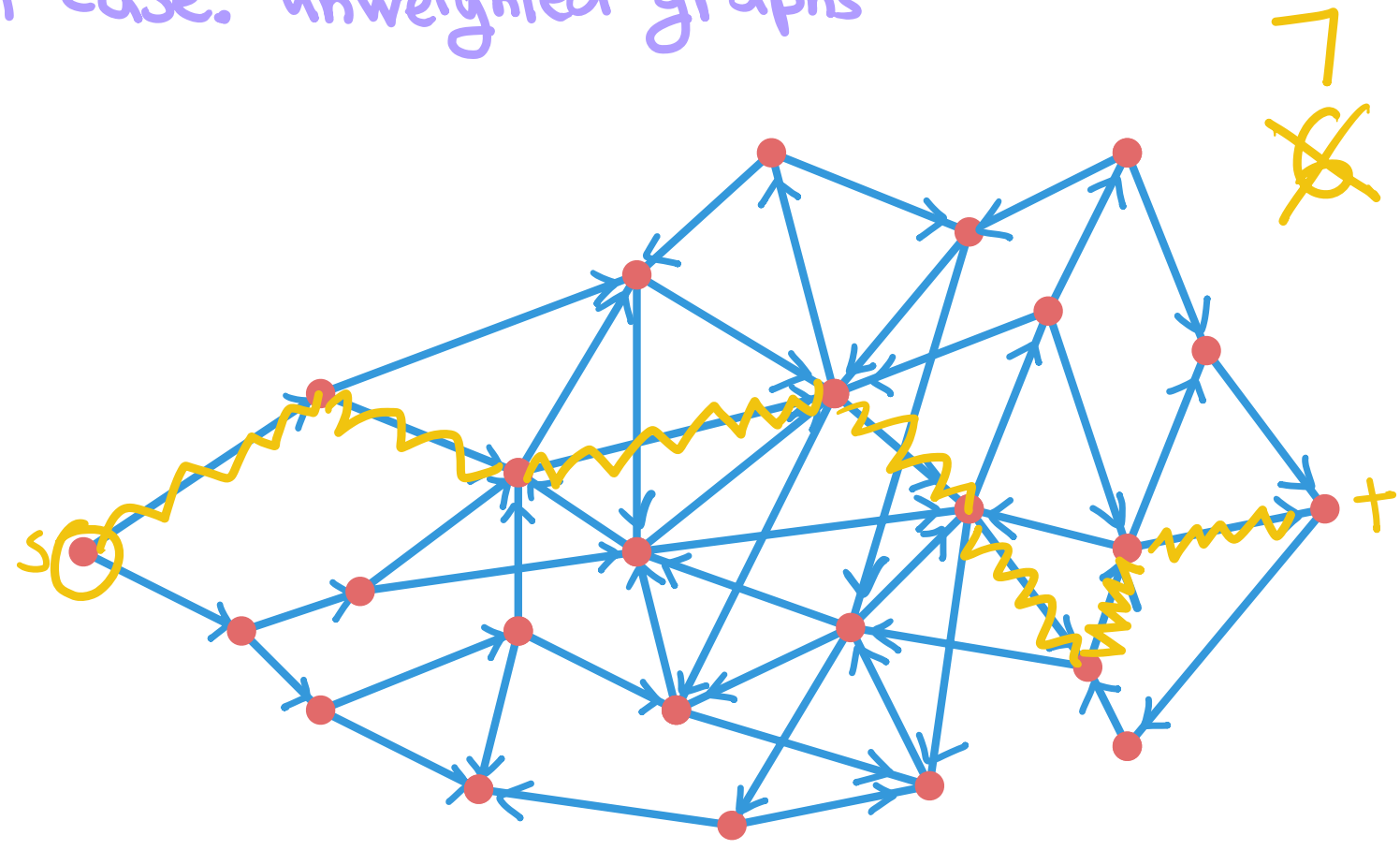
Shortest Paths

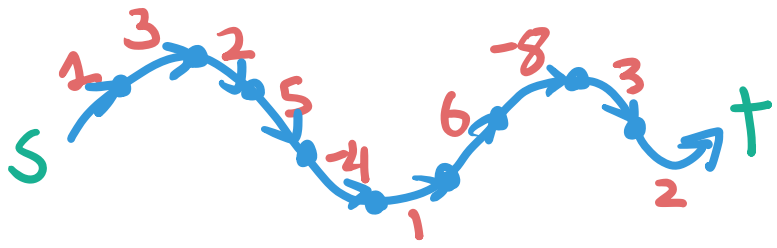
simplest case: unweighted graphs



shortest way from s to t?

Directed $G=(V,E)$, edge lengths $\ell: E \to \mathbb{R}$

for walk w, $\bar{\ell}(w) = \sum_{e \in w} \ell(e)$

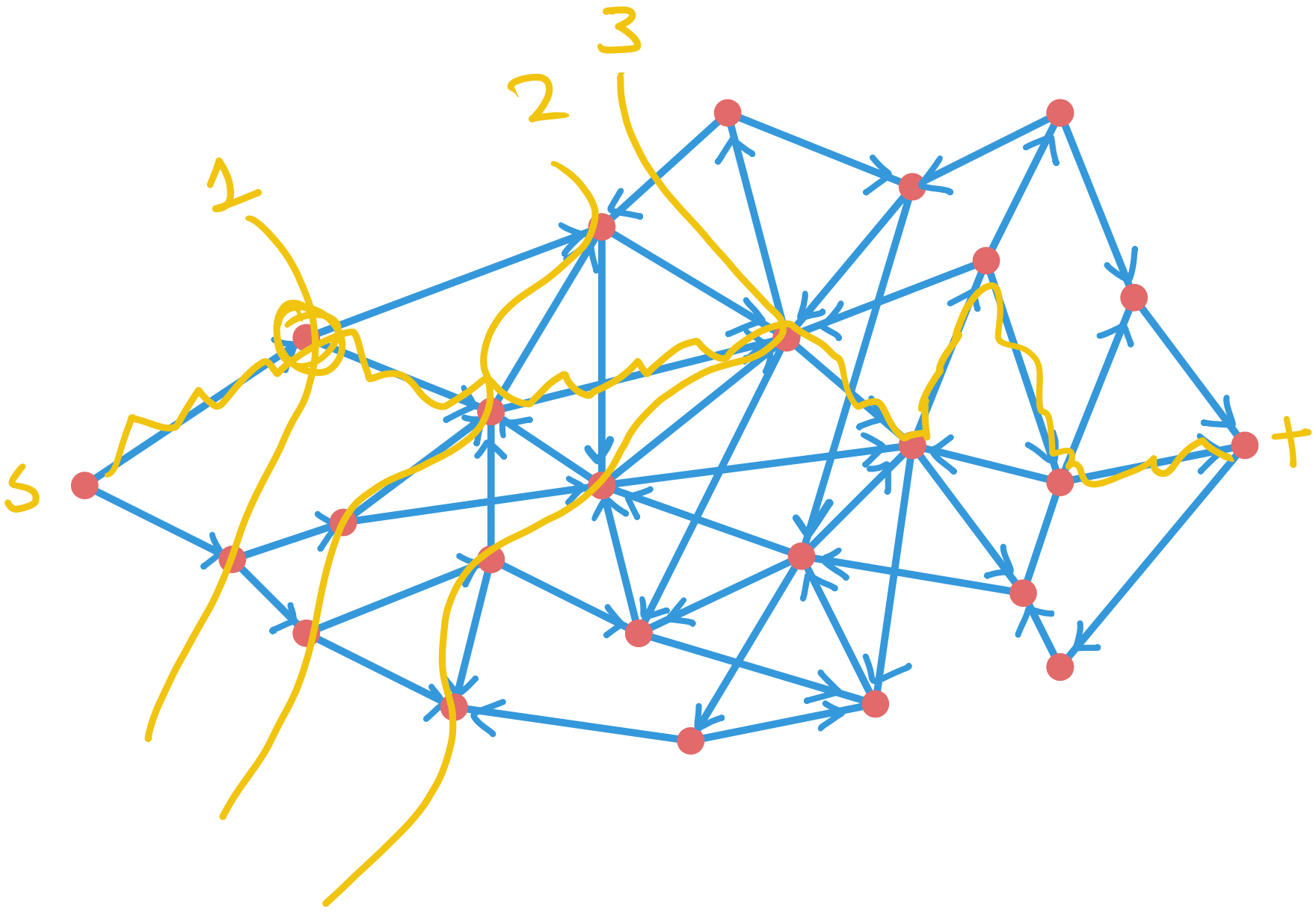


Distance $d(s,t) = \inf \{ \bar{\ell}(w) : w \text{ is an } (s,t)\text{-walk} \}$

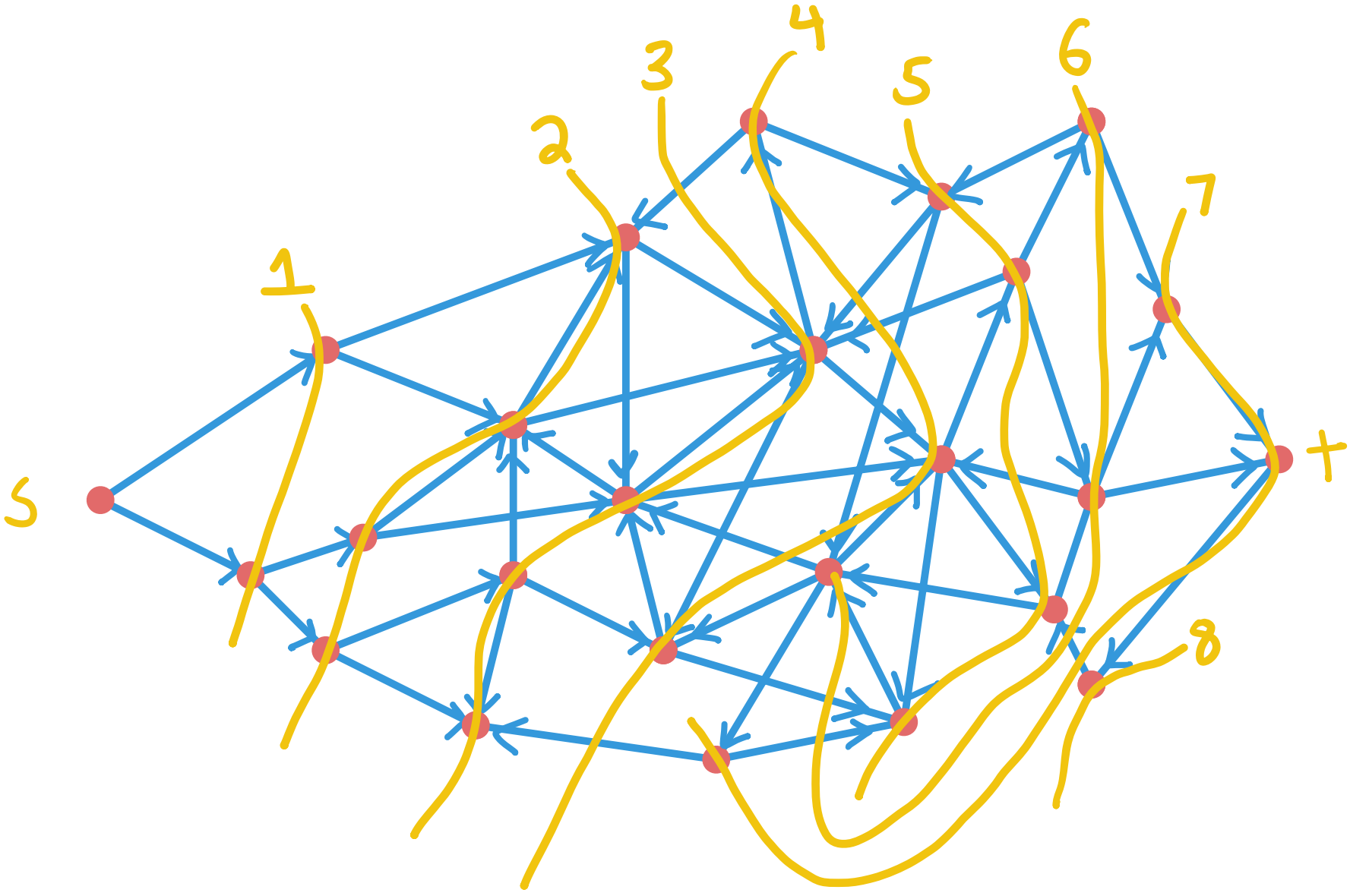
(infimum) (greatest lower bound)

- $d(s,t) = +\infty$ if s cannot reach t
- $d(s,t) = -\infty$ if there are arbitrarily negative length (s,t)-walks

Triangle inequality: $d(s,t) \leq d(s,u) + d(u,t)$



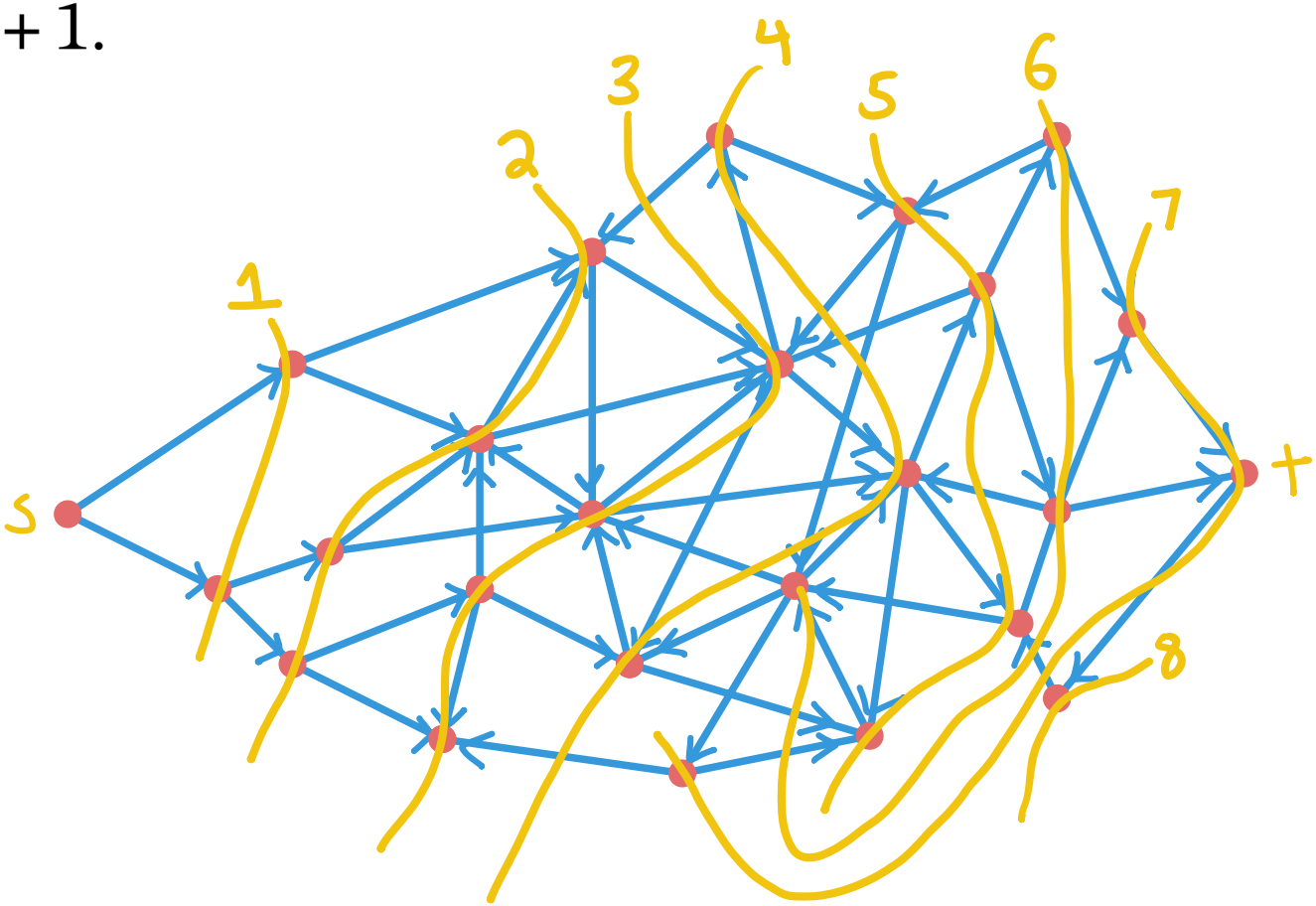
shortest way from s to t?



shortest way from s to t?

breadth-first-search (BFS)($G = (V, E)$, $s \in V$)

1. Set distance(s) = 0.
2. For $k = 0, 1, 2, \ldots$, as long as there is at least one vertex u with distance(u) = k:
   A. For all vertices u such that distance(u) = k:
      1. For all arcs (u, v) for which distance(v) is not yet set:
         a. set $d(v) = k + 1$.



O(mtn) time

(w/ simple bookkeeping)

Proof of correctness:

claim: for all $k \in \mathbb{Z}_{\geq 0}$,

$\{v : d(s,v) = k\} = \{v : \text{distance}(v) = k\}$

$\uparrow$ "A_k" "B_k" $\uparrow$

Base case $k=0$: $A_k = \{s\} = B_k$

General case $k>0$:

$B_k = \{v : v \notin B_0, \sim, B_{k-1}, (u,v) \in E \text{ for some } u \in B_{k-1}\}$

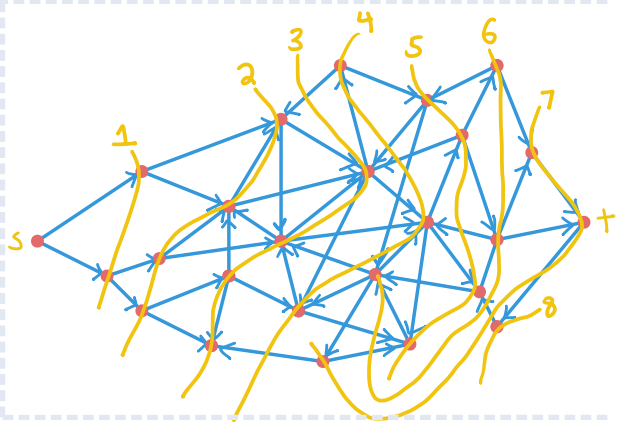
$=$ $=$ by induction $=$ by induction

$\{v : v \notin A_0, \sim, A_{k-1}, (u,v) \in E \text{ for some } u \in A_{k-1}\}$

$= A_k$

breadth-first-search (BFS)$(G = (V,E),\ s \in V)$

1. Set distance$(s) = 0$.
2. For $k = 0, 1, 2, \ldots$, as long as there is at least one vertex u with distance$(u) = k$:
   A. For all vertices u such that distance$(u) = k$:
      1. For all arcs (u,v) for which distance(v) is not yet set:
         a. set $d(v) = k+1$.



$O(m+n)$ time

breadth-first-search (BFS)($G = (V, E)$, $s \in V$)

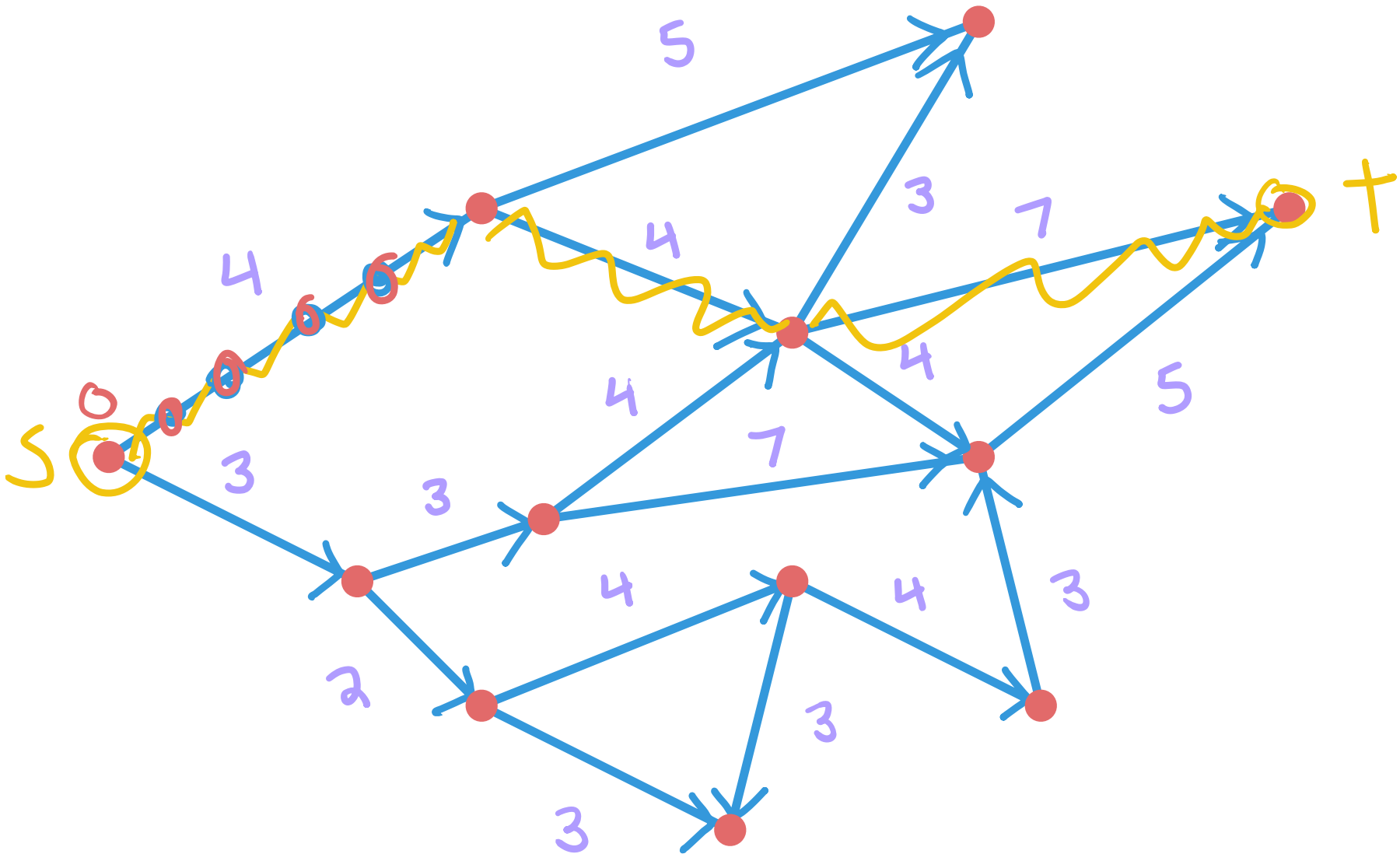
1. Set distance(s) = 0.
2. For $k = 0, 1, 2, \ldots$, as long as there is at least one vertex u with distance(u) = k:
   A. For all vertices u such that distance(u) = k:
      1. For all arcs (u, v) for which distance(v) is not yet set:
         a. set $d(v) = k + 1$.

BFS($G = (V, E)$, $s \in V$)

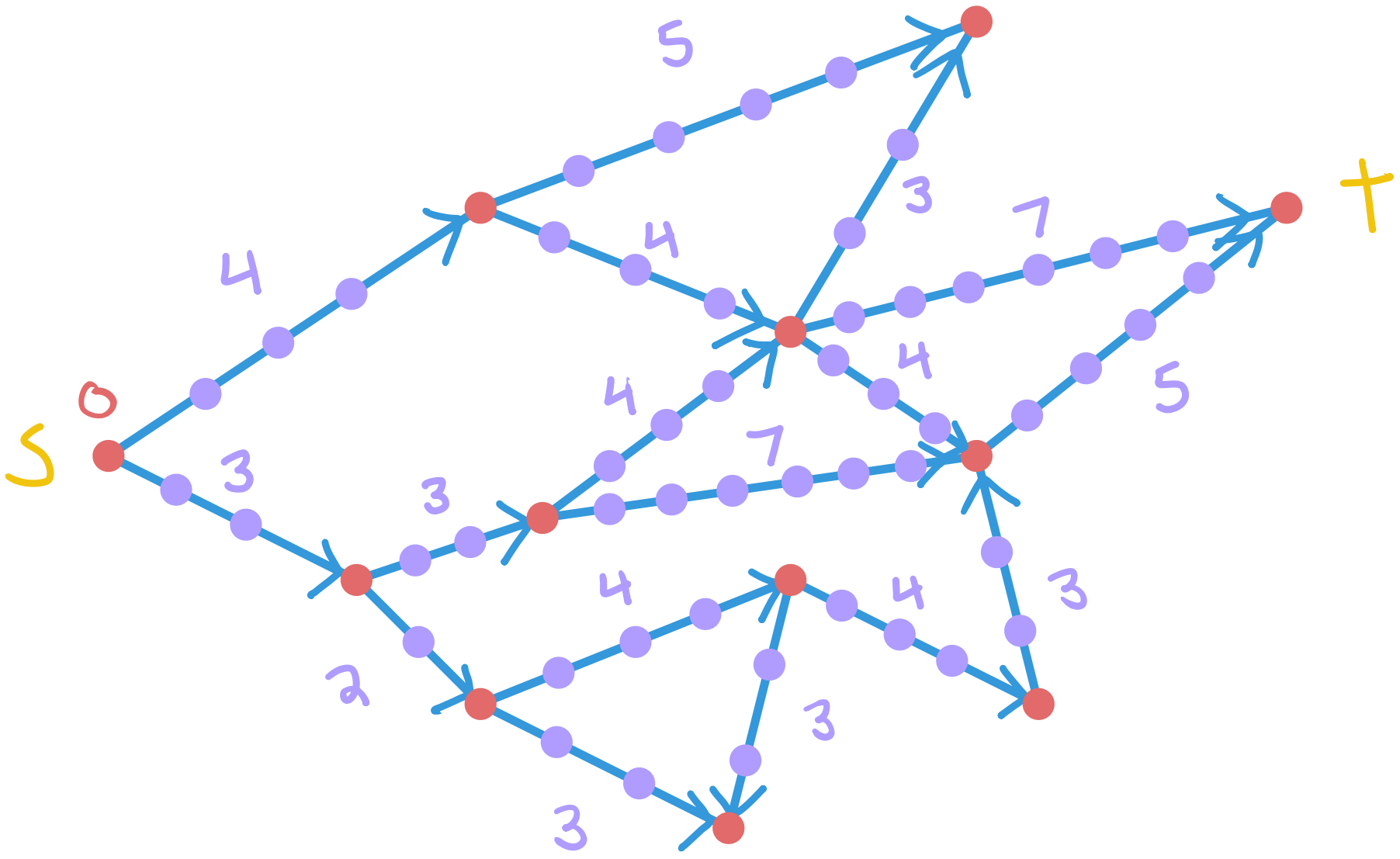
/* *A queue-based implementation of BFS.* */

1. Intialize an empty queue Q.
2. Set distance(s) $= 0$ and insert s into the queue Q.
3. While the queue Q is not empty.
   A. Dequeue the first vertex u in Q.
   B. For each arc (u, v) for which distance(v) is not yet set:
      1. Set distance(v) $=$ distance(u) $+ 1$.
      2. Enqueue v into Q.
4. Return all the distance labels.

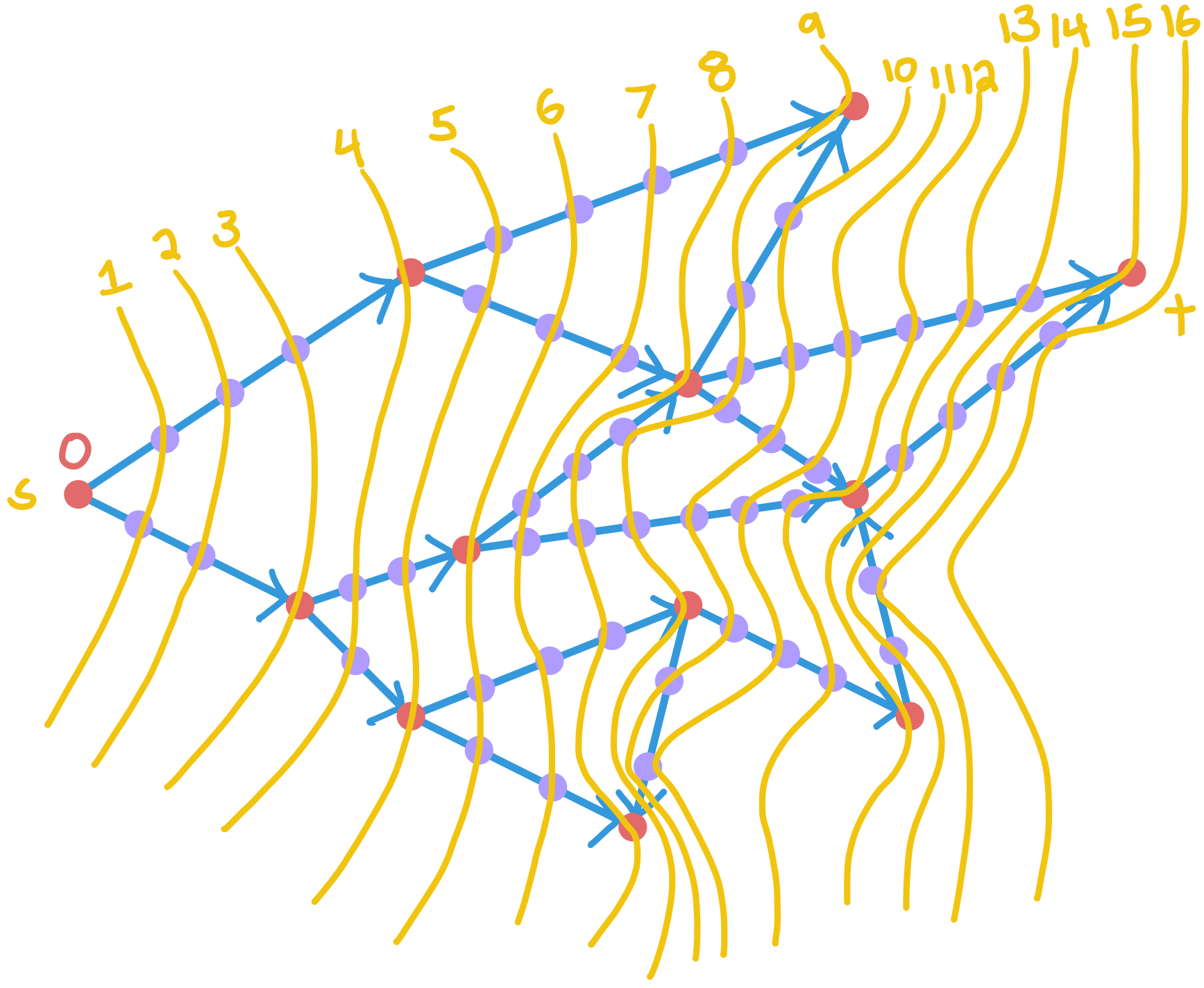
II positively weighted graphs



Suppose integer weights. how to reduce to unweighted?

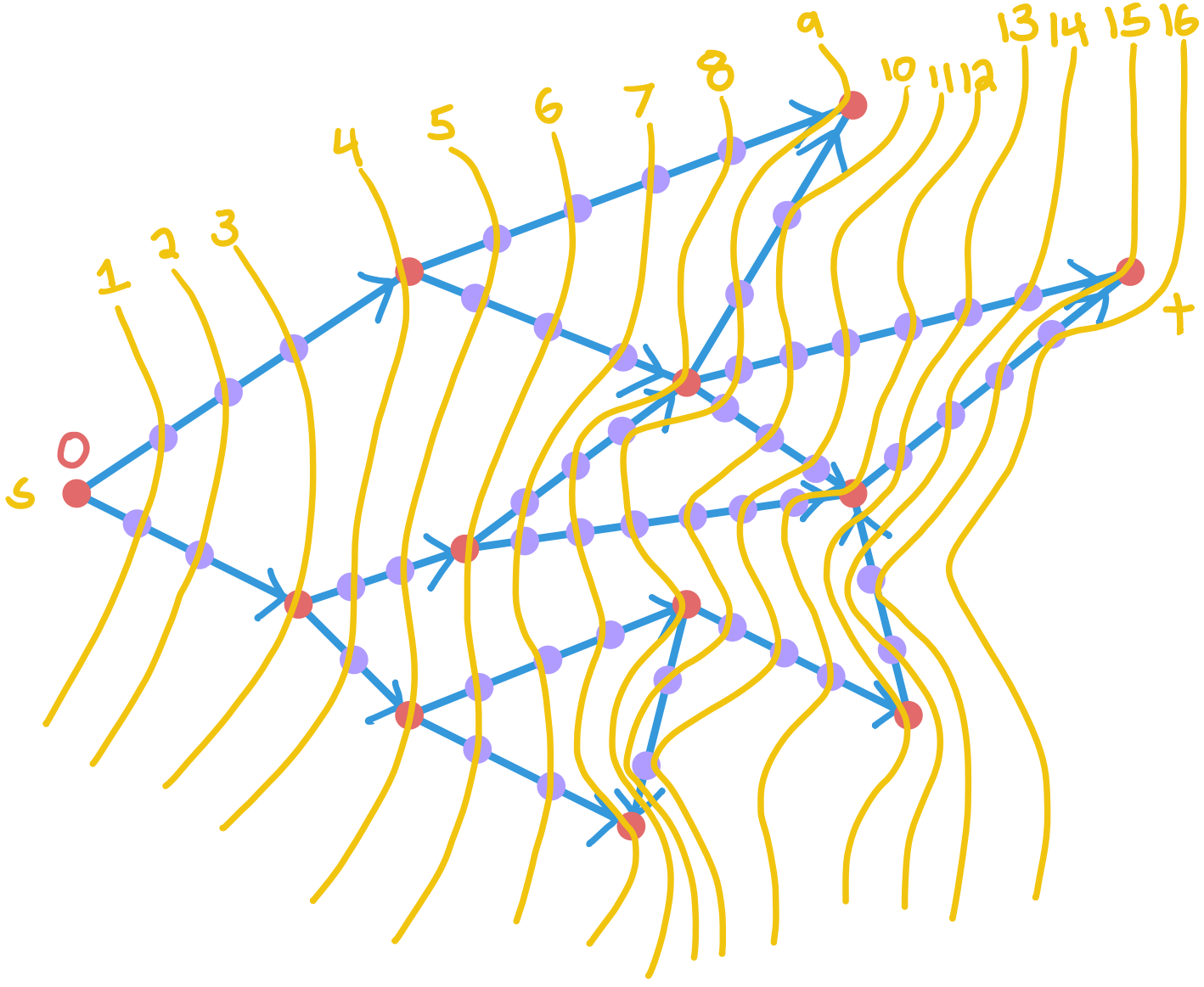


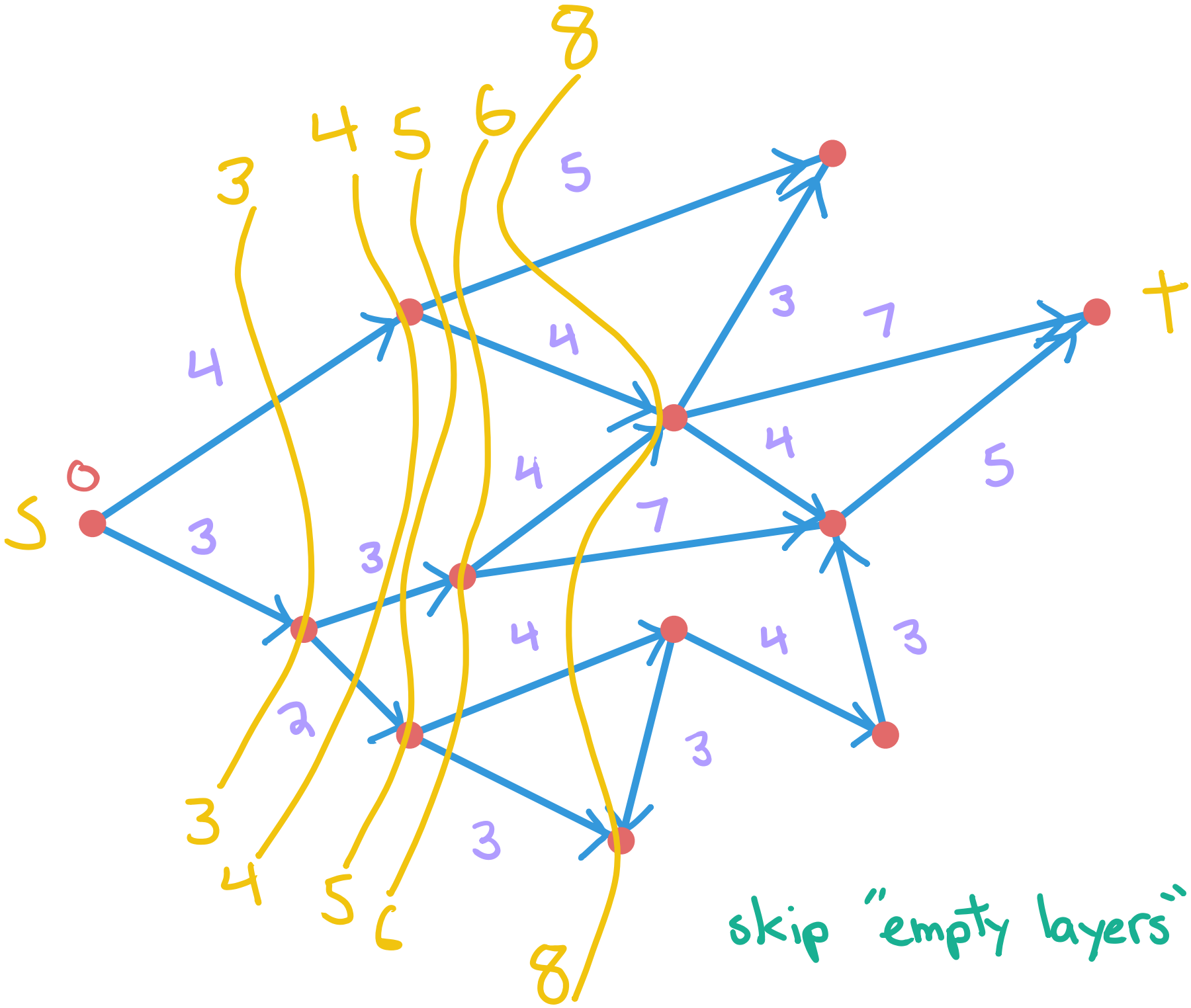
discretize


s
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
t

Only issue is running time

(lengths might be very large)

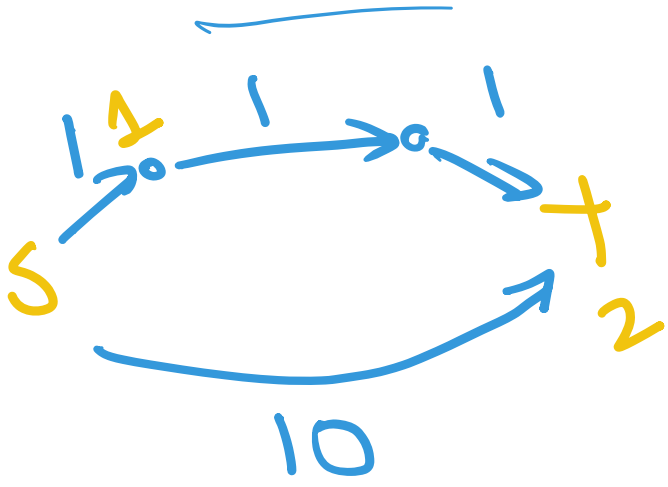


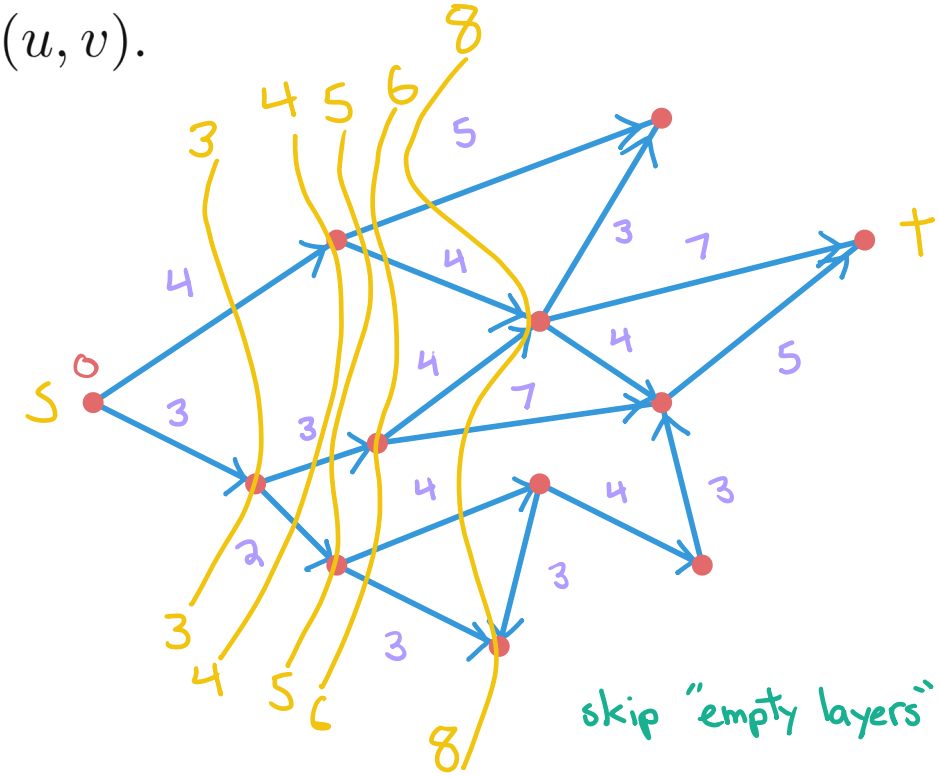

s
0
t
3
4
5
6
8
4
3
7
4
5
4
3
7
4
3
4
4
3
2
3
3
3
4
5
6
8
skip "empty layers"

Dijkstra($G = (V, E)$, $\ell : E \to \mathbb{R}_{>0}$, $s \in V$)

/ We assume for simplicity that all vertices are reachable from s; other vertices would be ignored. */*

1. Set distance(s) $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance(u) $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) $=$ distance(u) $+ \ell(u, v)$.
3. Return all the distance labels.





Proof of correctness.

let $v_1, v_2, \ldots$ list vertices in order that they are labeled

claim: for $i = 1, 2, \ldots$

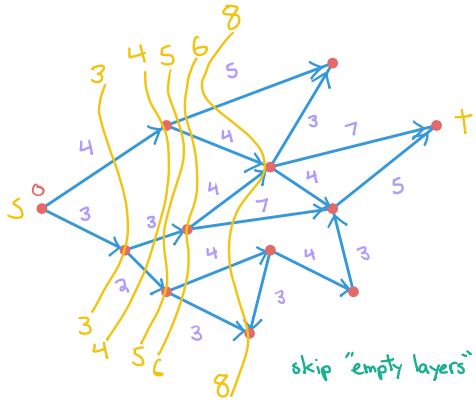
(a) v_i is the ith closest vertex from s

(b) $d(s, v_i) =$ distance(v_i)

Dijkstra$(G = (V, E),\ \ell : E \to \mathbb{R}_{>0},\ s \in V)$

/* *We assume for simplicity that all vertices are reachable from s; other vertices would be ignored.* */

1. Set distance$(s) = 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance$(u) + \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance$(v) =$ distance$(u) + \ell(u, v)$.
3. Return all the distance labels.



Proof of correctness.

let $v_1, v_2, \ldots$ list vertices in order that they are labeled

claim: for $i = 1, 2, \ldots$

(a) v_i is the ith closest vertex from s

(b) $d(s, v_i) = \texttt{distance}(v_i)$

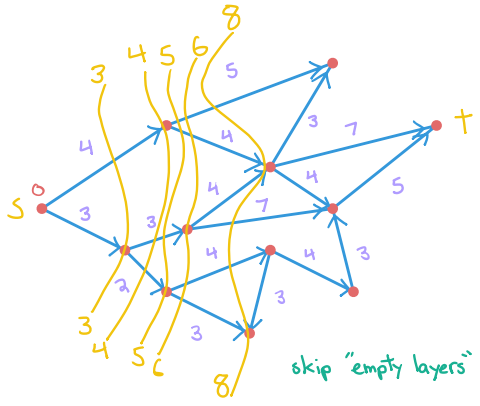
Base case: $i = 1$

$v_1 = s$, $d(s, s) = 0 = \texttt{distance}(s)$

`Dijkstra`$(G = (V, E),\ \ell : E \to \mathbb{R}_{>0},\ s \in V)$

/* *We assume for simplicity that all vertices are reachable from s; other vertices would be ignored.* */

1. Set `distance`$(s) = 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize `distance`$(u) + \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set `distance`$(v) =$ `distance`$(u) + \ell(u, v)$.
3. Return all the distance labels.



Proof of correctness.

let $v_1, v_2, \sim$ list vertices in order that they are labeled

claim: for $i = 1, 2, \sim$

(a) v_i is the ith closest vertex from s

(b) $d(s, v_i) =$ distance(v_i) true for $j < i$

General case: $i > 1$.

v_i = head x of edge $e = (v_j, x)$ minimizing

distance$(v_j) + \ell(e)$ s.t. $j < i$, $x \notin \{v_1, \sim, v_{i-1}\}$

becomes distance(v_i)

(ith closest vertex) = head x of edge $e = (v_j, x)$ minimizing

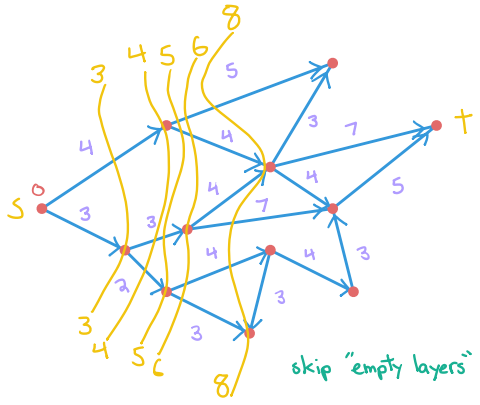
$d(s, v_j) + \ell(e)$ s.t. $j < i$, $x \notin \{v_1, \sim, v_{i-1}\}$

$= d(s, x)$

Dijkstra$(G = (V, E),\ \ell : E \to \mathbb{R}_{>0},\ s \in V)$

/* *We assume for simplicity that all vertices are reachable from s; other vertices would be ignored.* */

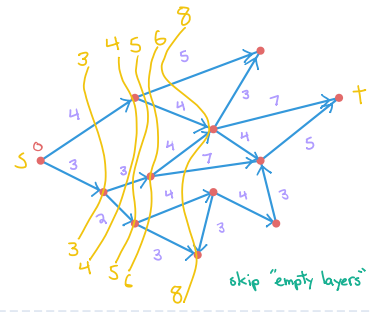
1. Set distance$(s) = 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance$(u) + \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) = distance$(u) + \ell(u, v)$.
3. Return all the distance labels.



Dijkstra($G = (V, E)$, $\ell : E \to \mathbb{R}_{\geq 0}$, $s \in V$)

/* *We assume for simplicity that all vertices are reachable from s; other vertices would be ignored.* */

1. Set distance(s) $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance(u) $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) = distance(u) $+ \ell(u, v)$.
3. Return all the distance labels.



Proof of correctness.

let $v_1, v_2, \sim$ list vertices in order that they are labeled

claim: for $i = 1, 2, \sim$

(a) v_i is the ith closest vertex from s

(b) $d(s, v_i)$ = distance(v_i)

General case: $i > 1$.

v_i = head x of edge $e = (v_j, x)$ minimizing

distance(v_j) $+ \ell(e)$ s.t. $j < i$, $x \notin \{v_1, \sim, v_{i-1}\}$

becomes distance(v_i)

equal by induction

(ith closest vertex) = head x of edge $e = (v_j, x)$ minimizing

$d(s, v_j) + \ell(e)$ s.t. $j < i$, $x \notin \{v_1, \sim, v_{i-1}\}$

$= d(s, x)$

Running time

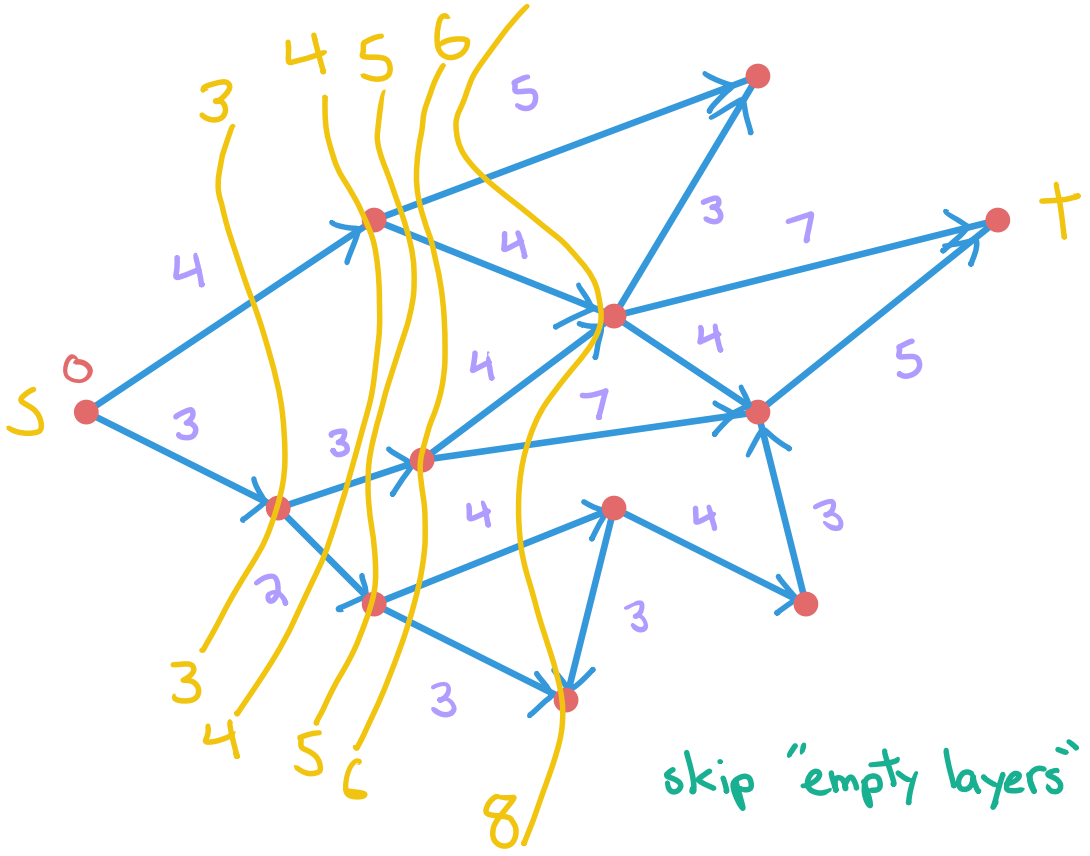
O(n) iterations →
× O(m) time →
O(mn) time (total)

Better?

Dijkstra($G = (V, E)$, $\ell : E \to \mathbb{R}_{>0}$, $s \in V$)

/ We assume for simplicity that all vertices are reachable from s; other vertices would be ignored. */*

1. Set distance(s) $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance(u) $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) $=$ distance(u) $+ \ell(u, v)$.
3. Return all the distance labels.

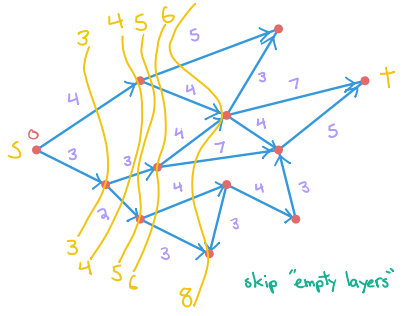


Tentative distances

Dijkstra($G = (V, E)$, $\ell : E \to \mathbb{R}_{>0}$, $s \in V$)

/* *We assume for simplicity that all vertices are reachable from s; other vertices would be ignored.* */

1. Set distance(s) $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize distance(u) $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) = distance(u) $+ \ell(u, v)$.
3. Return all the distance labels.



for unlabeled v,

"tentative(v)" = min distance(u) $+ \ell(u, v)$

w/ u labeled, $(u,v) \in E$

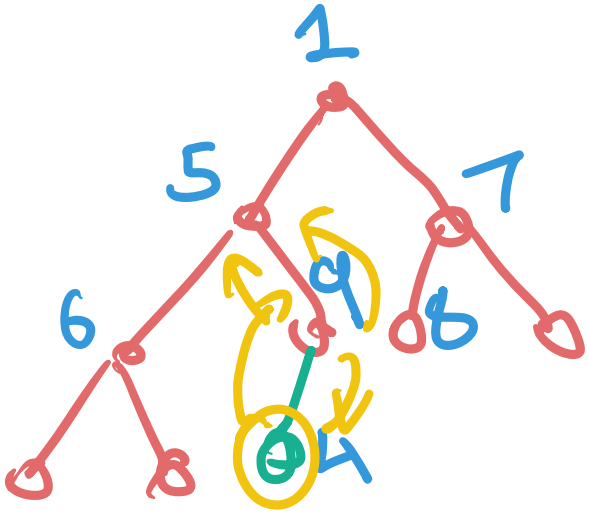
(best distance found so far)

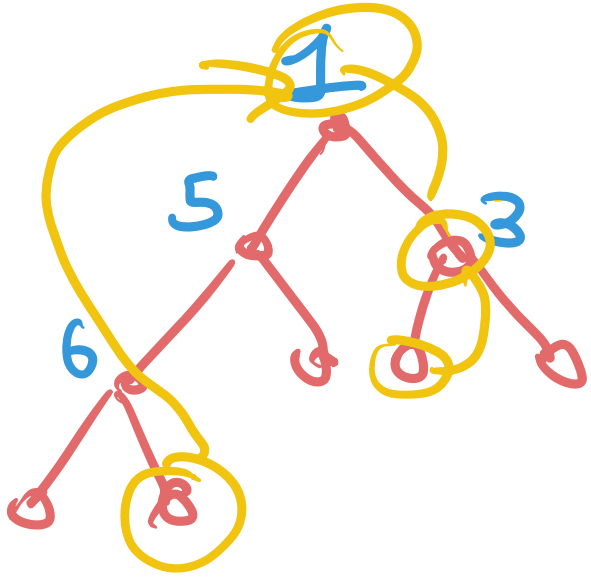
each iteration update some (not all) tentative(v)'s

Priority queues / heaps

$O(\log n)$

1. `insert`(k, p): inserts an key k with priority p.
2. `decrease`(k, p): Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.
3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.





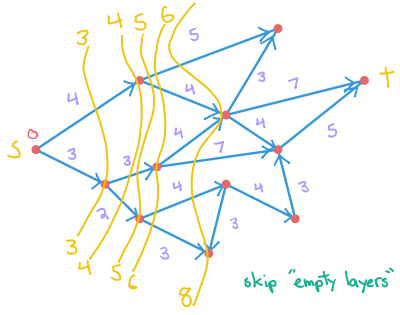
Dijkstra($G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V$)

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set tentative(s) = 0 and tentative(v) = $+\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in tentative(v).
3. While Q is not empty
   A. Remove the vertex v of minimum tentative(v) from Q.
   B. Set distance(v) = tentative(v).
   C. For each outgoing edge (v, w):
      1. Set

$$\text{tentative}(w) = \min\{\text{tentative}(w), \text{distance}(v) + \ell(v, w)\}$$

      and decrease w's priority in Q accordingly.
4. Return the distance labels.



Running time w/ O(log n)-time heaps

Dijkstra($G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V$)

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

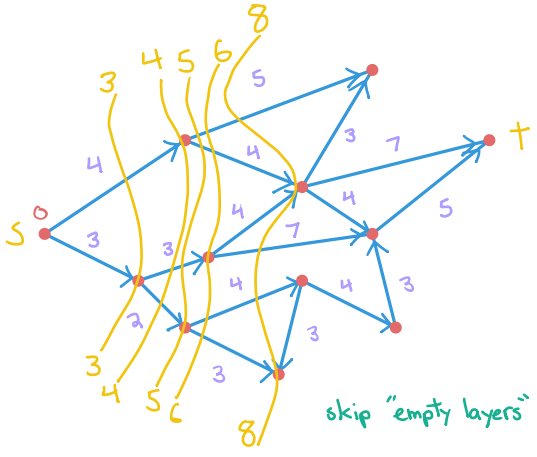
1. Set tentative(s) = 0 and tentative(v) = $+\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in tentative(v).
3. While Q is not empty
   A. Remove the vertex v of minimum tentative(v) from Q.
   B. Set distance(v) = tentative(v).
   C. For each outgoing edge (v, w):
      1. Set

$$\text{tentative}(w) = \min\{\text{tentative}(w), \text{distance}(v) + \ell(v, w)\}$$

      and decrease w's priority in Q accordingly.
4. Return the distance labels.

O(log n) — once per edge → log n

(m+n) log n



Priority queues / heaps

1. `insert(`k`,`p`)`: inserts an key k with priority p.
2. `decrease(`k`,`p`)`: Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.
3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.

Fact 8.1. *There exists a data structure, called a* **Fibonacci heap**, *that has the following guarantee. Over any sequence of I calls to* `insert-key`, D *calls to* `decrease-key`, *and R calls to* `remove-min`, *the Fibonacci heap takes*

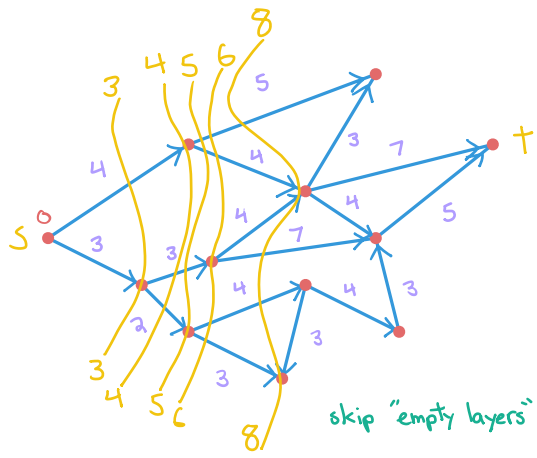
$$O(I + D + R \log n)$$

time in total.

Fact 8.1. *There exists a data structure, called a* **Fibonacci heap**, *that has the following guarantee. Over any sequence of I calls to* `insert-key`, *D calls to* `decrease-key`, *and R calls to* `remove-min`, *the Fibonacci heap takes*

$$O(I + D + R \log n)$$

time in total.



Running time w/ Fibonacci heaps

`Dijkstra`$(G = (V,E), \ell : E \to \mathbb{R}_{>0}, s \in V)$

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set `tentative`$(s) = 0$ and `tentative`$(v) = +\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in `tentative`(v).
3. While Q is not empty
   - A. Remove the vertex v of minimum `tentative`(v) from Q.
   - B. Set `distance`(v) = `tentative`(v).
   - C. For each outgoing edge (v,w):
     1. Set

$$\texttt{tentative}(w) = \min\{\texttt{tentative}(w), \texttt{distance}(v) + \ell(v,w)\}$$

     and decrease w's priority in Q accordingly.
4. Return the distance labels.

n iterations

O(log n)

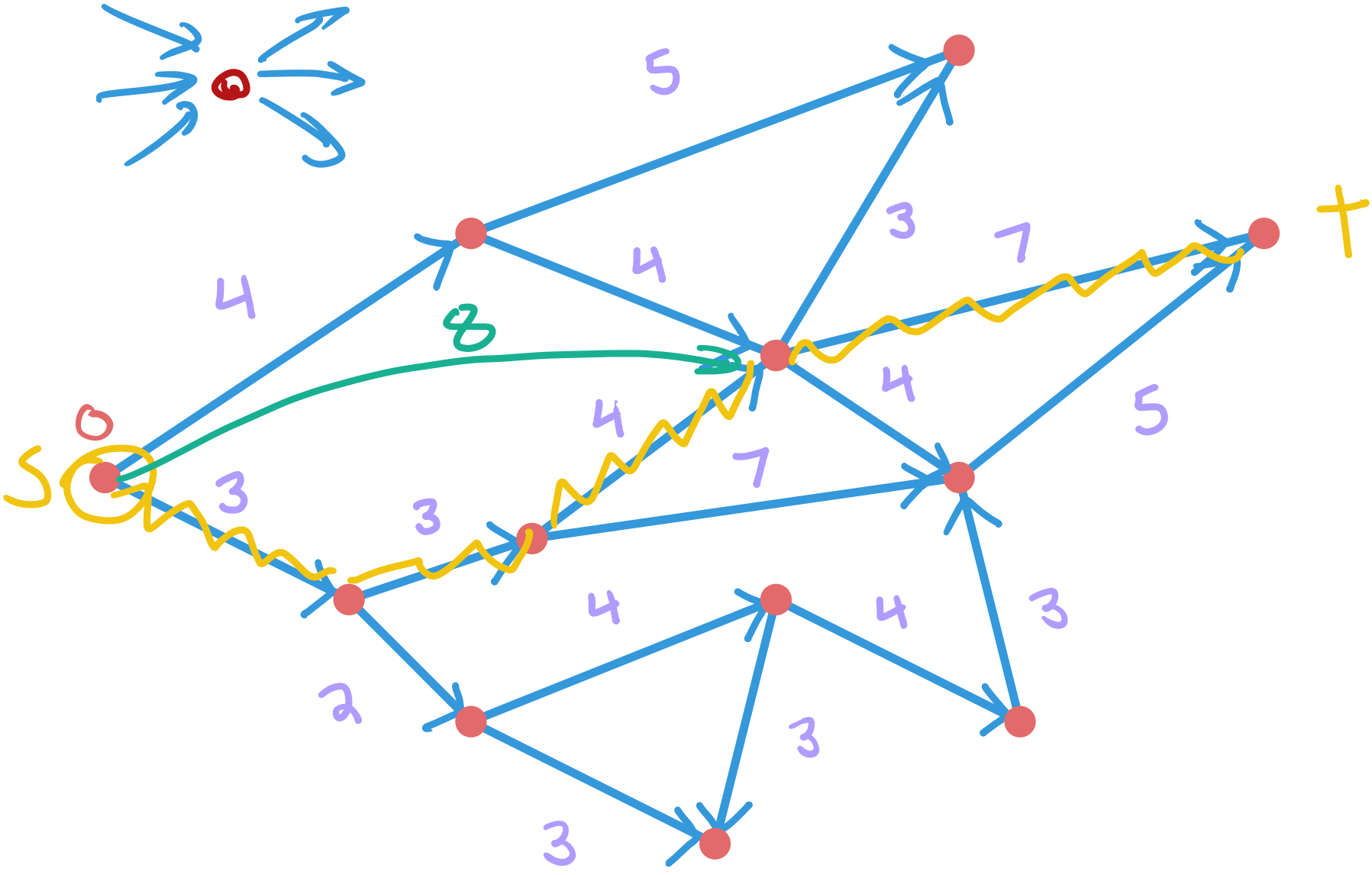
Once per edge

O(1)

O(m + n log n)

Exercise 7.1.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]

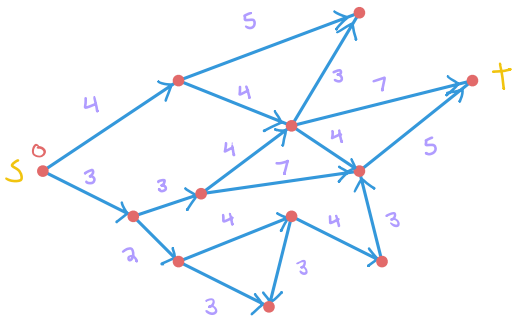


Exercise 7.1.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]

tentative(s, odd) = too even

even, odd



`Dijkstra`$(G = (V,E), \ell : E \to \mathbb{R}_{>0}, s \in V)$

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set $\texttt{tentative}(s) = 0$ and $\texttt{tentative}(v) = +\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in $\texttt{tentative}(v)$.

(v,p)

m + n log n

3. While Q is not empty
   A. Remove the vertex v of minimum $\texttt{tentative}(v)$ from Q.

   (v,p)

   B. Set $\texttt{distance}(v) = \texttt{tentative}(v)$.

   (v,p)

   C. For each outgoing edge (v,w):

   q = not p

      1. Set

      w,q w,q v,p

$$\texttt{tentative}(w) = \min\{\texttt{tentative}(w), \texttt{distance}(v) + \ell(v,w)\}$$

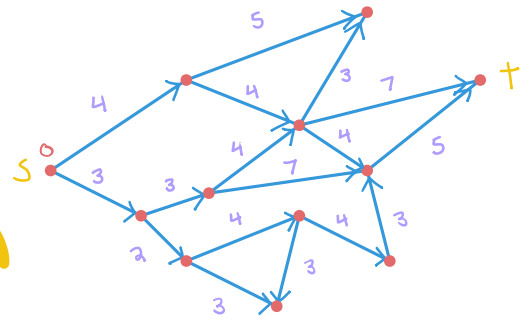
      and decrease w's priority in Q accordingly.

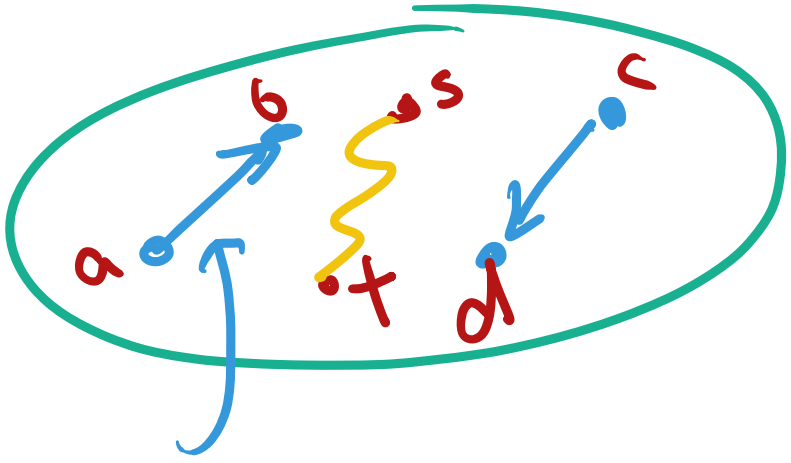
      w,q

4. Return the distance labels.

Exercise 7.1.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]





$|\bar{V}| = 2|V| = 2n$

$|\bar{E}| = 2|E| = 2m \Rightarrow O(2m + 2n\log(2n)) = O(m + n\log n)$

$G = (V, E)$

$\bar{G} = (\bar{V}, \bar{E})$

even

$\bar{V} = V_{odd} \cup V_{even}$

$\bar{E} = \{(u_{odd}, v_{even}) : (u,v) \in E\}$

odd

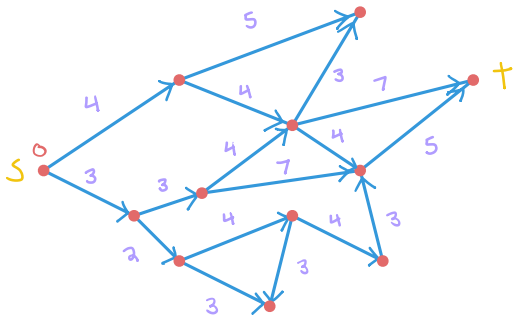
$\cup \{(u_{even}, v_{odd}) : (u,v) \in E\}$

$\bar{\ell}(u_{odd}, v_{even}) = \ell(u,v)$

$\bar{\ell}(u_{even}, v_{odd}) = \ell(u,v$

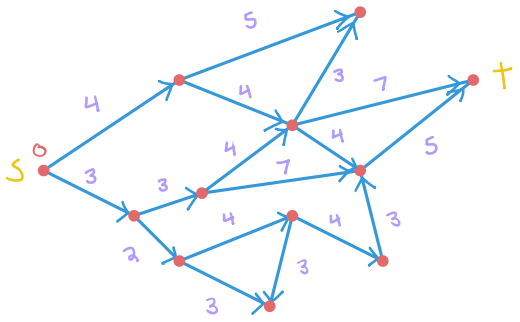
Exercise 7.1.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]



Exercise 7.1.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]



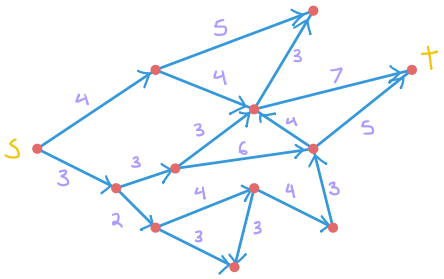
Chapter 7

Shortest walks

We have previously discussed algorithms for reachability: whether or not there *exists* a path between two vertices. Today's discussion is instead about finding the *shortest* path between two vertices.

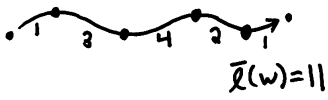
Of course this is a very practical problem; given a road map perhaps annotated with traffic (or weather!) conditions, we want the shortest driving directions from home to work. In a more general sense of planning problems, where vertices correspond to states, edges correspond to transitions between states, and edge lengths correspond to the time or cost of each transition, we may want to go from one state to another with minimum total cost.

7.1 Distances



Let $G = (V, E)$ be a directed graph and let $\ell : E \to \mathbb{R}$ assigned real-valued *lengths* to all the edges. The *length of a walk* w is defined as the sum of edge lengths. We denote the length of w as

$$\bar{\ell}(w) \stackrel{\text{def}}{=} \sum_{e \in w} \ell(e)$$



Here the sum is over all edges in the walk w (with repetition).

The *distance* between two vertices s and t is defined (mathematically) as the

infimum[1] over the lengths of all walks from s to t:

$$d(s,t) = \inf\Big\{\bar{\ell}(w) : w \text{ is a walk from } s \text{ to } t\Big\}.$$

By definition, this quantity is ∞ if s cannot reach t, and $-\infty$ if this quantity is arbitrarily small[2]. This chapter is about the problem of computing the distance from one vertex s to another vertex t.

The most basic setting is unweighted graphs (i.e., $\ell(e) = 1$ for all e). Then the distance is the minimum number of edges required to get from s to t. The next most basic setting is positively weighted graphs. That is, $\ell(e) > 0$ for all edges e. The most general allows for both positive and negative weights. In this chapter we will focus on the unweighted and positive settings and postpone negative edge weights for the following chapter.

Thus let $G = (V, E)$ have positive edge weights $\ell : E \to \mathbb{R}_{>0}$. For simplicity, the first-time reader may want to assume all edges have length 1. Given two vertices $s, t \in V$, the goal is to compute the length of the shortest walk from s to t. Observe that for positive weights, the shortest (s,t)-walk is always a path. (Why?) So the problem is the same as finding the shortest (s,t)-path.

7.2 Distances in a DAG

Let $G = (V, E)$ be a DAG, with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. Given $s, t \in V$, the goal is to compute the shortest (s,t)-path. Towards a recursive algorithm let us declare

> `distance(`v`)` as the length of the shortest path from s to v.

We want to compute `distance(`t`)`. Now, if $t = s$, then the distance is 0. Otherwise, there must be some vertex v preceding t on the shortest (s,t)-path. Then `distance(`t`)` $=$ `distance(`v`)` $+ \ell(v,t)$. But which in-neighbor v? Who knows. Let us recursively compute `distance(`v`)` over all incoming edges (v,t) and chose the one minimizing `distance(`v`)` $+ \ell(v,t)$. All put together we have the following recursive

[1] The infimum is defined as the greatest lower bound: the greatest value x that is less than all values in the corresponding set. It is usually the same as minimum but allows for $-\infty$ when the values are not bounded below and $+\infty$ when the set is empty.

[2] More formally: the distance is $-\infty$ if for all $L \in \mathbb{R}$, there exists an (s,t) walk w of total length $\bar{\ell}(w) \leq L$

algorithm.

$$\texttt{distance}(t) = \begin{cases} 0 & \text{if } t = s, \\ +\infty & \text{if } t \text{ has no incoming edges,} \\ \min_{(v,t)\in\delta^-(t)} \texttt{distance}(v) + \ell(v,t) & \text{otherwise.} \end{cases}$$

Why is the algorithm correct? Why does it avoid recursive loops? What is the underlying inductive hypothesis?

Proof of correctness. Recall that G is a DAG. Let $v_1, \dots, v_n$ list the vertices in topological order. Of every edge (v_i, v_j), we have $i < j$. We claim, by induction on i, that $\texttt{distance}(v_i)$ returns $d(s, v_i)$ for all i.

In the base case, $i = 1$, v_1 is a source (with no incoming arcs). In that case $d(s, v_1) = 0$ if $v_1 = s$ and $+\infty$ otherwise, which is same as $\texttt{distance}(v_1)$ returns. Now suppose $i > 1$. By induction, $\texttt{distance}(v_j) = d(s, v_j)$ for all $j < i$. If $v_i = s$ then $\texttt{distance}(v_j) = 0 = d(s, v_i)$.

Otherwise, $d(s, v_i)$ is the infimum length over all walks from s to v_j. If there is no walk from s to v_j, then for all arcs (v_j, v_i) with head v_i, we also have $d(s, v_j) = +\infty$. Moreover, an arc (v_j, v_i) implies that $j < i$, and by induction, $\texttt{distance}(s, v_j) = +\infty$. Consequently $\texttt{distance}(v_i)$ returns $+\infty$, as desire.

In the final case, $s \neq v_j$, and s can reach v_j. For all arcs (v_j, v_i) with head v_i, we have $j < i$, hence $\texttt{distance}(v_j) = d(s, v_j)$. Thus

$$\begin{aligned} \texttt{distance}(v_i) &= \inf_{(v_j,v_i)} \texttt{distance}(v_j) + \ell(v_j, v_i) \\ &= \inf_{(v_j,v_i)} d(s, v_j) + \ell(v_j, v_i). \end{aligned}$$

By the triangle inequality,

$$d(s, v_i) \leq d(s, v_j) + \ell(v_j, v_i)$$

for all arcs (v_j, v_i), hence

$$\texttt{distance}(v_i) \geq d(s, v_i).$$

Meanwhile, let w be the minimum length walk from s to v_i, and let v_k be the penultimate vertex on the walk. Then

$$d(s, v_i) = \bar{\ell}(w) \geq d(s, v_k) + \ell(v_k, v_i),$$

so

$$\begin{aligned}\texttt{distance}(v_i) &= \inf_{(v_j,v_i)} d(s,v_j) + \ell(v_j,v_i) \\ &\leq d(s,v_k) + \ell(v_k,v_i) \leq d(s,v_i).\end{aligned}$$

Thus $\texttt{distance}(v_i) = d(s,v_i)$, as desired.

Running time. The initial recursive algorithm may not be fast, but we can make it efficient by saving all our answers. With dynamic programming, the running time becomes the sum, over all $t \in V$, of the number of incoming edges to t (plus a constant). This gives a $O(m+n)$ running time in total.

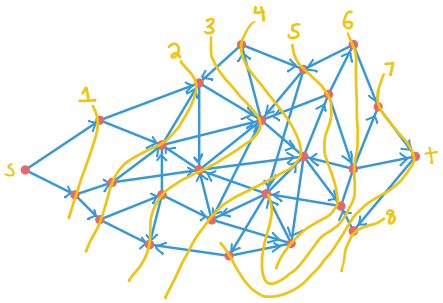
If the algorithm seems similar to the longest path algorithm from section 13.1, that's because they are essentially the same. One can extend the `longest-walk` algorithm for DAGs to real-valued edge lengths and moreover one can observe that the edge lengths do not have to be positive. Assuming this extension, then to find the shortest paths in a weighted DAG G, we could have negated all the edge lengths and found the longest path with respect to the negated edge lengths instead.

7.3 Unweighted shortest paths

We now consider shortest paths in general directed graphs. To ease into it we first focus on unweighted graphs: the length of a path is the number of edges.

When working with a DAG above, we could rely on the topological ordering to decide in which order of vertices to calculate the distances. General graphs do not have a topological order and we do not know *a priori* in which order to calculate the vertex distances. However, we do know:

0. *The vertices at distance* 0*:* This is just $\{s\}$.
1. *The vertices at distance* 1*:* These are just the vertices v with an arc (s,v) from s.
2. *The vertices at distance* 2*:* These are the vertices v that (a) have an arc (u,v) from some vertex u at distance 1, and (b) aren't at distance 0 or 1.
3. *The vertices at distance* 3*:* These are the vertices v that (a) have an arc (u,v) from some vertex u at distance 2, and (b) aren't at distance 0, 1, or 2.



Clearly a pattern has started to emerge. In general, for any integer k,

k. *The vertices at distance k:* the vertices v that (a) have an arc (u, v) from some vertex u at distance $k-1$, and (b) are not at distance $0, \dots, k-1$.

Note the inherent inductive structure in our observations: we can identify the distance k vertices only once we have identified the distances of all vertices at distance $\leq k-1$. We can encode our observations into an algorithm as follows.

`breadth-first-search (BFS)`$(G = (V, E),\ s \in V)$

1. Set `distance`$(s) = 0$.
2. For $k = 0, 1, 2, \dots$, as long as there is at least one vertex u with `distance`$(u) = k$:
   A. For all vertices u such that `distance`$(u) = k$:
      1. For all arcs (u, v) for which `distance`(v) is not yet set:
         a. Set $d(v) = k + 1$.

Proof of correctness. One can formally prove correctness by induction on k. The induction hypothesis is that for all $k \in \mathbb{Z}_{\geq 0}$, for all v with $d(s, v) = k$, `BFS` sets $d(v) = d(s, v)$. In the base case, $k = 0$, and s is the only vertex at distance 0. For $k > 1$, assume by induction that $d(u) = d(s, u)$ for all u with $d(s, u) < k$. As argued above, the vertices at distance k are those that (a) have an edge from a vertex at distance $k-1$, and (b) are not at distance less than k. Since we have already labeled all distances $< k$ by induction, (b) is the same as being unlabeled. Indeed, the algorithm labels all unlabeled vertices with arcs from vertices at label $k-1$, as desired.

Running time. It is easy to see that the algorithm runs in linear time with some obvious bookkeeping (like keeping track of all the vertices at distance k for each k). Each vertex assumes the role of u once in the loop at `(2.A)`. We then traverse each outgoing edge (u, v) from u and that is the only time we examine that edge. Overall we process each vertex and each edge once, and we have a $O(m + n)$ time overall.

An implicit DAG. The algorithm above is implicitly very similar to our algorithm for DAG's. Suppose we drop from G all the edges that do not appear in shortest paths from s. Every remaining edge goes from a vertex at distance k to a vertex distance $k+1$ for some k. (Such an edge is sometimes called a *tight edge*.) This subgraph is a DAG, and ordering the vertices by distance from s gives a topological ordering.

Queue-based implementation. There is a more compact way to implement BFS relying on a queue to order the vertices as they are processed. Consider the following code.

BFS($G = (V, E)$, $s \in V$)

/* *A queue-based implementation of BFS.* */

1. Intialize an empty queue Q.
2. Set distance(s) $= 0$ and insert s into the queue Q.
3. While the queue Q is not empty.
   A. Dequeue the first vertex u in Q.
   B. For each arc (u, v) for which distance(v) is not yet set:
      1. Set distance(v) $=$ distance(u) $+ 1$.
      2. Enqueue v into Q.
4. Return all the distance labels.

The high level idea is that the distance labels moving through the queue are increasing, and a vertex of distance label k is dequeued only after all vertices of distance label $k - 1$ have been completely processed. We leave it to the reader to formally verify that the queue-based version above simulates the first version. It can also be understood as a specialization of the upcoming algorithm for weighted graphs.

7.4 Weighted shortest paths

We now move on to the weighted case. Let G have positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. Again we want to find the lengths of the shortest paths from a fixed vertex s.

In the unweighted case, we knew that all the distances were integers, and we could inductively partition the vertices by distance layers. Weighted distances are not so simple to enumerate. That said, we do know that:

1. The closest vertex from s is s, at distance $d(s, s) = 0$.
2. The second closest vertex from s is the endpoint v_2 of the edge $e = (s, v_2)$ of minimum length $\ell(e)$; we then have $d(s, v_2) = \ell(e)$.

Things get a little more interesting when it comes to identifying the third closest vertex, which we denote v_3. v_3 might be the opposite endpoint of the second smallest edge (s, x) leaving s, but it does not have to be. It is possible that for an edge (v_2, y) leaving v_2, the combined distance $d(s, v_2) + \ell(v_2, y)$ is less than the length of any edge (s, x) leaving x. We will also consider this possibility and take the minimum of either scenario, as follows.

3. The third closest vertex v_3 is, among all vertices outside of $\{v_1 = s, v_2\}$ either (a) the endpoint x of the edge $e = (s, x)$ of minimum length $\ell(e)$ or (b) the endpoint x of the edge $e = (v_2, x)$ minimizing $d(s, v_2) + \ell(e)$. In case (a), the distance to x is $\ell(s, x)$; in case (b), the distance to x is the sum $d(s, v_2) + \ell(s, x)$.

This sets a pattern where the next vertex is based on the distances via all preceding vertices. Inductively, for $k \in \mathbb{N}$, if we let $v_1, \ldots, v_{k-1}$ denote the $k-1$ closest vertices from s:

k. The kth closest vertex v_k is, among all vertices outside of $\{v_1, \ldots, v_{k-1}\}$, the endpoint x of the edge $e = (v_i, x)$ minimizing $d(s, v_i) + \ell(e)$, over $i \in \{1, \ldots, k-1\}$.

Similarly to unweighted distances, an inductive structure has emerged. *A priori* it is not clear which vertex is the kth closest from s. But it is much easier after identifying the first $k-1$ vertices in distance from s.

We transfer our observations to the following algorithm sketched at a high-level below. (More concrete implementation details will be supplied momentarily). The algorithm is named after Edgar Dijkstra who published it in 1959.

`Dijkstra(`$G = (V, E)$`, ` $\ell : E \to \mathbb{R}_{>0}$`, ` $s \in V$`)`

/ We assume for simplicity that all vertices are reachable from s; other vertices would be ignored. */*

1. Set `distance(`s`)` $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize `distance(`u`)` $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to endpoints v that have not yet been assigned a distance.

   B. Set `distance(`v`)` $=$ `distance(`u`)` $+ \ell(u, v)$.
3. Return all the distance labels.

Figure 7.1: High-level implementation of Dijkstra's algorithm for single-source shortest-paths.

Proof of correctness. One can formally prove correctness by induction. Let $(v_1 = s), v_2, \ldots$ be the vertices processed by the algorithm, in order. We claim that for all $i \in \mathbb{Z}_{\geq 0}$, v_i is the ith closest vertex to s, and the algorithm sets `distance(`v_i`)` $= d(s, v_i)$.

The base case is $i = 1$, in which case $v_1 = s$ and $d(s) = 0 = d(s, s)$. Let $i > 1$. By induction, $v_1, \ldots, v_{i-1}$ are the $i-1$ closest vertices (in order), and each have

their distance label set to their distance from s. Above we observed that the ith closest vertex is the head x of the arc $e = (v_j, x)$ minimizing $d(s, v_j) + \ell(e)$, over $j \in \{1, \dots, i-1\}$. By induction, $d(s, v_j) = d(v_j)$ for all $j \in \{1, \dots, i-1\}$, so the ith closest vertex is also the head x of the arc $e = (v_j, x)$ minimizing $d(v_j) + \ell(e)$, over $j \in \{1, \dots, i-1\}$. The latter is exactly how the algorithm selects and labels v_i.

Remark 7.1. The proof above is unique from previous inductive proofs we've seen for the following reason: the ordering $v_1, v_2, \dots, v_n$ on which we make our induction argument is not given *a priori*, but emerges from the algorithm itself.

Running time. `Dijkstra`'s algorithm is following our inductive approach: the next distance we identify is based on the distances identified so far. As given, however, it is fairly slow. Each iteration of the outer loop produces one distance label. The minimization in `(2.A)` is implicitly another loop: over all labeled vertices u and all edges leaving u. This adds up to a loop over all the edges in the graph and takes $O(m)$ time. So we have a $O(mn)$ time algorithm for computing distances from s.

Speeding up Dijkstra's algorithm. The bottleneck is step `(2.A)`. Identifying the next closest unlabeled vertex takes $O(m)$ time, and we do this $O(n)$ times. This approach treats each search for u and v as completely independent from one iteration to the next. But of course the searches have a lot in common. They are all in the same graph. The only difference between one iteration of `(2.A)` and the next is that one more vertex has been labeled, opening up one more option for u.

For each unlabeled vertex v, consider the minimum of `distance(`u`)` $+ \ell(u, v)$ over all incoming edges (u, v) where u is already labeled (or $+\infty$ if there is no such u). This value represents the length of the shortest path to v *so far*; let us call it the "tentative distance" to v. Each iteration we are identifying the vertex v of minimum tentative distance, and finalizing that tentative distance the real distance to v. Labeling v may then decrease the tentative distance to vertices w where there is a directed edge (v, w). But all the other tentative distances stay the same. Perhaps we could track the tentative distances in a *data structure*, that only needs to be updated when tentative distances are revised. This data structure should also be able to quickly provide the next vertex of minimum tentative distance.

Such a data structure is given by a heap or prioirity queue. We assume the reader is acquainted with heaps but we briefly review the key points. A *priority queue / heap* is a data structure designed to keep track of the key with the smallest value over a collection of key-value pairs. In this context the value is sometimes called a "priority". For the sake of our discussion, a heap is defined by the following three basic operations. (Here we describe a min-heap but a max-heap can be defined analogously.)

1. `insert(`k`,`p`)`: inserts an key k with priority p.

`Dijkstra(`$G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V$`)`

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set `tentative(`s`)` $= 0$ and `tentative(`v`)` $= +\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in `tentative(`v`)`.
3. While Q is not empty
   A. Remove the vertex v of minimum `tentative(`v`)` from Q.
   B. Set `distance(`v`)` $=$ `tentative(`v`)`.
   C. For each outgoing edge (v, w):
      1. Set

$$\texttt{tentative}(w) = \min\{\texttt{tentative}(w), \texttt{distance}(v) + \ell(v, w)\}$$

      and decrease w's priority in Q accordingly.
4. Return the distance labels.

Figure 7.2: Accelerating Dijkstra's algorithm with a priority queue.

2. `decrease(`k, p`)`: Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.

3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.

A standard array-backed heap takes $O(\log n)$ time for each of the above operations, where n is the number of items in the heap.

To accelerate Dijkstra's algorithm, we can use a priority queue over the unlabeled vertices using the tentative distance as the priority. To keep the priority queue updated, whenever we label a vertex u, for each edge $(u, v) \in E$, we may revise the tentative distance to v and update the heap via the `decrease` operation. By keeping the priorities updated, we can retrieve the next vertex v with a call to `remove-min()`. See fig. 7.2.

As mentioned above, each heap operation takes $O(\log n)$ time in an array-backed heap. Thus, when using an array-backed heap, Dijkstra's algorithm takes

$$\begin{aligned} &O(\log n) \times (\#\ \texttt{decrease-key's}) + O(\log n) \times (\#\ \texttt{remove-min's}) \\ &= O(m \log n) + O(n \log n) = O(m \log n). \end{aligned}$$

This is a considerable improvement on $O(mn)$. It is also a very reasonable running time; it is as if we were sorting all the edges. Still, one can do better with the following remarkable data structure due to Fredman and Tarjan [FT87].

Fact 7.2. *There exists a data structure, called a* Fibonacci heap*, that has the following guarantee. Over any sequence of I calls to* `insert-key`*, D calls to* `decrease-key`*, and R calls to* `remove-min`*, the Fibonacci heap takes*

$$O(I + D + R \log n)$$

time in total.

Note that the running time above only guarantees that each `decrease-key` takes $O(1)$ time *on average*, but a single operation may take longer in the worst-case. This "on-average" guarantee is fine for our purposes where we are only concerned in the total running time over the course of a shortest path algorithm. Such averaging types of guarantees are proven by a technique called *amortized analysis*. We will come back to amortized analysis later in the course where we may analyze Fibonacci heaps among other data structures. For the moment, we take fact 7.2 as given.

If we use Fibonacci heaps, then the running time becomes

$$\begin{aligned} &O(1) \times (\#\ \texttt{decrease-key}\text{'s}) + O(\log n) \times (\#\ \texttt{remove-min}\text{'s}) \\ &= O(m + n \log n). \end{aligned}$$

In conclusion, we have the following remarkable running time for computing shortest paths with positive edge weights.

Theorem 7.3. *Let $G = (V, E)$ be a directed graph with positive edge weights $\ell : E \to \mathbb{R}_{>0}$ and let $s \in V$. Then Dijkstra's algorithm (with Fibonacci heaps) computes the shortest path from s to every other vertex $t \in V$ in $O(m + n \log n)$ time.*

Lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

7.5 Exercises

Exercise 7.1. Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with m edges and n vertices. Each problem

can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply, and (c) the resulting running time. No proof of correctness is required.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]

2. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is a multiple of 5.

3. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple of 2, or a multiple of 3.

4. Suppose G os a patriotic directed graph where all the edges are colored either red, white, or blue. An *American walk* is a walk where the edges alternate in color in the order of the American dream: red, white, blue, red, white, blue... Here the first edge could be any color and then the colors have to cycle through the American dream thereafter. Compute the length of the shortest American walk from s to t.

5. Suppose G is again a patriotic directed graph where all the edges are colored either red, white, or blue. An *un-American walk* is a walk that never cycles through the American dream; that is, a walk where no three consecutive edges in the walk have colors that form any of the following three sequences: $(\text{red}, \text{white}, \text{blue})$, $(\text{white}, \text{blue}, \text{red})$, or $(\text{blue}, \text{red}, \text{white})$. Compute the length of the shortest un-American walk from s to t.

6. Let $s, t \in V$ be two vertices. Consider the following game with two people Alice and Bob. Initially, Alice is on s, and Bob is on t. Each round, Alice and Bob are on two vertices, and they simultaneously take an outgoing edge to another vertex. The goal is to guide Alice and Bob to the same vertex with the minimum number of rounds or declare that it is impossible to have them meet. Compute the minimum number of (parallel) steps needed to have Alice and Bob meet at the same vertex.

Exercise 7.2. The racetrack problem in Chapter 5 of [Eri19].

[3] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

Exercise 7.3. Let $G = (V, E)$ be a directed graph with positive edge weights, and $s, t \in V$. Recall that Dijkstra's algorithm returns the lengths of the shortest paths from s (as well as the shortest paths themselves by incorporating parent pointers). However there may be many shortest (s, t)-paths for a particular t. Suppose we want to find the shortest (weighted) (s, t)-paths with the fewest number of edges (to break ties). Consider the problem of computing both the shortest path lengths as well as the minimum number of edges in each shortest path. Design and analyze an algorithm for this problem.

Exercise 7.4. Let $G = (V, E)$ be a directed and unweighted graph, and $s, t \in V$. Consider the problem of computing the *number* of shortest (s, t)-paths.[4] Design and analyze an algorithm for this problem.

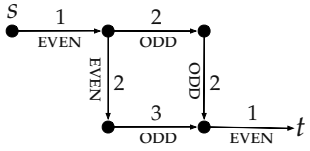
Exercise 7.5. Red light, green light! The following model is inspired by streetlights which, at a given point in time, allow traffic to enter some streets and block traffic from entering others.

Let $G = (V, E)$ be a directed graph with positive, integer edge lengths $\ell(e)$ for $e \in E$, which represents the amount of time it takes to traverse that edge. (Here time will always be an integer.) Suppose each edge is also annotated as being either "even" or "odd". The model is as follows.

We start at a vertex s at time $x = 0$, and want to get to a vertex t as soon as possible. In general, taking an edge e takes $\ell(e)$ units of time, which is added to x. However, we can only take an even edge when the time x is even, and we can only take an odd edge when the time x is odd. We can also wait at a vertex for one unit of time during which x increases by 1.

Loosely speaking, one can imagine each vertex as being like a stop light. An even edge is like a one-way street that traffic can enter only when the time x is even. An odd edge is like a one-way street that traffic can enter only when the time x is odd.

In the example on the right, the (s, t)-path along the top takes $1 + 2 + 2 + 1_{(\text{WAIT})} + 1 = 7$ units of time. (Here we annotated the 1's that correspond to waiting). The path along the bottom takes $1 + 1_{(\text{WAIT})} + 2 + 1_{(\text{WAIT})} + 3 + 1 = 9$ units of time.



For $s, t \in V$, consider the problem of computing the minimum amount of time needed to get from s to t in the time model described above. Design and analyze an algorithm for this problem.

[4] Technically, there could be as many as $n!$ paths in which case an arithmetic operation would take $O(\log(n!)) = O(n \log n)$ time. For this exercise, let us assume for simplicity that arithmetic operations take $O(1)$ time anyway.