

1 Priority Queue Implementations

The implementation of Dijkstra's algorithm is very much dependent on the priority queue that we use. That is, depending on how we implement our priority queue, we may or may not see a change in performance. So, we will go through some priority queue implementations.

1.1 Unsorted List

Store n elements in an unsorted list.

Operation	Runtime	Explanation
Insert	$\mathcal{O}(1)$	Add it at the end of the list.
DecreaseKey	$\mathcal{O}(1)$	Assuming the operation comes with a pointer to where the element is stored in your data structure, take that element and decrease its key value.
DecreaseMin	$\mathcal{O}(n)$	We need to potentially scan the entire list.

For Dijkstra, we would have $\mathcal{O}(|V|^2 + |E|)$.

1.2 Binary Heap

Store elements in a balanced binary tree with each element having smaller key value than its children.

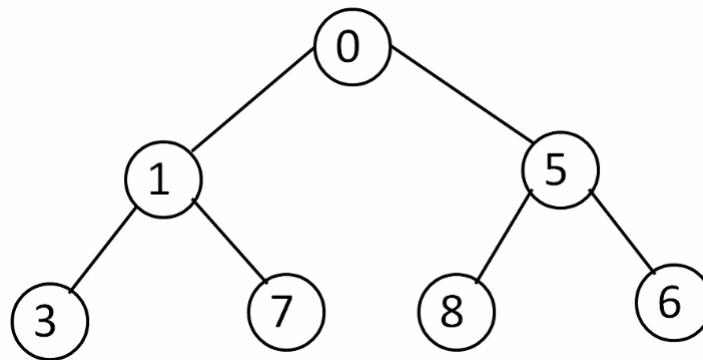


Figure: A binary heap.

The smallest key is at the top (0) and there are $\log n$ levels.

Operation	Runtime	Explanation
Insert	$\mathcal{O}(\log(n))$	Add the key at the bottom, then bubble the new key up until it's in the right place.
DecreaseKey	$\mathcal{O}(\log(n))$	We need to change the key. Then, we might need to bubble up the changed key until it's in the right place.
DecreaseMin	$\mathcal{O}(\log(n))$	We remove and then return the root node. Then, we move the bottom-most node to the root. After this, we might need to continuously bubble down the root node until it's in the right place.

For Dijkstra, we would have $\mathcal{O}(\log(|V|)(|V| + |E|))$.

1.3 d-ary Heap

This is like a binary heap, but each node has d children.

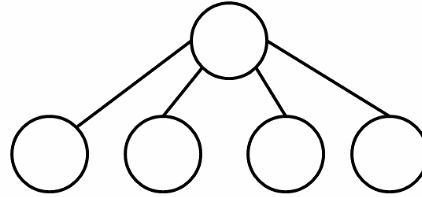


Figure: A 4-ary heap.

There are $\frac{\log(n)}{\log(d)}$ levels, so bubble up is faster. However, bubble down is slower since we need to compare more children.

Operation	Runtime	Explanation
Insert	$\mathcal{O}\left(\frac{\log(n)}{\log(d)}\right)$	Same thing as binary heap, essentially.
DecreaseKey	$\mathcal{O}\left(\frac{\log(n)}{\log(d)}\right)$	Same idea as binary heap, <i>but</i> the bubbling up is a lot faster.
DecreaseMin	$\mathcal{O}\left(\frac{d \log(n)}{\log(d)}\right)$	This is because, for bubble down, we need to compare more children; specifically, the d children.

For Dijkstra, we would have $\mathcal{O}\left(\frac{\log(|V|)(d|V|+|E|)}{\log(d)}\right)$. If the number of edges is substantially greater than the number of vertices, this can potentially be an improvement.

1.4 Fibonacci Heap

This is an advanced data structure that uses amortization¹. We are not concerned with its implementation.

Operation	Runtime
Insert	$\mathcal{O}(1)$
DecreaseKey	$\mathcal{O}(1)$
DecreaseMin	$\mathcal{O}(\log(n))$

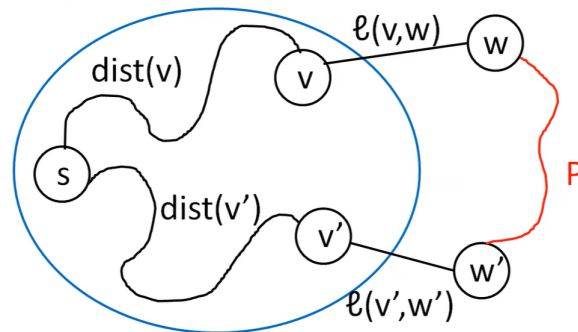
For Dijkstra, we would have $\mathcal{O}(|V| \log(|V|) + |E|)$. This is the most ideal runtime, and thus the runtime that we can assume.

¹So, you might spend more time on a particular operation, but the overall runtime will be “consistent.”

2 Negative Edge Weights

So far, we've talked about non-negative lengths. However, depending on what we're representing as lengths, we might have *negative* lengths. That being said, the problem statement is the same - find the path with the smallest sum of edge weight.

Right now, Dijkstra's algorithm doesn't actually work on negative edge values. To see why this is the case, consider the following visualization:



Here, the bubble represents all correctly assigned distances. In other words, here, s and v has edge length $\text{dist}(v)$, the shortest distance between the two. Then, from v (inside the bubble) to w (outside the bubble), if we consider the edge length $\ell(v, w)$, it follows that this must be the shortest path. The point was that this must be the right distance to w ; any other way to get to w means that we would have to leave the bubble some other way (in our case, from s to v' to w'). Then, there has to be a path connecting w' to w , of which we denote P . So, it follows that:

$$\text{dist}(v) + \ell(v, w) \leq \text{dist}(v') + \ell(v', w') + \ell(P)$$

This correctly works if $\ell(P)$ is non-negative. However, if $\ell(P)$ is negative, we run into an issue.

2.1 Negative Weight Cycle

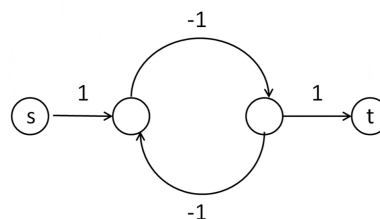
Definition 2.1

A **negative weight cycle** is a cycle where the total weight of edges is negative.

Remarks:

- If G has a negative weight cycle, then there are probably no shortest paths since we can go around the cycle over and over again.
- For an undirected graph G , a single negative weight edge gives a negative weight cycle by going back and forth on it. So, we usually don't talk about the negative edge weight in the context of an undirected graph.

For example, consider the following graph:



Here, the shortest path length from s to t is as small as we like it to be. So, really, the shortest path length is $-\infty$. This graph also answers an important question:

What stops us from adding a constant to every edge length so every edge length is non-negative, thus letting us run Dijkstra's algorithm?

Well, if we added a constant to every edge length so that every edge length is non-negative, then there would only be one answer. For example, if we added 1 to each edge length, the shortest path becomes $2 + 0 + 2 = 4$. But since we added 1 to each edge length and we went through three edges, our final answer is 1. *However*, this completely avoids the fact that we could just loop around the negative cycle many times.