CSE 373: More on Dijkstra's algorithm

Michael Lee
Wednesday, Feb 21, 2018

Dijkstra's algorithm

Initialization:

1. Assign each node an initial cost of ∞
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

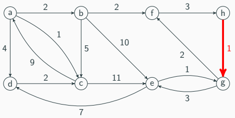
Metaphor: Treat edges as canals and edge weights as distance. Imagine opening a dam at the starting node. How long does it take for the water to reach each vertex?

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

Pronunciation: DYKE-struh ("dijk" rhymes with "bike")

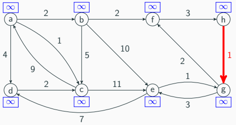
Dijkstra's algorithm

Suppose we start at vertex "a":



Dijkstra's algorithm

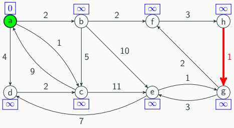
Suppose we start at vertex "a":



We initially assign all nodes a cost of infinity.

Dijkstra's algorithm

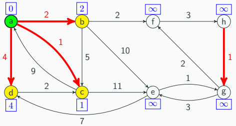
Suppose we start at vertex "a":



Next, assign the starting node a cost of 0.

Dijkstra's algorithm

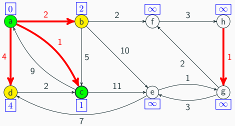
Suppose we start at vertex "a":



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

Dijkstra's algorithm

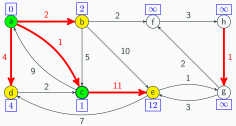
Suppose we start at vertex "a":



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

Dijkstra's algorithm

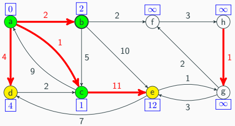
Suppose we start at vertex "a":



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 $c - e$.

Dijkstra's algorithm

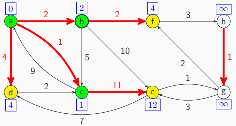
Suppose we start at vertex "a":



b is the next pending node with smallest cost.

Dijkstra's algorithm

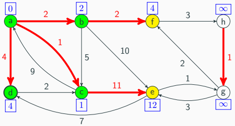
Suppose we start at vertex "a":



The adjacent nodes are c, e, and f. The only node where we can update the cost is f. Note the route $a - b - e$ has the same cost as $a - c - e$, so there's no point in updating the backpointer to e.

Dijkstra's algorithm

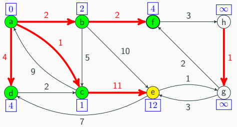
Suppose we start at vertex "a":



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.

Dijkstra's algorithm

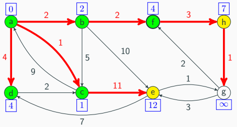
Suppose we start at vertex "a":



Next up is f.

Dijkstra's algorithm

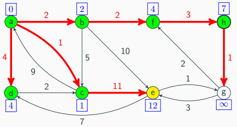
Suppose we start at vertex "a":



The only neighbor we is h.

Dijkstra's algorithm

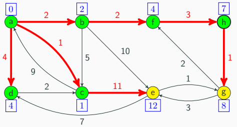
Suppose we start at vertex "a":



h has the smallest cost now.

Dijkstra's algorithm

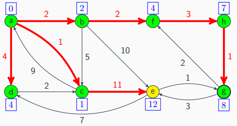
Suppose we start at vertex "a":



We update g.

Dijkstra's algorithm

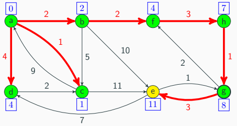
Suppose we start at vertex "a":



Next up is g.

Dijkstra's algorithm

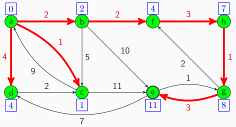
Suppose we start at vertex "a":



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.

Dijkstra's algorithm

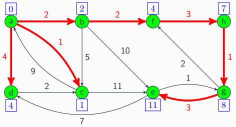
Suppose we start at vertex "a":



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

Dijkstra's algorithm

Suppose we start at vertex "a":



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:

```
def dijkstra(start):
    for (v : vertices):
        set cost(v) to infinity
    set cost(start) to 0

    while (we still have unvisited nodes):
        current = get next smallest node

        for (edge : 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

```
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>();
    heap.put([start, 0])
    cost.put(start, 0)
    while (heap is not empty):
        current, currentCost = heap.removeMin()
        skip if visited.contains(current), else visited.add(current)

        for (edge : current.getOutEdges()):
            skip if visited.contains(edge.dest), else visited.add(edge.dest)

            if (newCost < cost.get(edge.dest)):
                cost.put(edge.dest, newCost)
                heap.insert([edge.dest, newCost])
                backpointers.put(edge.dest, current)

    return backpointers dictionary
```

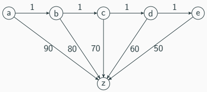

Dijkstra's algorithm

Another impl: after implementing decreasePriority

```
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 : 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()

        for (edge : 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
```

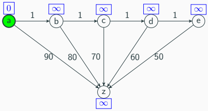

Example

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



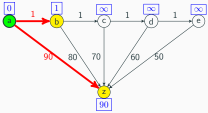
Example

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



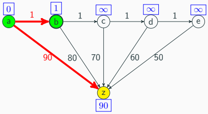
Example

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



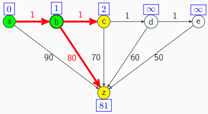
Example

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



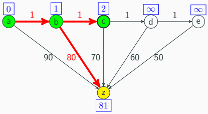
Example

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



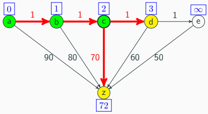
Example

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



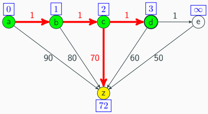
Example

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



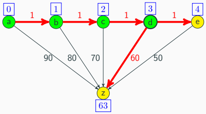
Example

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



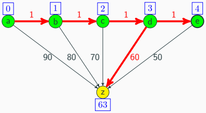
Example

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



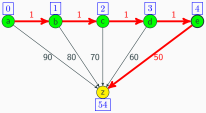
Example

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



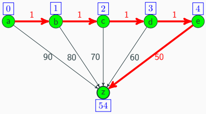
Example

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



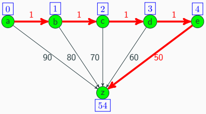
Example

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



Example

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



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:

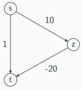
- Suppose a is the next unvisited node with the smallest cost. Suppose b is some unvisited vertex adjacent to a.
- The quickest path from the start to b is going to be through a. Any other route would be a longer detour (assuming edges are positive!).
- So, picking the shortest node will always accurately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

(Full proof beyond scope of class)

Dijkstra's: negative edges

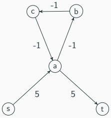
What if we have negative edges?

Question: What's the shortest path from s to t according to Dijkstra's? In reality?



Dijkstra's: negative edges

What's the shortest path now?



Dijkstra's: 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

Dijkstra's algorithm: analyzing runtime via code

Consider this (simplified) pseudocode. How do we analyze?

```
def dijkstra(start):
    for (v : vertices):
        set cost(v) to infinity
    set cost(start) to 0

    while (we still have unvisited nodes):
        current = get next smallest node

        for (edge : 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.)

Dijkstra's algorithm: analyzing runtime via code

Things we know:

- Initialization takes $\mathcal{O}(|V|)$ time
- The while loop repeats $|V|$ times
- The inner foreach loop repeats $|E|$ times (???)?
- The inner foreach loop does $\mathcal{O}(t_u)$ work per eiteration
- So while loop does $\mathcal{O}(t_s + |E| \cdot t_u)$ work per iteration

Final runtime:

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

Distribute:

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

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

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

Dijkstra's algorithm: analyzing runtime

Our runtime:

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

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 : current.getOutEdges()):
        newCost = min(cost(current) + edge.cost, cost(edge.dest))
        update cost(edge.dest) to newCost, update backpointers, etc
```


Dijkstra's algorithm: analyzing runtime

```
while (we still have unvisited nodes):
    current = get next smallest node

    for (edge : current.getOutEdges()):
        newCost = min(cost(current) + edge.cost, cost(edge.dest))
        update cost(edge.dest) to newCost, update backpointers, etc
```

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:

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

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

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

Dijkstra's algorithm: finding and updating nodes

Our runtime so far:

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

Question: So, what exactly is t_s and t_u?

Answer: Depends on how we store nodes and costs!

Dijkstra's algorithm: finding and updating nodes

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

Data structure	Remove min (t_s)	Update cost (t_u)
Hash map	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(\lvert 1\rvert)$
Sorted array	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert)$
AVL tree	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(\log(\lvert V\rvert))$
Binary heap	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(\lvert V\rvert)$

The AVL version looks actually pretty reasonable

Dijkstra's algorithm: finding and updating nodes

Another common approach: modify binary heaps so they can update the cost in $\mathcal{O}(\log(n))$ 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 $\mathcal{O}(\log(n))$.
- 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 $\mathcal{O}(\log(n))$.

Dijkstra's algorithm: finding and updating nodes

Data structure	removeMin (t_s)	updateCost (t_u)
Hash map	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(\lvert 1\rvert)$
Sorted array	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert)$
AVL tree	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(\log(\lvert V\rvert))$
Binary heap	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(\lvert V\rvert)$
"Hybrid" binary heap	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(\log(\lvert V\rvert))$
Fibonacci heaps	$\mathcal{O}(\log(\lvert V\rvert))$	$\mathcal{O}(1)$

Note: Fibonacci heaps are beyond the scope of this class

Dijkstra's algorithm: finding and updating nodes

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 $\mathcal{O}((\lvert V\rvert + \lvert E\rvert)\log(\lvert V\rvert + \lvert E\rvert))$ – the analysis is left as an exercise to the reader.

So, less efficient, but easiest to implement.