Lecture 16: Graph Modeling

CSE 373: Data Structures and Algorithms

Announcements

project 3 due (feedback quiz canvas for extra credit)

exercise 3 due friday

- note on N vs R and P: please actually use the R and P variables if the runtime is actually based on the number of possible reaction types or number of possible persons. You use N (the normal variable) bc there are two factors here – if you write N, we can't tell which one you mean so it's not correct.

project 4 out later today/maybe tomorrow morning, we're pushing for tonight though

lots of good lecture questions from Monday – check out Piazza if you want to see those!

Post-CSE 373 Pathways session Saturday 6pm PDT

- technical interviews / applying for software jobs / how CSE 373 fits in

- Q&A with TAs

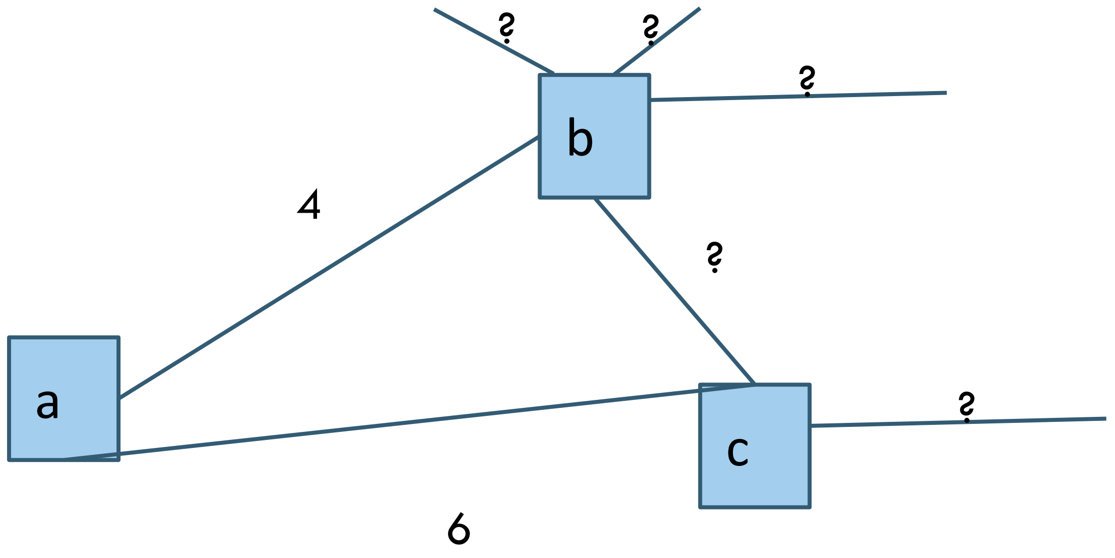
- recorded in case you can't make it

Review from last time

- BFS can be used to find shortest paths (path length = # of edges) for unweighted graphs, guaranteed to work because it traverses level by level
  - BFS can be used by keeping track of a predecessor edge (the edge that led to each vertex when we found it for the first time) and if you drew out all of those at the end, you'd see those edges represent the Shortest Path Tree (the short path from the source vertex to all the other vertices)
  - BFS shortest paths doesn't work on weighted graphs (paths lengths = sum of the edge weights along the path) because BFS's traversal order doesn't take into account weights. We can use the edge weights (total distance) to figure out the exact order to visit things in so our algorithm is correct
    - Dijkstra's algorithm = this differently ordered traversal / algorithm to find the shortest path on weighted graph
- Dijkstra's pseudocode / in English (it's pretty similar to BFS shortest paths)

Review from last time

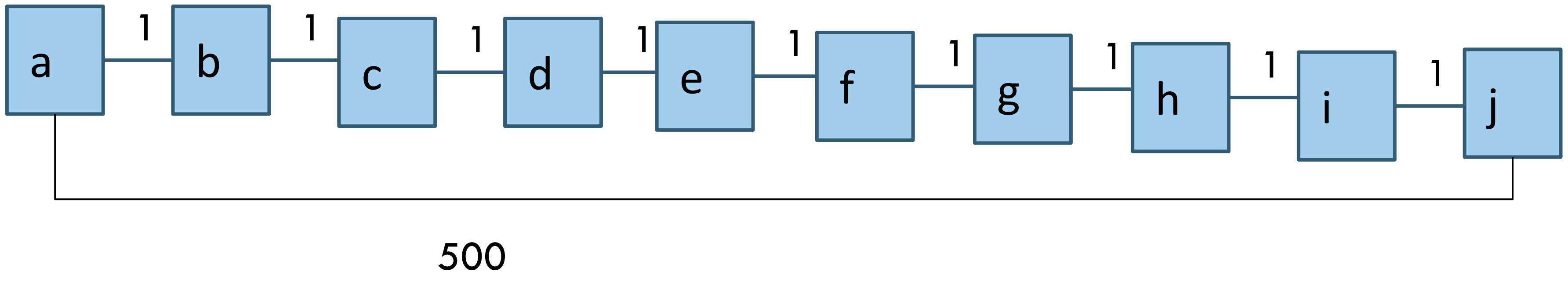
- BFS shortest paths doesn't work on weighted graphs (paths lengths = sum of the edge weights along the path) because BFS's traversal order doesn't take into account weights. We can use the edge weights (total distance) to figure out the exact order to visit things in so our algorithm is correct.



Idea: when choosing the next node to process next, choose the next smallest distanced node that you know. This should give you the flexibility to be comprehensive about your search / traversal.

In the above example, imagine that after we finish processing A (record that there are possible distances of 4 to b and 6 to c) we want to figure out which node we should process next. Without analyzing all the paths coming out from B and C, we know that since the distance to b is smaller, there's a possibility b could have a better shortest path that ends up leading to c (or other nodes) that we'd want to incorporate and propagate that information.

Another example of Dijkstra's ordering



BFS processing in level order would make sure the things at level 1 away (b and j) both get processed / completed before nodes at level 2 +.

Dijkstra's processing in distance order would visit B and see that the distance to C is only 2, which is still smaller than 500. Hmmm there might still be a better path to J (which we propose has distance 500) if we keep going this way. Repeat for C → D, distance of 3 still less than 500, so it's possible and we should look at D first.

To summarize: Dijkstra's algorithm will visit things in next-closest-distance order to make sure we're comprehensive (just like how BFS goes level by level, we go distance by distance)

Review from last time

- BFS can be used to find shortest paths (path length = # of edges) for unweighted graphs, guaranteed to work because it traverses level by level
  - BFS can be used by keeping track of a predecessor edge (the edge that led to each vertex when we found it for the first time) and if you drew out all of those at the end, you'd see those edges represent the Shortest Path Tree (the short path from the source vertex to all the other vertices)
  - BFS shortest paths doesn't work on weighted graphs (paths lengths = sum of the edge weights along the path) because BFS's traversal order doesn't take into account weights. We can use the edge weights (total distance) to figure out the exact order to visit things in so our algorithm is correct
    - Dijkstra's algorithm = this differently ordered traversal / algorithm to find the shortest path on weighted graph

- Dijkstra's pseudocode / in English (it's pretty similar to BFS shortest paths)

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 to be the current 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

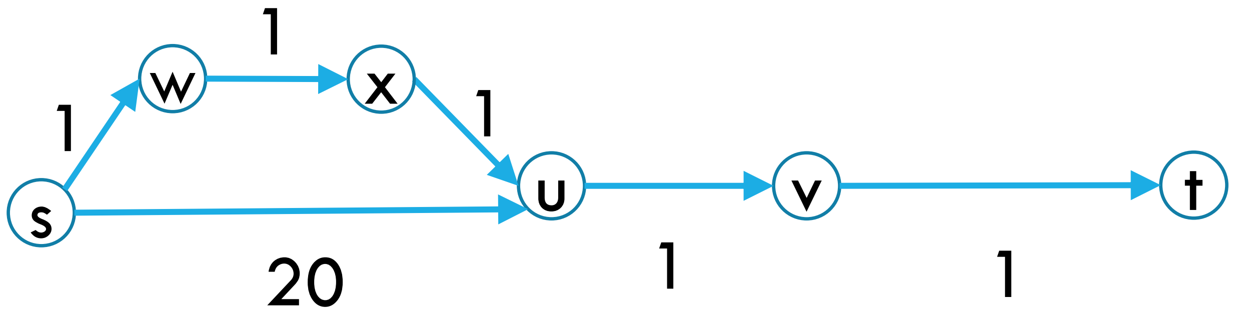
2 differences from BFS pseudocode:

- “let u be the closest unprocessed vertex” (not going necessarily by level order anymore)
- checking if the current distance + weight is better than what we’ve seen so far (BFS assumes its correct the first time it discovers a node, but Dijkstra’s has the potential to update to a better distance later, so we check if that’s the case)

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’s dist+weight(u,v)
            v’s predecessor = (u,v)
         }
      }
    mark u as processed
  }
```

Vertex	Distance	Predecessor	Processed
s			
w			
x			
u			
v			
t			

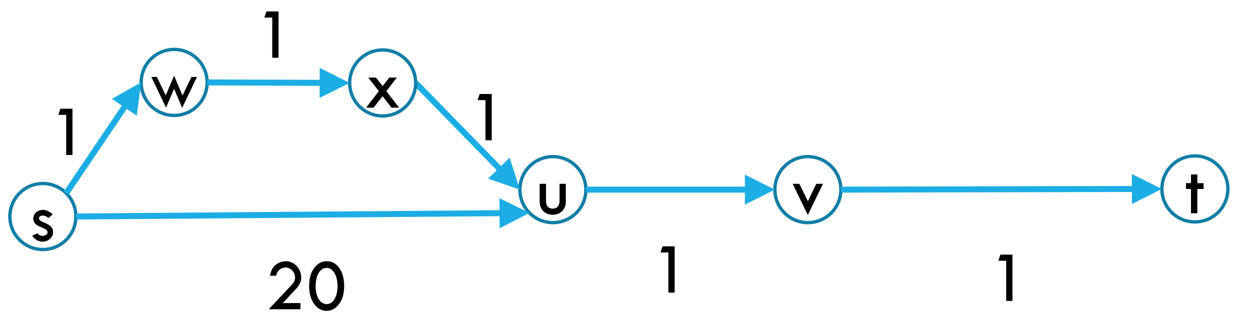


$$\begin{tikzcd} & w \arrow[r, "1"] & x \arrow[dr, "1"] & & & \\ s \arrow[ur, "1"] \arrow[rrr, "20"'] & & & u \arrow[r, "1"'] & v \arrow[r, "1"'] & t \end{tikzcd}$$

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’s 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 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  ←——— Huh?
      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
   }
```

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){ ← How?
    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 ←
  }
```

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)
             }
         }
     }
 }
```

BFS to find shortest paths tree

```
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);
                }
            }
        }
    }
}
```

Dijkstra's to find shortest paths tree (use this for p4)

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

BFS to find shortest paths	Dijkstra’s to find shortest paths
only on unweighted graphs (breaks on weighted graphs) 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.	works on weighted graphs 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:
- both produce an SPT (the set of edges used to find the shortest path to every vertex)
- both can be used on undirected or directed graphs
- 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.

Overall intuition: we look at every vertex once (when we take it out of the queue/stack) and we look at every edge that exists in the graph twice (there will be two instances when our current vertex is one of the vertices attached to this edge in question), so the runtime is just Theta(n + m).

BFS/DFS runtime

Runtime details -- a different strategy than before: calculating how many times each line runs in the whole method / not across one particular loop:

```
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)
        }
    }
}
```

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). perimeter.remove() will run n times.

And we know that since all the edges will be looped through, we know that edge.to() and discovered.contains() will run m times.

How to think about it by multiplying loops: 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.add(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 m instead of m * n.

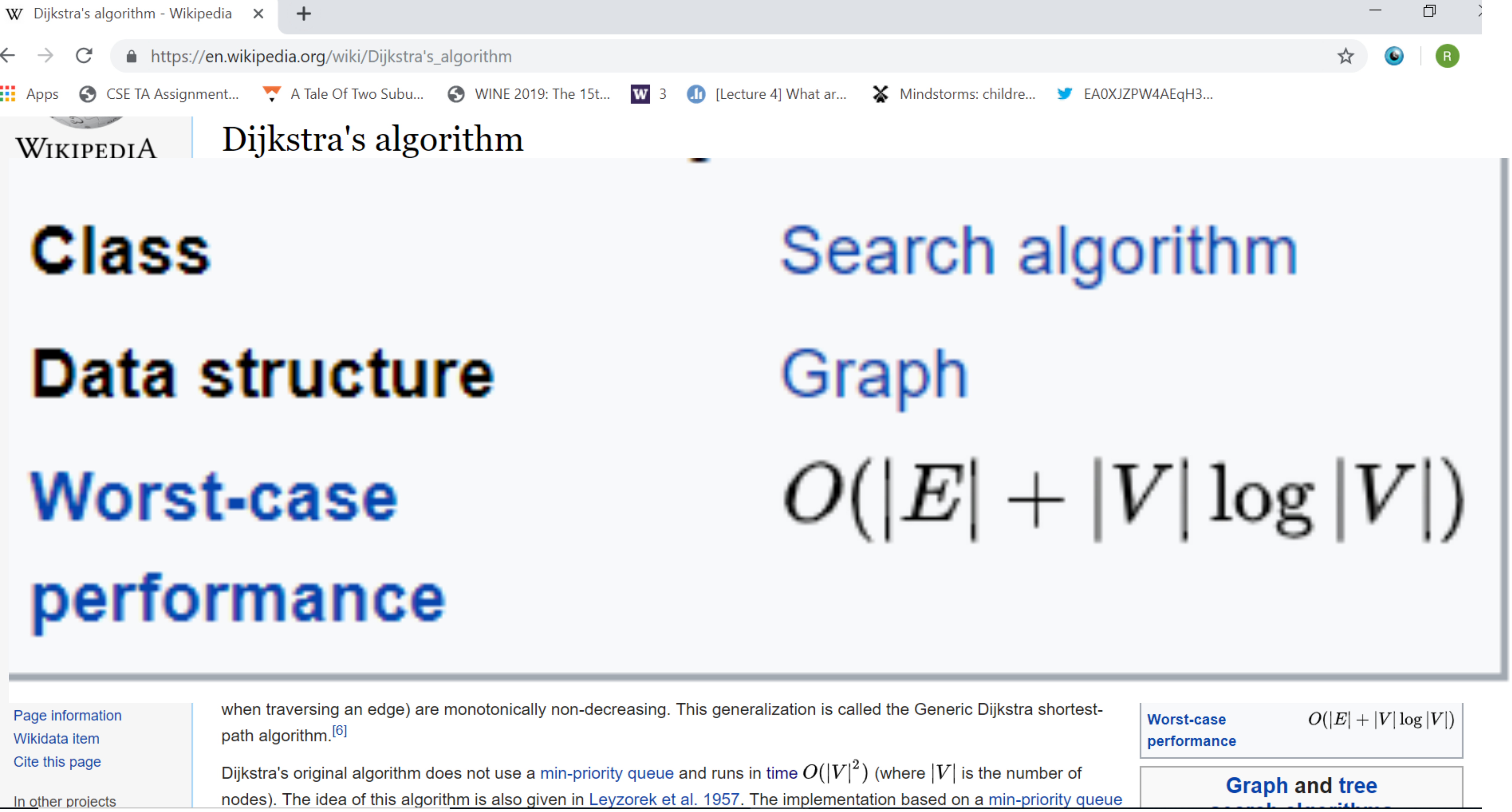
Tight O Bound = O(n log n + m log n)

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

Class	Search algorithm
Data structure	Graph
Worst-case performance	$O(\|E\| + \|V\| \log \|V\|)$

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!

Questions?

Review: Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your “fundamental” objects are

- Those become your vertices.

Then think about how they’re related

- Those become your edges.

Review: Some examples

For each of the following think about what you should choose for vertices and edges.

The internet

- **Vertices**: webpages. **Edges** from a to b if a has a hyperlink to b.

Family tree

- **Vertices**: people. **Edges**: from parent to child, maybe for marriages too?

Input data for the "6 Degrees of Kevin Bacon" game

- **Vertices**: actors. **Edges**: if two people appeared in the same movie
- Or: **Vertices** for actors and movies, **edge** from actors to movies they appeared in.

Course Prerequisites

- **Vertices**: courses. **Edge**: from a to b if a is a prereq for b.

Graph Modeling Activity

Note Passing - Part I

Imagine you are an American High School student. You have a very important note to pass to your crush, but the two of you do not share a class so you need to rely on a chain of friends to pass the note along for you. A note can only be passed from one student to another when they share a class, meaning when two students have the same teacher during the same class period.

Unfortunately, the school administration is not as romantic as you, and passing notes is against the rules. If a teacher sees a note, they will take it and destroy it. Figure out if there is a sequence of handoffs to enable you to get your note to your crush.

In your breakouts you'll discuss possible graph designs to help you solve this problem given the following student schedules.

	Period 1	**Period 2**	**Period 3**	**Period 4**
You	Smith	Patel	Lee	Brown
Anika	Smith	Lee	Martinez	Brown
Bao	Brown	Patel	Martinez	Smith
Carla	Martinez	Jones	Brown	Smith
Dan	Lee	Lee	Brown	Patel
Crush	Martinez	Brown	Smith	Patel

Break Outs!

Instructions

- Instructor will trigger breakout rooms
  - Detailed instructions on how breakouts work
- Accept the invite that pops up
- Turn on your mic and camera (if possible) ☺
- Work with your partners to answer the questions on Part I of the worksheet
  - how would you represent this scenario as a graph?
  - how would you implement this graph?
  - what graph algorithm would you use to find a route to your crush?
  - TAs will be coming in and out. Fill out this form to request a TA’s assistance: https://forms.gle/b9NiC1s11FKBcpm89
- Fill out the poll everywhere activity with your solution and upvote others
- Instructor will end the breakouts in ~5 minutes.

Possible Design

Algorithm

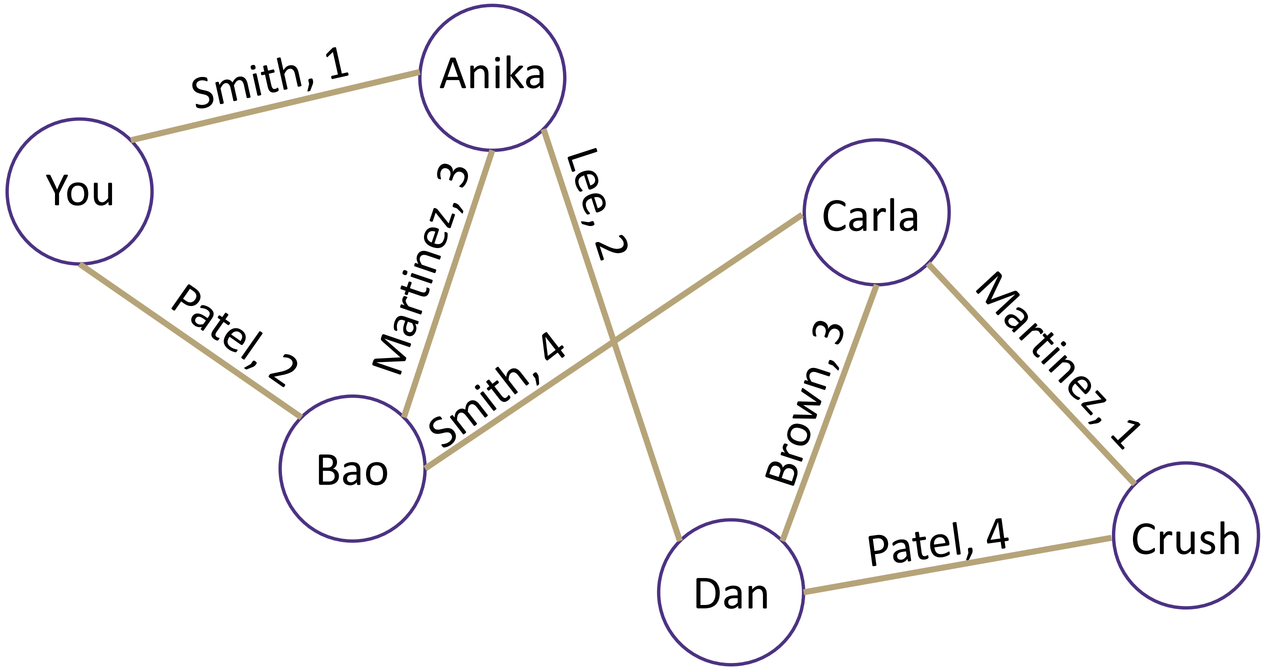
BFS or DFS to see if You and your Crush are connected

Vertices

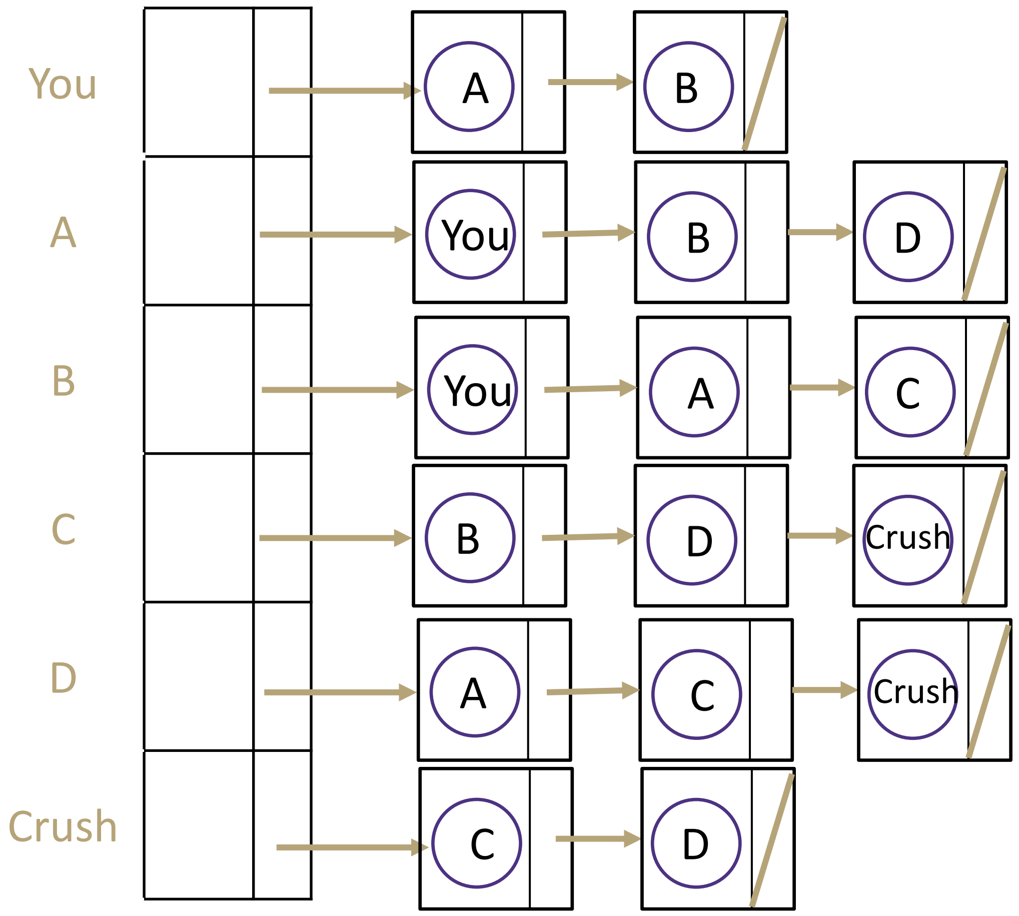
- Students
- Fields: Name, have note

Edges

- Classes shared by students
- Not directed
- Could be left without weights
- Fields: vertex 1, vertex 2, teacher, period



Adjacency List



More Design

Note Passing - Part II

Now that you know there exists a way to get your note to your crush, we can work on picking the best hand off path possible.

Thought Experiments:

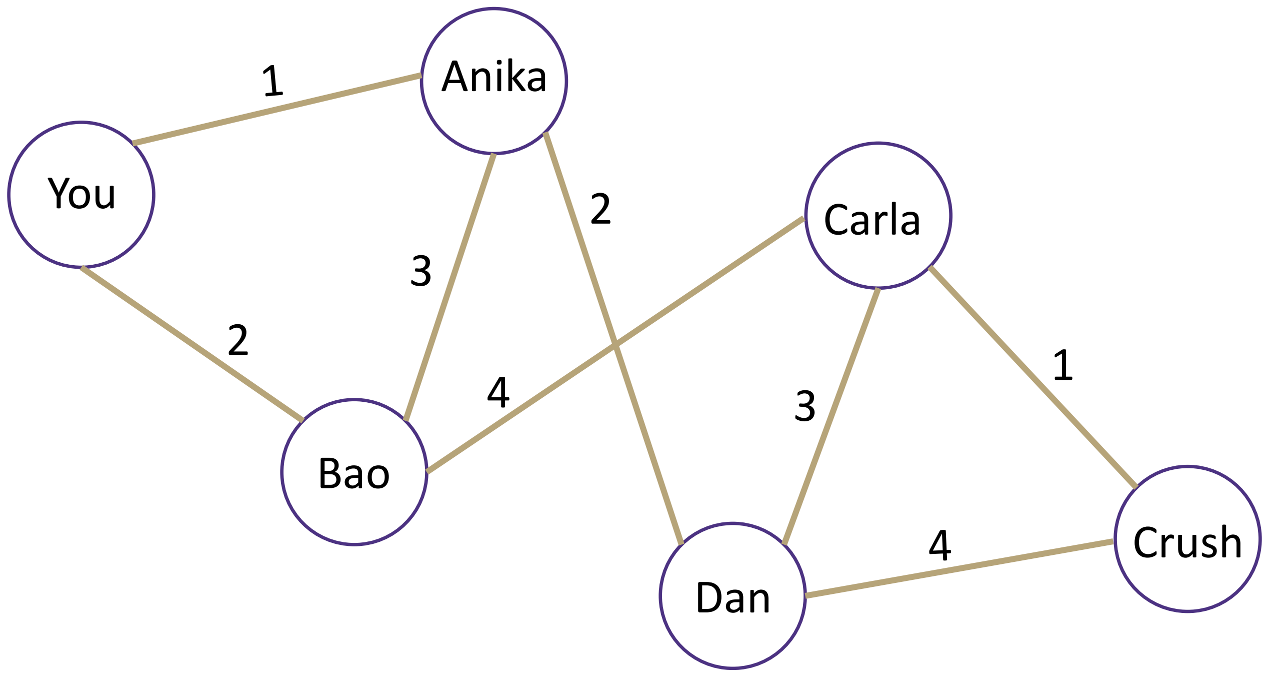
1. What if you want to optimize for time to get your crush the note as early in the day as possible?
    - How can we use our knowledge of which period students share to calculate for time knowing that period 1 is earliest in the day and period 4 is later in the day?
    - How can we account for the possibility that it might take more than a single school day to deliver the note?

2. What if you want to optimize for rick avoidance to make sure your note only gets passed in classes least likely for it to get intercepted?
    - Some teachers are better at intercepting notes than others. The more notes a teacher has intercepted, the more likely it is they will take yours and it will never get to your crush. If we knew how many notes each teacher has intercepted how might we incorporate that into our graph to find the least risky route?

Optimize for Time

"Distance" will represent the sum of which periods the note is passed in, because smaller period values are earlier in the day the smaller the sum the earlier the note gets there except in the case of a "wrap around"

1. Add the period number to each edge as its weight
2. Run Dijkstra's from You to Crush

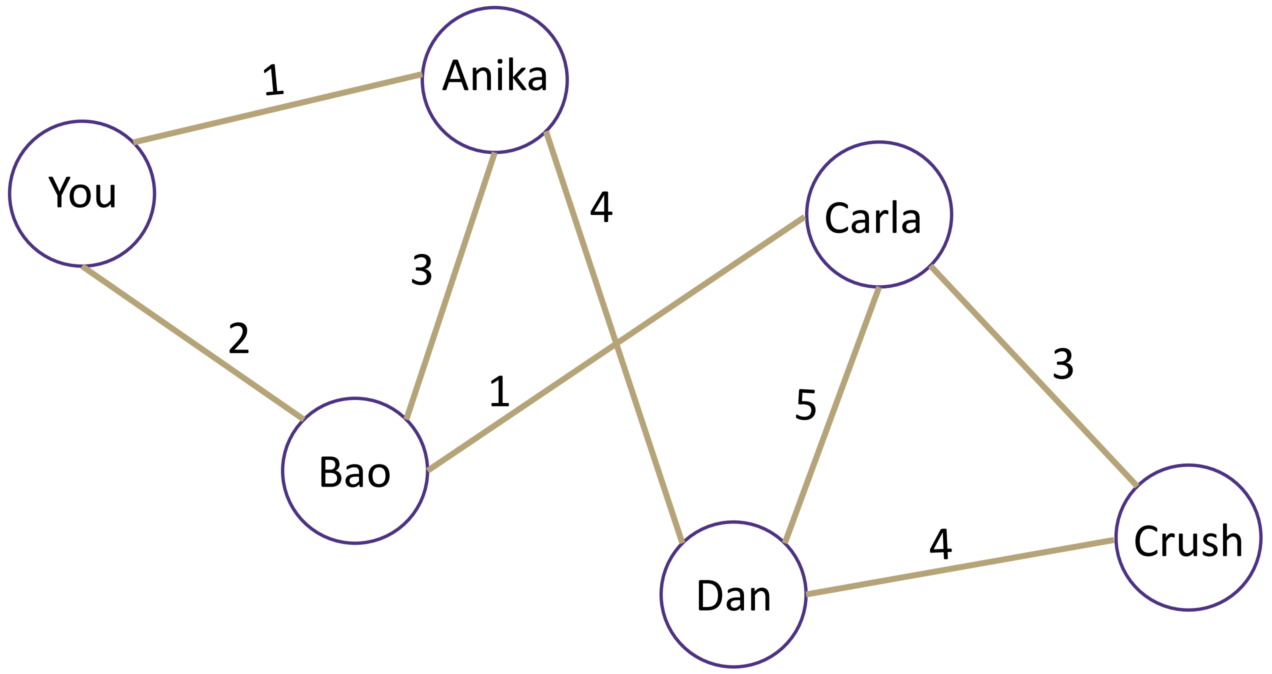


Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	2	You	5
Carla	6	Dan	3
Dan	3	Anika	2
Crush	7	Carla	4*

*The path found wraps around to a new school day because the path moves from a later period to an earlier one
- We can change our algorithm to check for wrap arounds and try other routes

Optimize for Risk

"Distance" will represent the sum of notes intercepted across the teachers in your passing route. The smaller the sum of notes the "safer" the path.



1. Add the number of letters intercepted by the teacher to each edge as its weight

2. Run Dijkstra's from You to Crush

Teacher	Notes Intercepted
Smith	1
Martinez	3
Lee	4
Brown	5
Patel	2

Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	4	Anika	2
Carla	5	Bao	3
Dan	10	Carla	5
Crush	8	Carla	4