CSE 373: Graph traversal

Michael Lee

Friday, Feb 16, 2018

Warmup

Warmup

Given a graph, assign each node one of two colors such that no two adjacent vertices have the same color. (If it's impossible to color the graph this way, your algorithm should say so).

Solution: This algorithm is known as the 2-color algorithm. We can solve it by using any graph traversal algorithm, and alternating colors as we go from node to node.

Goal: How do we traverse graphs?

Today's goal: how do we traverse graphs?

Idea 1: Just get a list of the vertices and loop over them

Problem: What if we want to traverse graphs following the edges?

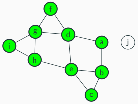
For example, can we...

- Traverse a graph to find if there's a connection from one node to another?
- Determine if we can start from our node and touch every other node?
- Find the shortest path between two nodes?

Solution: Use graph traversal algorithms like breadth-first search and depth-first search

Breadth-first search (BFS) example

```
search(v):
    visited = empty set

    queue.enqueue(v)
    visited.add(v)

    while (queue is not empty):
        curr = queue.dequeue()

        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(curr)
```



Current node: a b d c e f g h i

Queue: a, b, d, c, e, f, g, h, i,

Visited: a, b, d, c, e, f, g, h, i,

Breadth-first search (BFS)

Breadth-first traversal, core idea:

1. Use something (e.g. a queue) to keep track of every vertex to visit
2. Add and remove nodes from queue until it's empty
3. Use a set to store nodes we don't want to recheck/revisit
4. Runtime:
    - We visit each node once.
    - For each node, check each edge to see if we should add to queue
    - So we check each edge at most twice

    So, $\mathcal{O}(|V| + 2|E|)$, which simplifies to $\mathcal{O}(|V| + |E|)$.

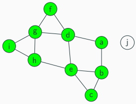
Breadth-first search (BFS)

Pseudocode:

```
search(v):
    visited = empty set

    queue.enqueue(v)
    visited.add(v)

    while (queue is not empty):
        curr = queue.dequeue()

        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(curr)
```

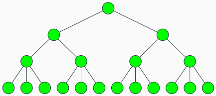
An interesting property...

Note: We visited the nodes in “rings” – maintained a gradually growing “frontier” of nodes.



An interesting property...

What does this look like for trees?



The algorithm traverses the width, or “breadth” of the tree

Depth-first search (DFS)

Question: Why a queue? Can we use other data structures?

Answer: Yes! Any kind of list-like thing that supports appends and removes works! For example, what if we try using a stack?

The BFS algorithm:

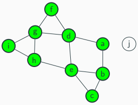
```
search(v):
    visited = empty set

    queue.enqueue(v)
    visited.add(v)

    while (queue is not empty):
        curr = queue.dequeue()

        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(curr)
```

The DFS algorithm:

```
search(v):
    visited = empty set
    stack.push(v)
    visited.add(v)

    while (stack is not empty):
        curr = stack.pop()
        visited.add(curr)

        for (w : v.neighbors()):
            if (w not in visited):
                stack.push(w)
                visited.add(v)
```


Depth-first search (DFS) example

```
search(v):
    visited = empty set
    stack.push(v)

    while (stack is not empty):
        curr = stack.pop()
        visited.add(curr)

        for (w : v.neighbors()):
            if (w not in visited):
                stack.push(w)
```



Current node: adgihfecb

Stack: a, b, d, e, f, g, h, i, c,

Visited: a, b, d, e, f, g, h, i, e, c,

Depth-first search (DFS)

Depth-first traversal, core idea:

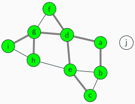
1. Instead of using a queue, use a stack. Otherwise, keep everything the same.
2. Runtime: also $\mathcal{O}(|V| + |E|)$ for same reasons as BFS

Pseudocode:

```
search(v):
    visited = empty set

    stack.push(v)
    visited.add(v)

    while (stack is not empty):
        curr = stack.pop()

        for (w : v.neighbors()):
            if (w not in visited):
                stack.push(w)
                visited.add(curr)
```

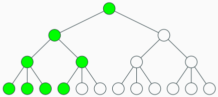

An interesting property...

Note: Rather the growing the node in “rings”, we randomly wandered through the graph until we got stuck, then “backtracked”.



An interesting property...

What does this look like for trees?



The algorithm traverses to the bottom first: it prioritizes the "depth" of the tree

Note: rest of algorithm omitted

Compare and contrast

Question: When do we use BFS vs DFS?

Related question: How much memory does BFS and DFS use in the worst case?

- BFS: $\mathcal{O}(|V|)$ – what if every node is connected to the start?
- DFS: $\mathcal{O}(|V|)$ – what if the nodes are arranged like a linked list?

So, in the worst case, BFS and DFS both have the same worst-case runtime and memory usage.

They only differ in what order they visit the nodes.

Compare and contrast

How much memory does BFS and DFS use in the **average** case?

Related question: how much memory do they use when we want to traverse a tree?

- BFS: $\mathcal{O}(\text{"width" of tree}) = \mathcal{O}(\text{num leaves})$
- DFS: $\mathcal{O}(\text{height})$

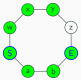
For graphs:

- Use BFS if graph is "narrow", or if solution is "near" start
- Use DFS if graph is "wide"

In practice, graphs are often large/very wide, so DFS is often a good default choice. (It's also possible to implement DFS recursively!)

Design challenge

Question: How would you modify BFS to find the shortest path between every node?



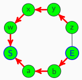
Observation: Since BFS moves out in rings, we will reach the end node via the path of length 3 first.

Idea: when we enqueue, store where we came from in some way. (e.g. mark node, use a dictionary...)

After BFS is done, *backtrack*.

Design challenge: pathfinding

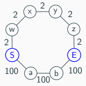
Question: How would you modify BFS to find the shortest path between every node?



Now, start from any node, follow arrows, then reverse to get path.

Design challenge: pathfinding

Question: What if the edges have weights?



Weighted graph

A **weighted graph** is a kind of graph where each edge has a numerical "weight" associated with it.

This number can represent anything, but is often (but not always!) used to indicate the "cost" of traveling down that edge.

Pathfinding and DFS

We can use BFS to correctly find the shortest path between two nodes in an unweighted graph...

...but it fails if the graph is weighted!

We need a better algorithm.

Today: **Dijkstra's algorithm**

Dijkstra's algorithm

Core idea:

1. Assign each node an initial cost of ∞
2. Set our starting node's cost to 0
3. Update all adjacent vertices costs to the minimum known cost
4. Mark the current node as being "done"
5. Pick the next unvisited node with the minimum cost. Go to step 3.

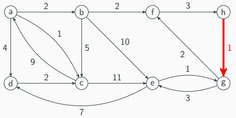
Metaphor: Treat edges as canals and edge weights as distance. Imagine opening a dam at the starting node. How long does it take for the water to reach each vertex?

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

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

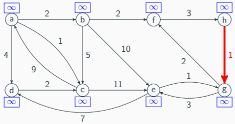
Dijkstra's algorithm

Suppose we start at vertex "a":



Dijkstra's algorithm

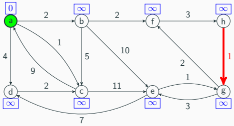
Suppose we start at vertex "a":



We initially assign all nodes a cost of infinity.

Dijkstra's algorithm

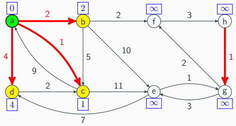
Suppose we start at vertex "a":



Next, assign the starting node a cost of 0.

Dijkstra's algorithm

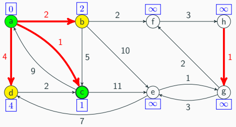
Suppose we start at vertex "a":



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

Dijkstra's algorithm

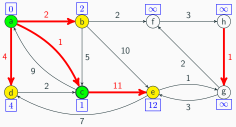
Suppose we start at vertex "a":



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

Dijkstra's algorithm

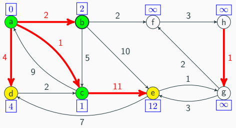
Suppose we start at vertex "a":



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

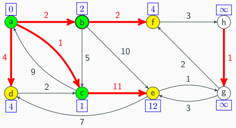
Suppose we start at vertex "a":



b is the next pending node with smallest cost.

Dijkstra's algorithm

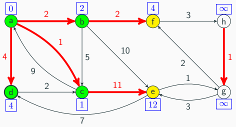
Suppose we start at vertex "a":



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

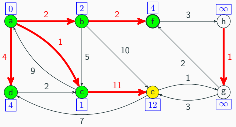
Suppose we start at vertex "a":



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

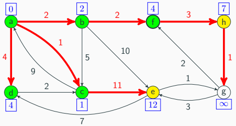
Suppose we start at vertex "a":



Next up is f.

Dijkstra's algorithm

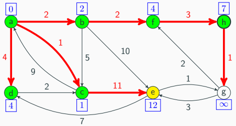
Suppose we start at vertex “a”:



The only neighbor we is h.

Dijkstra's algorithm

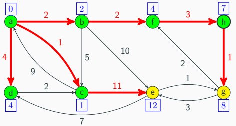
Suppose we start at vertex “a”:



h has the smallest cost now.

Dijkstra's algorithm

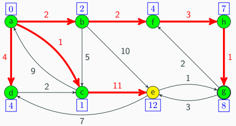
Suppose we start at vertex “a”:



We update g.

Dijkstra's algorithm

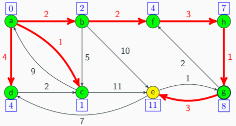
Suppose we start at vertex “a”:



Next up is g.

Dijkstra's algorithm

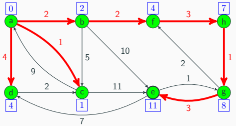
Suppose we start at vertex “a”:



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

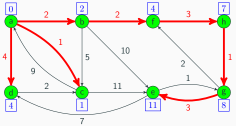
Suppose we start at vertex “a”:



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

Some implementation details...

- How do we keep track of the node costs?
  - Could use a dictionary
  - Could manually mark each node
- How do we find the node with the smallest cost?
  - Could maintain a sorted list
  - Could use a heap!
- If we're using a heap, how do we update node costs?
  - Could add a `changeKeyPriority(...)` method to heap
  - Alternatively, add the node and the cost to the heap again (and ignore duplicates)

Dijkstra's algorithm

The pseudocode

```
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)

            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)

    use backpointers dictionary to get path
```