Shortest Paths I: Dijkstra's Algorithm

We have now seen breadth-first search and depth-first search, which allow us to find which vertices are reachable from a particular source vertex, i.e., paths between vertices in a graph. Today we are interested in finding *shortest paths*. Depth-first search in particular can create very long paths, so it is not suitable for this task, while breadth-first search does in fact find shortest paths in *unweighted graphs*, i.e., it finds paths that use as few edges as possible.

On an unweighted graph (or a graph where every edge has the same weight), this property of breadth-first search is useful. On a weighted graph, this is not as helpful—the shortest path is not necessarily the one that uses the fewest edges. This is what motivates algorithms for solving problems involving shortest paths, which find the paths that minimize the *sum of the edge weights* along the path.

1 Shortest Path Problems

Previously, we have mostly been working with unweighted graphs, which did not have any weights assigned to each of the edges. When it comes to shortest paths problems, we will generally be working with **weighted graphs**, which are graphs where a real number is assigned to each edge as its weight.

One way to represent weighted graphs is as an adjacency list with associated weights. That is, we might represent edges as pairs of `(int, real)` where (v, w) denotes a directed edge pointing at v with weight w.

In an unweighted graph, we defined the "length" of the shortest path from u to v as the fewest number of edges of any path from u to v in G.

Now, we will consider weighted graphs. If G is weighted, then $\delta_G(u, v)$ is defined to be the "weight" of the shortest path from u to v. The "weight" of a path now refers to the sum of the weights of the edges along that path, rather than the number of edges.

There are then three types of shortest path problems that we might want to solve on a weighted graph:

- **Single-pair shortest path**: Given vertices u and v, find the shortest path from u to v.
- **Single-source shortest paths**: Given a vertex s, find the shortest paths from s to every vertex.
- **All-pairs shortest paths**: Find the shortest paths between every pair of vertices.

Single-pair shortest path problems are frequently solved with single-source shortest paths algorithms — the single-source algorithm moves outward from the source, building up a set of known shortest paths as it goes along. Once it has found the desired destination vertex, it can terminate, however, this is often not asymptotically better than simply finishing the algorithm.

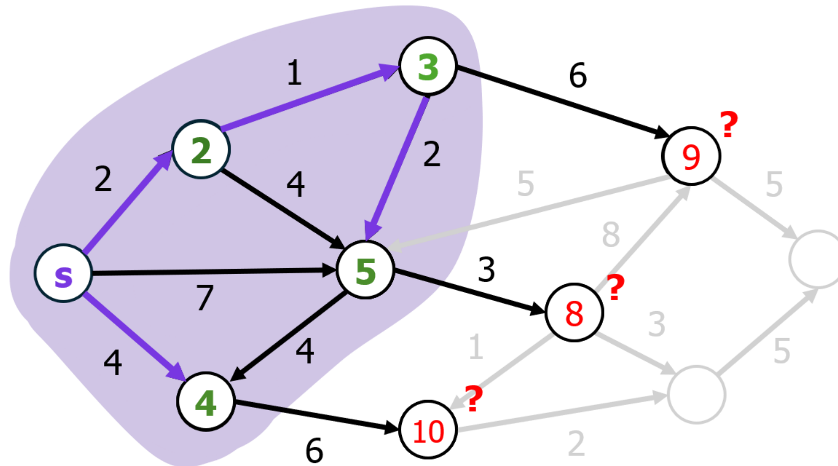
For the rest of our discussion of shortest paths, we will assume that every graph that we are analyzing is a weighted, directed graph unless stated otherwise. We'll look at a few different algorithms which work on graphs with various properties, and analyze their work and span.

2 Dijkstra's Algorithm

The first algorithm we will look at is **Dijkstra's algorithm**, which solves the single-source shortest paths problem. Dijkstra's is a fully sequential algorithm, and it only works on graphs where the edge weights are all nonnegative. It returns a sequence containing the shortest path weight from a given source vertex s for every vertex.

Dijkstra's algorithm is based on the following **greedy** observation. If we know some subset of the shortest paths, then we can find another shortest path by adding one more edge to those paths and taking the shortest one.

Intuitively, this works because there can't be a better path for that vertex, as it would have to go through on of the other paths, and since all edge weights are non-negative and the other paths are heavier, any path resulting from extending those paths must be heavier.



For example, consider the following graph and suppose the algorithm knows the shortest paths to the vertices in the purple region, whose distances are labeled in green and whose shortest paths are edges bolded in purple. The algorithm can extend the paths found so far by adding one additional edge, and it will do so greedily by choosing the shortest such path, which connects the edge of weight 3 to the vertex at distance 5, connecting it to a vertex of distance 8.

This is guaranteed to be a shortest path since any alternate path would have to either detour through the distance 10 or 9 vertex, and since the weights are non-negative, nothing we add to 10 or 9 will ever be lower than 8. These tentative distances are called *distance labels*.

That is the core idea of the algorithm. Instantiating this greedy approach with appropriate data structures gives us Dijkstra's algorithm.

2.1 Implementation

The algorithm is, at each step, selecting a minimum weight path from a pool of candidates. It is therefore natural to use a *priority queue* to maintain these candidates, since a priority queue allows us to efficiently retrieve the minimum one.

Algorithm: Dijkstra's Algorithm

```
fun dijkstra(G : Graph, s: int) -> sequence<T>:
    dist = [0 if (u == s) else ∞ for u in 0...|G.vertices|-1]
    visited = [False for _ in 0...|G.vertices|-1]

    pq = PriorityQueue()
    pq.insert((0,s))

    while pq not empty:
        d, u = pq.deleteMin()
        if visited[u]: continue
        visited[u] ← True
        for (neighbor, weight) in neighbors(u):
            dist' = d + weight
            if dist' < dist[neighbor]:
                dist[neighbor] ← dist'
                pq.insert((dist', neighbor))
    return dist
```

Note that as written, this algorithm is entirely sequential and uses mutable data structures. Dijkstra's algorithm is essentially not parallelizable at all, so we keep it this way.

One interesting quirk of our Dijkstra's implementation is that a vertex can end up in the priority queue multiple times if there are several possible previous paths that could be extended to reach it. It is possible to eliminate this redundancy, but it requires a fancier priority queue data structure—one that allows us to modify the key of an element already present in the queue.

Since most priority queue implementations do not actually include this operation, we do not do this strategy. Thus, our implementation just inserts a vertex every time it shows up in the neighborhood of a visited vertex and produces a smaller distance estimate, even if it is already in the queue, and ignore it if it comes up again after having been visited.

Theorem: Cost of Dijkstra's

On a simple graph on n vertices and m edges, Dijkstra's Algorithm costs $O(m \log(n))$

Proof. The cost of Dijkstra's algorithm is dominated by the costs of the priority queue operations. An efficient priority queue implements both `delete_min` and `insert` in $O(\log|Q|)$ time. Thus, we want to find the number of times that these operations are used, and the maximum size of the queue at any point during the algorithm.

If Dijkstra's visits every vertex, then throughout the execution, we will be performing m insertions. This comes about because every edge (u, v) will lead to one insertion, which happens

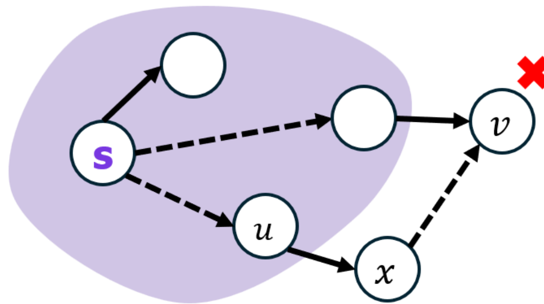
when u is visited. The maximum size of the priority queue at any one time is $O(m)$, meaning that each insertion costs $O(\log m)$. There are $O(m)$ insertions, and each insertion leads to a deletion so there are also $O(m)$ calls to `delete_min`. Therefore, both the insertions and the deletions have a total cost of $O(m \log m)$.

Since the graph is a simple graph, $m \leq n^2$ and hence $\log m = O(\log n)$, so we can say the cost of Dijkstra's Algorithm is $O(m \log n)$. □

Theorem: Correctness of Dijkstra's

On a directed graph with non-negative weights, Dijkstra's Algorithm outputs the shortest path weights for every vertex.

Proof. Assume for the sake of contradiction that Dijkstra's algorithm outputs a wrong distance. Let v be the first vertex discovered by the algorithm that has the wrong distance. By assumption, there must be a shorter path to v. Let u be the last visited vertex on that path, and let x be the subsequent unvisited vertex.



- If $x = v$, the v is adjacent to u which by assumption has the correct distance, which means v must have the correct distance since the distance estimate for v would have been set correctly when u was discovered.
- If $x \neq v$ then **since the weights are non-negative**, the distance estimate of x must be lower than the distance estimate of v. Therefore Dijkstra's should have visited x, not v.

In either case, we get a contradiction. □