FIT2004: Supplementary notes for lectures 8 and 9

Graphs: Shortest Path Problems

DANIEL ANDERSON [1]

In this section, we will study the **shortest path** problem. Shortest paths are one of the most natural and clearly useful graph problems that you will encounter. If you've ever used Google maps to find your way to a destination, then you've seen shortest paths problems in action. We have already seen a simple example of shortest paths, which were shortest paths on an unweighted graph, which could be determined as a by-product of a breadth-first search. Now we will explore more general shortest path algorithms for weighted graphs.

Summary: Shortest paths

In this lecture, we cover:

- Properties of shortest paths
- The single-source shortest path problem
- The Bellman-Ford algorithm
- Dijkstra's algorithm
- The all-pairs shortest path problem
- The Floyd-Warshall algorithm and its relation to the transitive closure problem

Recommended Resources: Shortest paths

- CLRS, Introduction to Algorithms, Chapter 24 and Section 25.2
- Weiss, Data Structures and Algorithm Analysis, Section 9.3
- `http://users.monash.edu/~lloyd/tildeAlgDS/Graph/`
- `https://visualgo.net/en/sssp`

Definitions of Shortest Paths

The notion of a shortest path is intuitive, and may be stated informally as a path between vertices of a graph that minimises the total weight of the edges used in the path. One of the most obvious examples would be the idea of finding the shortest route on a road map from one location to another, in which we must decide which roads to take in order to minimise either the total distance travelled, or the time taken, or indeed any other quantity that we are interested in, such as toll costs. Note that we usually refer to **a** shortest path rather than **the** shortest path since there may be multiple equally short paths between two vertices.

Shortest paths make sense for both directed and undirected graphs, and none of the algorithms that we will study differ for either kind. Indeed, for a shortest path problem on an undirected graph, we can easily see that it is equivalent to a shortest path problem on a corresponding directed graph in which each undirected edge $e = (u, v)$ is simply replaced with two directed edges (u, v) and (v, u) of the same weight.

Properties of Shortest Paths

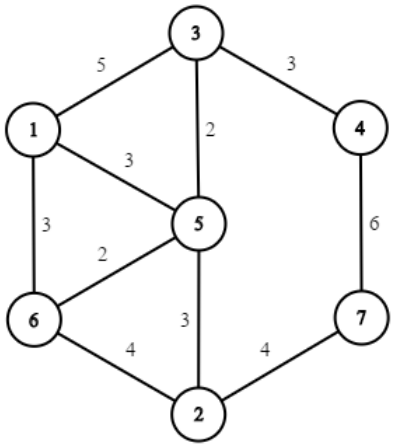
Before diving straight in to the algorithms, we'll first take a look at some useful properties that shortest paths have which will help us in understanding the algorithms and how they work.

Shortest paths have optimal sub-structure

The first and most important observation that we can make about shortest paths is that they contain massive amounts of sub-structure. Consider the graph in the figure below. We can see that the shortest path from $1 \rightsquigarrow 7$

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

is through vertices $1 \to 5 \to 2 \to 7$ with a total weight of 10. Given that this is true, we know for sure that a shortest path from vertex 1 to vertex 2 must be $1 \to 5 \to 2$. Why? Since $1 \to 5 \to 2$ is part of the shortest path from $1 \rightsquigarrow 7$ (it is a *sub-path*), if it were not itself a shortest path, then we could replace this part of the original path with a shorter one, making our supposed shortest path even shorter. (Think about how you would prove this formally. **Hint:** Use contradiction.)



An edge weighted, undirected graph. A shortest path from vertex 1 to 7 has a total weight of 10.

Shortest paths satisfy the "triangle inequality"

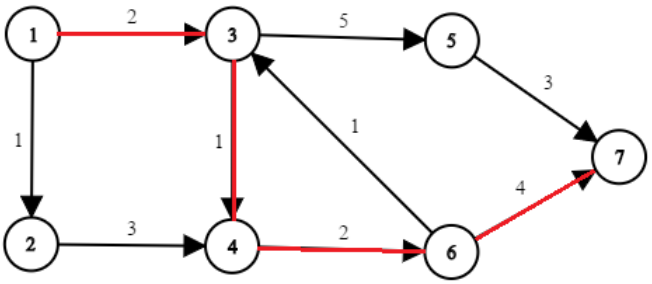
Let us denote by $\delta(s, v)$ the total length of a shortest path from vertex s to vertex v. The triangle inequality says that for any edge $(u, v) \in E$ with weight $w(u, v)$, it is true that

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Behind these fancy symbols is a simple and intuitive but very critical fact that underpins most shortest path algorithms. All that this says in plain English is that if we have a shortest path $s \rightsquigarrow v$, then we can't find a path $s \rightsquigarrow u$, followed by an edge $u \to v$ with a shorter overall length, because this would mean that our shortest path was not actually the shortest path. Expressed even more simply, you can't find a path that is shorter than your shortest path!

Why is this true? Simply put, if the shortest path from vertex s to vertex u has total length $\delta(s, u)$ and the edge (u, v) has weight $w(u, v)$ then we can form a path to vertex v by taking a shortest path from s to u and then traversing the edge (u, v), forming a path from s to v. The total length of this path is $\delta(s, u) + w(u, v)$, so this length can not be shorter than the length of the shortest path, or it means that our shortest path was wrong. (Again, try proving this formally by contradiction.)

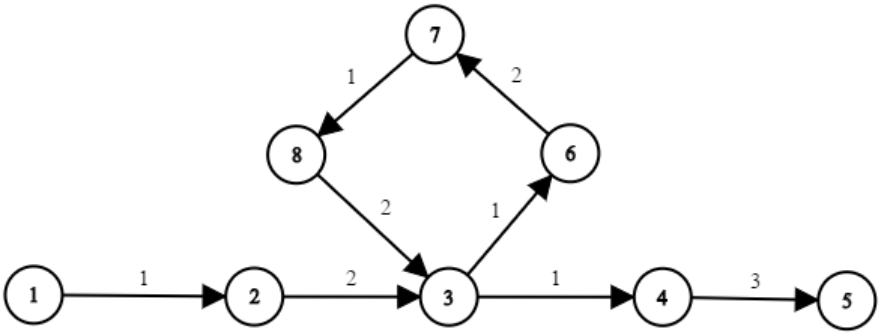
This is a **critical fact** to understand about shortest paths. Make sure that it sinks in before attempting to absorb the rest of the material.



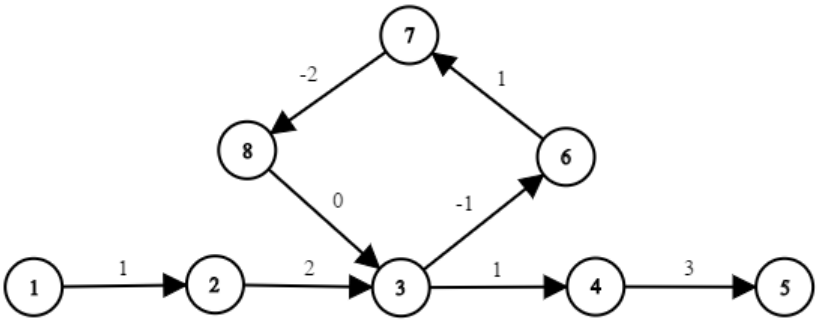
An edge weighted, directed graph. A shortest path from vertex 1 to 7 has a weight of 9. We could instead travel from vertex 1 to 5 at a weight of 7, and then from vertex 5 to 7 for a total weight of 10, which is not as short as the shortest path.

We can disregard cycles

Consider a path that contains a cycle in it. If this cycle has a positive total weight, then we can remove it from the path and the total weight decreases, so this path was not a shortest path. If the cycle has a negative total weight, then we may travel around the cycle over and over for as long as we wish and accumulate arbitrarily short paths, so the notion of a shortest path is not well defined in this case. Finally, if the path contains a cycle of zero total weight, then we may remove it and obtain a path with the same total weight. Therefore without loss of generality, we may assume that shortest paths are simple paths (contain no cycles) when looking for them. This will greatly simplify the ideas to come.



An edge weighted, directed graph containing a directed cycle. The cycle can not possible be a part of a shortest path since it only makes the path length longer without going anywhere!.



An edge weighted, directed graph containing a directed cycle with a negative total weight. The shortest path between vertex 1 and 5 is now **undefined**. For any path that you give me, I can always make an even shorter one by going around the cycle over and over, making the weight arbitrarily small.

For further details and proofs of shortest path properties, see CLRS Chapter 24.

Shortest Path Problem Variants

There are three main variants of the shortest path problems that we might wish to consider.

Problem Statement: The single-pair shortest path problem

Given a weighted, directed or undirected graph $G = (V, E)$ and a pair of vertices $u, v \in V$, find a shortest path between u and v. Formally, find a sequence of adjacent vertices $(v_1, v_2, ..., v_k)$ such that the sum

$$\sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

is minimised. In an unweighted graph, we consider $w(u, v) \equiv 1$, which is equivalent to asking for a path from u to v containing the fewest possible edges.

Although single-pair is seemingly the simplest possible shortest path problem, it turns out that the more general variants are typically simpler and asymptotically no more difficult to solve.

Problem Statement: The single-source shortest path problem

Given a directed or undirected graph $G = (V, E)$ and a starting vertex $s \in V$, find for every vertex $v \in V$ a shortest path from s to v.

Given the copious amounts of substructure that is present in shortest paths, it should be intuitive that we can solve the single-source problem much faster than we could solve a single-pair problem separately for every possible vertex v. The final variant is a further generalisation that is actually computationally harder then the previous ones.

Problem Statement: The all-pairs shortest path problem

Given a directed or undirected graph $G = (V, E)$, find a shortest path between every pair of vertices $u, v \in V$.

There is much overlap between these problems, and indeed each of them can be used to solve the former one. We will only explore algorithms for the last two of them since in general, it is usually the same difficulty to solve the first problem as the second[2].

Distance estimates and the relaxation technique

Most of the algorithms that we will study for shortest paths hinge on the technique of **relaxation**. For each of these algorithms, we are going to maintain a *distance estimate* for each vertex. That is, we will maintain for each vertex, the length of the shortest path to that vertex that we have found so far. Throughout the algorithms, we will continuously update these estimates and improve them as we find better paths. The notion of *relaxation* simply means to update a current shortest path to an even shorter path found by **enforcing the triangle inequality.**

In other words, if we find an edge that violates the triangle inequality, then it means that we have found a path that is shorter than one of our current estimates, so we should update our estimate to the newer, shorter path.

Algorithm: Edge Relaxation

```
1: function RELAX(e = (u,v))
2:     if dist[v] > dist[u] + w(u,v) then
3:         dist[v] = dist[u] + w(u,v)
4:         pred[v] = u
5:     end if
6: end function
```

The predecessor array `pred` is maintained in the same way as it was in a breadth-first search and can be used in the same way to reconstruct the shortest paths once we have found all of the correct distances.

A very simple algorithm for finding single-source shortest paths would then be to simply relax all of the edges of the graph over and over until all of them satisfy the triangle inequality. Once the triangle inequality is satisfied for all of our distance estimates, then we know that our estimates are correct and we have all of the true shortest paths. This is in fact the basis of a shortest path algorithm called *Bellman-Ford.*

The Bellman-Ford Algorithm

The Bellman-Ford algorithm is arguably the simplest algorithm for solving the single-source shortest paths problem in a weighted graph. Simply put, the Bellman-Ford algorithm repeatedly relaxes all of the edges of the

[2]This is not always the case. Meet-in-the-middle approaches are commonly employed in search problems with an exponential search space size to cut down the number of states that need to be visited significantly. Often a search size of 2^N can be cut down to a size of $2^{\frac{N}{2}}$, which can mean the difference between a completely impossible computation and a very feasible one.

graph until all of the shortest paths are obtained. More specifically, we relax every edge of the graph $|V| - 1$ times. The reason for this will become clear shortly.

Algorithm: Bellman-Ford

```
1: function BELLMAN_FORD(G = (V[1..N], E[1..M], s))
2:     Set dist[1..N] = ∞
3:     Set pred[1..N] = 0
4:     Set dist[s] = 0
5:     for i = 1 to N-1 do
6:         for each edge e in E[1..M] do
7:             relax(e)
8:         end for
9:     end for
10:    return dist, pred
11: end function
```

That's it! This a simple implementation of Bellman-Ford that will find all of the shortest paths from the vertex `s`, if they exist. The time complexity of Bellman-Ford is easy to establish: the outer loop runs for $|V| - 1$ iterations, the inner loop runs for $|E|$ iterations, and each relaxation takes $O(1)$, so the total time taken is $O(|V||E|)$. Why do we need $|V| - 1$ iterations? Let's explore that now.

Theorem: Correctness of Bellman-Ford

The Bellman-Ford algorithm terminates with the correct distance estimates to all vertices whose shortest path is well-defined after at most $|V| - 1$ iterations.

Proof

In a well-defined shortest path, the number of edges on the path can not be greater than $|V| - 1$, why? If a path contained more than $|V| - 1$ edges, then it must visit some vertex multiple times and hence contain a cycle, which we know will never occur in a valid shortest path. We claim that after k iterations of the Bellman-Ford algorithm, that all valid shortest paths consisting of at most k edges are correct.

We can prove this formally by induction on the number of edges in each shortest path. For the base case, we initialise `dist[s] = 0`, which is correct if there is no negative cycle containing `s`. Hence all valid shortest paths containing zero edges are correct.

Inductively, suppose that all valid shortest paths containing at most k edges are correct after k iterations. Let's argue that all valid shortest paths containing at most $k+1$ edges are correct after one more iteration. Consider some shortest path from $s \rightsquigarrow v$ consisting of at most $k+1$ edges. By the substructure of shortest paths, the sub-path $s \rightsquigarrow u$ consisting all but the final edge (u, v) is a shortest path, and by our inductive hypothesis, is correctly estimated at the current iteration of the algorithm. If the current distance estimate of v is incorrect, it will be relaxed in this iteration by the edge (u, v) and hence be made correct since $\delta(s, v) = \delta(s, u) + w(u, v) =$ `dist[u] + w(u,v)`.

Hence by induction on the number of edges in the shortest paths, all valid shortest paths consisting of at most k edges are correctly estimated after k iterations. Therefore the Bellman-Ford algorithm terminates correctly after at most $|V| - 1$ iterations.

Dealing with negative cycles

Now, an important problem that needs to be addressed is how to deal with the existence of negative cycles. As mentioned earlier, the presence of a negative cycle may cause the shortest path to a particular vertex to become undefined since there may exist paths of arbitrarily short length. In this case, the algorithm above will terminate with a distance estimate to such vertices that is incorrect. Detecting and dealing with such paths is actually quite easy though, we simply appeal once more to **the triangle inequality.**

If a subset of the distance estimates satisfy the triangle inequality, then they must be valid shortest paths, so we can keep them. We can then seek out the distance estimates that violate the triangle inequality and conclude that those are the vertices that must be reachable via a negative weight cycle, and mark them as such.

Algorithm: Bellman-Ford Post-Processing to Mark Undefined Shortest Paths

```
1: function MARK_UNDEFINED_PATHS(G = (V[1..N], E[1..M]))
2:     for i = 1 to N do
3:         for each edge e = (u,v) in E[1..M] do
4:             if dist[u] + w(u,v) < dist[v] then
5:                 dist[v] = −∞
6:             end if
7:         end for
8:     end for
9: end function
```

After running this post-processing step, all of the distances given in the `dist` array will be:

1. ∞ if the vertex in question was never reached (it is not connected to the source s)
2. $-\infty$ if the vertex in question is reachable via a negative cycle and hence does not have a shortest path
3. The correct distance if the vertex has a well-defined shortest path

Optimising Bellman-Ford

There are some simple optimisations that can be made to Bellman-Ford that improve its running time:

1. During the iterations of the main `i` loop, if no relaxations are successful, then the shortest paths have already been found and we can terminate early.
2. If such an early termination occurs, then we are guaranteed that no negatives cycles exist and hence can skip the post-processing step that checks for undefined shortest paths.
3. Rather than iterating over every single edge, we could instead maintain a queue of edges that need to be relaxed. If an edge is not successfully relaxed in one phase, then it clearly need not be relaxed in the next phase either unless its source vertex has its distance estimate improved. For sparse graphs, this improves the empirical average case complexity of the algorithm to just $O(|E|)$, although the worst-case behaviour is unfortunately still $O(|V||E|)$. This variant is often referred to as the "Shortest Path Faster Algorithm."

Dijkstra's Algorithm

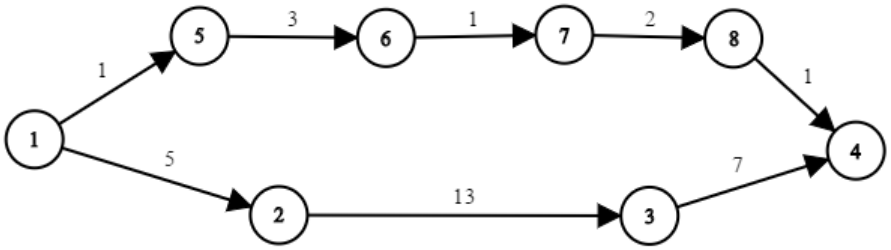
Although the Bellman-Ford algorithm is a generally applicable algorithm for the single-source shortest path problem, ie. it works on any graph that has well-defined shortest paths, it is not very efficient in practice even with optimisations. *Dijkstra's algorithm* presents us with a useful trade-off. Dijkstra's algorithm, unlike the Bellman-Ford algorithm is no longer capable of handing negative edge weights, but it is significantly faster both theoretically and in practice.

The idea of finding a more efficient solution to a problem by restricting the kinds of input that you want to handle should be one that you become more and more used to (for example, radix sort allowed us to sort faster, but only if we were willing to make strict assumptions about the things that we were sorting.) Dijkstra's algorithm is a perfect example of trading-off generality for a significant performance increase, and is still a widely applicable algorithm since many graphs encountered in practice do not require negative edge weights.

Key Ideas: Dijkstra's Algorithm

1. If all edge weights are non-negative, ie. $w(u, v) \geq 0$ for all edges (u, v), then any sub-path $s \rightsquigarrow u$ of some longer path $s \rightsquigarrow v$ necessarily has a weight that is no greater than the entire path. This is not true when negative weights appear, as a path may grow in number of edges but decrease in total weight due to the negative weights.
2. Unlike Bellman-Ford, which finds shortest paths with a very "shotgun" like approach (relax edges in an arbitrary order over and over), Dijkstra's follows a much more similar idea to breadth-first search by searching for and visiting nodes that are a small distance away from the source before fanning out and gradually visiting further away vertices.

Owing to the first observation, Dijkstra's algorithm employs a greedy strategy for relaxing edges in a provably optimal order. Just like breadth-first search, Dijkstra's algorithm visits nodes in order of their distance from the source node. The primary difference is that because the edge weights are no longer all the same, we can not simply use an ordinary queue to store the vertices that we need to visit, since a shortest path might consist of a long chain of edges with very small weights, which may indeed have a smaller total weight than some short chain consisting of large weights.



A graph where the shortest path from $1 \rightsquigarrow 4$ has far more edges than the path with the fewest edges.

Instead, Dijkstra's algorithm employs a priority queue to ensure that vertices are visited in the correct order of their distance from the source[3].

Algorithm: Dijkstra's Algorithm

```
1: function DIJKSTRA(G = (V[1..N], E[1..M], s))
2:     Set dist[1..N] = ∞
3:     Set pred[1..N] = 0
4:     Set dist[s] = 0
5:     Set Q = priority_queue(V[1..N], key(v) = dist[v])
6:     while Q is not empty do
7:         u = Q.pop_min()
8:         for each edge e that is adjacent to u do
9:             relax(e)
10:        end for
11:    end while
12:    return dist, pred
13: end function
```

The priority queue `Q` is a minimum priority queue initially containing all vertices that serves each vertex `u` in order of their current distance estimate `dist[u]`. At each iteration, the next vertex `u` that has not been visited yet and which has the smallest current distance estimate is processed. Note that since vertices are removed in distance order, that once a vertex has been removed from the queue, its distance estimate must be correct and will never be further modified.

The time complexity of Dijkstra's algorithm depends on the way in which we chose the implement the priority queue of vertices. In general, the time taken depends on two things, the amount of time required to find the next minimum, and the amount of time required to update the distance estimates when we relax them. If we denote by T_{update} and T_{find}, the time taken to update the distance estimates and to find the minimum respectively, then the total time complexity of Dijkstra's algorithm will be

$$O(|E| \cdot T_{\text{update}} + |V| \cdot T_{\text{find}}).$$

This is due to the fact that we relax every edge exactly once, performing an update operation each time in the worst-case, and we find and extract each vertex exactly once when it is the current minimum. If we implement Dijkstra's in the simplest way possible, using an array and looping over every vertex to find the minimum, then it takes constant $O(1)$ time to update the distances (we just update the `dist` array and nothing else) and linear $O(|V|)$ time to find the next vertex. Therefore the total running time would be

$$O(|E| + |V| \cdot |V|) = O(|V|^2).$$

[3]Actually, Dijkstra's original implementation of his algorithm did not use a priority queue, but rather simply looped over the vertices to find which one was the next closest. This is rather inefficient however, and was improved by Fredman and Tarjan, who described the min-heap based priority queue implementation.

This can be improved significantly for sparse graphs by using a min-heap based priority queue. If we use a standard binary heap data structure, we achieve $O(\log(|V|))$ time per operation, and hence the time complexity of Dijkstra's algorithm becomes

$$O(|E|\log(|V|) + |V|\log(|V|)) = O(|E|\log(|V|)).$$

For dense graphs, this is actually slightly worse than the simple, array-based implementation, but for sparse graphs, this is a huge performance increase. Even better asymptotic complexities can be achieved by using fancier heap data structures such as the Fibonacci heap, but these often turn out to be less efficient in practice than their simpler counterparts.

Theorem: Correctness of Dijkstra's Algorithm

Given a graph $G = (V, E)$ with non-negative weights and a source vertex s, Dijkstra's algorithm correctly finds shortest paths to each vertex $v \in V$.

Proof

We can prove the correctness of Dijkstra's algorithm using a similar inductive argument to the one used for Bellman-Ford. Let's use induction on the set of vertices S that have been removed from the queue, ie. the set $S = V \setminus Q$. We will show for any vertex $v \in S$, that `dist[v]` is correct.

For the base case, we will always remove the source vertex s first which has `dist[s] = 0`, which is correct since the graph contains no negative weights.

Suppose for the purpose of induction that at some point it is true that for all $v \in S$, `dist[v]` is correct. Let u be the next vertex removed from the priority queue, we will show that `dist[u]` must be correct.

Suppose for contradiction that there exists a shorter path for $s \rightsquigarrow u$ with a shorter distance $\delta(s \rightsquigarrow u) <$ `dist[u]`. Let x be the furthest vertex along the correct $s \rightsquigarrow u$ path that is in S. Since $x \in S$, by the inductive hypothesis, its distance estimate is correct. Let y be the next vertex on the shortest path after x. Since $\delta(s \rightsquigarrow u) <$ `dist[u]` **and all edge weights are non-negative**, it must be true that $\delta(s \rightsquigarrow y) \leq \delta(s \rightsquigarrow u) <$ `dist[u]`. But y is adjacent to x on a shortest path, which means the edge (x, y) was relaxed when x was removed from Q. This means that `dist[y]` $= \delta(s \rightsquigarrow y) <$ `dist[u]`. If `dist[y] < dist[u]` and $y \neq u$, then Dijkstra's algorithm would have popped y from the priority queue instead of u, a contradiction. Alternatively if $y = u$ and `dist[y] < dist[u]`, this is also a contradiction.

By contradiction, we conclude that `dist[u]` must be correct and hence by induction on the set S, when Dijkstra's algorithm terminates, `dist[v]` is correct for all $v \in V$.

Notice how the key step of the proof requires us to invoke the fact that the edge weights are non-negative. Without this restriction, this step of the proof would not be true, since the sub-path $s \rightsquigarrow y$ might actually have a higher total weight than the total path $s \rightsquigarrow u$ due to negative weights.

Practical considerations: Implementing Dijkstra's Algorithm

When implementing Dijkstra's algorithm, careful thought needs to be put into the way in which the priority queue is going to be implemented. The most common description of the priority queue used for Dijkstra's algorithm is a min-heap that supports the following operations:

1. Insert the initial items into the priority queue,
2. Remove the item with the minimum key,
3. Decrease the key of an item already in the priority queue (whenever an edge is relaxed.)

In practice however, most language's standard library priority queues do not support operation 3, and implementing this operation in a custom min-heap can be somewhat tricky. Most implementations therefore take the following alternate but equivalent approach:

1. Begin with only the source vertex s in the priority queue,
2. Whenever an edge is relaxed, insert the target vertex in the priority queue keyed by its current distance estimate, regardless of whether it is already in there or not,

3. When a vertex is removed from the queue, check whether this vertex has already been removed before, and if so, just ignore it.

The key idea is that instead of updating the keys of vertices that are in the priority queue, we simply allow it to contain multiple entries for the same vertex with different distance estimates. When a vertex is removed that has already been removed once before, we know that we just removed an "out-of-date" entry and can safely ignore it.

This does not hurt the asymptotic complexity of the algorithm, since the only difference encountered is that the priority queue may grow to a worst-case size of $|E|$ instead of $|V|$, leading to operations that cost $O(\log(|E|))$ instead of $O(\log(|V|))$. But since

$$O(\log(|E|)) = O(\log(|V|^2)) = O(2\log(|V|)) = O(\log(|V|)),$$

this is asymptotically equivalent. In practice, the number of entries in the priority queue at any given time is likely to be much smaller than $|E|$, so this method is quite often actually a speed-up too.

The All-Pairs Shortest Path Problem

Finally, we consider the all-pairs shortest path problem, in which we seek a shortest path between every pair of vertices in the given graph. The most obvious solution to this problem is to simply solve the single-source shortest path problem using every possible vertex as the source. This might sound inefficient, but for certain graphs this is actually the optimal solution.

- Consider an unweighted graph. If we perform a breadth-first search from every possible source, each taking $O(|V| + |E|)$ time, then the complexity of finding all-pairs shortest paths is $O(|V|(|V| + |E|))$.
- In graph with non-negative weights, we can run Dijkstra from every possible source and achieve a runtime of $O(|V||E|\log(|V|))$ assuming that our priority queue is a binary heap.
- In a graph with negative weights, multiple invocations of Bellman-Ford from every source would yield a time complexity of $O(|V|^2|E|)$.
- **(Not examinable)** For graphs with negative weights, multiple Bellman-Fords is actually extremely slow. However, a very cheeky trick exists called the "potential method" that allows us to re-weight the graph in such a way that we get rid of all of the negative weight edges without changing the shortest paths. Multiple invocations of Dijkstra's algorithm can then be made on the modified graph before reverting the weights and recovering the shortest paths. This algorithm is called Johnson's algorithm and has a runtime of $O(|V||E|\log(|V|))$, assuming Dijkstra's is implemented with a binary heap.

The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a very simple all-pairs shortest path algorithm that is asymptotically optimal for dense graphs, but can be beaten in performance on sparse graphs by multiple invocations of the single-source algorithms as described above.

Floyd-Warshall relies on the observation that a shortest path from $u \rightsquigarrow v$ that has at least two edges can be decomposed into two shortest paths $u \rightsquigarrow k$ and $k \rightsquigarrow v$ for some intermediate vertex k. The algorithm simply iteratively increases the pool of intermediate vertices k from 1 to N and updates all paths that can be improved by visiting vertex k as an intermediate step.

We initialise the all-pairs distance estimate as a 2D matrix `dist[1..N][1..N]` such that `dist[v][v] = 0` for all vertices v and `dist[u][v] = w(u,v)` for all edges $e = (u, v)$. Observe that if the graph is represented as an adjacency matrix, then initialising the distance matrix `dist` is easy as this is nothing but a copy of the adjacency matrix.

Algorithm: Floyd-Warshall

```
1: function FLOYD_WARSHALL(G = (V[1..N], E[1..M]))
2:     Set dist[1..N][1..N] = ∞
3:     Set dist[v][v] = 0 for all vertices v
4:     Set dist[u][v] = w(u,v) for all edges e = (u,v) ∈ E[1..M]
5:     for k = 1 to N do
6:         for u = 1 to N do
7:             for v = 1 to N do
8:                 dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
9:             end for
10:        end for
11:    end for
12:    return dist
13: end function
```

The time complexity of Floyd-Warshall is obvious, there are three loops from 1 to N and a constant time update, so the total runtime is $O(|V|^3)$. Comparing this against the running times of the repeated single-source solution, we can see that for dense graphs ($|E| \approx |V|^2$), Floyd-Warshall is as good or strictly better in all cases. For sparse graphs however ($|E| \approx |V|$), Floyd-Warshall is beaten by $O(|V|^2 \log(|V|))$.

Finally, note that just like Bellman-Ford, Floyd-Warshall can handle negative weights just fine, and can also detect the presence of negative weight cycles. To detect negative weight cycles, simple inspect the diagonal entries of the `dist` matrix. If a vertex v has `dist[v][v]` < 0, then it must be contained in a negative weight cycle (since were are able to travel from a vertex to itself with a negative total distance.)

Application of all-pairs shortest paths: Transitive closure of a graph

Definition: Transitive Closure

The transitive closure of a graph G is a new graph G' on the same vertices such that there is an edge between a pair of vertices if there was a path between those vertices in G.

Finding the transitive closure of a graph can be reduced to the problem of finding all-pairs shortest paths by noting that there is an edge between u and v in the transitive closure if and only if `dist[u][v]` $< \infty$.

We can save some practical time and space by observing that we only ever care about whether or not two vertices are connected, not what their distance value is. We can store our connectivity matrix using individual bits (a data structure called a *bitset* in some languages,) and update them by noting that vertex u is connected to vertex v via vertex k if and only if `connected[i][k]` and `connected[k][j]` are both set to true.

Algorithm: Warshall's Transitive Closure Algorithm

```
1: function FLOYD_WARSHALL(G = (V[1..N], E[1..M]))
2:     Set connected[1..N][1..N] = 0
3:     Set connected[v][v] = 1 for all vertices v
4:     Set connected[u][v] = 1 for all edges e = (u,v) ∈ E[1..M]
5:     for k = 1 to N do
6:         for u = 1 to N do
7:             for v = 1 to N do
8:                 connected[u][v] = connected[u][v] or (connected[u][k] and connected[k][v])
9:             end for
10:        end for
11:    end for
12:    return connected
13: end function
```