Dijkstra’s algorithm

Initialization:
1. Assign each node an initial cost of $\infty$
2. Set our starting node’s cost to 0

Core loop:
1. Get the next (unvisited) node that has the smallest cost
2. Update all adjacent vertices (if applicable)
3. Mark current node as “visited”

Idea: Greedily pick node with smallest cost, then update everything possible. Repeat.

Dijkstra’s algorithm

Metaphor: Treat edges as canals and edge weights as distance. Imagine opening a dam at the starting node. How long does it take for the water to reach each vertex?

Caveat: Dijkstra’s algorithm only guaranteed to work for graphs with no negative edge weights.

Pronunciation: DYKE-struh (“dijk” rhymes with “bike”)

Dijkstra’s algorithm

Suppose we start at vertex “a”:

We initially assign all nodes a cost of infinity.

Next, assign the starting node a cost of 0.

And we’re done! Now, to find the shortest path, from a to a node, start at the end, trace the red arrows backwards, and reverse the list.
Dijkstra's algorithm

Suppose we start at vertex "a":

Next, update all adjacent node costs as well as the backpointers.

The pending node with the smallest cost is c, so we visit that next.

b is the next pending node with smallest cost.

We consider all adjacent nodes. a is fixed, so we only need to update e. Note the new cost of e is the sum of the weights for a − c and c − e.

We consider all adjacent nodes. a is fixed, so we only need to update e. Note the new cost of e is the sum of the weights for a − c and c − e.

Both d and f have the same cost, so let's (arbitrarily) pick d next. Note that we can’t adjust any of our neighbors.
Suppose we start at vertex "a":

Next up is f.

The only neighbor we is h.

h has the smallest cost now.

We update g.

The two adjacent nodes are f and e. f is fixed so we leave it alone. We however will update e: our current route is cheaper then the previous route, so we update both the cost and the backpointer.

And we're done! Now, to find the shortest path, from a to a node, start at the end, trace the red arrows backwards, and reverse the list.
Dijkstra's algorithm

Suppose we start at vertex "a":

The last pending node is e. We visit it, and check for any unfixed adjacent nodes (there are none).

And we're done! Now, to find the shortest path, from a to a node, start at the end, trace the red arrows backwards, and reverse the list.

Dijkstra's algorithm

Core idea in simplified pseudocode:

```python
def dijkstra(start):
    for v in vertices:
        set cost(v) to infinity
        set cost(start) to 0
    while (we still have unvisited nodes):
        current = get next smallest node
        for edge in current.getOutEdges():
            newCost = min(cost(current) + edge.cost, cost(edge.dest))
            update cost(edge.dest) to newCost, update backpointers, etc
    return backpointers dictionary
```

Dijkstra's algorithm

One implementation: inserting extra values into heap

```python
def dijkstra(start):
    backpointers = empty Dictionary of vertex to vertex
    costs = Dictionary of vertex to double, initialized to infinity
    visited = empty Set
    heap = new Heap<Node with cost>();
    for v in vertices:
        heap.put([v, infinity])
        costs.put(v, infinity)
    heap.decreasePriority([start, 0])
    costs.put(start, 0)
    while (heap is not empty):
        current, currentCost = heap.removeMin()
        skip if visited.contains(current), else visited.add(current)
        for edge in current.getOutEdges():
            newCost = currentCost + edge.cost
            if (newCost < cost.get(edge.dest)):
                cost.put(edge.dest, newCost)
                heap.insert([edge.dest, newCost])
                backpointers.put(edge.dest, current)
    return backpointers dictionary
```

Dijkstra's algorithm

Another impl: after implementing decreasePriority

```python
def dijkstra(start):
    backpointers = empty Dictionary of vertex to vertex
    costs = empty Dictionary of vertex to double
    heap = new Heap<Node with cost>();
    for v in vertices:
        heap.put([v, infinity])
        costs.put(v, infinity)
    heap.decreasePriority([start, 0])
    costs.put(start, 0)
    while (heap is not empty):
        current, currentCost = heap.removeMin()
        skip if heap.contains(current), else heap.removeMin()
        for edge in current.getOutEdges():
            newCost = currentCost + edge.cost
            if (newCost < cost.get(edge.dest)):
                cost.put(edge.dest, newCost)
                heap.decreaseKey([edge.dest, newCost])
                backpointers.put(edge.dest, current)
    return backpointers dictionary
```

Example

What does Dijkstra's algorithm do when run on vertex a?
Example

What does Dijkstra's algorithm do when run on vertex a?

Example

What does Dijkstra's algorithm do when run on vertex a?

Example

What does Dijkstra's algorithm do when run on vertex a?

Example

What does Dijkstra's algorithm do when run on vertex a?
What does Dijkstra's algorithm do when run on vertex a?

Example

What does Dijkstra's algorithm do when run on vertex a?

Example

What does Dijkstra's algorithm do when run on vertex a?
Misc announcements

- Project 1, part 2 regrades will be released later tonight
- Project 3, part 1 grades also released later tonight
  Reminder: if you fix the errors in your Friday submission, you can get up to half credit back.
- If you’ve emailed me, and you haven’t heard back, email me again

Dijkstra’s: why does it work?

Rough intuition:
- Suppose $a$ is the next unvisited node with the smallest cost. Suppose $b$ is some unvisited vertex adjacent to $a$.
- The quickest path from the start to $b$ is going to be through $a$. Any other route would be a longer detour (assuming edges are positive!).
- So, picking the shortest node will always accurately update the adjacent nodes.

(Full proof beyond scope of class)

Dijkstra’s: negative edges

What if we have negative edges?

**Question:** What’s the shortest path from $s$ to $t$ according to Dijkstra’s? In reality?

![Diagram showing a graph with negative edges]

Punchline:
- If there are negative edges, Dijkstra’s doesn’t work
  (There exist other algorithms that can handle negative edges – e.g. see Bellman-Ford.)
- If there are negative cycles, nothing works

(Where do negative edges show up? Examples: modeling credit and debit, modeling flow of energy, etc.)

Dijkstra’s algorithm: analyzing runtime

**Question:** what is the worst-case runtime of Dijkstra’s algorithm?

**Strategy 1:** Analyze the code, like we’ve been doing all quarter
**Strategy 2:** Analyze the algorithm more holistically, like we did for DFS and BFS
Consider this (simplified) pseudocode. How do we analyze?

```python
def dijkstra(start):
    for v in vertices:
        set cost(v) to infinity
        set cost(start) to 0
    while (we still have unvisited nodes):
        current = get next smallest node
        for edge in current.getOutEdges():
            newCost = min(cost(current) + edge.cost, cost(edge.dest))
            update cost(edge.dest) to newCost, update backpointers, etc
    return backpointers dictionary
```

(Note: let $t_s$ be the time needed to get the next smallest node, and let $t_u$ be the time needed to update vertex costs. We’ll treat these as unknowns for now.)

**Things we know:**
- Initialization takes $O(|V|)$ time
- The while loop repeats $|V|$ times
- The inner foreach loop repeats $|E|$ times (???)
- The inner foreach loop does $O(t_u)$ work per iteration
- So while loop does $O(t_s + |E| \cdot t_u)$ work per iteration

**Final runtime:**

$$O(|V| + |V| \cdot t_s + |V| \cdot |E| \cdot t_u)$$

Distribute:

$$O(|V| \cdot t_s + |V| \cdot |E| \cdot t_u)$$

The lone $|V|$ is dominated by $|V| \cdot t_s$:

$$O(|V| \cdot t_s + |V| \cdot |E| \cdot t_u)$$

**Observations about the foreach loop:**
- We don’t know how many times it runs per each iteration
- …but we do know num times it runs across all iterations!

**Original bound:**

$$O(|V| \cdot t_s + |V| \cdot |E| \cdot t_u)$$

We update at most once per edge — so, a tighter bound:

$$O(|V| \cdot t_s + |V| \cdot |E| \cdot t_u)$$

**Question:**
Do we really need to update vertex costs $|V| \cdot |E|$ times?

**While (we still have unvisited nodes):**
- current = get next smallest node
- for edge in current.getOutEdges():
  - newCost = min(cost(current) + edge.cost, cost(edge.dest))
  - update cost(edge.dest) to newCost, update backpointers, etc

**Observation:**
there are two operations we care about: finding the node with the min cost, and given a node, updating its cost

**Ideas:**
- Use a binary heaps: lets us find a node with min cost easily
- Use a dictionary: lets us update the value corresponding to a node easily
Dijkstra’s algorithm: finding and updating nodes

Exercise: fill out this table

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Remove min ($t_s$)</th>
<th>Update cost ($t_u$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash map</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>AVL tree</td>
<td>$O(\log(</td>
<td>V</td>
</tr>
<tr>
<td>Binary heap</td>
<td>$O(\log(</td>
<td>V</td>
</tr>
</tbody>
</table>

The AVL version looks actually pretty reasonable

Another common approach: modify binary heaps so they can update the cost in $O(\log(n))$ time (a “hybrid” binary heap):

- Two fields: the same heap internal array, and a hash table mapping vertices to their index in the array.
- Assumptions: each vertex is unique; we only decrease the cost
- Implementing **removeMin**:
  - Run the standard removeMin heap algorithm. As we swap nodes, add some extra code to keep the hash map up-to-date.
  - This is still $O(\log(|V|))$.
- Implementing **updateCost**:
  - Use the hash map to get the index of the given node. Run percolateUp, updating the hash map as we go.
  - This is still $O(\log(|V|))$.

Observation: Gosh, this all sounds exhausting

What if we replace the binary heap’s call to **updateCost** with **insert** and just allow duplicates?

Runtime is now $O((|V| + |E|) \log(|V| + |E|))$ – the analysis is left as an exercise to the reader.

So, less efficient, but easiest to implement.

Note: Fibonacci heaps are beyond the scope of this class