Lecture 15: Shortest Paths

CSE 373: Data Structures and Algorithms

Roadmap

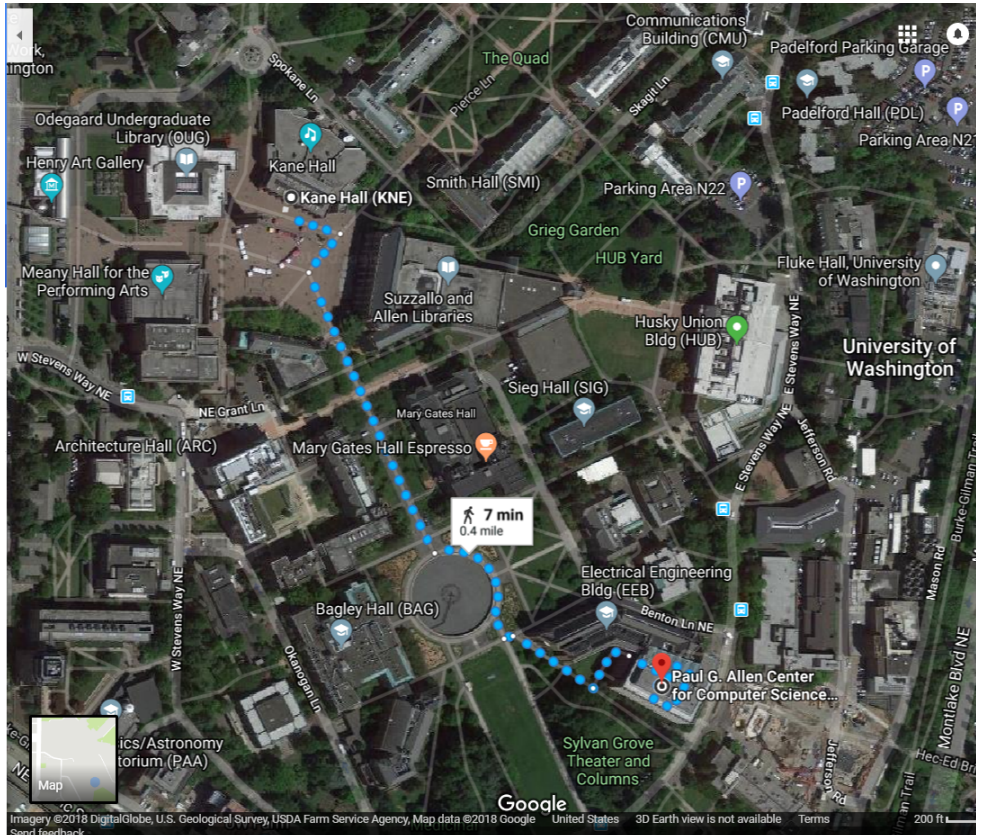
- Graphs examples, shortest paths for unweighted graphs

- using BFS to find the shortest paths

- shortest paths for weighted graphs
  - Idea 1: using BFS directly
  - Idea 2: modifying the graph
  - Idea 3: modifying the order of visiting nodes
  - examples
  - runtime if time?

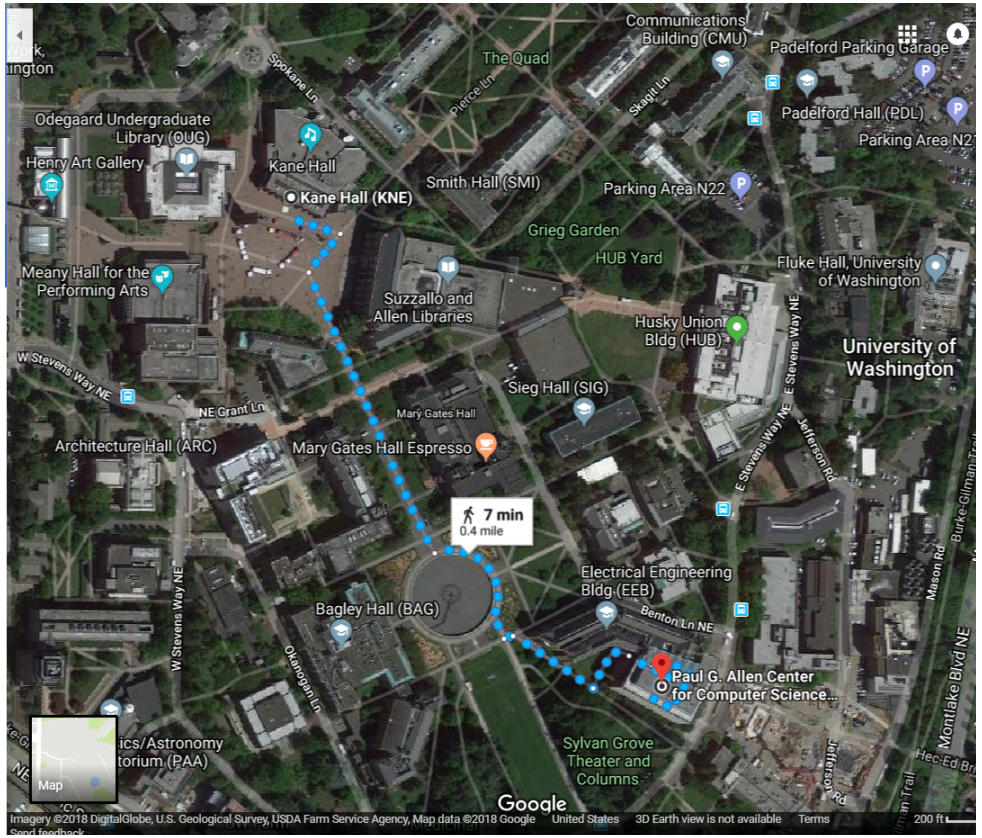
Shortest Paths

How does Google Maps figure out this is the fastest way to get from Kane Hall to the CS building?

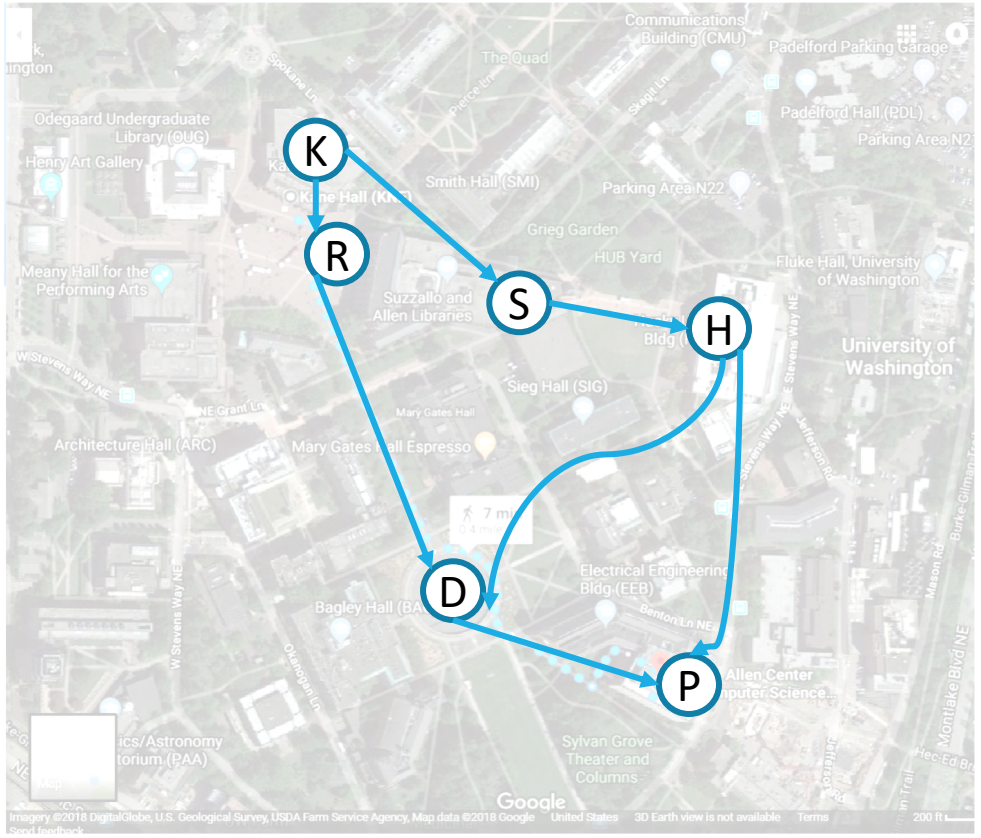


Representing Maps as Graphs

How do we represent a map as a graph? What are the vertices and edges?

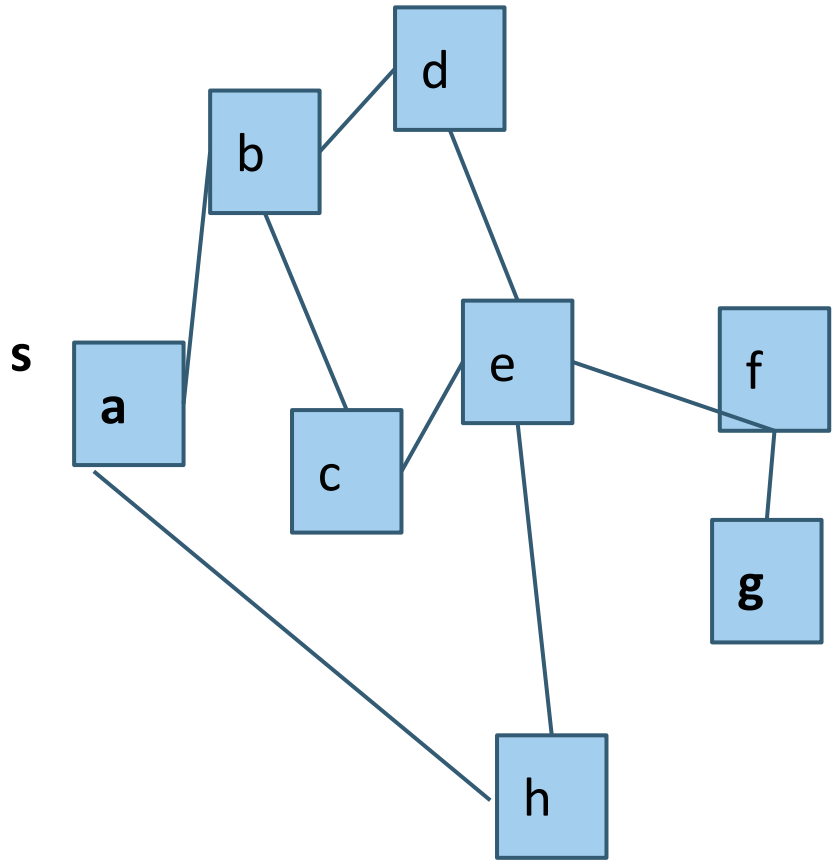


Representing Maps as Graphs



Shortest Path problem (unweighted graph)

- For the graph on the right, find **the shortest path (the path that has the fewest number of edges)** between the a node and the g node. Describe the path by describing each edge (i.e. (a, b) edge).
- What's the answer? How did we get that as humans? How do we want to do it comprehensively defined in an algorithm?



Shortest Path problem (unweighted graph)

What's the shortest path from a to a?

- Well….we're already there.

What's the shortest path from a to b or h?

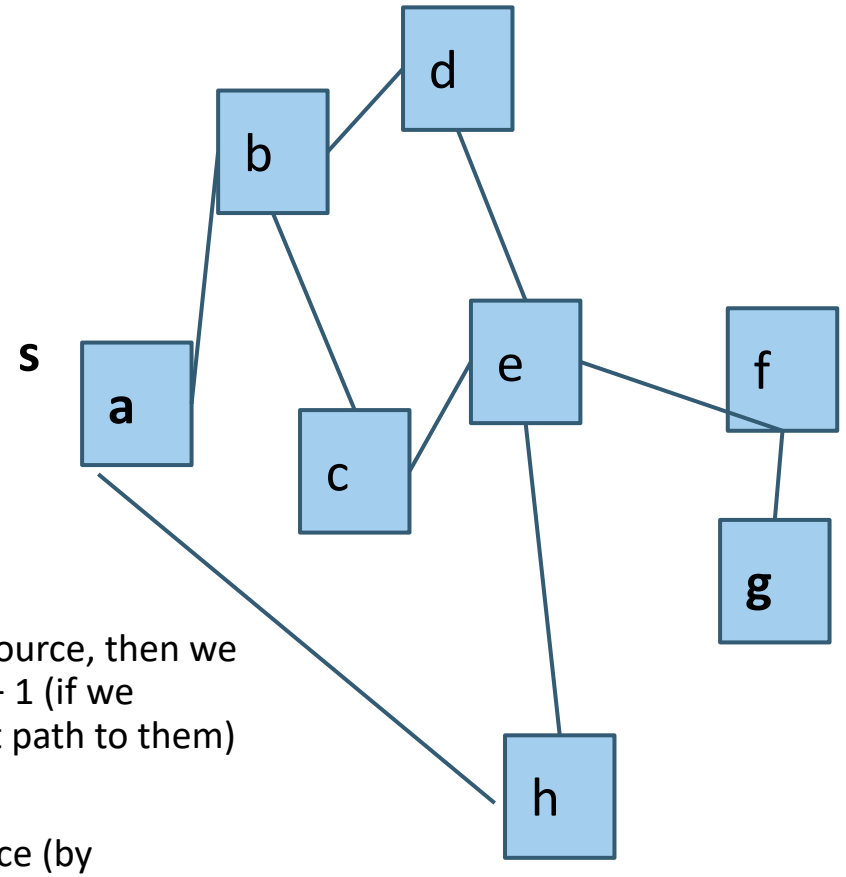
- Just go on the edge from 0

From a to d or c or e?

- Can't get there directly from a, if we want a length 2 path, have to go through b or h.

From a to f?

- Can't get there directly from a, if we want a length 3 path, have to go through e.



big idea for solving shortest paths: If we have all the nodes at distance k away from the source, then we can check all the outgoing edges from those nodes and get to all the nodes at distance k + 1 (if we haven't seen these nodes at k +1 distance before then we're just now seeing the shortest path to them)

As long as we have all the current vertices at a given distance, we can find the next distance (by traveling to all the neighbors) and the next distance and the next distance… until we finally find our target vertex and can stop.

Shortest Path problem (unweighted graph) key idea

Do we already know an algorithm that can help us get all the nodes at a given level and help us keep going through the graph level by level?

Yes! BFS! Let's modify it to fit our needs.

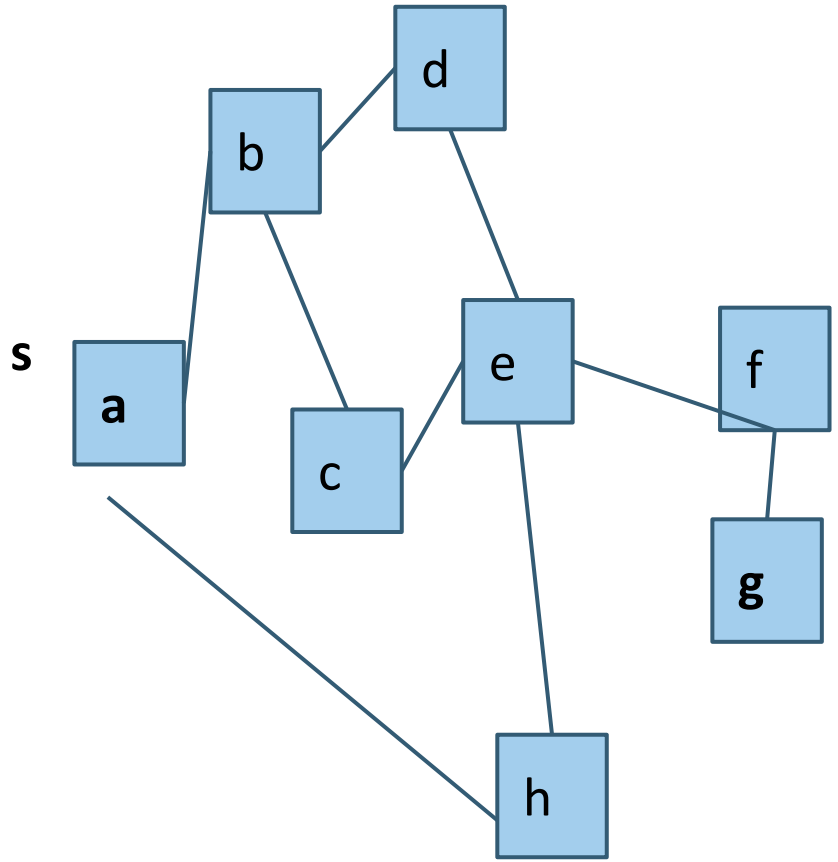
Changes from traversal BFS:

- Every node now will have an associated distance (for convenience)
- Every node V now will have an associated predecessor edge that is the edge that connects V on the shortest path from S to V. The edges that each of the nodes store are the final result.

```
perimeter.add(start);
discovered.add(start);
start's distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to's distance = from.distance + 1;
            to's predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

Shortest Path problem

Use BFS to find shortest paths in this graph.



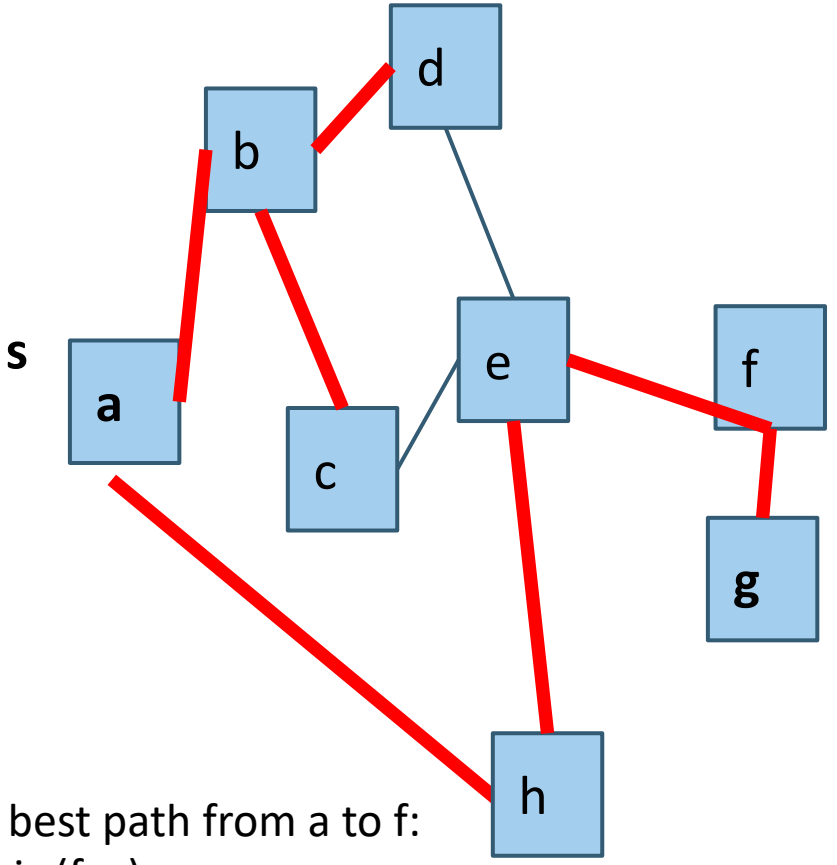

```
perimeter.add(start);
discovered.add(start);
start's distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to's distance = from.distance + 1;
            to's predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

In English:

- starting from the start vertex as the current node:
- look at all your undiscovered neighbors and record them as distance + 1, and keep track of the edge that led to them. Add them to a queue to be processed
- repeat until we traverse all that can be reached by the start node

Shortest Path problem

Use BFS to find shortest paths in this graph.

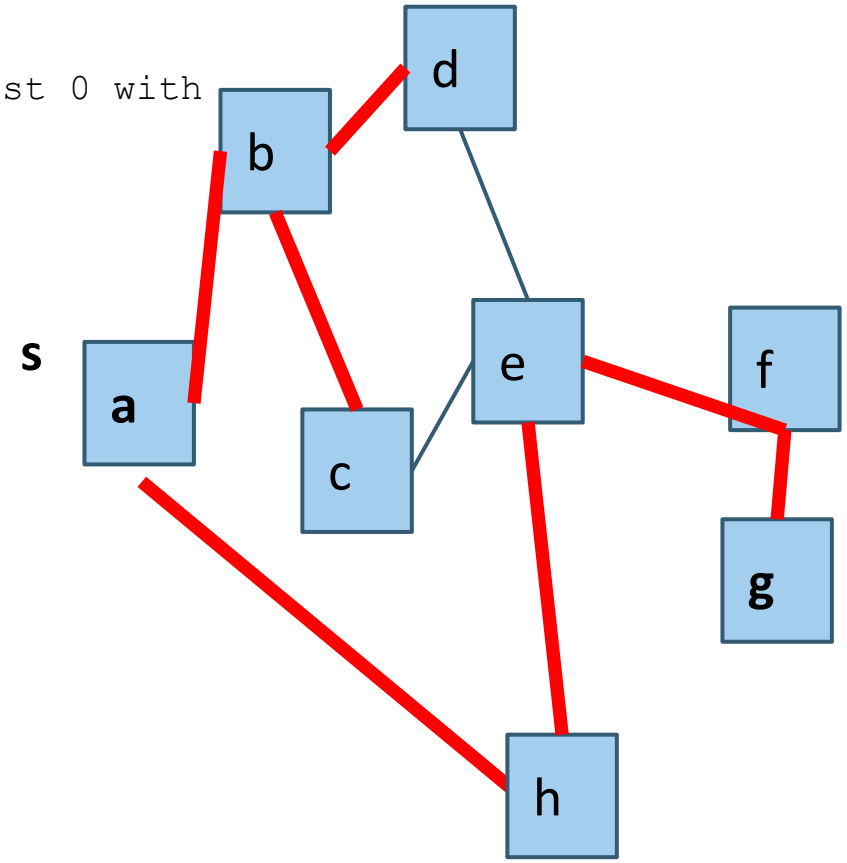



```
perimeter.add(start);
discovered.add(start);
start's distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to's distance = from.distance + 1;
            to's predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

If trying to recall the best path from a to f:
f's predecessor edge is (f, e)
e's predecessor edge is (e, h)
h's predecessor edge is (a, h)
a was the start vertex

Note: this BFS modification stores these edges individually, but there's extra work to figure out a specific path from a start / target

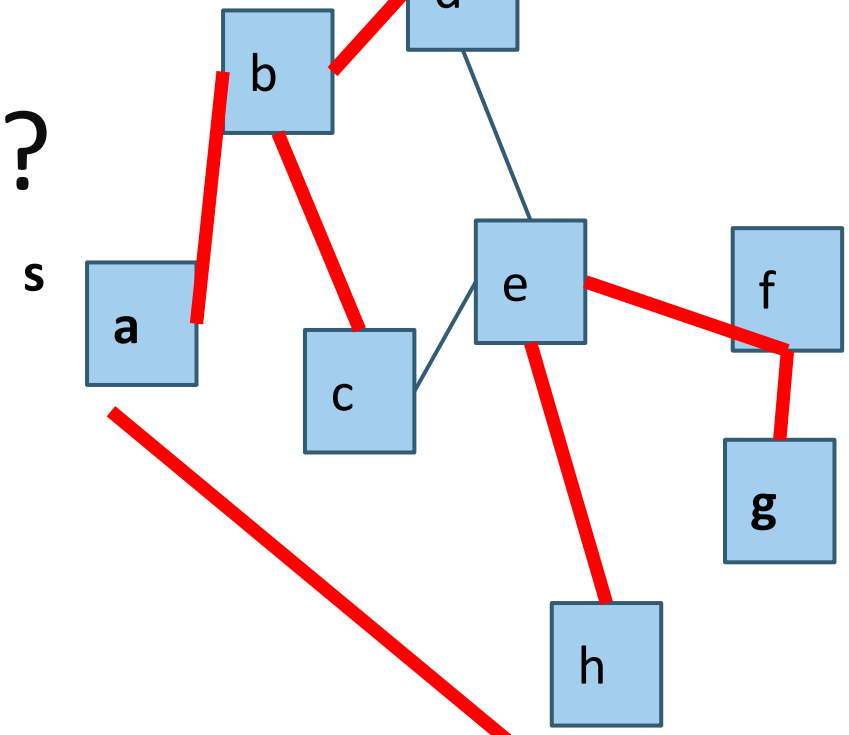
```
Map<V, E> bfsFindShortestPathsEdges(G graph, V start) {
    // stores the edge `E` that connects `V` in the shortest path from `start` to V
    Map<V, E> edgeToV = empty map

    // stores the shortest path length from `start` to `V`
    Map<V, Double> distToV = empty map


    Queue<V> perimeter = empty queue
    Set<V> discovered = empty set

    // setting up the shortest distance from start to start is just 0 with
    // no edge leading to it
    edgeTo.put(start, null);
    distTo.put(start, 0.0);

    perimeter.add(start);

    while (!perimeter.isEmpty()) {
        V from = perimeter.remove();
        for (E e : graph.outgoingEdgesFrom(from)) {
            V to = e.to();
            if (!discovered.contains(to)) {
                edgeTo.put(to, e);
                distTo.put(to, distTo(from) + 1);
                perimeter.add(to);
                discovered.add(to)
            }
        }
    }
    return edgeToV;
}
```



What about the target vertex?

Shortest Path Problem

Given: a directed graph G and vertices s,t
Find: the shortest path from s to t.



BFS didn't mention a target vertex…
It actually finds the distance from s to **every** other vertex since we needed to traverse the graph anyways. The resulting edges from our BFS shortest paths algorithm are called **the shortest path tree** (SPT: the set of all edges that are used in the shortest paths from a particular start vertex to every other vertex – see red edges above)

Both BFS modified to find the shortest paths and Dijkstra's algorithm need to start computing a shortest path tree in order to find the target.

But if you only care about one target, you can sometimes stop early (in `bfsShortestPaths`, when the target gets removed from the queue)

BFS Shortest Path Summary

- BFS works to find the shortest path summary because BFS traverses the graph level by level outwards from the start -- because we're making sure we look at all the neighbors of all the vertices on the current level, it means that the first time that we see some vertex u means that we've found the shortest path to u. There's no way there's a shorter path because of how comprehensively we're searching the graph. (If there were a shorter path, it should have been found at an earlier level!)

slightly more concrete example:

- If you start at level 0, there's only the start node, and you've definitely found the shortest path to that by default (0 edges)
- If you look at the neighbors of the start node who are all at 1 edge/distance away, you've definitely found the shortest path to all of those since they're just 1 edge length away and it's not like they could be 0 away.
- If you look at all the neighbors of the nodes who were at 1 edge/distance away, you're looking at all the nodes who are 2 distance away (when you ignore previously discovered nodes). Since you made sure we were comprehensive about finding all the nodes at 0 and 1 distance away, the nodes we're seeing for the first time here must be have 2 as the shortest possible distance.

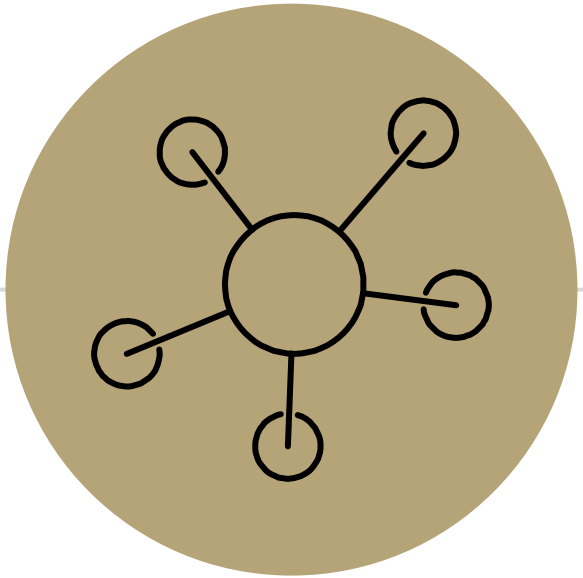
- BFS shortest paths keeps track of the actual paths by just keeping track of for every vertex V, the predecessorEdge that was used to attach V in the SPT. Which means to trace back a full path you have to loop through all of the predecessorEdges to go back to the root of the SPT/start vertex.

BFS Shortest Path Summary

another of way rephrasing this is through invariants:

- because we add vertices to the queue in level order and have the guarantee that the information in the

when a vertex is *added to the queue* with its predecessorEdge and distance recorded, we've guaranteed to have found the shortest path /distance to that vertex



Questions?

- Shortest path problem in general
- BFS to solve and traverse in level order
- solving it like this produces a SPT
- to recover an actual path have to follow the predecessor edges on the SPT

Roadmap

- Graphs examples, shortest paths for unweighted graphs
- using BFS to find the shortest paths
- **shortest paths for weighted graphs**
  - **Idea 1: using BFS directly**
  - **Idea 2: modifying the graph**
  - **Idea 3: modifying the order of visiting nodes**
  - **examples**
  - runtime if time?

Weighted Graphs

Each edge should represent the “time” or “distance” it takes to travel from one vertex to another.

Sometimes those aren’t uniform, so we put a weight on each edge to record that number.

The length of a path in a weighted graph is the sum of the weights along that path (instead of just counting the # of edges).

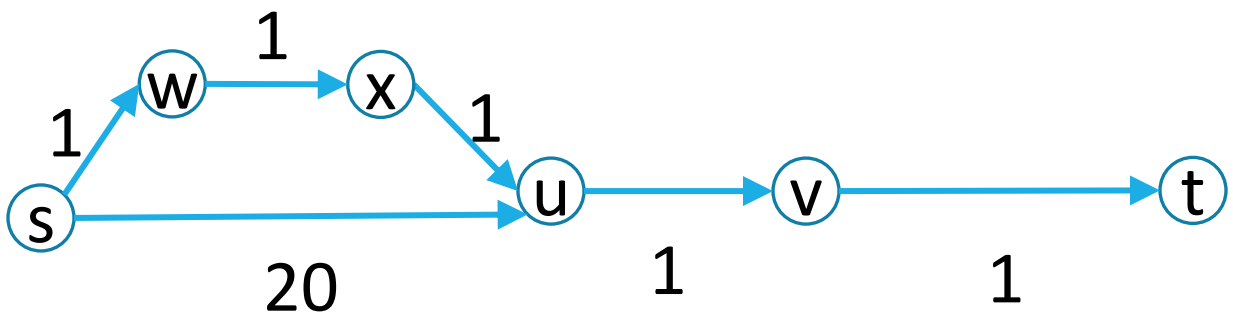
We’ll assume all of the weights are positive

- For Google Maps that definitely makes sense.
- Sometimes negative weights make sense. **Today’s algorithm doesn’t work for those graphs**
- There are other algorithms that do work.

Weighted Graphs: Take 1

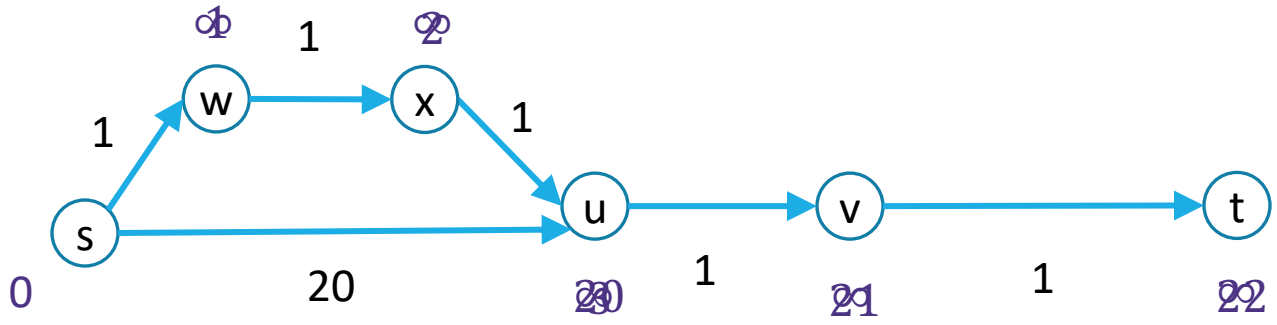
BFS works if the graph is unweighted.

Maybe it just works for weighted graphs too?



Weighted Graphs: Take 1

BFS works if the graph is unweighted. Maybe it just works for weighted graphs too?



What went wrong? When we found a shorter path from s to u, we needed to update the distance to v (and anything whose shortest path went through u) but BFS doesn't do that.

Weighted Graphs: Take 2

Reduction (informally)

Using an algorithm for Problem B to solve Problem A.

You already do this all the time.

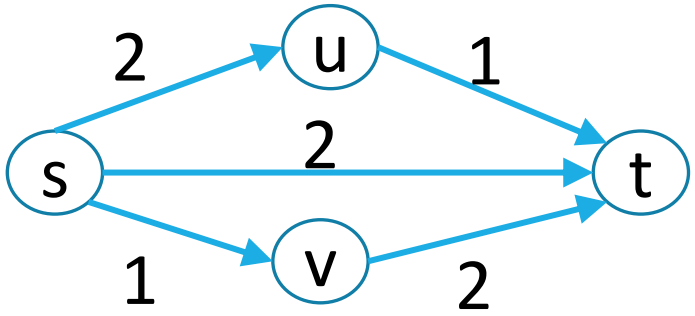
Any time you use a library, you're reducing your problem to the one the library solves.

Can we reduce finding shortest paths on weighted graphs to finding them on unweighted graphs?

Weighted Graphs: Take 2

Given a weighted graph, how do we turn it into an unweighted one without messing up the path lengths?

Weighted Graphs: A Reduction



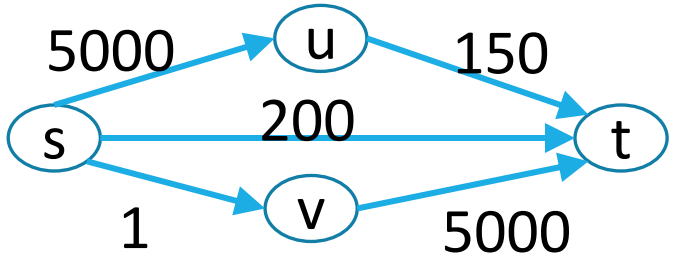
Transform Input

Unweighted Shortest Paths

Transform Output

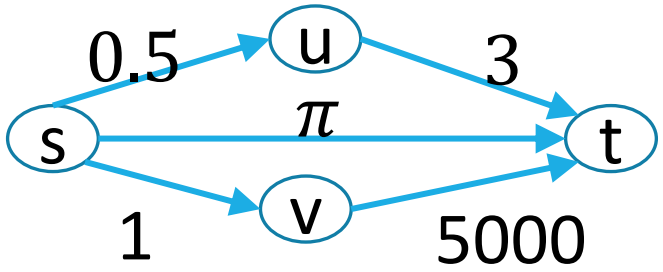
Weighted Graphs: A Reduction

What is the running time of our reduction on this graph?



BFS relies on traversing the vertices and edges, so if there are suddenly 5000x more of each the runtime will get a lot worse.

Does our reduction even work on this graph?



Ummm….

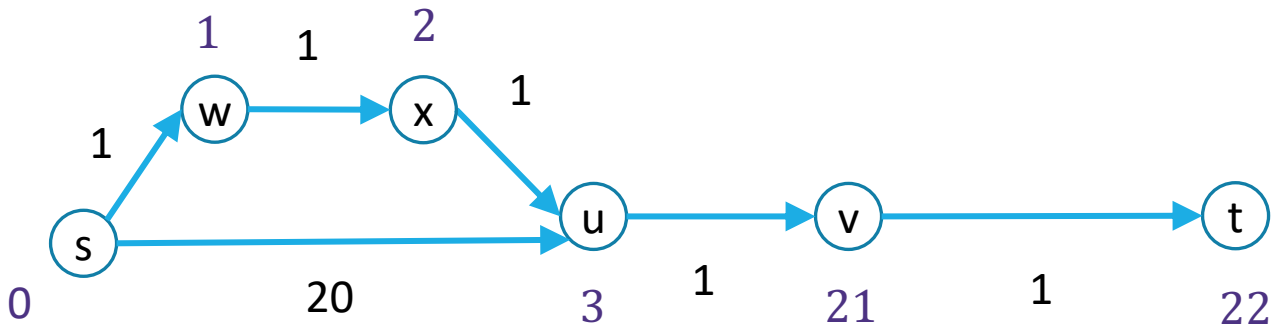
tl;dr: If your graph's weights are all small positive integers, this reduction might work great.
Otherwise we probably need a new idea.

Weighted Graphs: Take 3

So we can't just do a reduction.

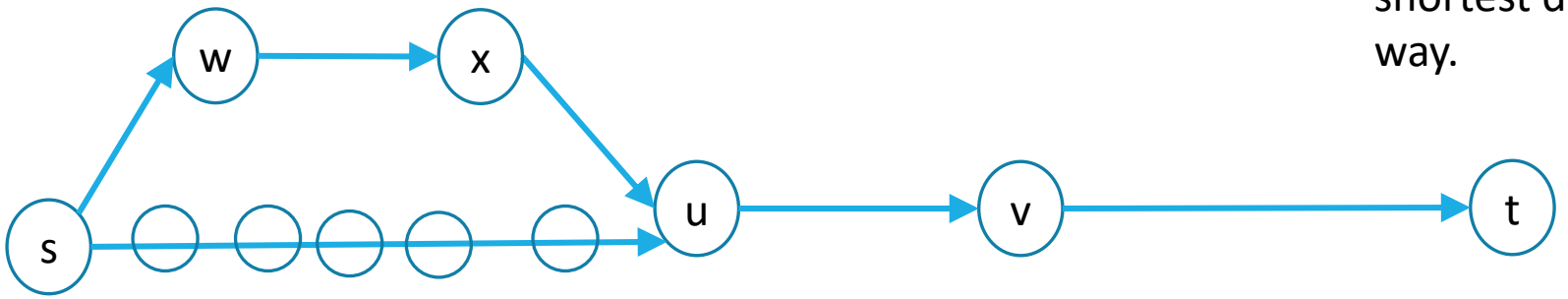
Instead figure out why BFS worked in the unweighted case, try to make the same thing happen in the weighted case.

How did we avoid this problem:

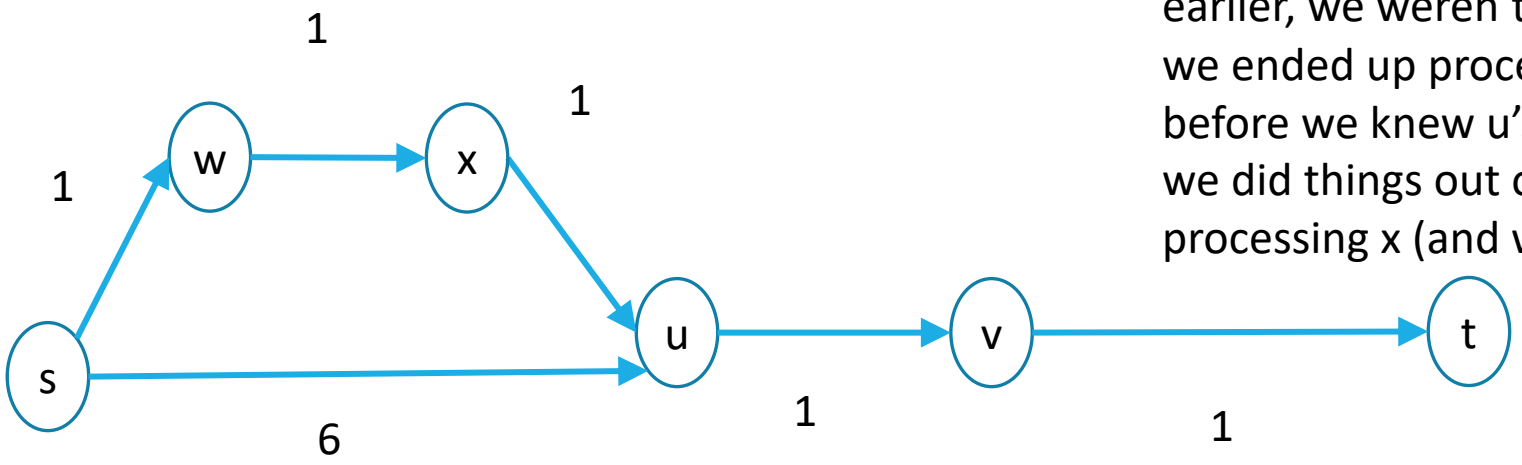


Weighted Graphs: Take 3

In the unweighted case, we have this guarantee that the way we traverse the graph level-by-level, we're always finding the shortest distance to each vertex along the way.

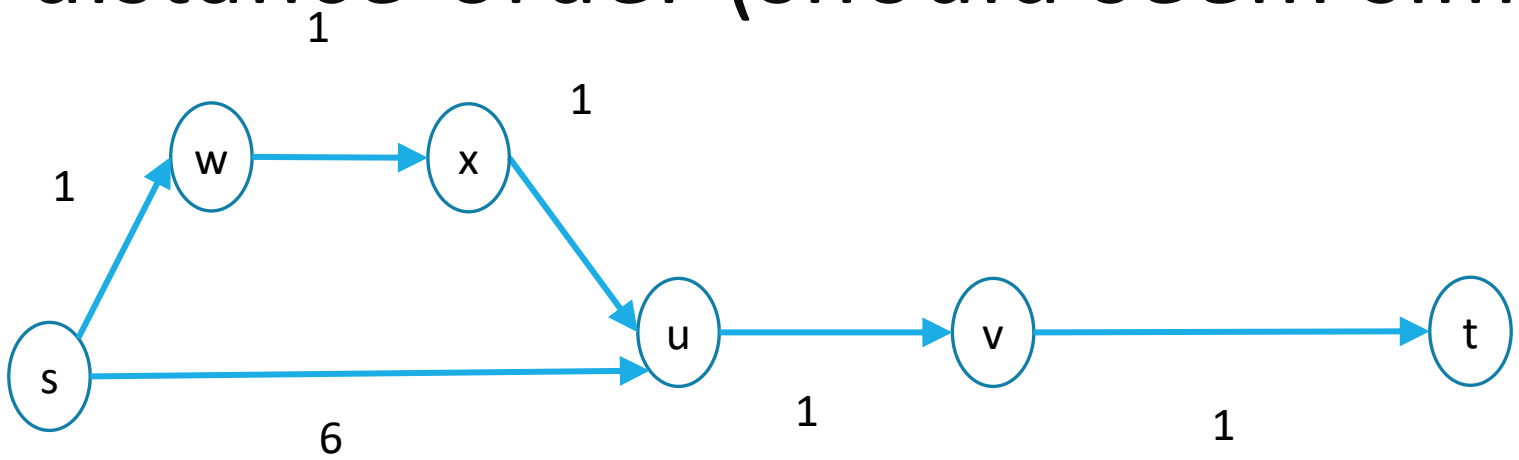


When we tried to do the same BFS on the weighted graph earlier, we weren't taking the edge weights into account and we ended up processing u and updating its out neighbors before we knew u's true shortest path / distance. So it seems we did things out of order and should have processed u after processing x (and w).



Idea: what if we copy the level-by-level idea but instead traverse distance-by-distance?

Weighted Graphs: Take 3 -Visiting nodes in distance-order (should seem similar to BFS still)



- One way to think about our new algorithm: imagine that every edge weight represents the distance it takes to traverse that edge, and that we'll simulate the order of traversal by trying to explore all of our outgoing edges at the same time but with respect to the edge weight distances.

- Say we start at search at s and want to traverse to our neighbors 1 distance/time away: since we're thinking of this process traversing in all possible directions, how far do we get? Well S → W is only distance 1, so we actually get to W. The S →U edge that we know about is distance 6, so even though we started traversing it, we only get 1/6 of the way through that edge.

- Say another unit of time passes in our constant traversal and we traverse 1 more distance in all the directions, and now the upper path has traversed W→X (total distance from S is 2), and meanwhile S → U is only 2/6 done.

- Say another unit of time passes and we get to U through X→U (total dist from S is 3), and S→U is 3/6 done.

This order of processing nodes guarantees that when we finally reach a node U (with the constant rate of travel idea) we've guaranteed we found the shortest path to it (just like BFS if there were a shorter path, we would've gotten here sooner since we're exploring all the paths simultaneously).

Weighted Graphs: Take 3

Goal: Process the vertices in order of **distance from s** (instead of level order)

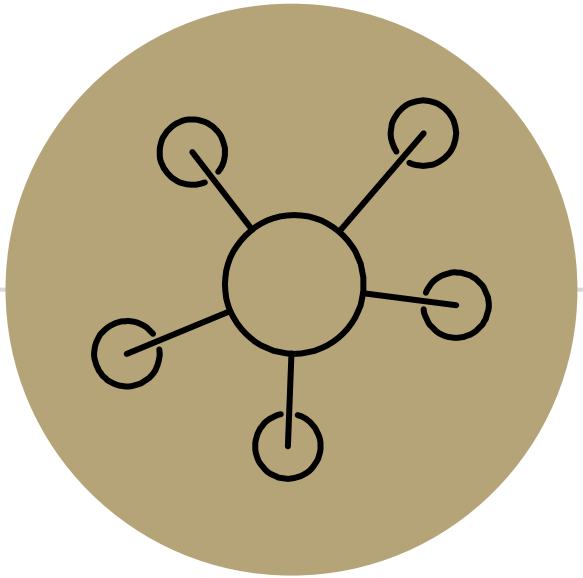
Idea:

Have a set of vertices that are "known"
- (we know at least one path from s to them).

Record an estimated distance
- (the best way we know to get to each vertex).

If we process only the vertex closest in estimated distance, we won't ever find a shorter path to a processed vertex. This takes a jump from the previous way of distance-by-distance, but is the same high-level idea.

For example, there are some cases in the distance-by-distance approach where you could say we'd be 3/7 of the way through one edge and 3/9 way through another edge. Instead of manually stepping through +1 each time, this is like cutting to the answer and just realizing that the 7 edge would finish before the 9 (we should process the edge with weight 7 first) if they both started at the same time.
- This statement is the key to proving correctness.



Questions?

- weighted graph shortest paths
- using BFS directly
- modifying the graph
- traversing in distance-order

Dijkstra's algorithm (pseudocode + English)

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
     mark u as processed
   }
```

- propose all the estimated distances to all nodes is infinity, except for the start which is 0

- start at your start vertex

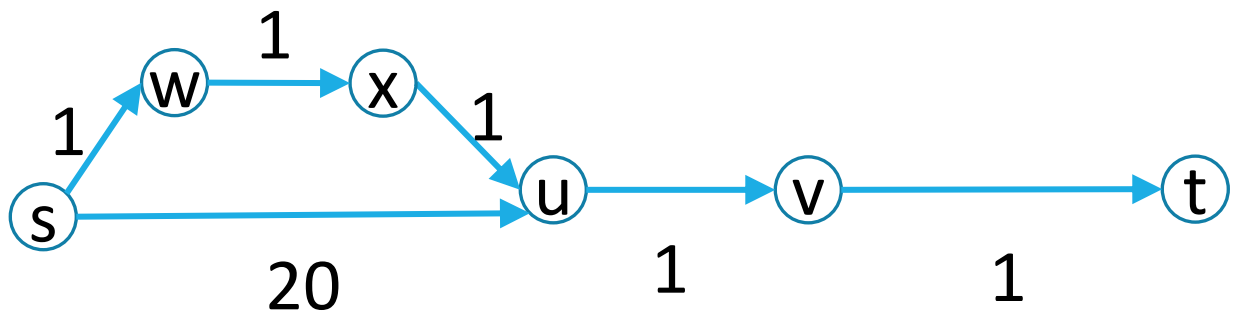
  - for the current vertex, look at all of the outgoing neighbors/their edges. If the distance to the current node + that edge weight is smaller than the proposed estimated distance for that neighbor, ***relax*** the neighbor node (update the proposed estimated distance and predecessor edge).

  - update the current vertex to be the vertex w the next smallest estimated distance that hasn't been processed

Dijkstra's Algorithm

```
Dijkstra(Graph G, Vertex source)
    initialize distances to ∞
    mark source as distance 0
    mark all vertices unprocessed
    while(there are unprocessed vertices){
        let u be the closest unprocessed vertex
        foreach(edge (u,v) leaving u){
            if(u's dist+weight(u,v) < v's dist){
                v's dist = u.dist+weight(u,v)
                v's predecessor = (u,v)
            }
        }
       mark u as processed
    }
```

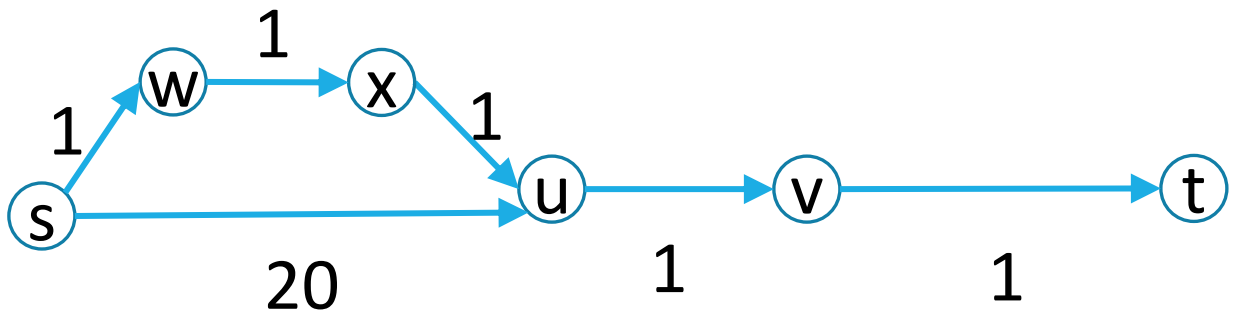
Vertex	Distance	Predecessor	Processed
s			
w			
x			
u			
v			
t			



Dijkstra's Algorithm

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
     mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
s	0	--	Yes
w	1	(s,w)	Yes
x	2	(w, x)	Yes
u	~~20~~ 3	~~(s,u)~~ (x, u)	Yes
v	4	(u, v)	Yes
t	5	(v, t)	Yes

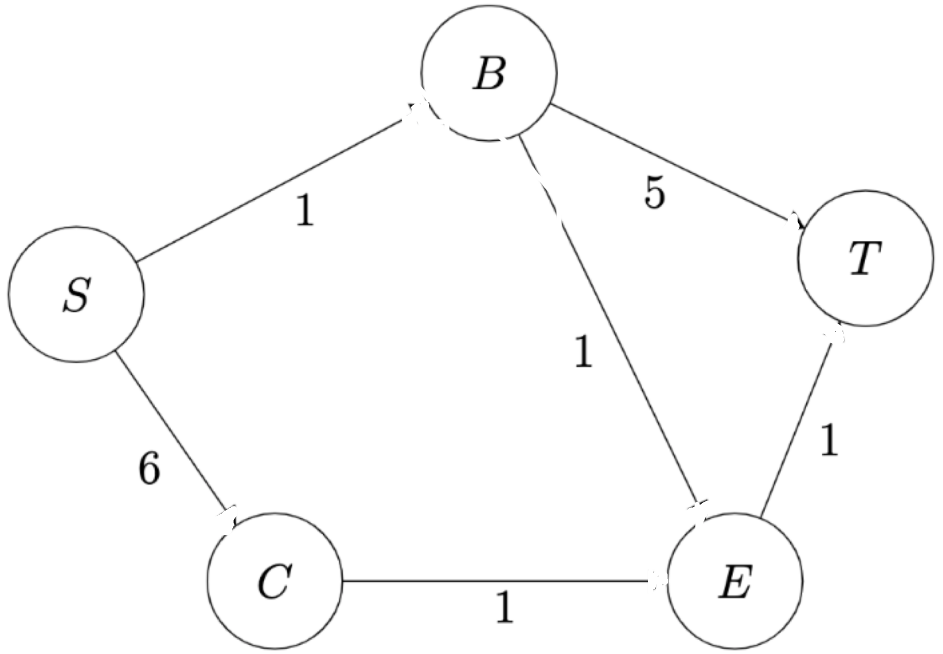


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S			
C			
B			
T			
E			

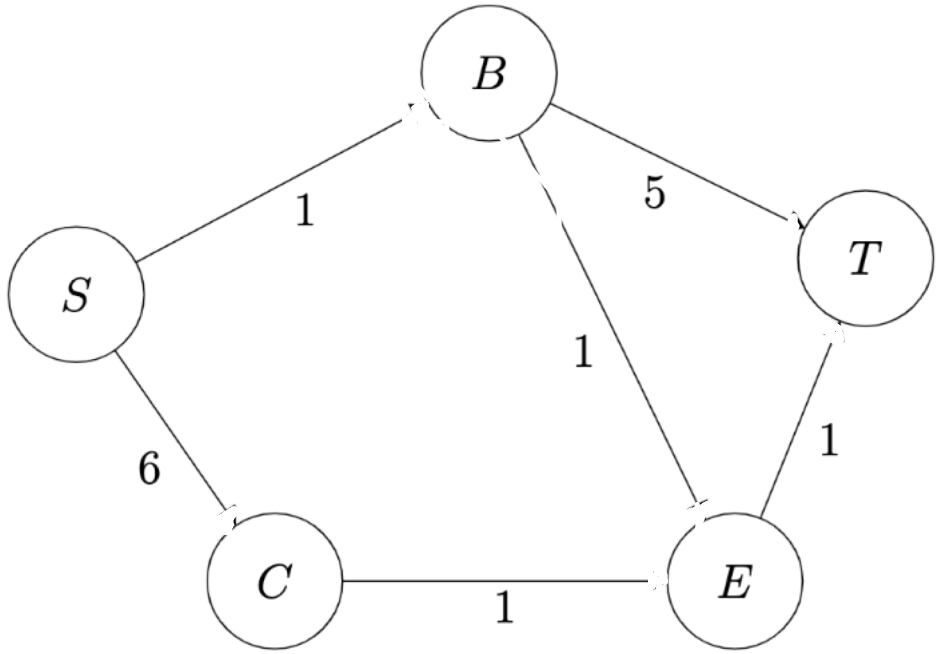


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0		No
C	∞		No
B	∞		No
T	∞		No
E	∞		No

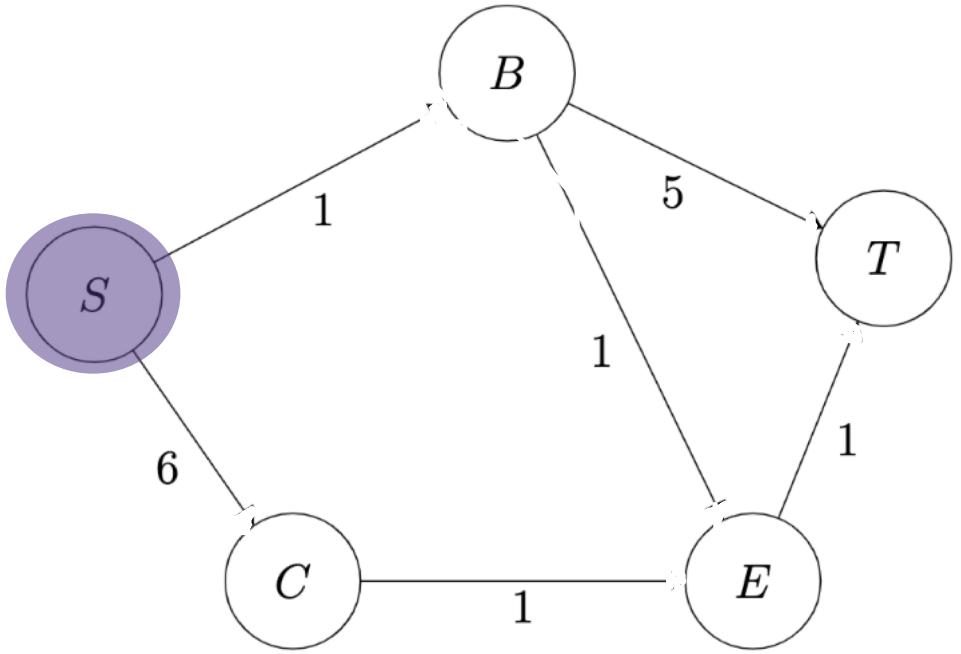


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	No
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

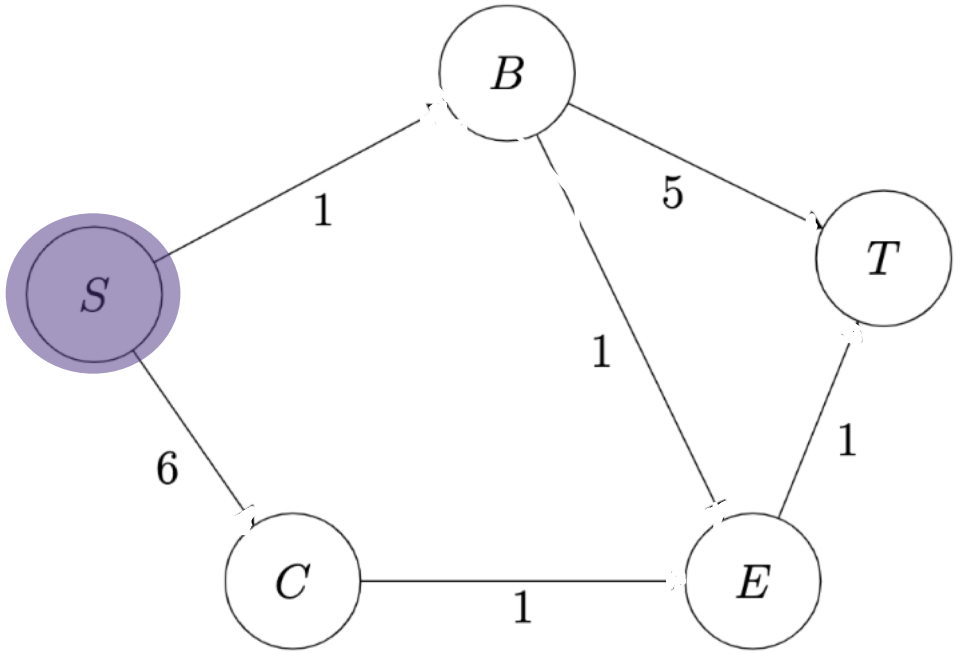


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

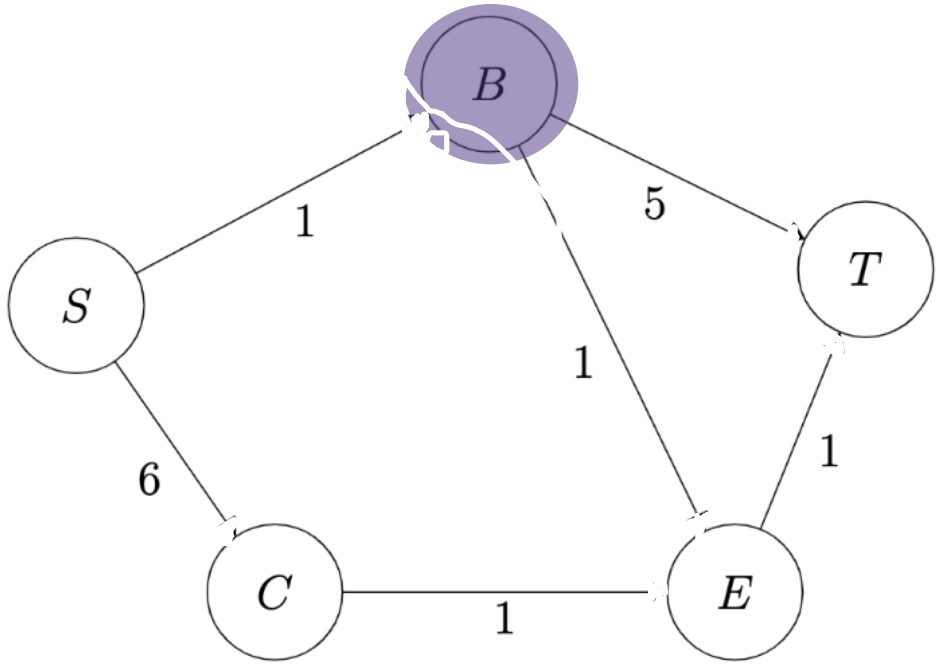


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	6	B	No
E	2	B	No

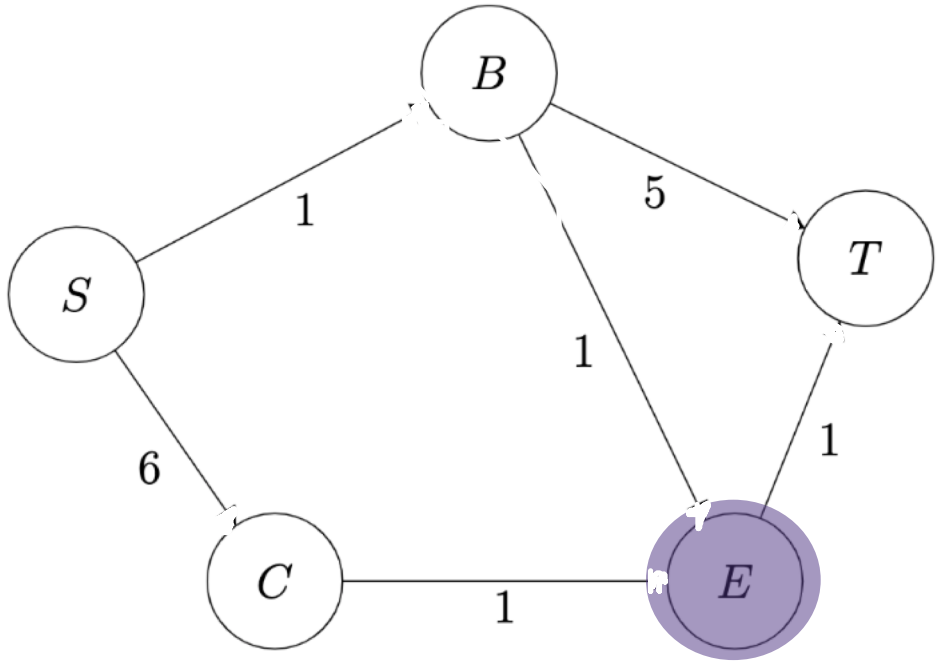


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	No
E	2	B	Yes

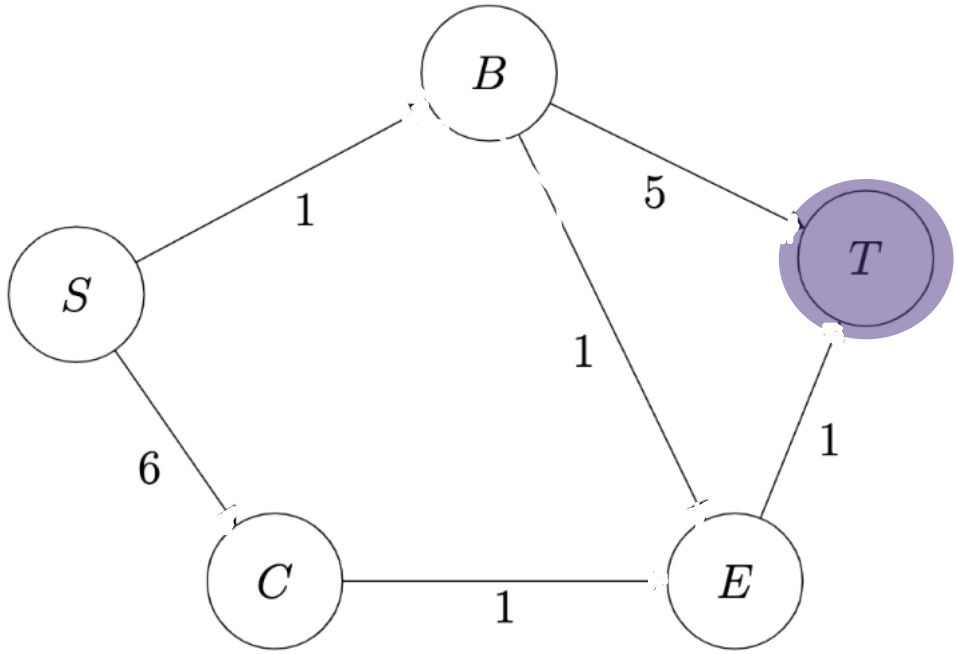


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v's dist = u.dist+weight(u,v)
            v's predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	Yes
E	2	B	Yes



Dijkstra's Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

Huh?

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

Dijkstra's Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   initialize MPQ as a Min Priority Queue, add source
   while(there are unprocessed vertices){
      u = MPQ.removeMin();
      foreach(edge (u,v) leaving u){
         if(u's dist+weight(u,v) < v's dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = (u,v)
         }
      }
      mark u as processed
   }
```

How?

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

What are the high-level differences between using BFS and Dijkstra's algorithm?

```
findShortestPathsTree(G unweightedGraph, V start) {
     Map<V, E> edgeToV = empty map
     Map<V, Double> distToV = empty map

     Queue<V> perimeter = empty queue
     Set<V> discovered = empty set



     perimeter.add(start);
     distTo.put(start, 0.0);

     while (!perimeter.isEmpty()) {
         V from = perimeter.remove();
         for (E e : unweightedGraph.outgoingEdgesFrom(from)) {
             V to = e.to();
             if (!discovered.contains(to)) {
                 edgeTo.put(to, e);
                 distTo.put(to, distTo(from) + 1);
                 perimeter.add(to);
                 discovered.add(to)
             }
         }
     }
 }
```

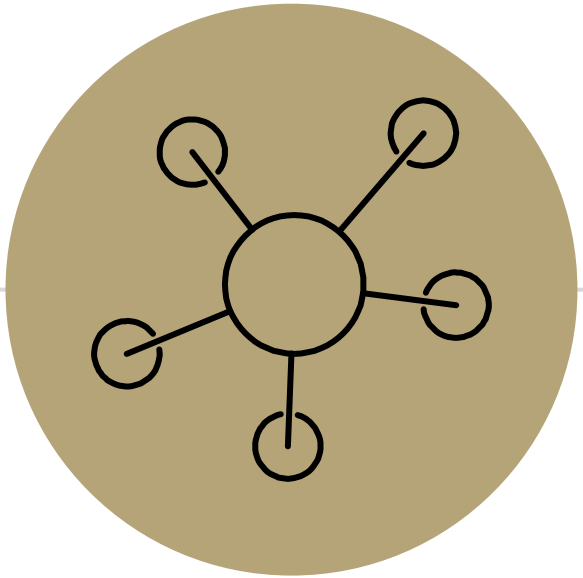
```
findShortestPathsTree(G weightedGraph, V start) {
     Map<V, E> edgeToV = empty map
     Map<V, Double> distToV = empty map


     PQ<V> orderedPerimeter = empty pq

     initialize all distTo's to ∞ so the best paths
     can be updated if any path is found to that vertex

     orderedPerimeter.add(start, 0);
     distTo.put(start, 0.0);

     while (!orderedPerimeter.isEmpty()) {
         V from = orderedPerimeter.removeMin();
         for (E e : weightedGraph.outgoingEdgesFrom(from)) {
             V to = e.to();
             double oldDist = distTo.get(to);
             double newDist = distTo.get(from) + e.weight();
             if (newDist < oldDist) {
                 edgeToV.put(to, e);
                 distToV.put(to, newDist);
                 if (pq contain to) {
                    orderedPerimeter.changePriority(to, newDist);
                 } else {
                    orderedPerimeter.add(to, newDist);
                 }
             }
         }
     }
}
```

What are the high-level differences between using BFS and Dijkstra's algorithm?

BFS iterates in level order, ordering in-between levels is not important by default. So BFS uses a Queue as it's internal data structure to keep track of the ordering.

Dijkstra's iterates in priority order, prioritizing processing nodes with the next smallest estimated distance (so that we get that guarantee that we're looking at correct information). So Dijkstra's uses a PriorityQueue as it's internal data structure to keep track of the ordering.

Overall they're really similar and Dijkstra's just has a few more steps than BFS, so if you're confused, start with understanding BFS shortest paths and then after you feel comfortable with that, tackle practicing / understanding Dijktra's.



Questions?

BFS/DFS runtime

```
perimeter.add(start);
discovered.add(start);
start's distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to's distance = from.distance + 1;
            to's predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

All of the data structure operations (remove, add, contains) are all constant runtime and so are getting values out of the graph (neighbors, distance, etc.).

So the main runtime is going to come from just how much we're looping / how many things we're looking at. Since we know that we could loop through all vertices, we know it's going to take at least n time (where n is the number of nodes)

And for each of those vertices, we loop through all of it's edges. Caution: m represents the number of edges in the whole graph, so we can't just use n * m for our runtime, because each vertex only looks at its own neighbors, not the entire edge list for the whole graph.

It turns out the way to model this standard traversal is n + m runtime (in the worst case if the whole graph is actually explored). This is because we look at every vertex once and we actually look at every unique edge twice (from the perspectives of the 2 different vertices attached to the edge).

Another way to think about this is that the inner for each loop actually runs in m/n time, where m/n represents the average number of edges per node. If you multiply this happening actually n times, then you get that the code inside the inner for loop runs m times. (And the code inside the while loop and outside of the for loop like the Queue.remove runs n times).

Dijkstra's Runtime

Just like when we analyzed BFS, don't just work inside out; try to figure out how many times each line will be executed.

```
Dijkstra(Graph G, Vertex source)
   for (Vertex v : G.getVertices()) { v.dist = INFINITY; }
   G.getVertex(source).dist = 0;
   initialize MPQ as a Min Priority Queue, add source
   while(MPQ is not empty){
      u = MPQ.removeMin();                                  +logV
      for (Edge e : u.getEdges(u)){
         oldDist = v.dist; newDist = u.dist+weight(u,v)
         if(newDist < oldDist){
            v.dist = newDist
            v.predecessor = u
            if(oldDist == INFINITY) { MPQ.insert(v) }       +logV
            else { MPQ.updatePriority(v, newDist) }
         }
      }
   }
```

This actually doesn't run m times for every iteration of the outer loop. It actually will run m times in total; if every vertex is only removed from the priority queue (processed) once, then we examine each edge once. Each line inside this foreach gets multiplied by a single E instead of E * V.

Tight O Bound = O(n log n + m log n)

Dijkstra's Wrap-up

The details of the implementation depend on what data structures you have available.

Your implementation in the programming project will be different in a few spots.

Our running time is $\Theta(E \log V + V \log V)$ i.e. $\Theta(m \log n + n \log n)$.

Dijkstra's algorithm - Wikipedia

https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Apps | CSE TA Assignment... | A Tale Of Two Subu... | WINE 2019: The 15t... | 3 | [Lecture 4] What ar... | Mindstorms: childre... | EA0XJZPW4AEqH3...

WIKIPEDIA

Dijkstra's algorithm

Class Search algorithm

Data structure Graph

Worst-case performance $O(|E| + |V| \log |V|)$

Page information
Wikidata item
Cite this page

In other projects

when traversing an edge) are monotonically non-decreasing. This generalization is called the Generic Dijkstra shortest-path algorithm.[6]

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (where $|V|$ is the number of nodes). The idea of this algorithm is also given in Leyzorek et al. 1957. The implementation based on a min-priority queue

Worst-case performance	$O(\|E\| + \|V\| \log \|V\|)$

Graph and tree

Dijkstra's Wrap-up

The details of the implementation depend on what data structures you have available.

Your implementation in the programming project will be different in a few spots.

Our running time is $\Theta(E \log V + V \log V)$ i.e. $\Theta(m \log n + n \log n)$.

If you go to Wikipedia right now, they say it's $O(E + V \log V)$

They're using a Fibonacci heap instead of a binary heap.

$\Theta(E \log V + V \log V)$ is the right running time for this class.

Shortest path summary:

- BFS works great (and fast -- $\Theta(m + n)$ time) if graph is unweighted.
- Dijkstra's works for weighted graphs with no negative edges, but a bit slower $\Theta(m \log n + n \log n)$
- Reductions!