The shortest walk

(high-level comment)

"only two algorithms"*

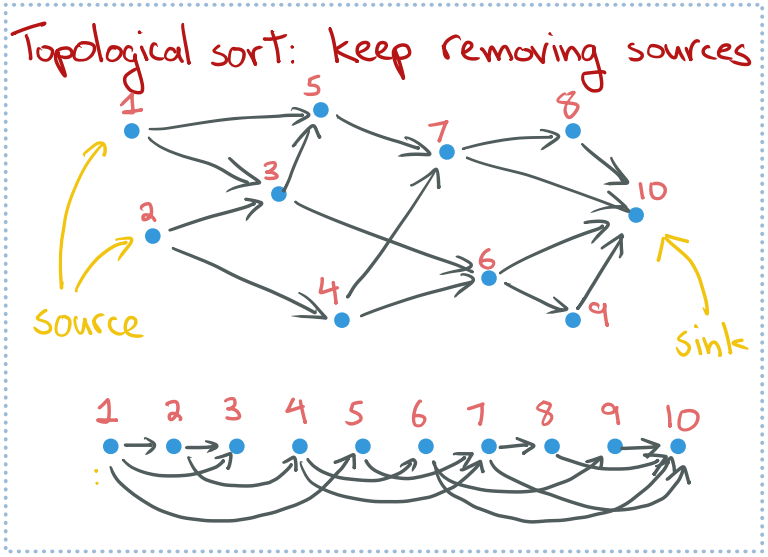
① identify subproblems

② change the input

*(mostly; we also had a little divide+conquer extend. merge-sort)

One week ago:

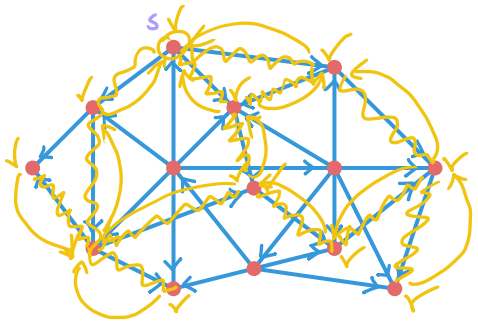
Connectivity in Directed Graphs



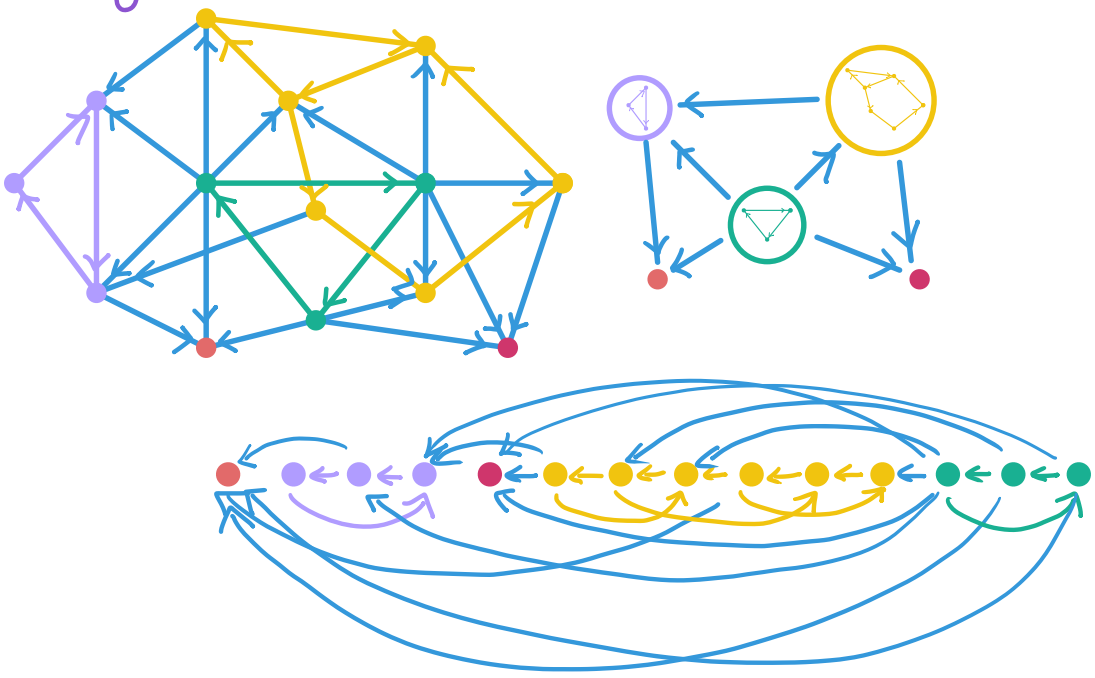
"depth-first search"

DFS(v):

① if v is unmarked
  ⓐ mark v
  ⓑ for each edge v→w
    ① DFS(w)



strongly connected component (SCC)



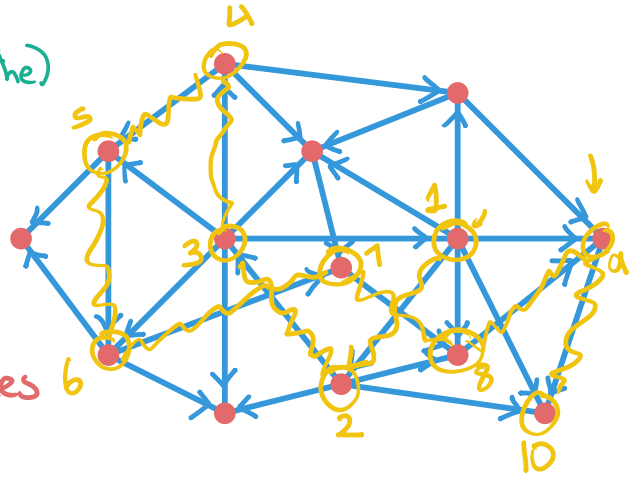
`sink-first-SCC(`$G = (V, E)$`)`

1. Compute a post-DFS ordering of V in the *reversed* graph G_r
2. Until $V = \emptyset$
   A. Let v be the *last* (remaining) vertex in V in the post-DFS order of G_r
   // *v is in a sink component of G.*
   B. $C \leftarrow$ all vertices marked by `dfs(`v`)`
   C. Append C to the output
   D. Remove C and all incident edges from G

All SCC's in $O(mtn)$ time

Last time: longest path

Find the (length of the) longest path in G (eg. # edges)



Hamiltonian path:

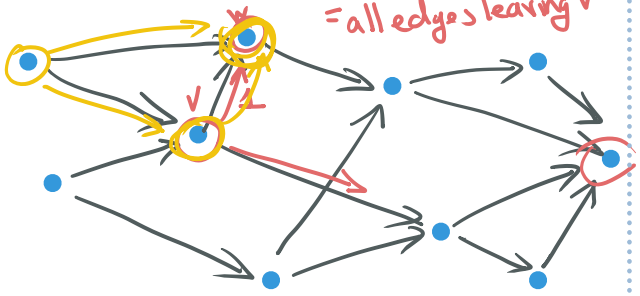
path visiting all vertices

does G have a Hamiltonian path?

Longest ~~path~~ walk in a DAG

no cycles
⇒ walk = path

$\delta^+(v)$ = "out-cut" of v = all edges leaving v

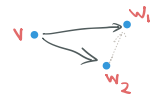


Recursion

$LW(v)$ = length of Longest Walk from v

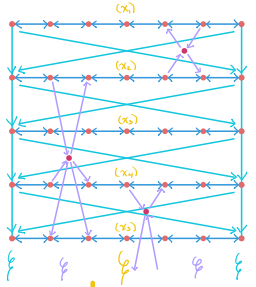
$$LW(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w)\in\delta^+(v)} LW(w) & \text{otherwise} \end{cases}$$

+ caching



Theorem 7.1. *A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.*

$\varphi(x_1, \sim, x_n) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \sim$



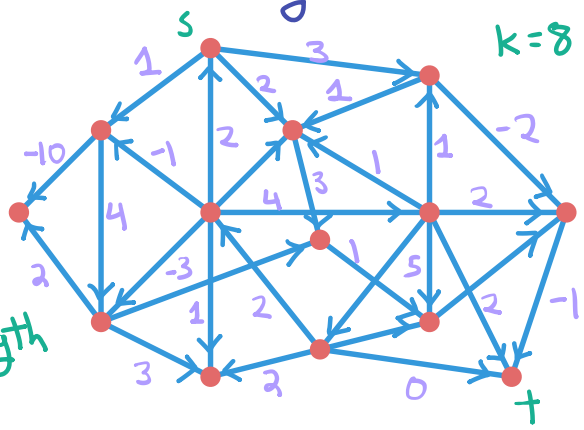
yes, no

Hamiltonian path solver

Shortest (s,t)-walk w/ ≤k edges (next time: remove this constraint) (time permitting)

edges have lengths

Goal:
go from s to t
w/ ≤k edges
and min total length

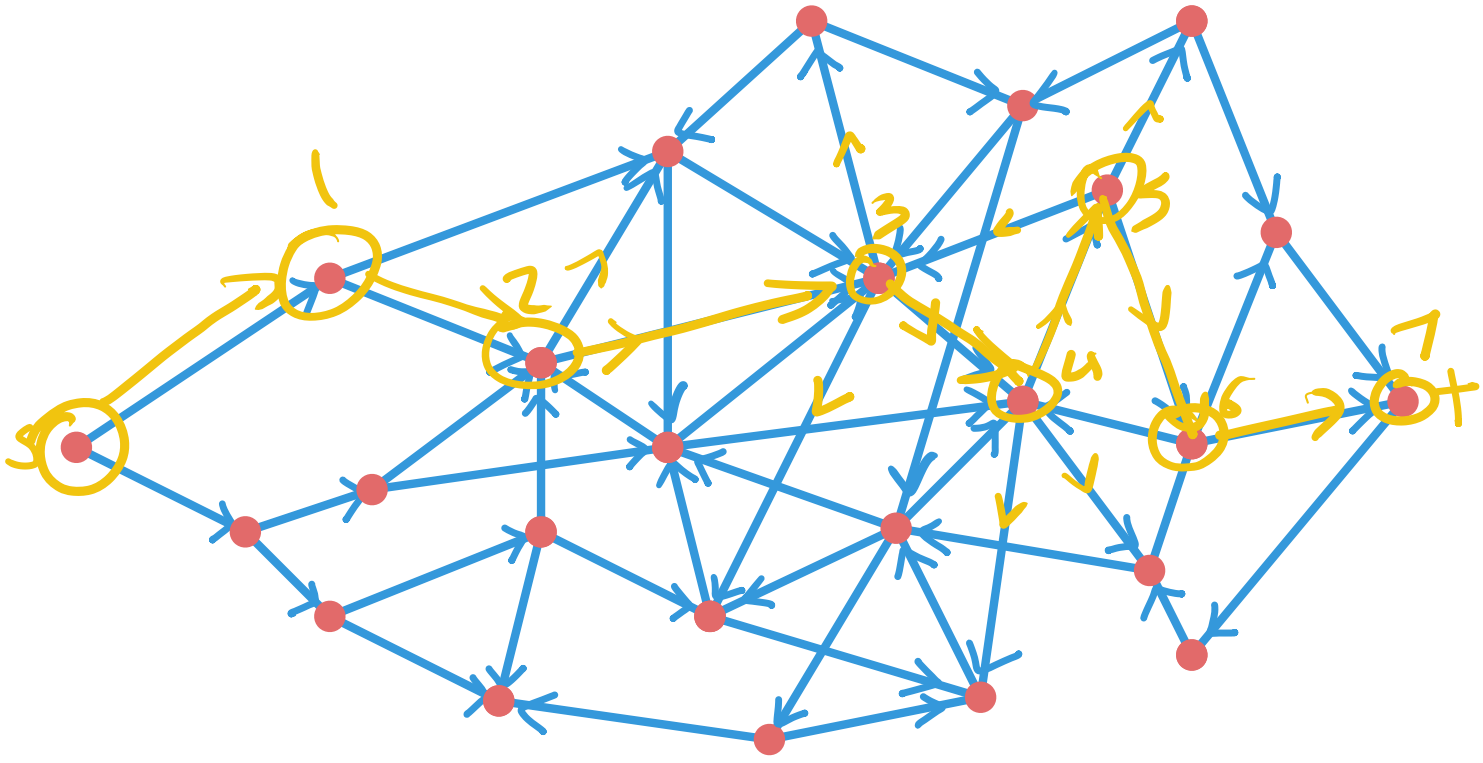


Recursion:

$SW(v,i)$ = length of shortest walk from v to t w/ ≤ i edges

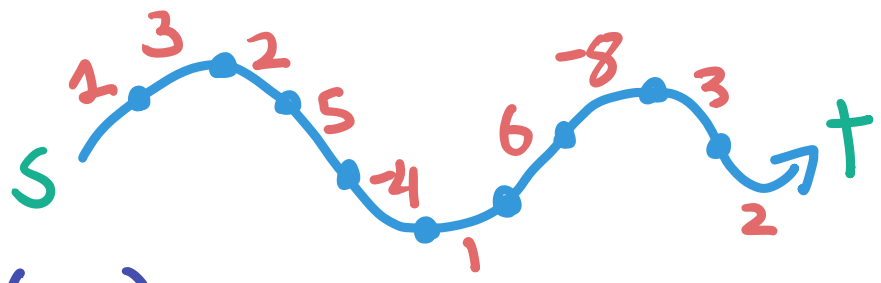
today: shortest walks

simplest case: unweighted graphs



shortest way from s to t?

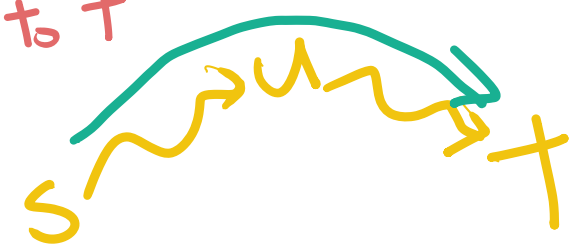
Directed $G=(V,E)$, edge lengths $\ell: E \to \mathbb{R}$



Distance $d(s,t)$

$d(s,t)$ = min length of any (s,t) walk

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



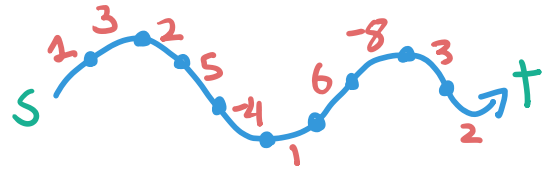
Triangle inequality

$$d(s,t) \leq d(s,u) + d(u,t)$$

local

goal:

compute $d(s,t)$ (for fixed s,t)



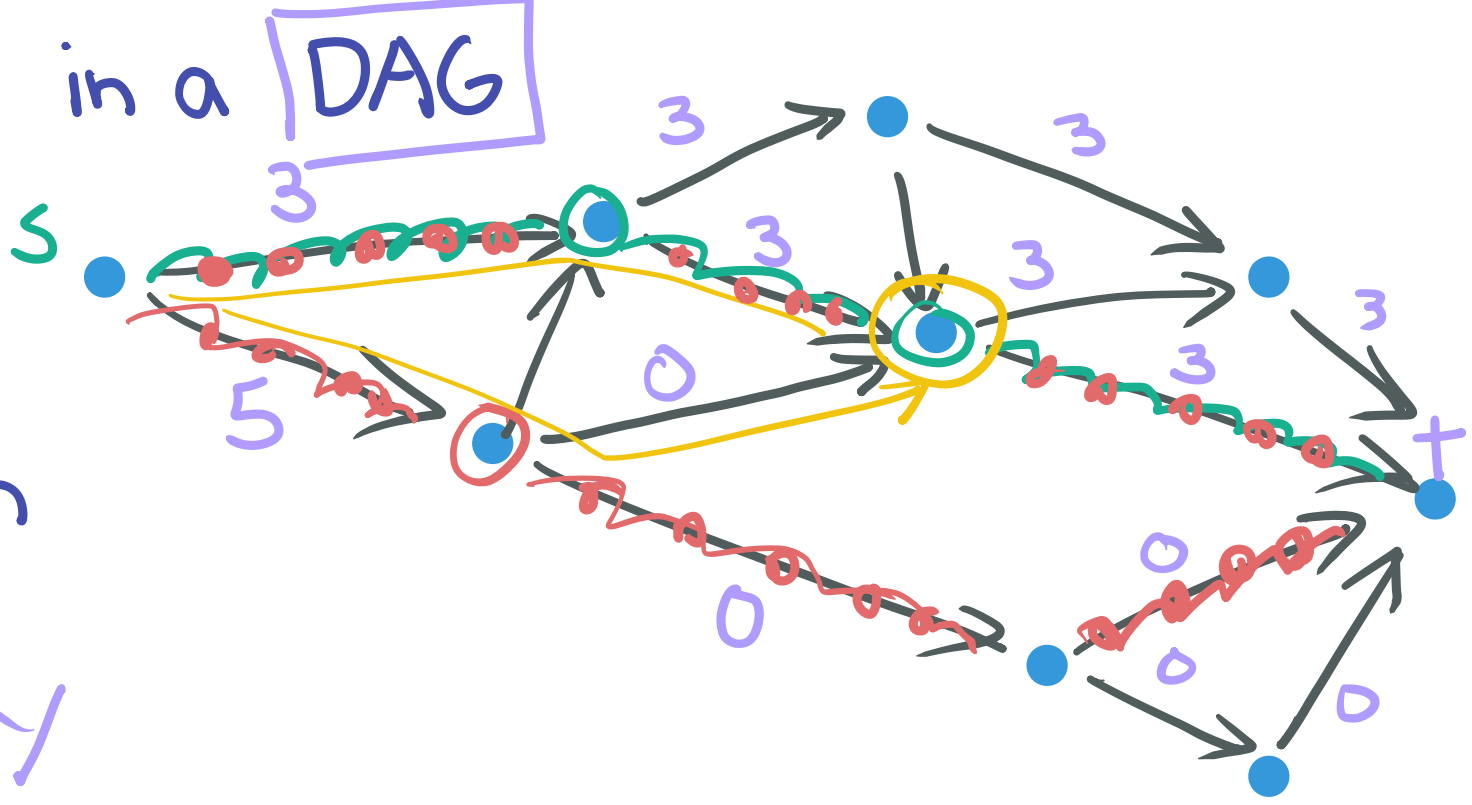
Distance $d(s,t)$
= min length of any (s,t) walk

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

settings:

1. DAG's ←
2. unweighted ←
3. positively weighted ←
4. real weighted (time permitting) ↙

Shortest walk in a DAG

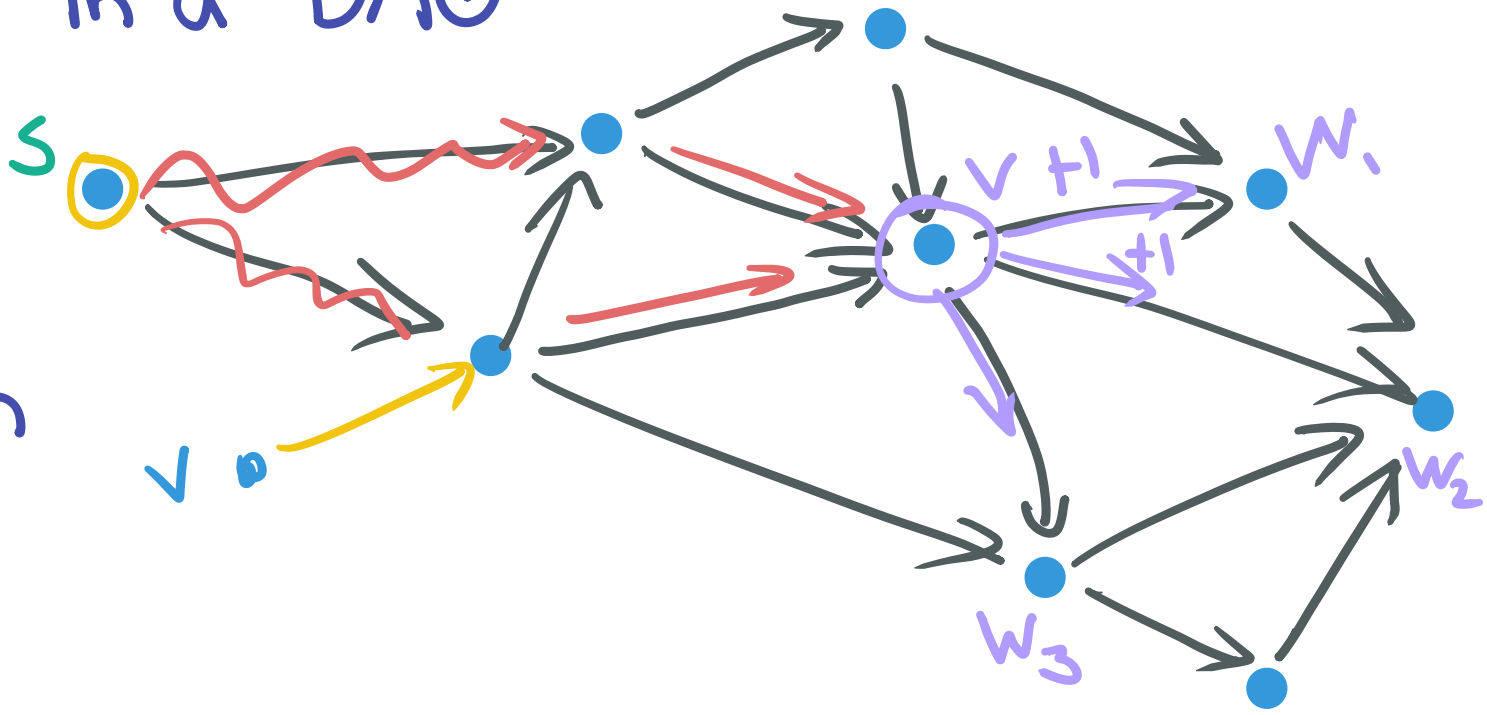


no cycles

⇒ walk = path

Recursively

Shortest walk in a DAG



no cycles

$\Rightarrow$ walk = path

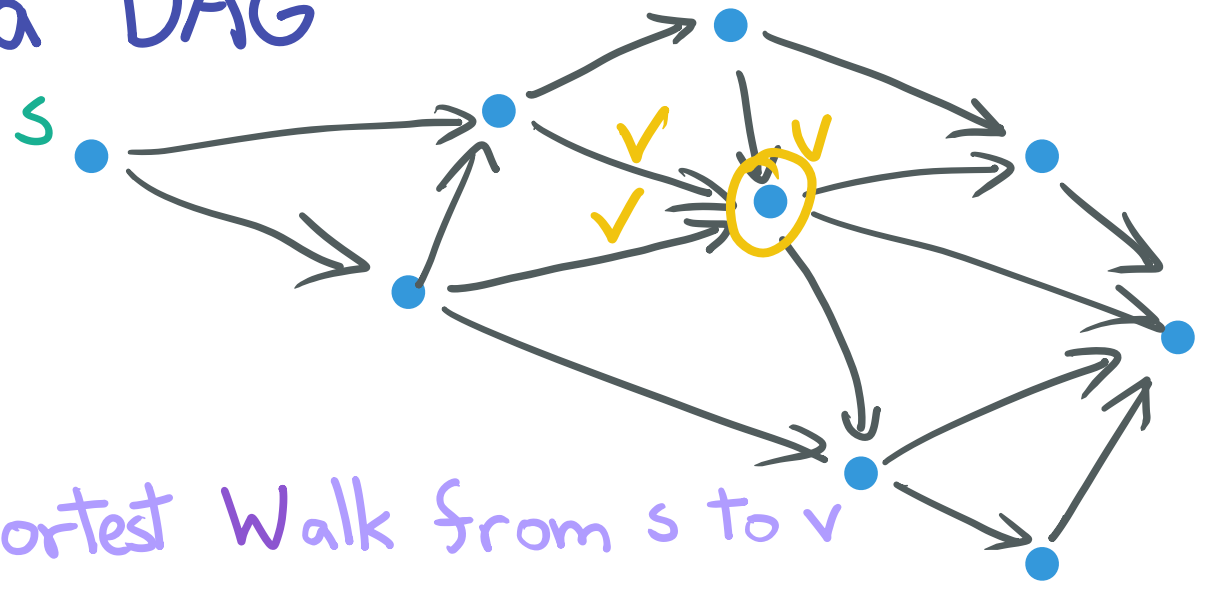
Recursion

$SW(v)$ = length of Shortest Walk from s to v

$$SW(v) = \begin{cases} 0 & \text{if } v = s \\ +\infty & \text{if } v \text{ is a source} \\ \min_{(u,v) \in \partial^-(v)} len(u,v) + SW(u) & \text{otherwise} \end{cases}$$

before v?

Shortest walk in a DAG



Recursion

$SW(v)$ = length of Shortest Walk from s to v

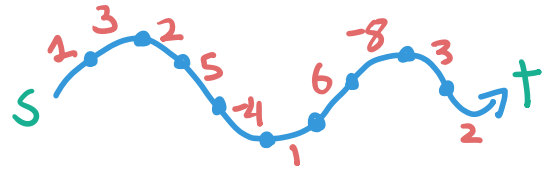
$$SW(v) = \begin{cases} 0 & \text{if } v = s \\ +\infty & \text{if } v \text{ is a source} \\ \min_{(u,v) \in \vec{\partial}^-(v)} SW(u) + \ell(u,v) & \text{otherwise} \end{cases}$$

Running time w/ caching

$$\sum_{v \in V} (\text{time for } SW(v)) = \sum_{v \in V} 1 + \deg^-(v) = O(n+m)$$

goal:

compute $d(s,t)$ (for fixed s,t)



Distance $d(s,t)$
= min length of any (s,t) walk

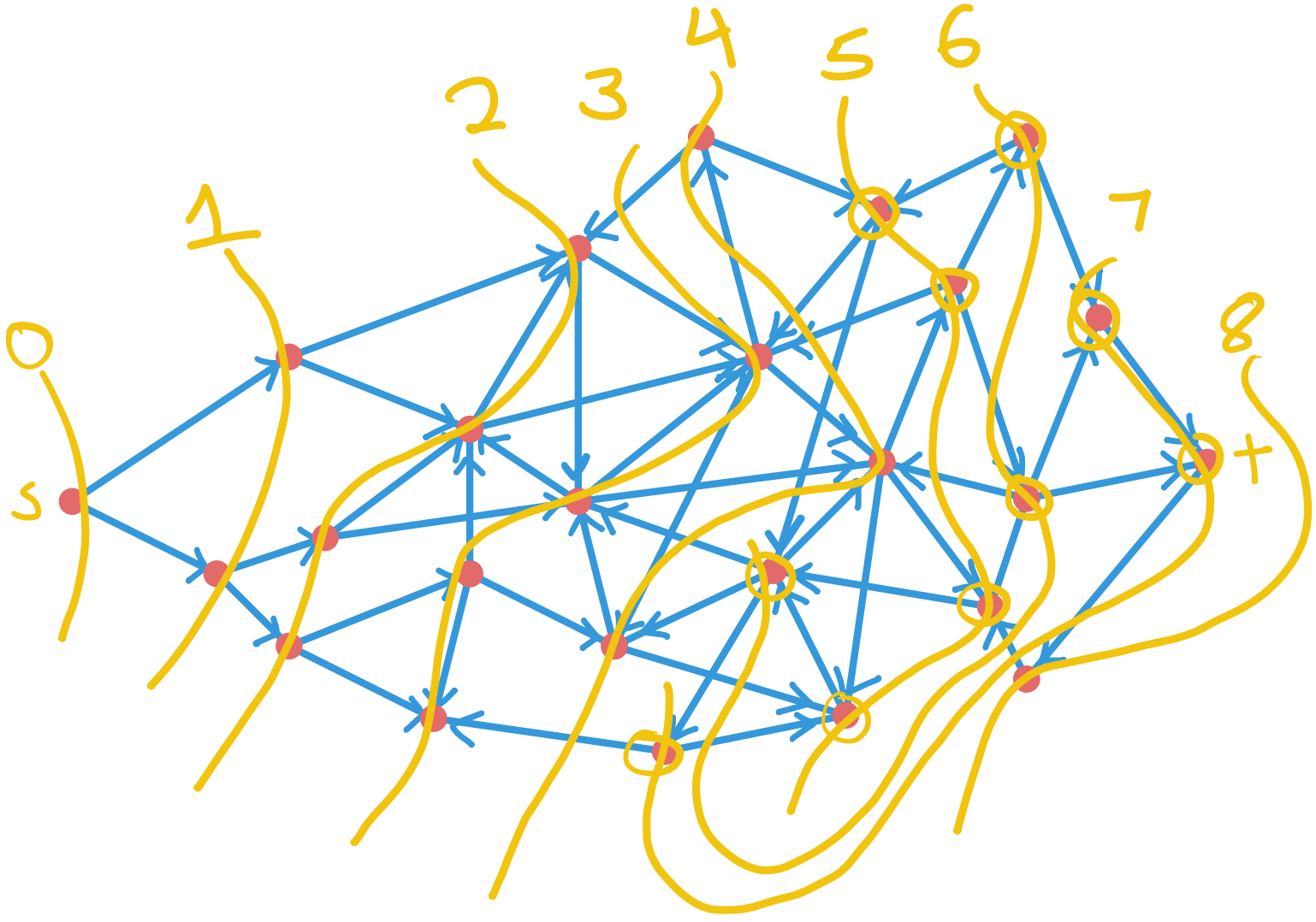
- $d(s,t) = +\infty$ if s cannot reach t
- $d(s,t) = -\infty$ if there are arbitrarily negative length walks from s to t

settings:

1. ~~DAG's~~
2. Unweighted
3. positively weighted
4. real weighted (time permitting)

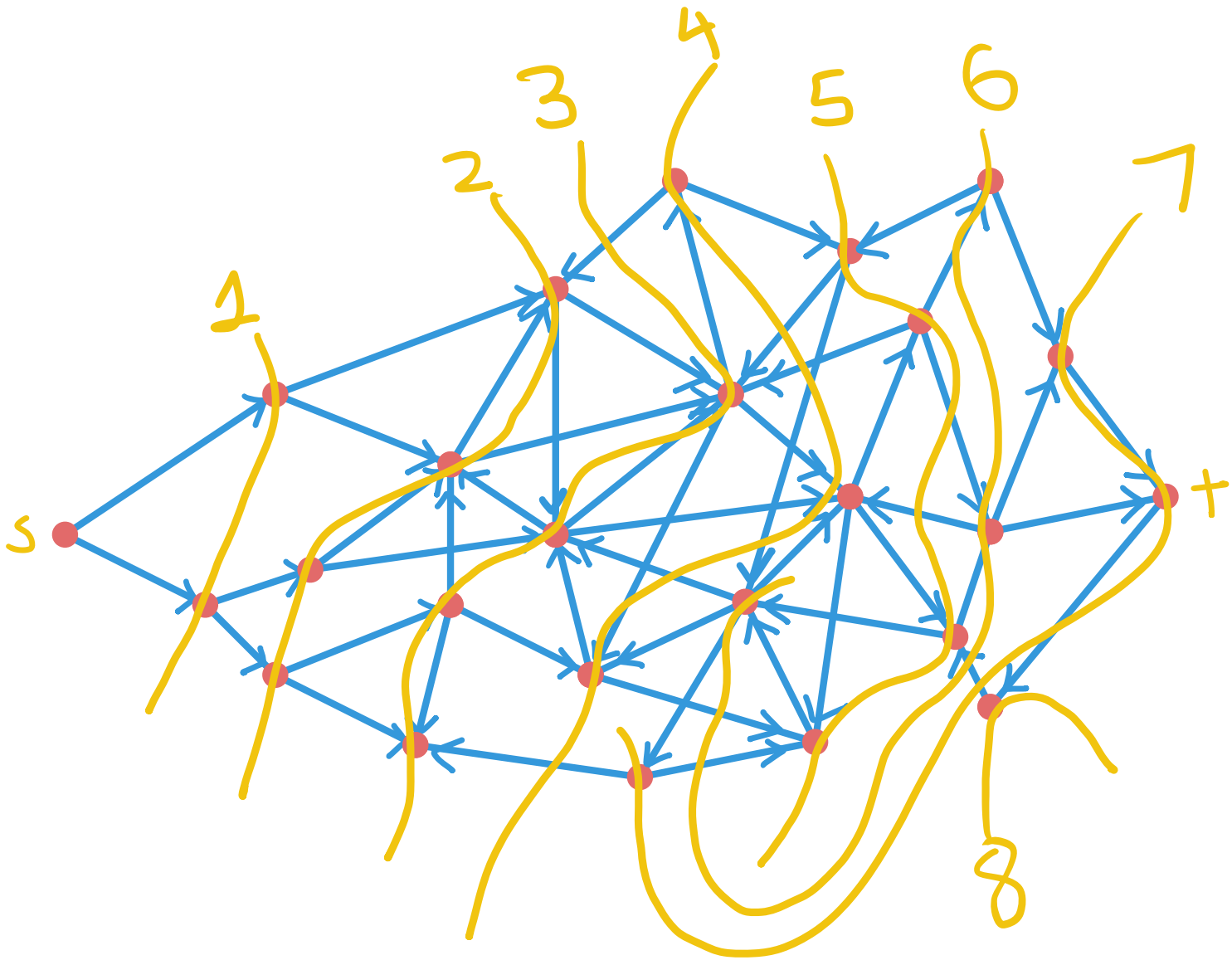
unweighted graphs

shortest way from s to t?



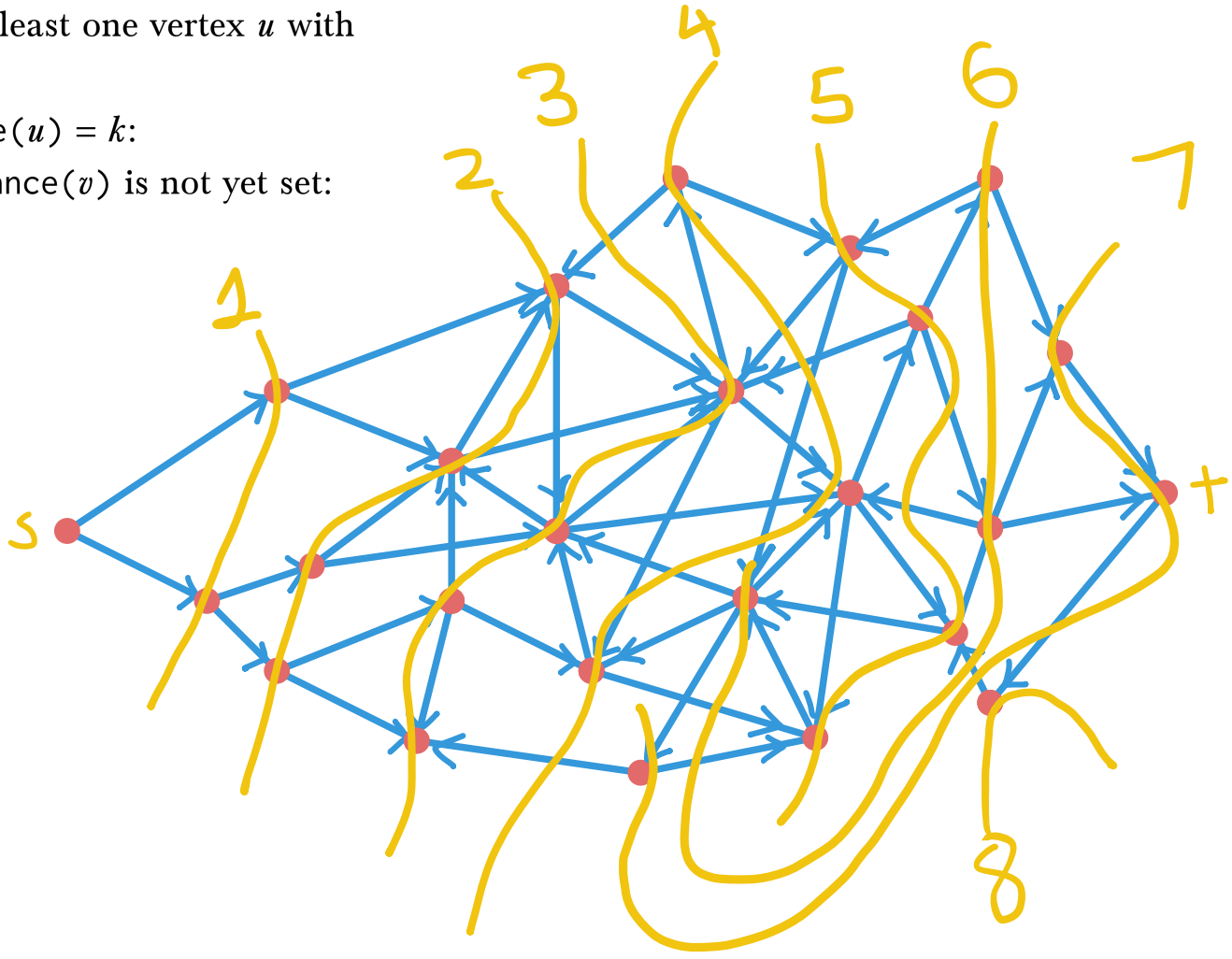
unweighted graphs

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 $d(v) = k + 1$.

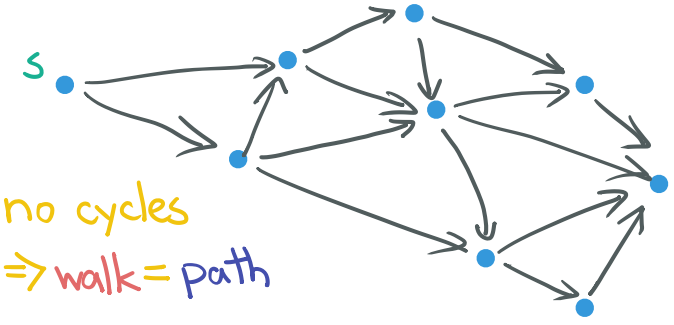


O(mtn) time

(w/ simple bookkeeping)

DAG's vs Graphs

Shortest walk in a DAG



no cycles
⇒ walk = path

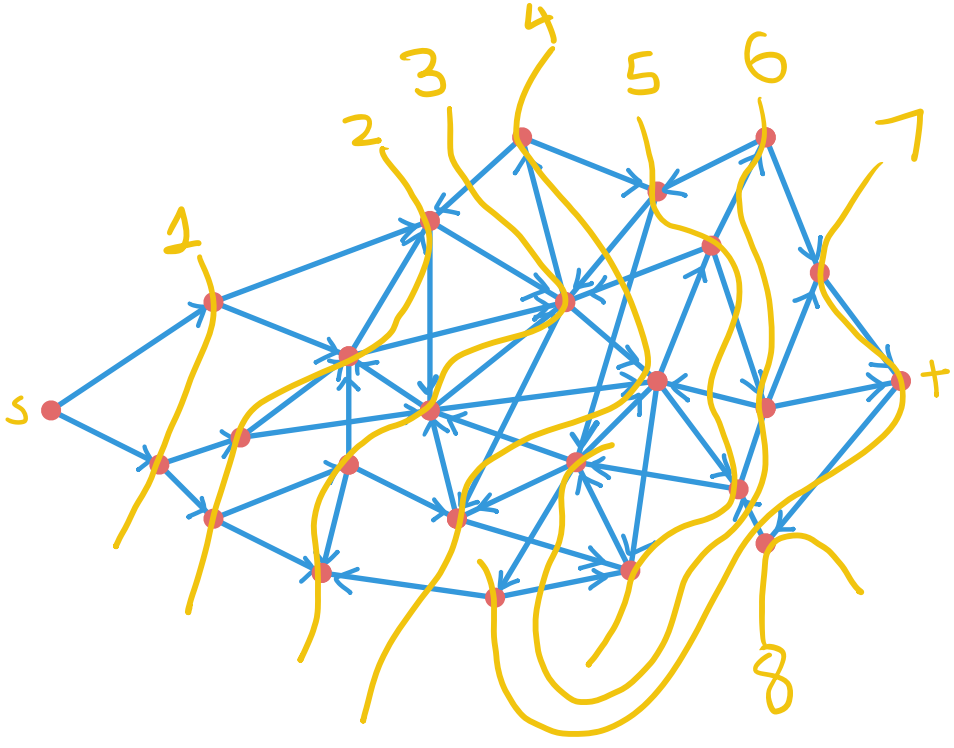
Recursion

$SW(v)$ = length of Shortest Walk from s to v

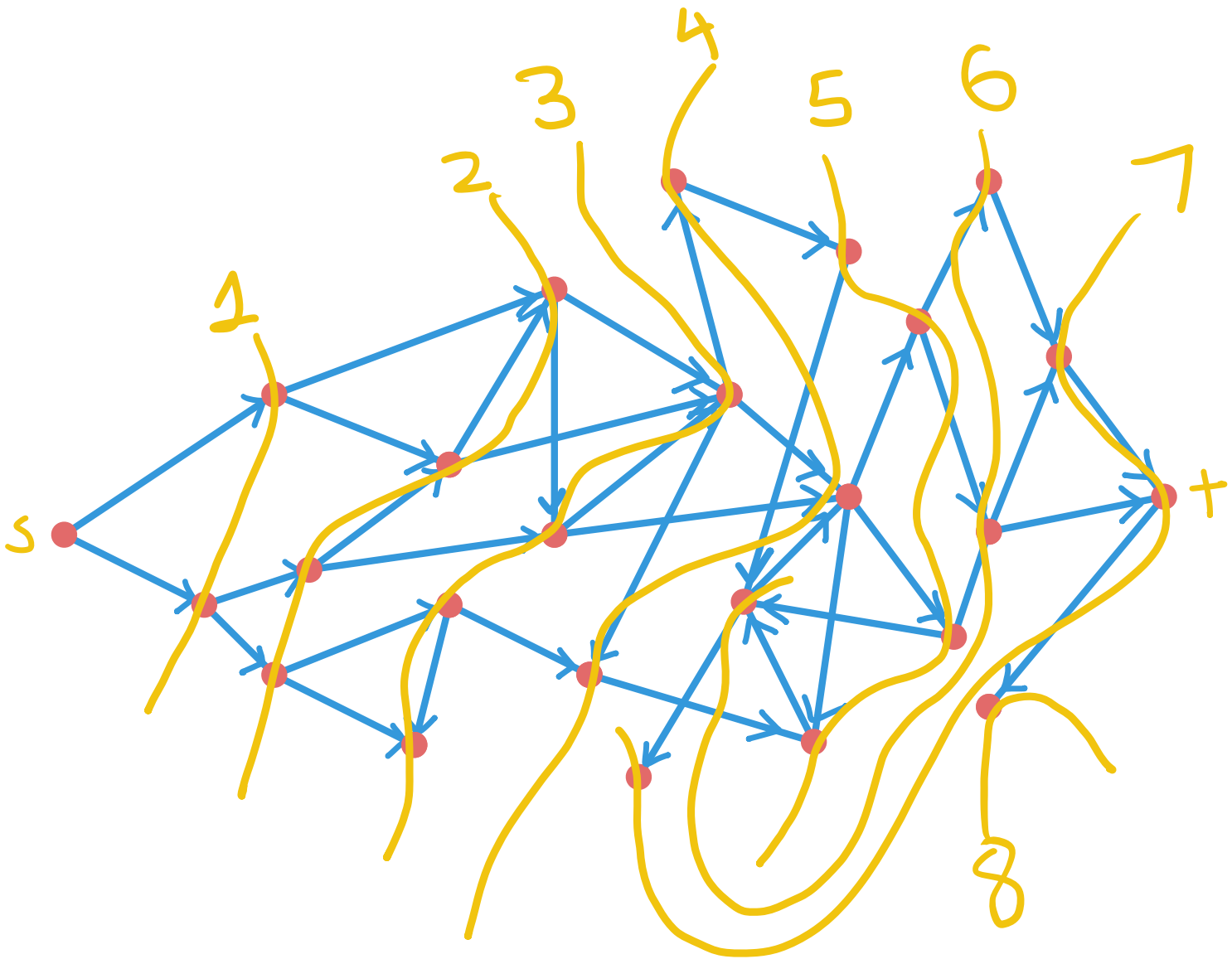
$$SW(v) = \begin{cases} 0 & \text{if } v = s \\ +\infty & \text{if } v \text{ is a source} \\ \min_{(u,v) \in \vec{J}(w)} LW(u) + \ell(u,v) & \text{otherwise} \end{cases}$$

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

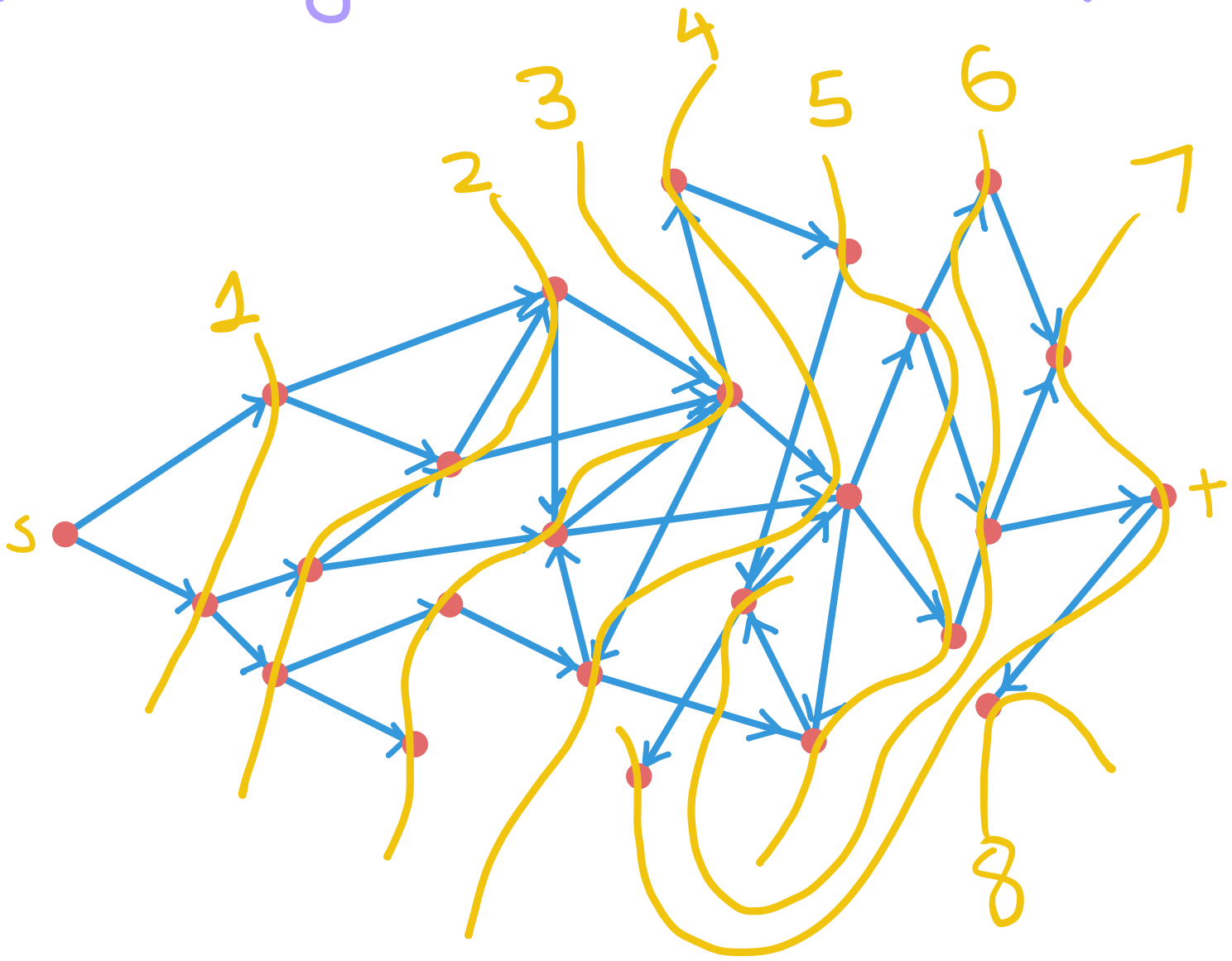
1. Set `distance`$(s) = 0$.
2. For $k = 0, 1, 2, \ldots$, as long as there is at least one vertex u with `distance`$(u) = k$:
   A. For all vertices u such that `distance`$(u) = k$:
      1. For all arcs (u,v) for which `distance`(v) is not yet set:
         a. set $d(v) = k + 1$.



drop all edges not in shortest paths



drop all edges not in shortest paths

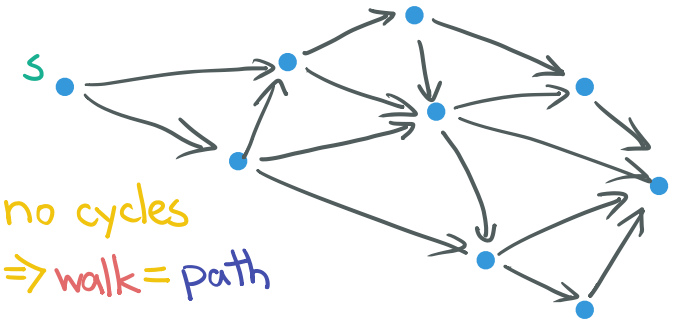


what's left? a dag!

(distances give topological order)

DAG's vs Graphs

Shortest walk in a DAG



no cycles
⇒ walk = path

Recursion

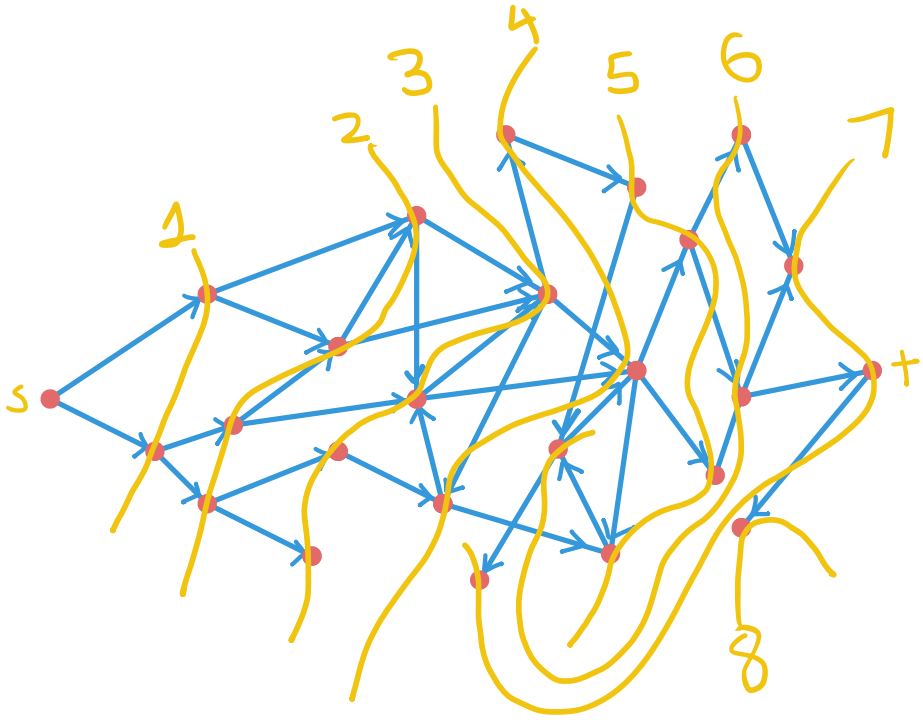
$SW(v)$ = length of Shortest Walk from s to v

$$SW(v) = \begin{cases} 0 & \text{if } v = s \\ +\infty & \text{if } v \text{ is a source} \\ \min_{(u,v) \in \vec{J}(w)} LW(u) + \ell(u,v) & \text{otherwise} \end{cases}$$

---

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 $d(v) = k + 1$.



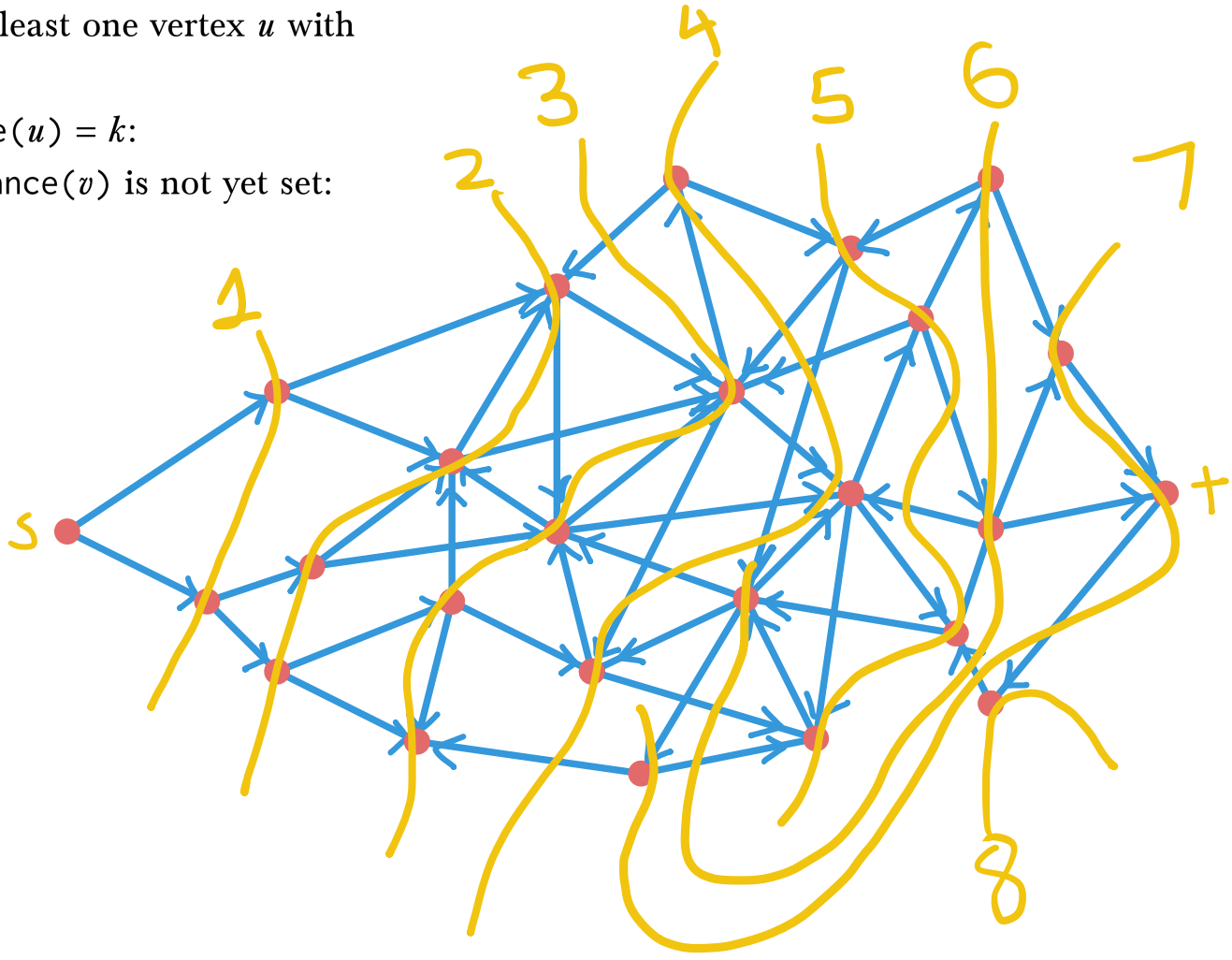
implicitly DP on DAG of "tight edges"

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)

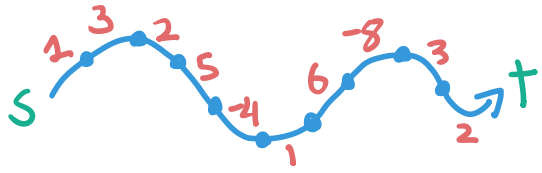


↑

alternative implementation based on queues

goal:

compute $d(s,t)$ (for fixed s,t)

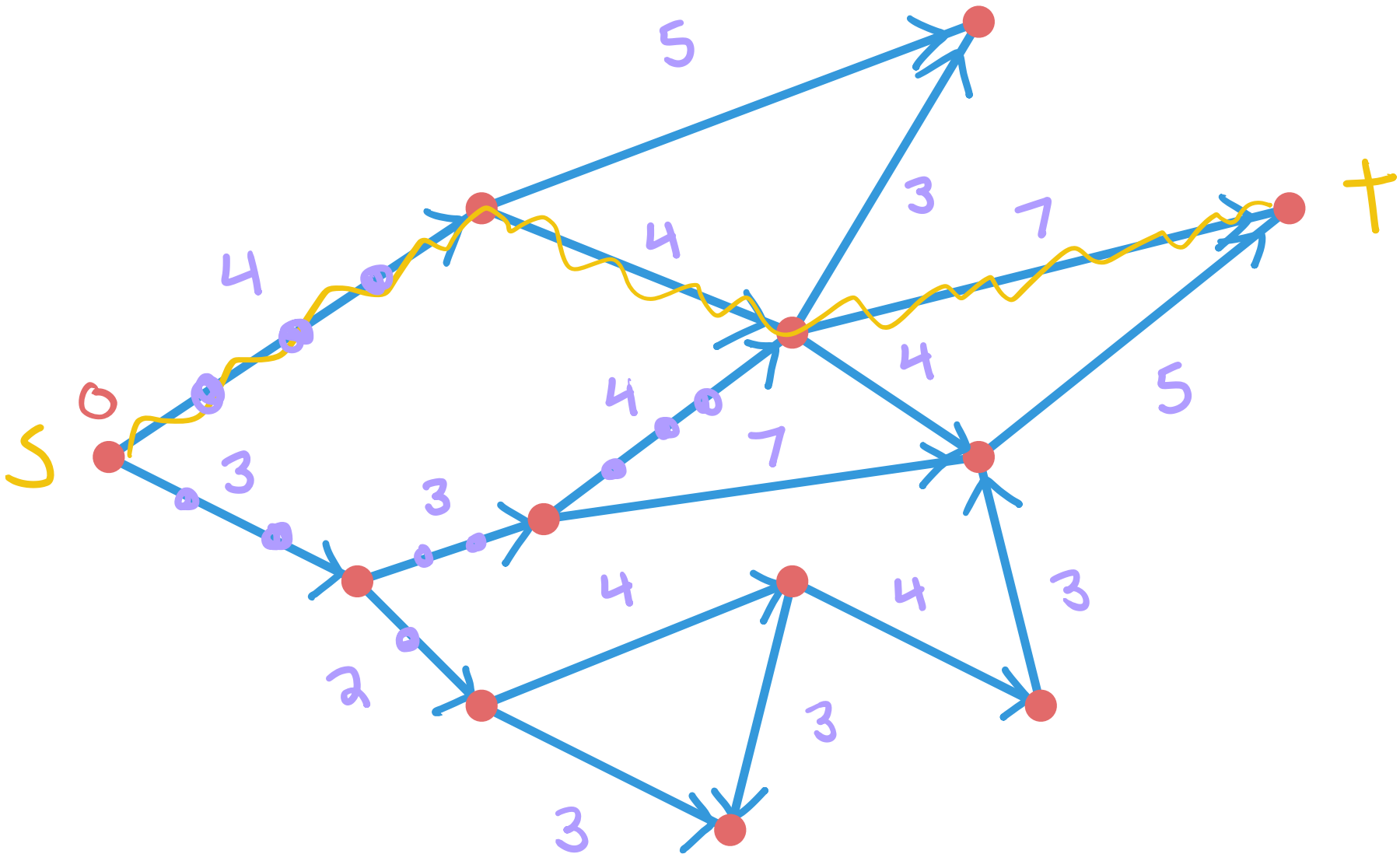


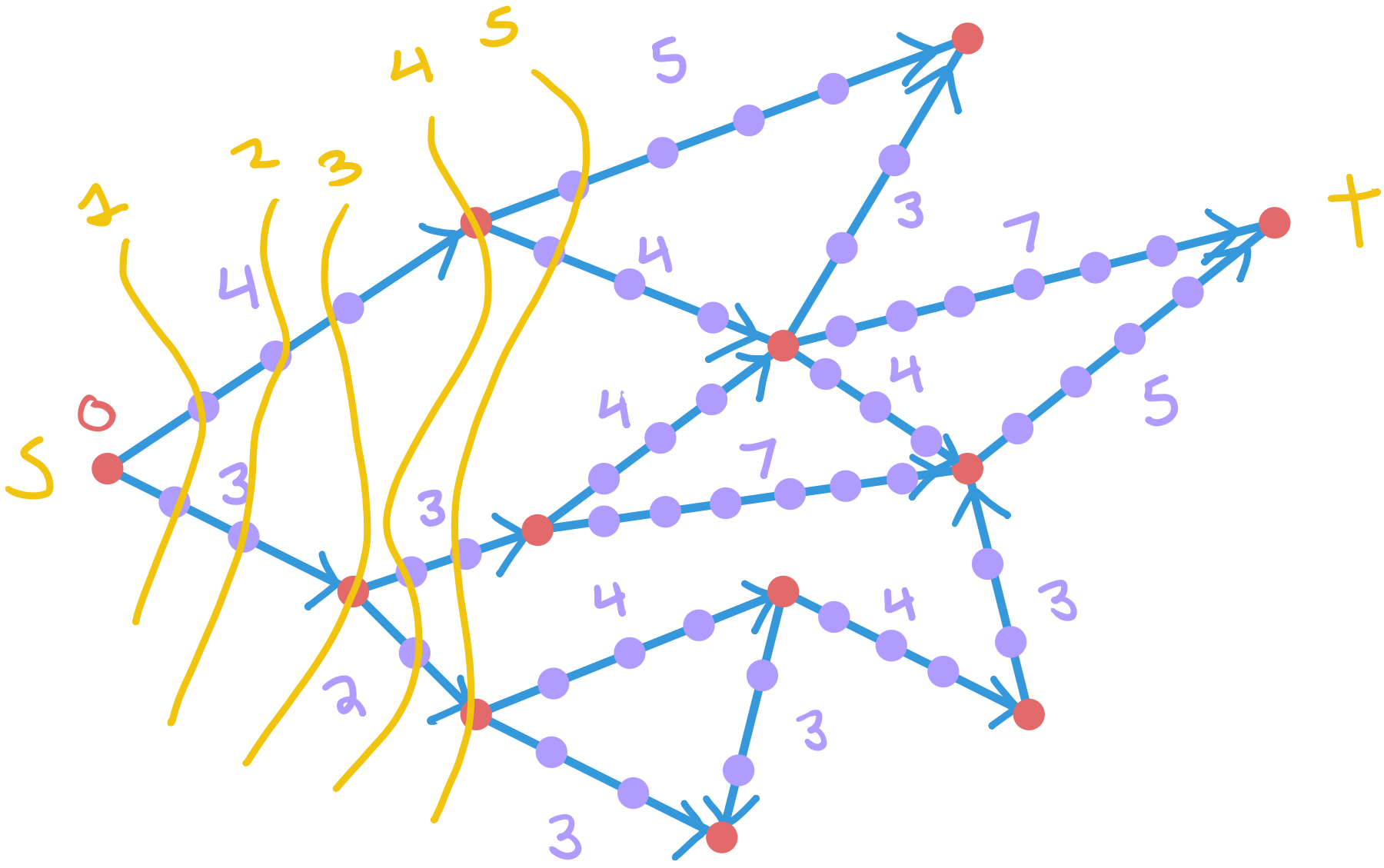
Distance $d(s,t)$
= min length of any (s,t) walk

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

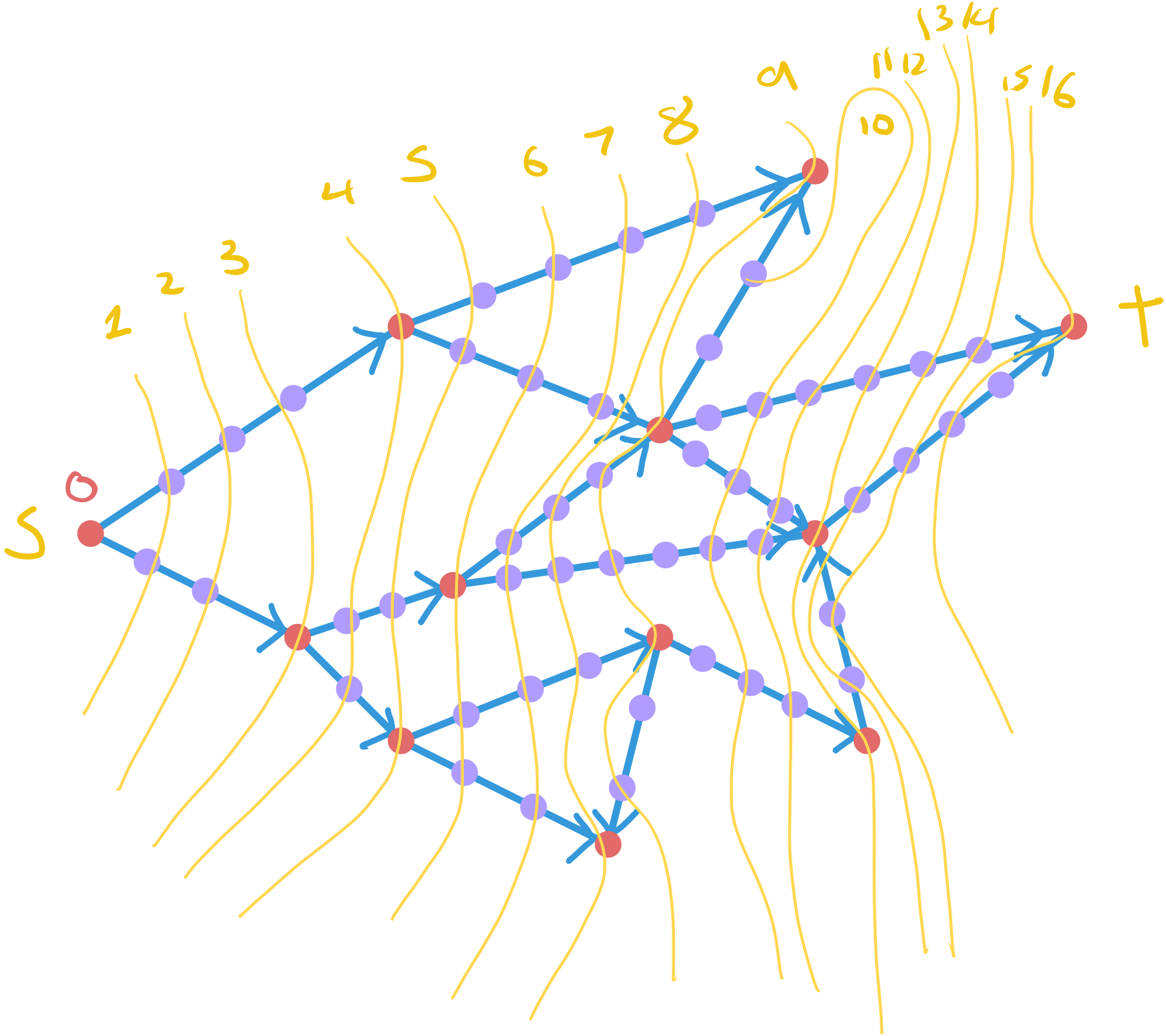
settings:

1. DAG's
2. unweighted
3. positively weighted
4. real weighted (time permitting)


s
0
5
4
4
3
4
7
t
4
4
5
3
7
3
4
4
3
2
3
3

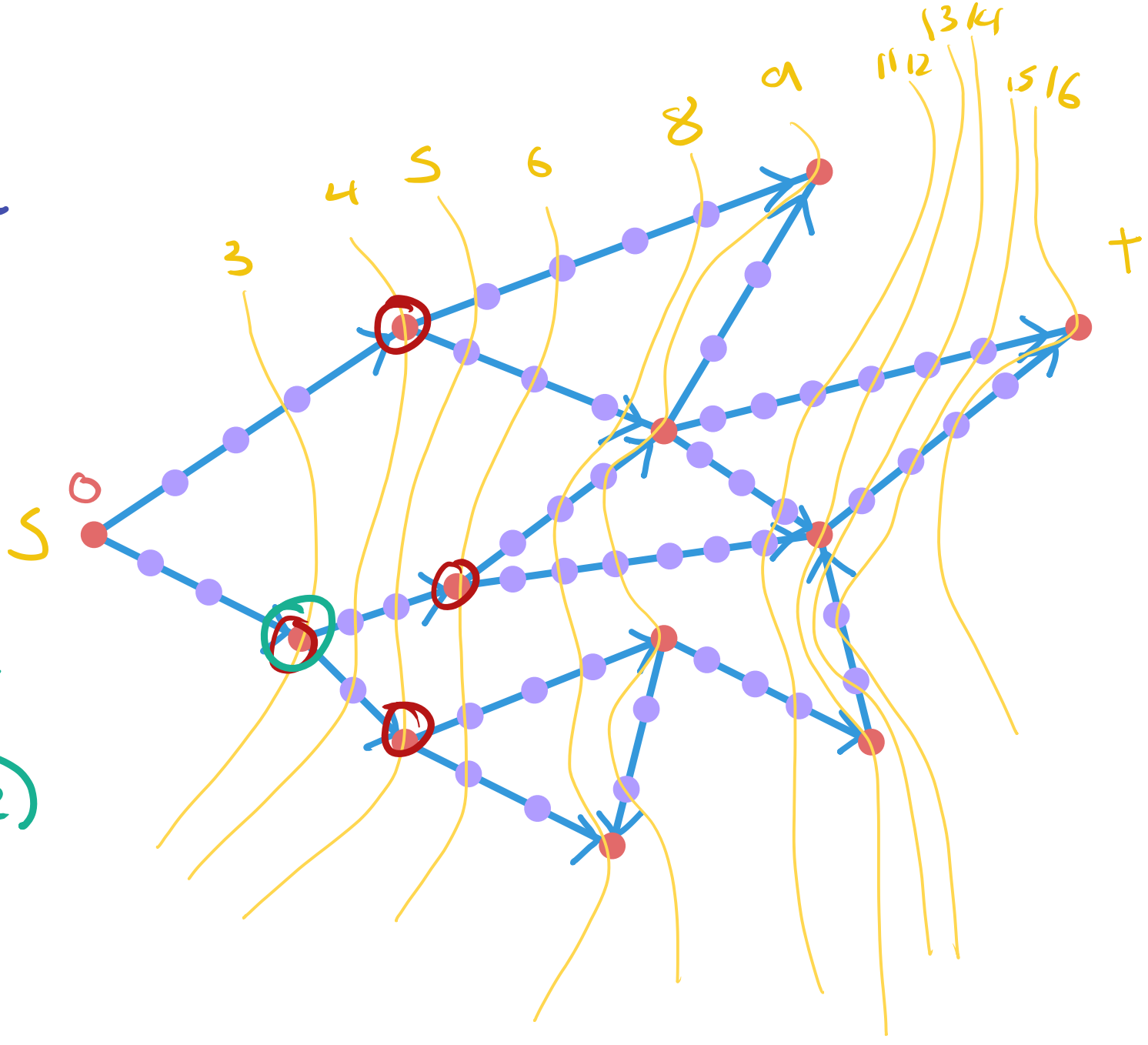


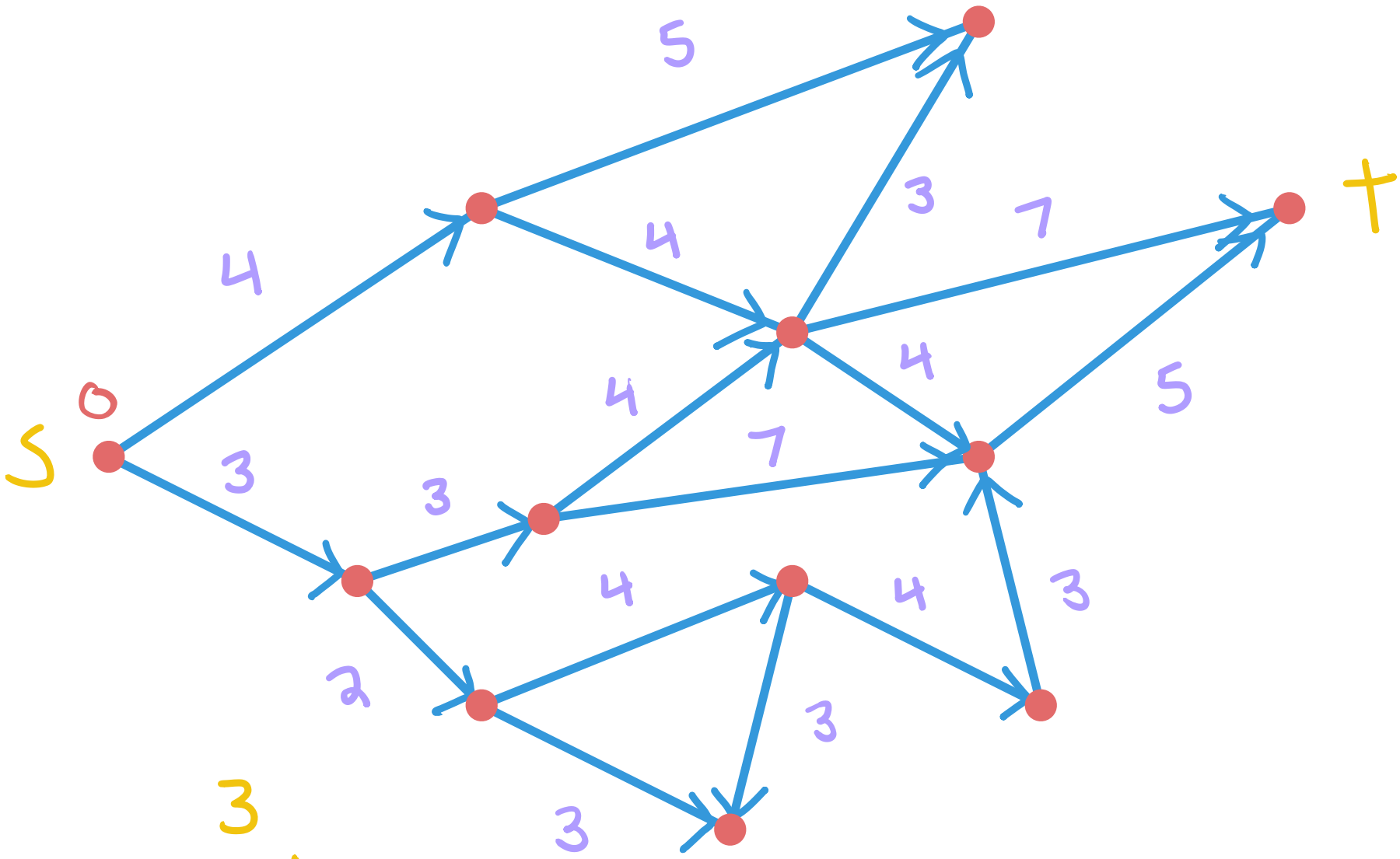
discretize


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)




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

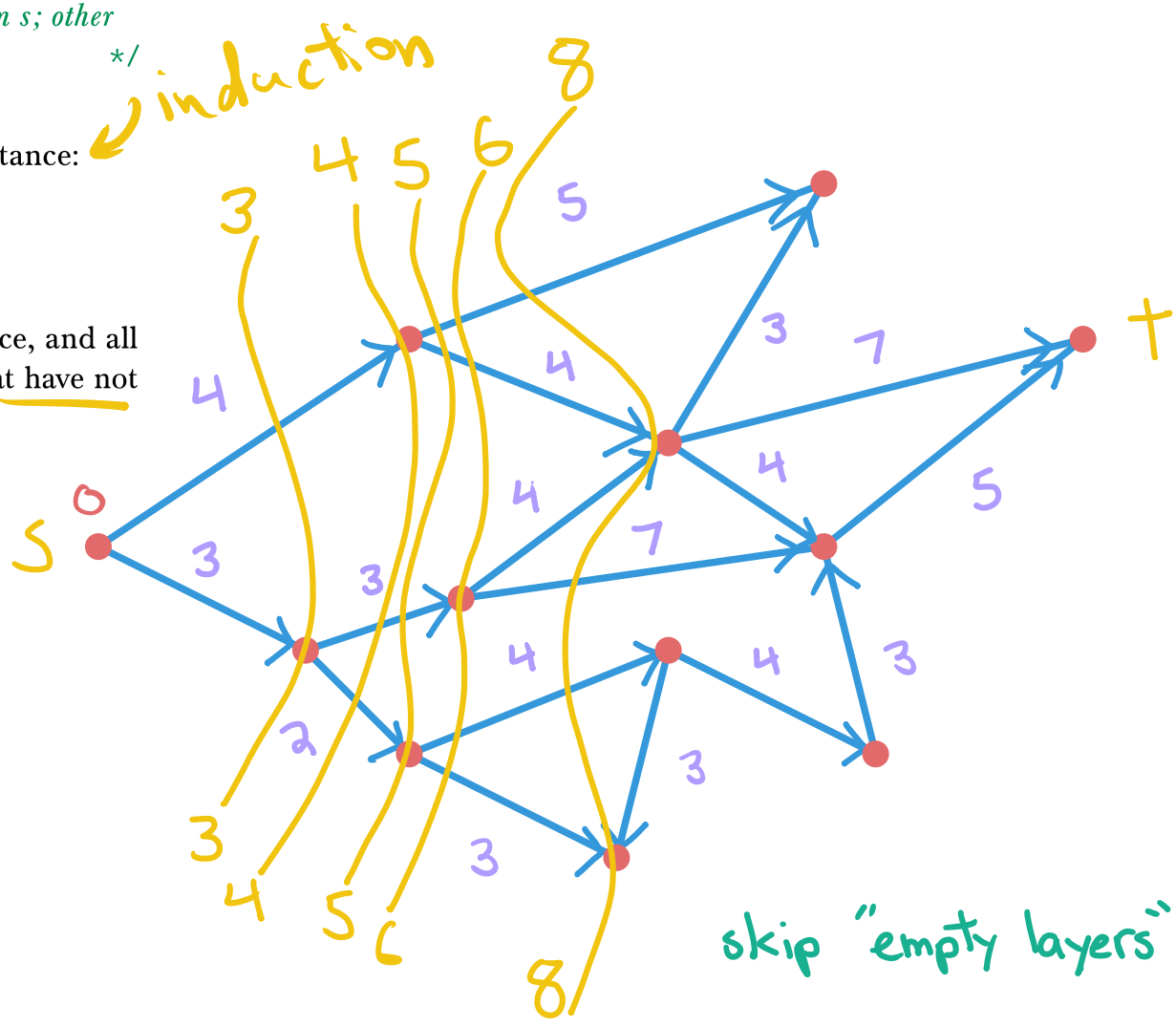
`Dijkstra`$(G = (V,E),\ \ell : E \to \mathbb{R}_{>0},\ s \in V)$

/ 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

$$\texttt{distance}(u) + \ell(u,v)$$

   over all vertices u that have been assigned a distance, and all edges (u,v) leaving u, 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($G = (V, E)$, $\ell : E \to \mathbb{R}_{>0}$, $s \in V$)

/* *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

$$\text{distance}(u) + \ell(u, v)$$

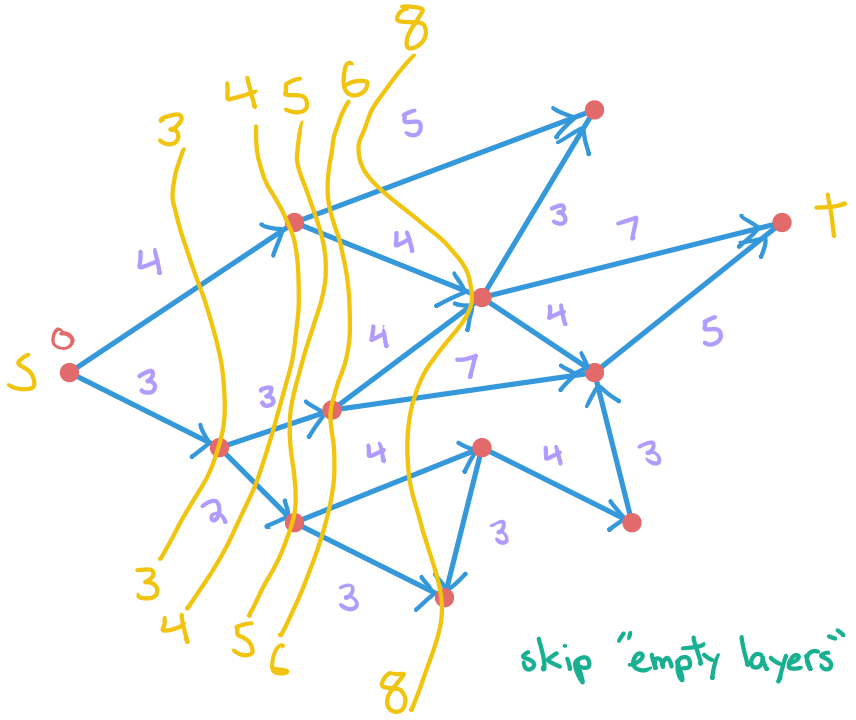
   over all vertices u that have been assigned a distance, and all edges (u, v) leaving u, restricting to endpoints v that have not yet been assigned a distance.
   B. Set distance(v) = distance(u) + $\ell(u, v)$.
3. Return all the distance labels.

n iterations

× m time

O(mn) time

better?



"tentative distance"

for unlabeled v:

"tentative(v)" =

min distance(u) + ℓ(u,v)

where u labeled, (u,v) ∈ δ⁻(v)

(best distance found so far)

each iter. updates only some (not all) tentative(v)'s

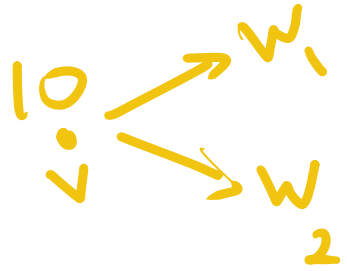
Dijkstra($G = (V,E)$, $\ell : E \to \mathbb{R}_{>0}$, $s \in V$)

/* *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, and all edges (u,v) leaving u, 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.



Priority queues / heap

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.

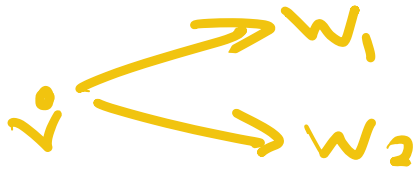
`Dijkstra`$(G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V)$

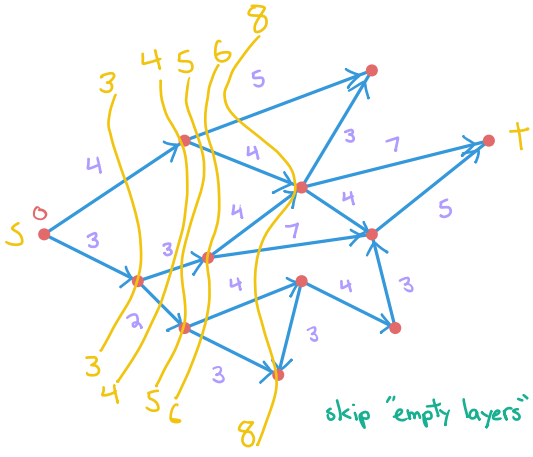
/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

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

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

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





Running time w/ 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

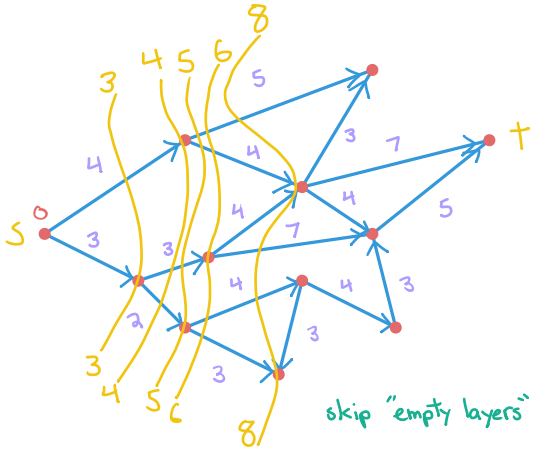
$$\text{tentative}(w) = \min\{\text{tentative}(w), \text{distance}(v) + \ell(v, w)\}$$

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

n iterations
O(n)
O(log n)
deg+(v)
O(log n)

decr. priority

$n \log n + m \log n$



Priority queues / heap

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.

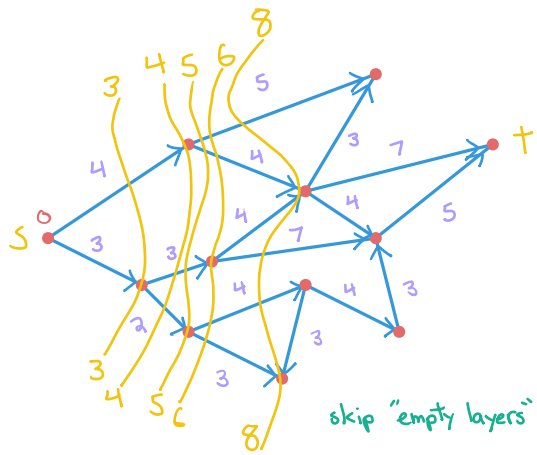
decrease takes O(1) time "on average"

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.

↑ Tarjan (Feldman)



`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 → $O(\log n)$ →

$\deg^+(v)$ → $O(1)$ (on avg.) →

$\sum_v \deg^+(v) = m$

$O(n \log n + m)$

mn?

faster than sorting n number

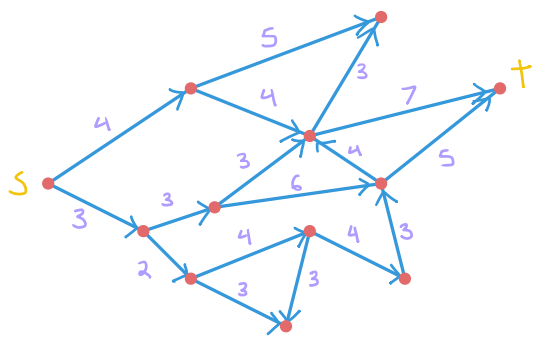
Chapter 8

Shortest walks

We have previously discussed algorithms for reachability: whether or not there *exists* a path between two vertices. For reachability, a beautiful structure emerges from the strongly connected components and the DAG of contracted SCC's, making the problem and many variations of it very tractable. We have also considered the problem of finding the *longest* path between vertices; this problem turns out to be very hard. 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.

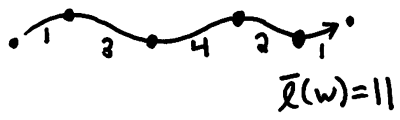
8.1 Distances



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:

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

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. Let us first point out 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.

This problem seems similar to the longest path problem previously discussed. We will approach the shortest path problem the same way. First we will consider DAG's, which are simplest. Then we will consider general graphs.

8.2 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.

[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 $\bar{\ell}(w) \leq L$

We want to compute $\texttt{distance}(t)$. 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 $\texttt{distance}(t) = \texttt{distance}(v) + \ell(v \mid t)$. But which in-neighbor v? Who knows. Let us recursively compute $\texttt{distance}(v)$ over all incoming edges (v,t) and chose the one minimizing $\texttt{distance}(v) + \ell(v \mid t)$. All put together we have the following recursive algorithm. (Note that we avoid infinite loops because G is a DAG.)

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

As usual with DAG's, 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 7.1, that's because they are essentially the same. One can easily 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.

8.3 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) don't have an arc (s,v).
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) don't have any arcs (x,v) from a vertex x at distance 1 or less.

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) do not have any arcs (x,v) from a vertex x at distance $k-2$ or less.

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 $k=0,1,2,\ldots$, 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 $d(v)=k+1$.

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 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.

We remark that the algorithm above is implicitly very similar to our algorithm for DAG's. We are incrementally building out distances in increasing order from s. To make the algorithm clearer, 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 as 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.

8.4 Weighted shortest paths

We now move onto 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 don 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 $d(s,v_2) + \ell\big(v_2 \,\big|\, y\big)$ 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 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 $d(s,v_2) + \ell(s \mid x)$.

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:

1. To find the kth closest vertex v_k from s, we consider all the closer vertices v_i (where $i \in \{1, \dots, k-1\}$) and all edges (v_i, x) where x is not yet labeled, and choose x that minimizes $d(s, v_i) + \ell(v_k)$.

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. (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

$$\texttt{distance}(u) + \ell(u,v)$$

   over all vertices u that have been assigned a distance, and all edges (u,v) leaving u, 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`'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.

So 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`$(u) + \ell(u \mid v)$ 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.
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()`.

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].

`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.

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*

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

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 8.1 as given.

If we use Fibonacci heaps, then the running time becomes

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

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

Theorem 8.2. *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 $O\big(m + n \log n\big)$ time.*

8.5 Exercises

Exercise 8.1. Let $G = (V, E)$ be an unweighted, directed graph, and 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. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 8.2. Here we consider an extension of shortest (s, t)-walks where one has to visit a family of vertices specified by the input. The input consists of a directed graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, as well as a list of vertices $x_1, \ldots, x_k \in V$. The high-level goal in both of the following problems is to compute the shortest walk that visits all k vertices.

1. Suppose you are allowed to visit $x_1, \ldots, x_k$ in any order. Consider the problem of computing the length of the shortest walk visiting all of $x_1, \ldots, x_k$ in any order. (You may assume no vertices repeat in $x_1, \ldots, x_k$.) For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Suppose you have to visit $x_1, \ldots, x_k$ in the listed order. (Here vertices may repeat.) Consider the problem of computing the length of the shortest walk visiting $x_1, \ldots, x_k$ in order. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

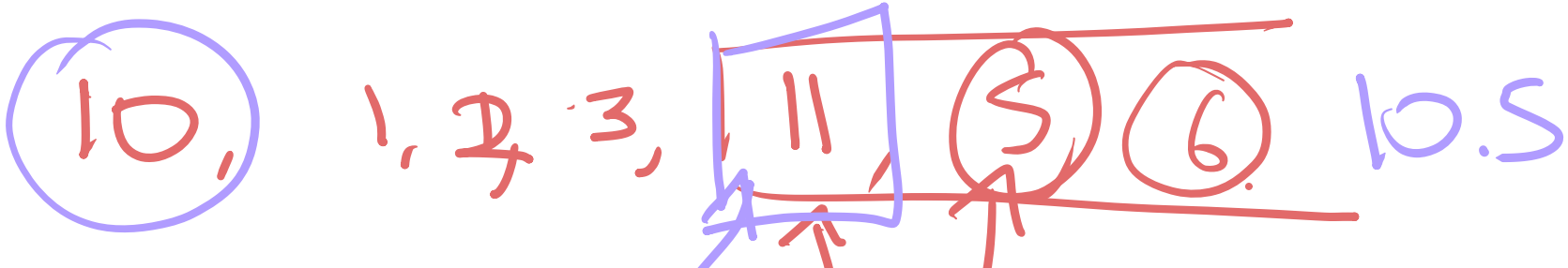
Exercise 8.3. The racetrack problem in Chapter 5 of [Eri19].

LIS(i) = longest increasing subseq of A[i...n]

$$LIS(i) = \begin{cases} 1 + LIS(j) \\ \quad j > i \\ \quad A[j] > A[i] \\ LIS(i+1) \end{cases}$$

// incl. A[i]

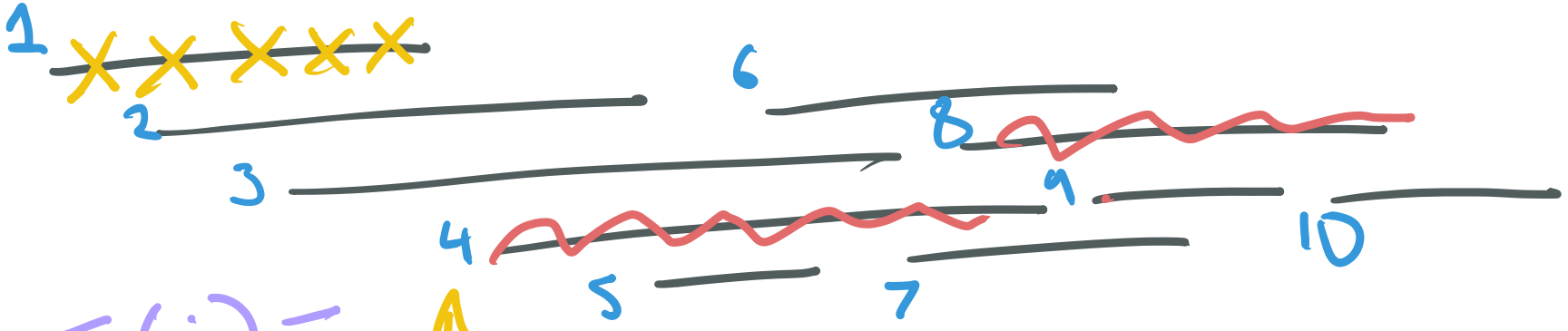
// don't include A[i]



LIS

LIS(i) = Length of the LIS of A[i+1, n] starting w/ A[i]

$$LIS(i) = \begin{cases} 1 & \text{if } A[i] > A[j] \ \forall j > i \\ \max\{1 + LIS(j)\} & \text{for } j > i, \ A[j] > A[i] \end{cases}$$



MDSI(i) = ↑

min weight of any dominating set of intervals $I_1, \ldots, I_n$

MDSI(i) =

$v^*(i)$

weight

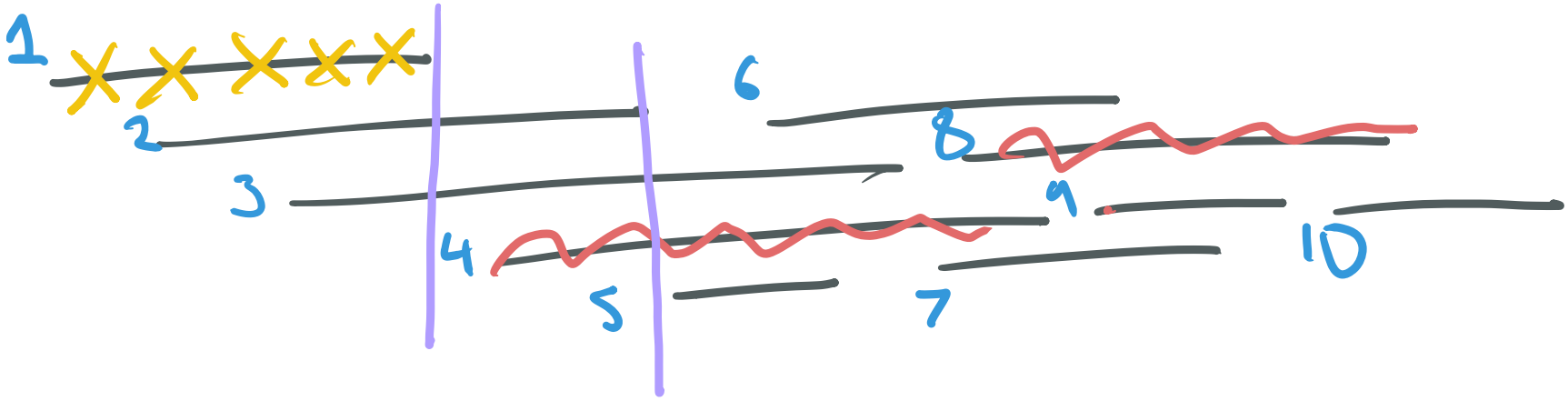
w_i + MDSI(j) (include I_i)

(j: first nonoverlapping index)

~~MDSI(i+1)~~ (don't include I_i)

MDSI(1) = min loop { w(I_1) + MDSI(4) // try I_1
                   { w(I_2) + MDSI(5) // try I_2
                   { w(I_3) + MDSI(7) // try I_3

loop


1
2
3
4
5
6
7
8
9
10