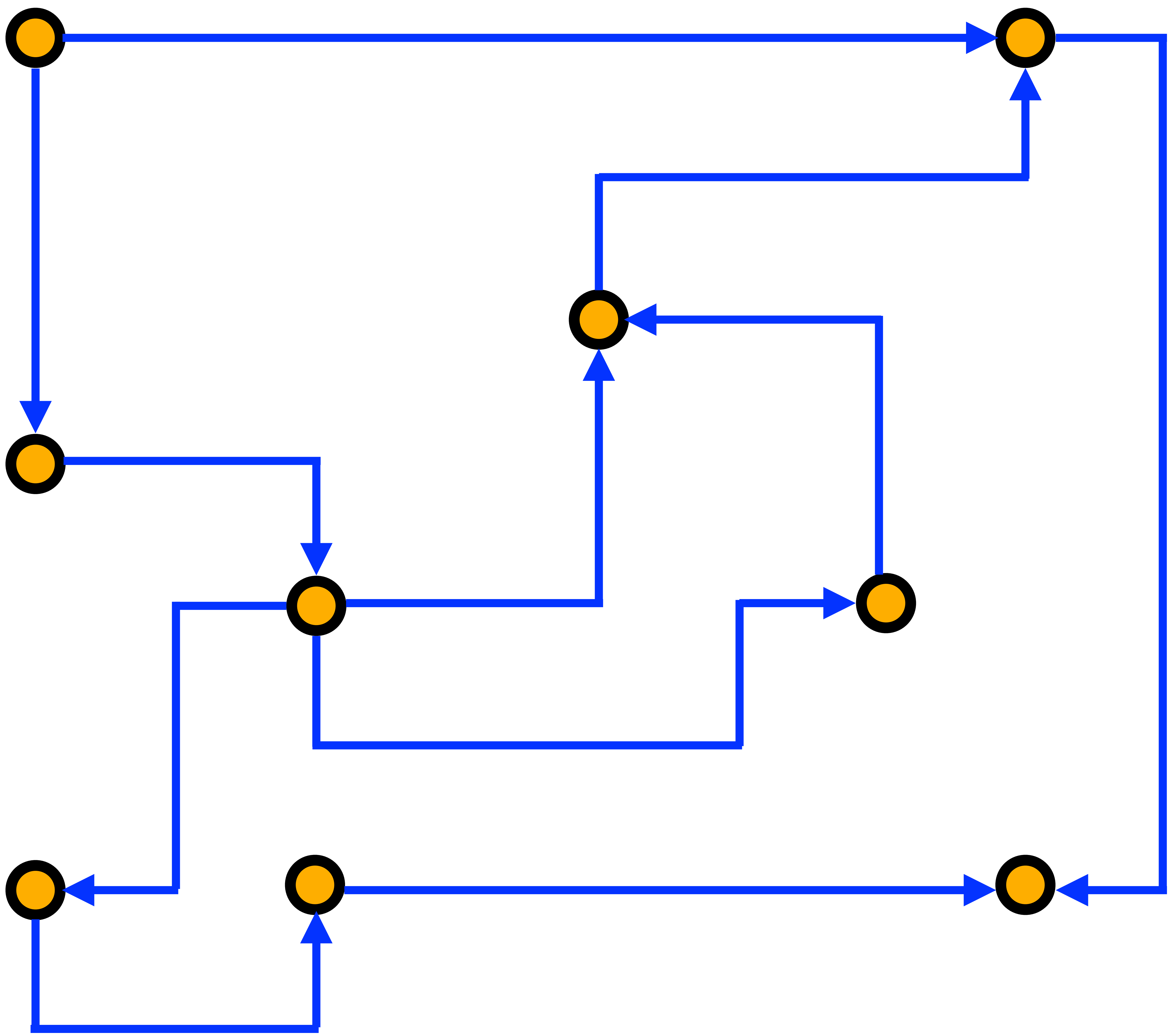
- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph G, create a new directed graph G' by replacing each edge $\{u, v\}$ in G by (u, v) and (v, u) in G' .
  - set $l(u, v) = l(v, u) = l(\{u, v\})$
  - Exercise: show reduction works. Relies on non-negativity!


N

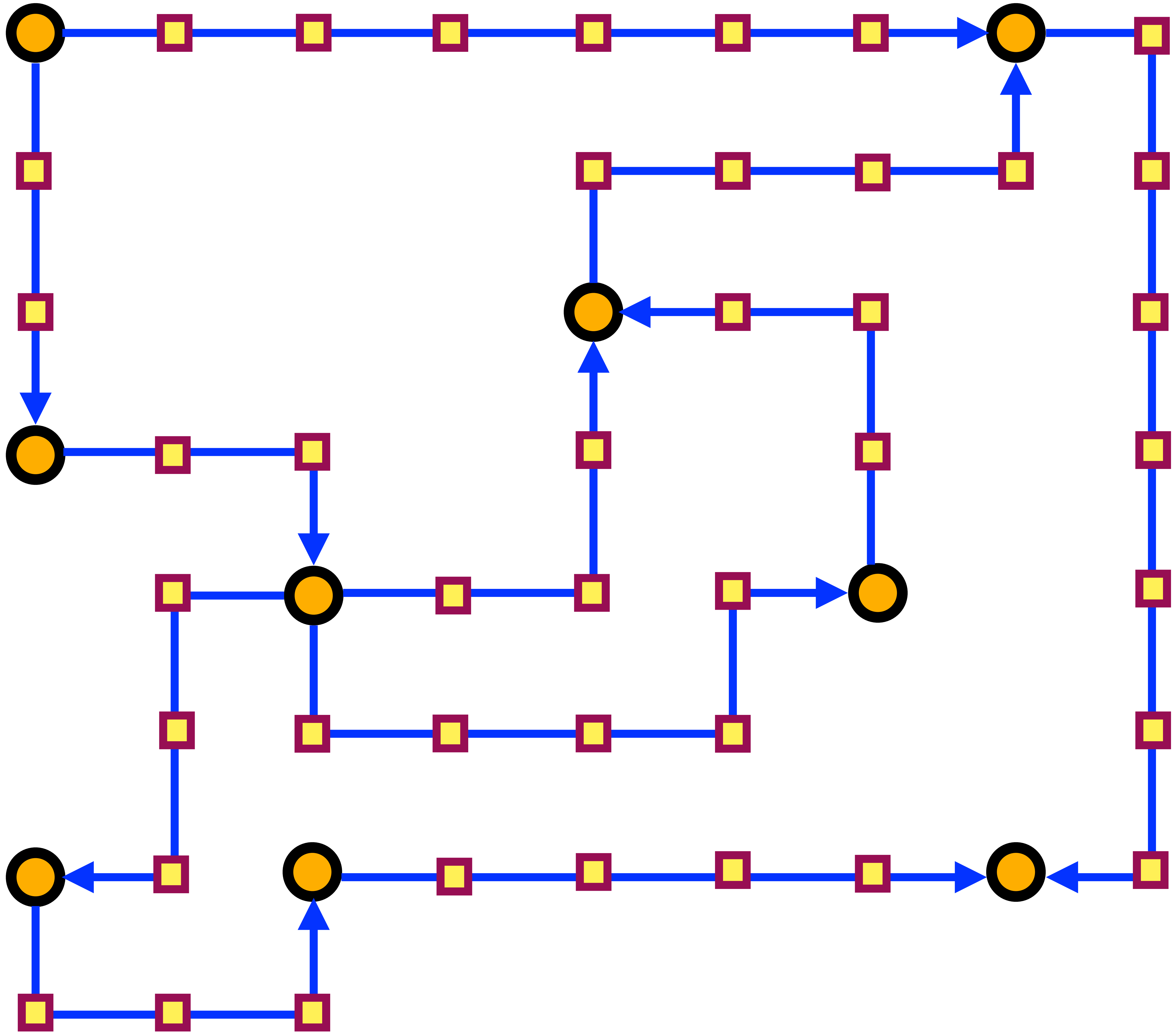
Single-source shortest paths via BFS

- **Special case:** All edge lengths are 1.
  - Run BFS(s) to get shortest path distances from s to all other nodes.
  - O(m + n) time algorithm.
- **Special case:** Suppose $l(e)$ is an integer for all e? Can we use BFS? Reduce to unit edge-length problem by placing $l(e) - 1$ dummy nodes on e.
- Let $L = \max_e l(e)$. New graph has $O(mL)$ edges and $O(mL + n)$ nodes. BFS takes $O(mL + n)$ time. Not efficient if L is large.

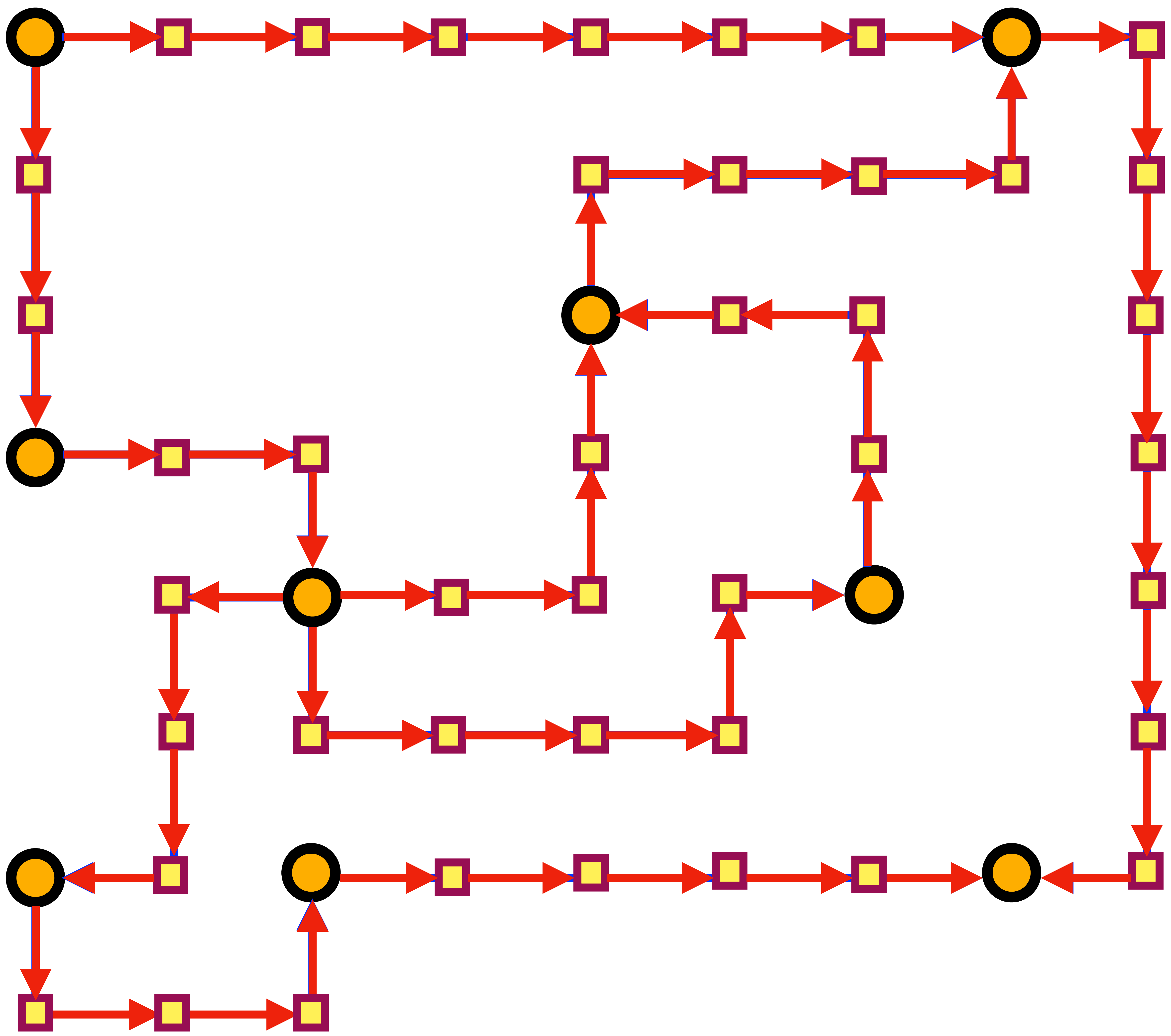
Example of edge refinement



Example of edge refinement



Example of edge refinement



You can not shortcut a shortest path

Lemma (… also goes by Bellman's principle of optimality)

Let G be a directed graph with *non-negative* edge lengths. Suppose that

$$p = v_0 \to v_1 \to v_2 \to \ldots \to v_k$$

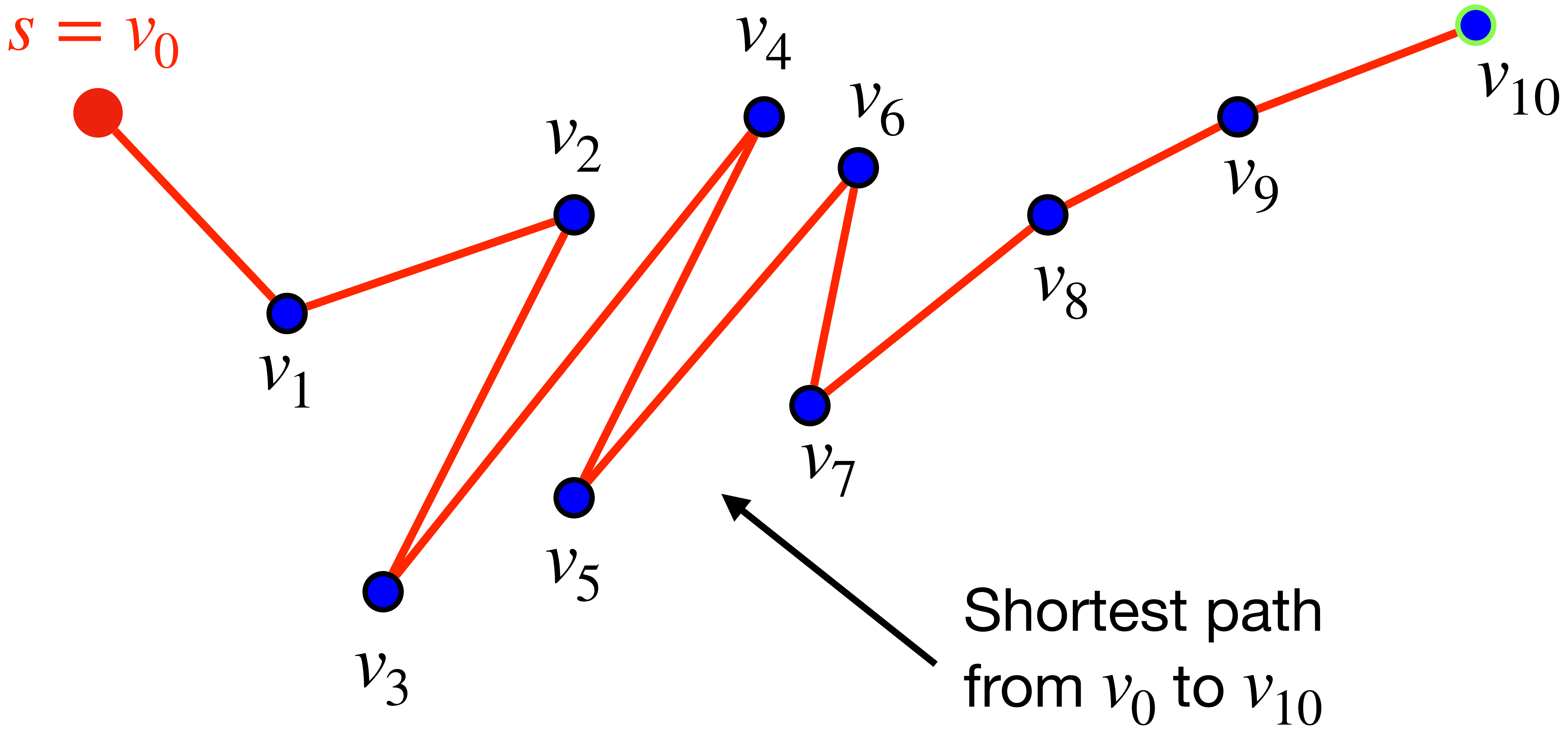
is the shortest path from v_0 to v_k.

Then for any $0 \leq i < j \leq k$ we have that

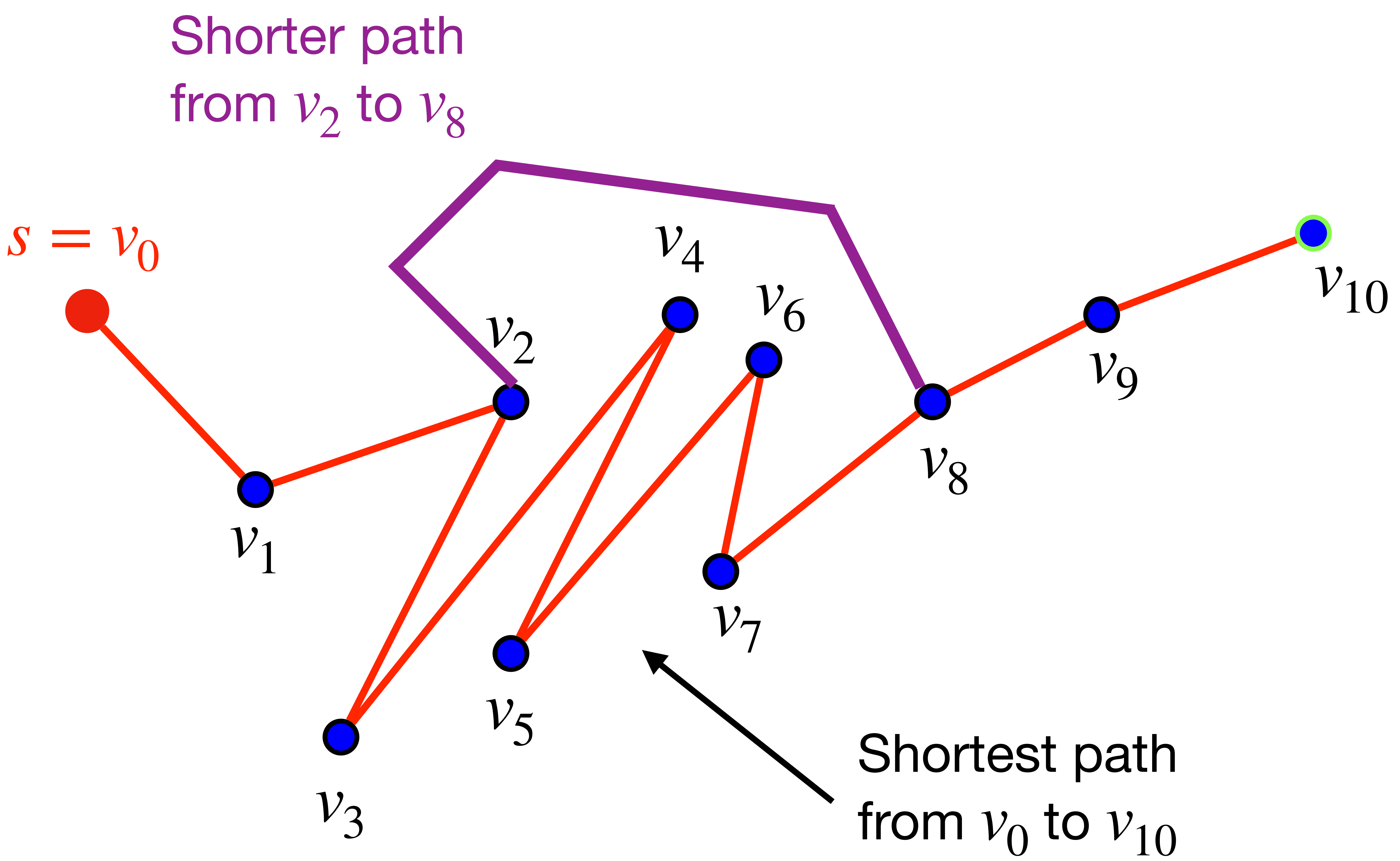
$$v_i \to v_{i+1} \to \ldots \to v_j$$

is the shortest path from v_i to v_j.

A proof by picture

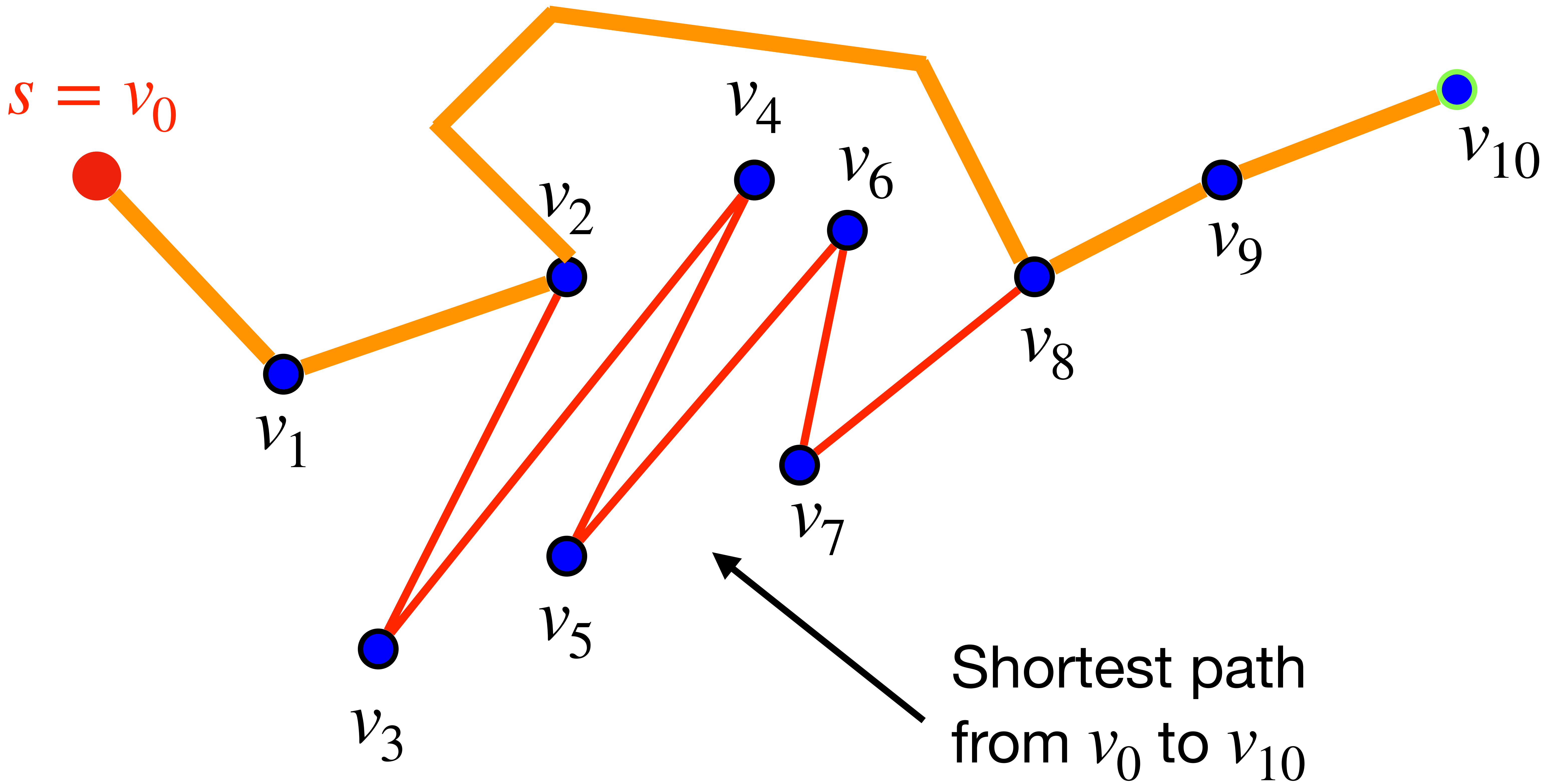


A proof by picture



A proof by picture

A shorter path from v_0 to v_{10}.
A contradiction



Shortest path from v_0 to v_{10}

What we really need…
Stated in terms of distance

Let G be a directed graph with non-negative edge lengths and let $\text{dist}(s, v)$ denote the length of the shortest path from s to v.

If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$

is the shortest path from $s = v_0$ to v_k then for any $0 \leq i < j \leq k$ we have that

$s = v_0 \to v_1 \to v_2 \ldots \to v_i$ is shortest path from s to v_i and

$$\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$$

Find the i^{th} closest vertex

A basic strategy

Explore vertices in increasing order of distance from s: (For simplicity, assume that nodes are at different distances from s and that no edge has zero length)

```
Initialize for each node v, dist(s,v) = ∞
Initialize X = {s},
    for i = 2 to |V| do
        (* Invariant: X contains the i−1 closest nodes to s *)
        Among nodes in V\X, find the node v that is the
        i^th closest to s
        Update dist(s,v)
        X = X ∪ {v}
```

How can we implement the step in the for loop?

Finding the i^{th} closest node

What we have ...

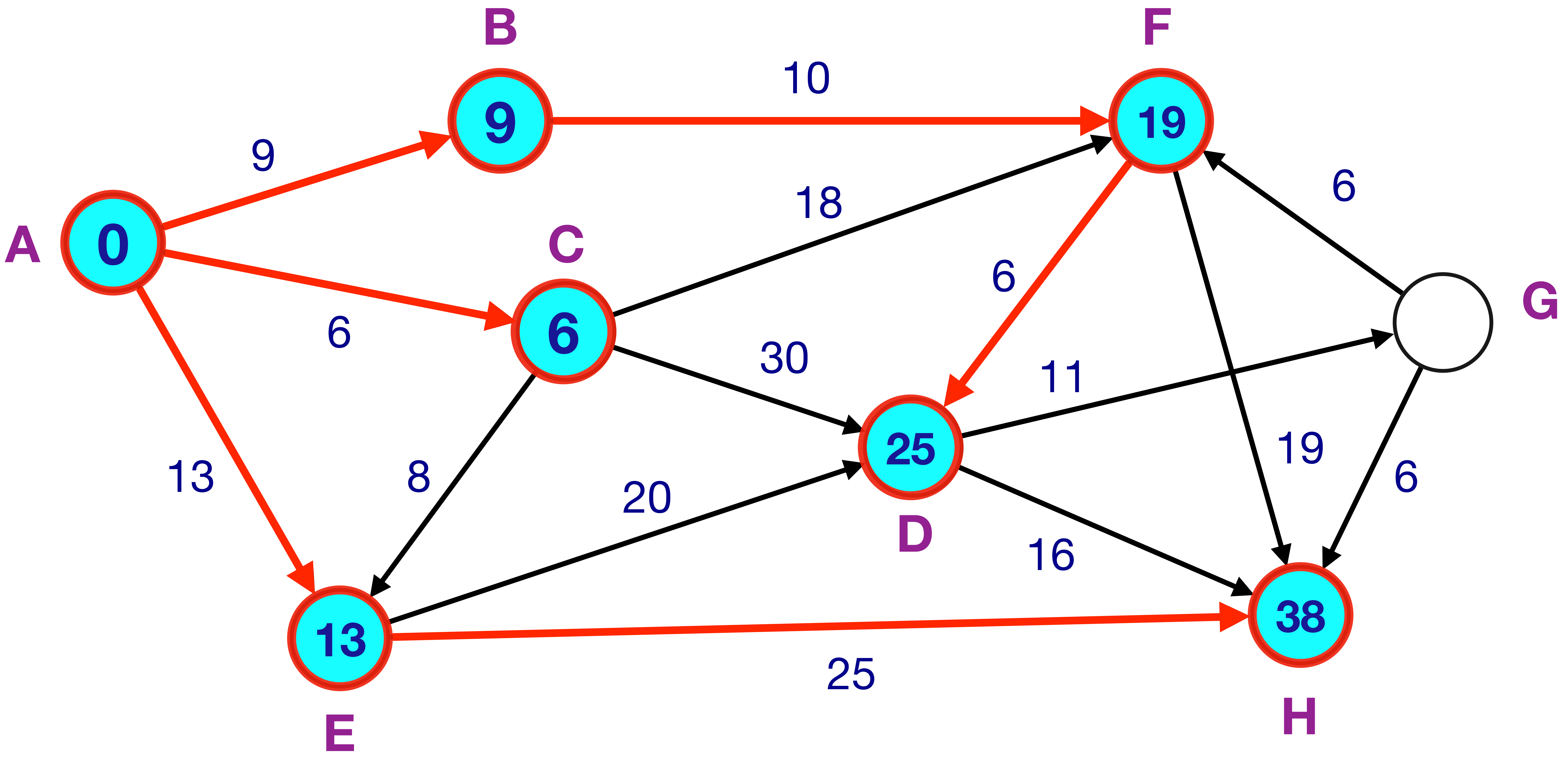
- X contains the $i-1$ closest nodes to s
- Want to find the i^{th} closest node from $V \backslash X$.

What do we know about the i^{th} closest node?

Claim: Let P be a shortest path from s to v where v is the i^{th} closest node. Then, all intermediate nodes in P belong to X.

Proof: If P had an intermediate node u not in X then u will be closer to s than v. Implies v is **not** the i^{th}closest node to s - recall that X already has the $i-1$ closest nodes!

Finding the i^{th} closest node



Algorithm

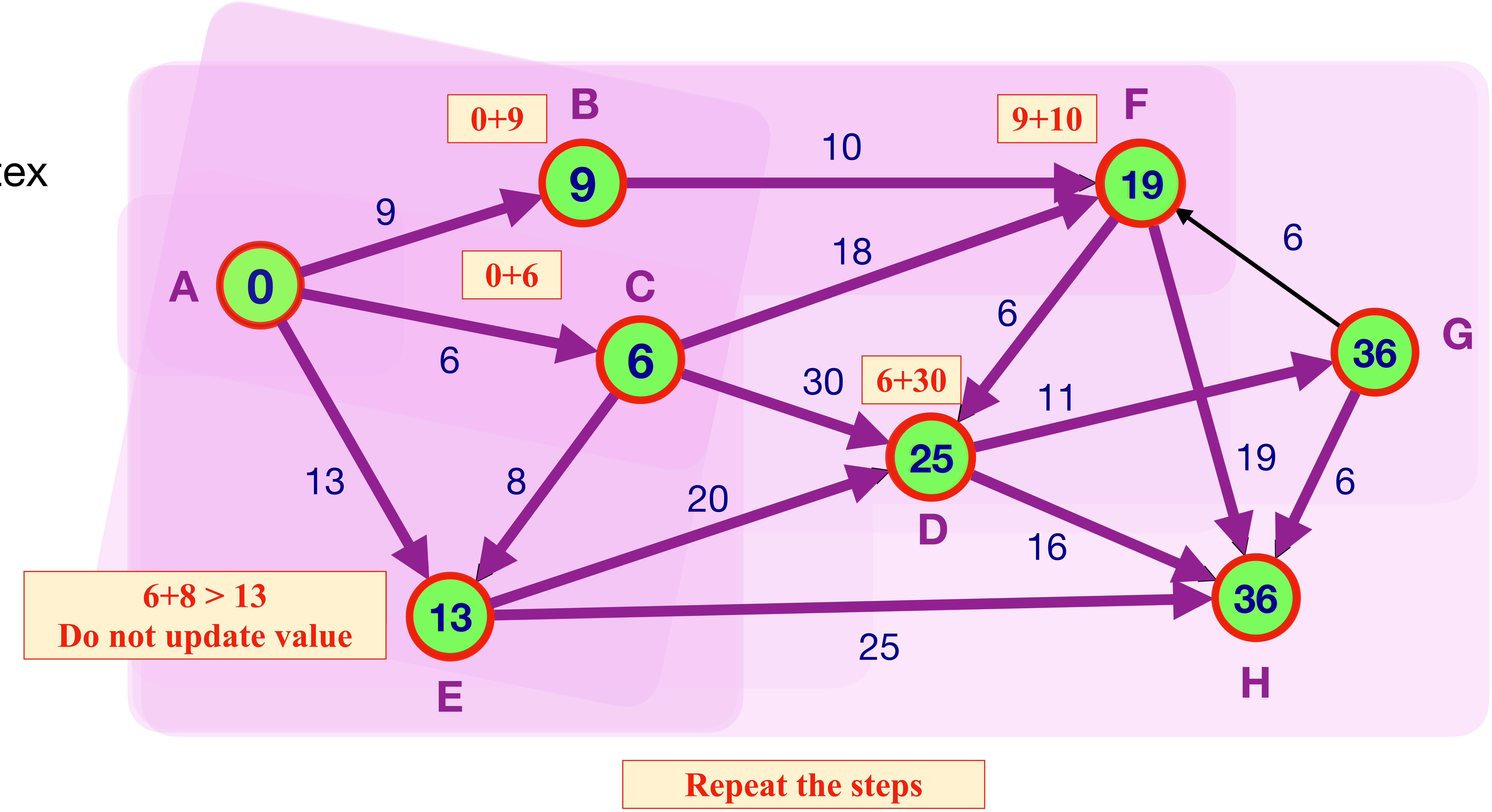
```
Initialize for each node v: dist(s,v) = ∞
Initialize X = ∅, d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s
    using only X as intermediate nodes*)
        Let v be such that d'(s,v) = min_{u∈V−X} d'(s,u)
        dist(s,v) = d'(s,v)
        X = X ∪ {v}
        for each node u in V − X do
            d'(s,u) = min_{t∈X}(dist(s,t) + l(t,u))
```

Running time: $O(n\,.\,(n+m))$ time

There are n outer iterations. In each iteration, $d'(s,u)$ for each u by scanning all edges out of nodes in X; $O(m+n)$ time/iteration

Dijkstra algorithm
Example

- Choose a starting vertex



Improved algorithm

- Main work is to compute the $d'(s, u)$ values in each iteration
- $d'(s, u)$ changes from iteration i to $i + 1$ only because of the node v that is added to X in iteration i (previous step)

```
Initialize for each node v: dist(s, v) = d'(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    // X contains the i − 1 closest nodes to s,
    // and the values of d'(s, u) are current
        Let v be node realizing d'(s, v) = min_{u ∈ V\X} d'(s, u)
        dist(s, v) = d'(s, v)
        X = X ∪ {v}
        Update d'(s, u) for each u in V − X as follows:
            d'(s, u) = min(d'(s, u), dist(s, v) + l(v, u))
```

Improved algorithm

Running time: $O(m+n^2)$ time.

- n outer iterations and in each iteration following steps take place:
  - updating $d'(s,u)$ after v is added takes $O(\deg(v))$ time so ***total*** work is $O(m)$ since a node enters X at most once
  - Finding v from $d'(s,u)$ values takes $O(n)$ time

Dijkstra's Algorithm

- Eliminate $d'(s,u)$ and let dist(s,u) maintain it
- Update dist values after adding v by scanning edges out of v

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V\X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + l(v, u))
```

Can use Priority Queues to maintain dist values for even faster running time

- Using heaps and standard priority queues: $O((m+n)\ \log n)$
- Using Fibonacci heaps: $O(m + n \log n)$

Dijkstra using Priority Queues

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ alongwith that the following operations:

- makePQ: create an empty queue.
- findMin: find the minimum key in S.
- extractMin: Remove $v \in S$ with smallest key and return it.
- insert($v, k(v)$): Add new element v with key $k(v)$ to S.
- delete(v): Remove element v from S.
- decreaseKey($v, k'(v)$): decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
- meld: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time - decreaseKey is implemented via delete and insert.

Dijkstra's algorithm using priority queues

```
Q ← makePQ()
insert(Q, (s, 0))
for each node u ≠ s do
    insert(Q, (u, ∞))
X ← ∅
for i = 1 to |V| do
  (v, dist(s, v)) = extractMin(Q)
X = X ∪ {v}
for each u in Adj(v) do
            decreaseKey(Q, (u, min(dist(s, u), dist(s, v) + l(v, u))))
```

PQ operations:

- $O(n)$ **insert** operations
- $O(n)$ **extractMin** operations
- $O(m)$ **decreaseKey** operations

Shortest Path Tree

Dijkstra's alg. finds the shortest path distances from s to V.

Question: How do we find the paths themselves?

```
Q ← makePQ()
insert(Q, (s, 0))
prev(u) ← null
for each node u ≠ s do
    insert(Q, (u, ∞))
    prev(u) ← null
X ← ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + l(v, u) < dist(s, u)) then
            decreaseKey(Q, (u, dist(s, u) + l(v, u)))
            prev(u) = v
```

Shortest Path Tree

Lemma: The edge set $(u, \text{prev}(u))$ is the reverse of a shortest path tree rooted at s. For each u, the reverse of the path from u to s in the tree is a shortest path from s to u.

Proof Sketch:

- The edge set $\{(u, \text{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at s (Why?)
- Use induction on $|X|$ to argue that the obtained tree is a shortest path tree for nodes in V.

Shortest paths *to* s?

Dijkstra's alg. gives shortest paths from s to all nodes in V.

How do we find shortest paths from all of V to s?

- In undirected graphs shortest path from s to u is a shortest path from u to s so there is no need to distinguish.

- In directed graphs, use Dijkstra's algorithm in G^{rev}!