CS 3510 — Algorithms, Spring 2022

Lecture 14: BFS & Dijkstra

March 10, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

1 BFS

BFS is the same as DFS, but it is based on a different data structure. Rather than using a stack, we use a queue (FIFO: first in first out).

It is primarily used for finding shortest paths as it will explore everything closest to it (root, then all children with distance 1, then all children with distance 2, etc).

An implementation of BFS could look like:

```
1  Function DFS (G = (V, E)) :
2      q ← Queue
3      dist ← {} // map of <vertex, distance>
4      enqueue(q, root)
5      while q is not empty do
6          v ← dequeue(q) for v, w ∈ E do
7              if ¬ (w ∈ dist) then
8                  enqueue(q, w)
9                  dist[w] ← dist[v] + 1
10     return dist
```

The runtime of this algorithm is $\mathcal{O}(|V| + |E|)$ as we will only explore each edge once, hence it is linear in runtime.

Issue: What if the nodes had weights? If a real example, two cities (nodes) could be connected by a highway (edge), but the length of the highway could vary (weight on edge). How do we consider that?

Solution. Modify the graph where each edge of weight k becomes k different nodes weight weight 1. Hence, we can use BFS on this modified graph to find distances. However, this has runtime challenges as the runtime depends on the weight (since we are adding new edges).

2 Dijkstra

Dijkstra's algorithm is an algorithm that keeps track of distances between nodes by using a priority queue rather than a standard queue. During the execution of Dijkstra, we keep track of the best distance to each node we have seen so far and the next node to explore comes from the priority queue (to evaluate distances).

We have two things to keep track of: (1) Distances thus far and (2) vertices that have not been finished yet (fully explored). Here is an implementation:

```
1  Function Dijkstra(G = (V, E), weights, root):
       /* set up priority queue with distance ∞ for all vertices      */
2      seen ← PriorityQueue(V)
       /* distance map to be returned                                 */
3      dist ← {}
4      DecKey(seen, root, 0)
5      while seen is not empty do
6          curr ← PopMin(seen)
7          for (curr, w) ∈ E do
8              alt ← dist[curr] + weight[curr, w]
9              if alt < dist[w] then
10                 dist[w] ← alt
11                 DecKey(seen, w, alt)
12     return dist
```

In the above, we call `PopMin` $\mathcal{O}(|V|)$ and call `DecKey` $\mathcal{O}(|E|)$ times. Further, the running time of this algorithm is dependent on this fancy data structure that we used (the min heap). With this current data structure, it takes $\mathcal{O}(\log n)$ to pop the minimum and $\mathcal{O}(\log n)$ to decrement the key. Hence, this implementation has a runtime of $\mathcal{O}\left(|V|\log|V| + |E|\log|V|\right) = \mathcal{O}\left((|V| + |E|)\log|V|\right)$.

2.1 So, what is a min heap?

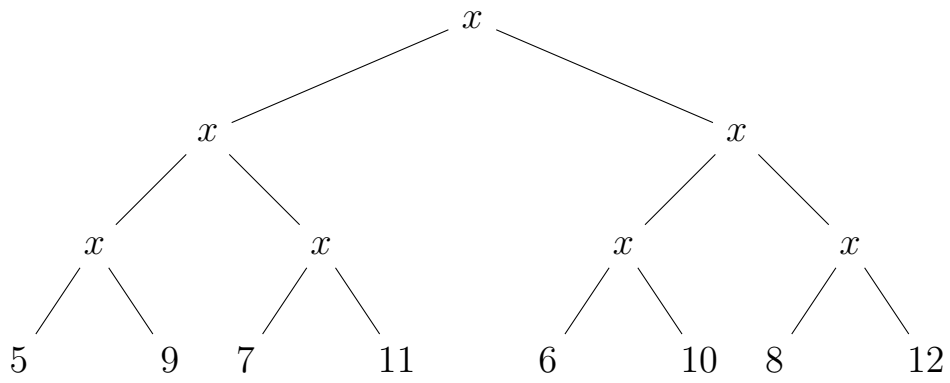
A minimum heap is a binary tree with two invariants:

1. Each node is smaller than its children
2. For each node, either the left and right subtrees are equal size or the left subtree is greater than the right subtree by only 1.

So, how do we add to a minimum heap? Take the scenario where we already start with a minimum heap given by the xs below and we are trying to add elements $5, 6, \ldots, 11, 12$. We are concerned with placing these items into the heap, but not too much on maintaining the first invariant, yet.

We add them in a manner such that the second invariant is satisfied which states that the left subtree is not longer than the right subtree, but if it is: it's not greater than it by more than 1.

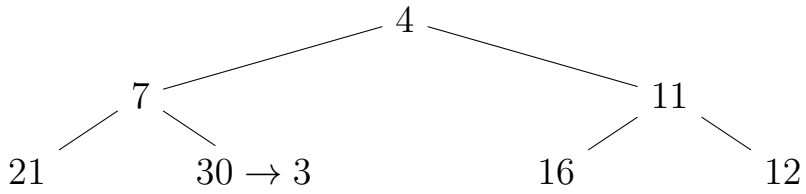
Hence, we add 5 (the first element) to the left-most position in the tree. Afterwards, we add 6 to the left-most portion of the right subtree (to maintain the invariant). We repeat this procedure for the rest of the elements shown below.



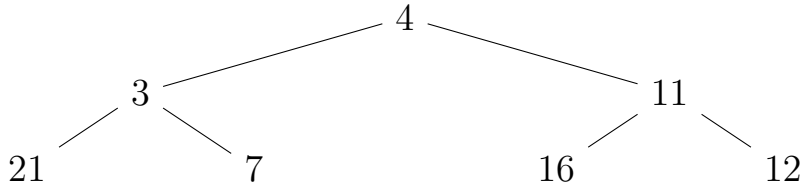
As said, this is only half of the add procedure. Next, we need to decide where this node should go? It may not satisfy the first invariant (the nodes have to be smaller than the children). So, how do we perform `DecKey`?

We essentially compare the node with the parent of that node and swap if the node is smaller and we repeat until there is no swap to be done.

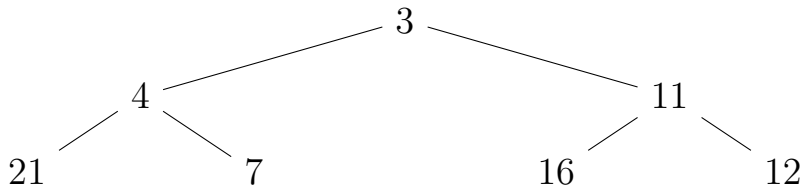
In the below, we change 30 to 3 which immediately invalidates the heap invariant because 7 should be smaller than both its children, but it is not.



Hence, we perform repeated swaps. First, swapping 7 and 3.



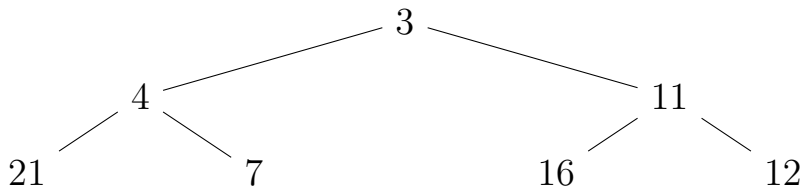
Now, swap 3 and 4 to finish:



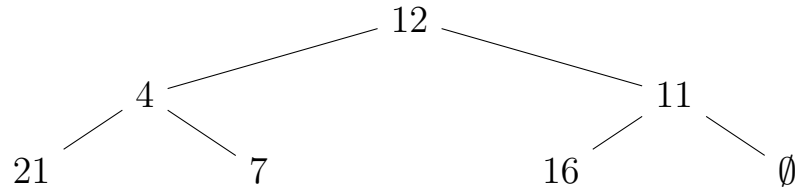
Since this could propogate all the way to the root and this is a binary tree, there are a total of $\log n$ swaps that could be needed making `DecKey` runtime $\mathcal{O}(\log n)$.

Finally, for `PopMin`, we know that the item that we want is at the top so we can simply take that item, but what do we put there instead? We can simply take the last element added (bottom most, right most element) and put it in the root and swap it down until no swaps are mode (similar to `DecKey`).

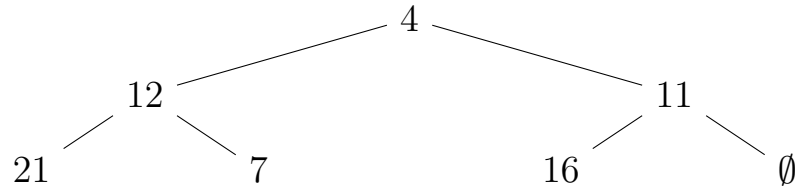
Remark: If the element is greater than both its children, swap with the smallest of the two. Now, let's see an example of this in practice.



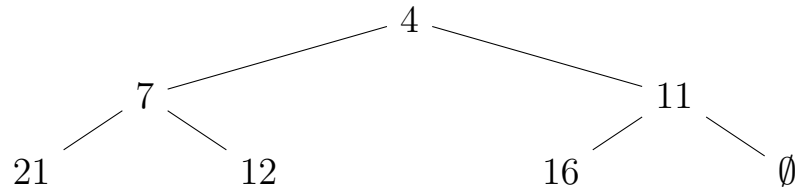
If we choose to pop the minimum, we will remove 3 and put 12 in its place.



Now, since 12 is greater than 4 and 11, we swap it with 4.



Since, 12 is greater than 7, we swap them.



Hence, we have achieved the heap invariant again. Note that there is no element in the bottom right of the tree and $\emptyset$ is only used as a placeholder for formatting reasons. In the same manner as before, the runtime of this $\mathcal{O}(\log n)$.

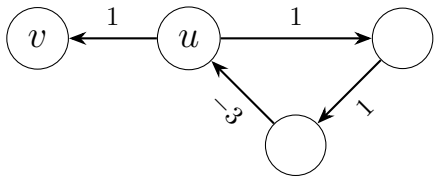
3 Challenges

1. What if we used an unsorted array instead of using a minheap? How does this change your algorithm for Dijkstra?

   Solution. Since we are using an unsorted array, finding and popping the minimum takes $\mathcal{O}(|V|)$ time, but `DecKey` takes $\mathcal{O}(1)$ time because we can simply update values in the array without needing to adjust the array.

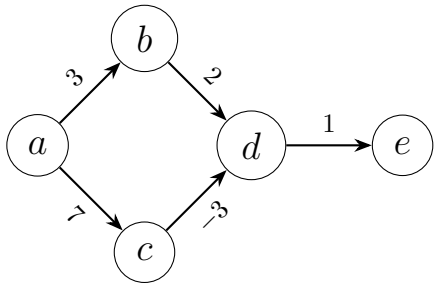
   This changes the runtime from $\mathcal{O}\left((|V| + |E|) \log |V|\right)$ to $\mathcal{O}(|V|^2)$. When is this better? If our graph has a lot of edges, then $\mathcal{O}(|V|^2)$ is better meaning using an unsorted array would be more efficient. This is because if we have a lot of edges $|E| \to |V|^2$ hence the min heap runtime would be $\mathcal{O}\left(|V|^2 \log |V|\right)$ which is a factor of $\log |V|$ worse than using an unsorted array.

2. What is the mimimum path from u to v in the following graph?



   Solution. The minimum path cannot be determined as we have a negative cycle as we can traverse from u to u in a cycle with -1 cost, meaning for every path $u \to v$, there is another path with 1 less cost. Hence, we can say that $u \to v$ has a minimum path of $-\infty$.

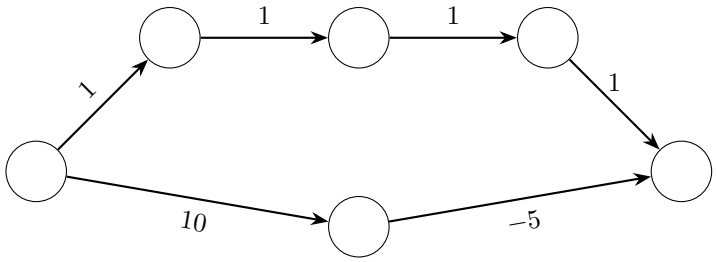
3. Why does Dijkstra fail on this example?



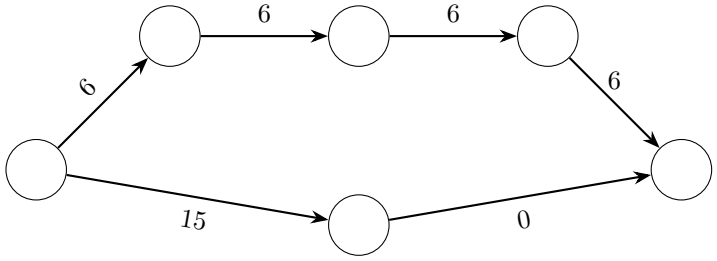
Solution. Dijkstra fails on this example because it will compute the minimum distance for a, then b, then d, then e. From there, it will pull c out of the priority queue and compute distance to c, then d again, but it won't propogate this to e. Hence, Dijkstra will say the distance to e is 6 when it should be 5. This is an example of Dijkstra failing with negative edges.

4. In general, can we fix Dijkstra on graphs with negative edges by adding some fixed value to each edge such that each edge is positive?

Solution. Disproved by counterexample, imagine the following graph:



If we added some number such that the minimum edge weight was 0, the graph would become:



The top route (which previously took 4) now takes 24 while the bottom unit (which previously took 5) takes $15 + 0 = 15$, so Dijkstra would suggest to take the bottom route as the minimal route when the top route is the one that should be taken.