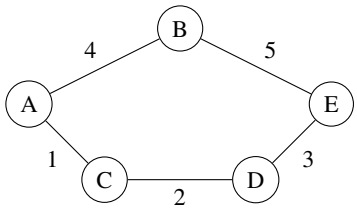
Chapter 25
Shortest Path Algorithms

25.1 The Shortest Path Problem

Back in chapter 19, we introduced several algorithms that can be used to solve a variety of different graph problems. One of these algorithms was the breadth-first search (BFS), which can be applied to find the shortest path between any two vertices of an unweighted graph. However, many graphs in real-world applications are *weighted* rather than unweighted. For weighted graphs, a simple BFS cannot be used to find the path between two vertices with the minimal weight, as it is designed to pick the route with the fewest number of edges rather than the one that minimizes overall edge weight. As a result, we will need to consider other algorithms if we want to find the lowest-weighted path between two vertices of a weighted graph.

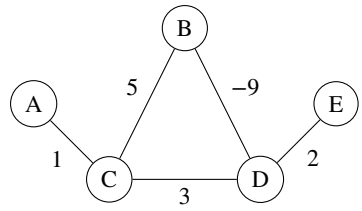
Before we begin exploring these algorithms, we will first define the problem we are trying to solve: the **shortest path problem**. In the shortest path problem, you are given an edge-weighted graph $G = (V, E)$ and two vertices $v_s \in V$ and $v_d \in V$, and you want to find the path starting at v_s and ending at v_d with the lowest overall path weight. For example, the shortest path between nodes A and E in the following graph is $A \to C \to D \to E$ with total weight $1 + 2 + 3 = 6$. Notice that the *number* of edges in our path is irrelevant here, as the value we want to minimize is the total *weight* of the edges from source to destination.



One important feature shared by shortest path algorithms is a reliance on *optimal substructure*, as the shortest path between any two points in a graph must itself be composed of the shortest paths between intermediary nodes along the way. For instance, we know that the shortest path from $A \to E$ is $A \to C \to D \to E$ for a total weight of $1 + 2 + 3 = 6$. However, this also implies that the shortest path from $A \to D$ must be $A \to C \to D$ for a total weight $1 + 2 = 3$. This is because, if there were a shorter path that goes from A to D without going through C, then the best path from A to E could not possibly be $A \to C \to D \to E \ldots$ a contradiction to a claim we already know to be true! Similarly, we also know that the shortest path from C to E must be $C \to D \to E$. In general, given any shortest path $v_0 \to v_1 \to \ldots \to v_k$ from v_0 to v_k, any subpath from vertex v_i to v_j such that $0 \leq i \leq j \leq k$ must also be a shortest path from v_i to v_j.

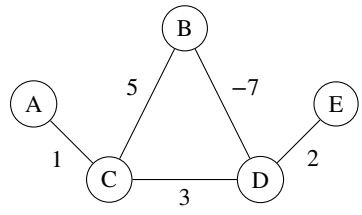
Recall that optimal substructure serves as a foundation for the greedy and dynamic programming algorithm families discussed in earlier chapters. Thus, it comes at no surprise that shortest path algorithms typically rely on these algorithmic techniques to solve the shortest path problem. The reliance on an optimal substructure also leads to another consequence of shortest path algorithms: since smaller subproblems are used to build up to larger subproblems, many of these algorithms find the shortest path from the source vertex v_s to *every other vertex* in the given graph V as a byproduct of solving the original problem from v_s to some destination v_d. Because of this, such algorithms also solve the **single-source shortest path problem**, which seeks to find the shortest paths from a single source vertex to *every other vertex* in the graph.

Finally, there is one notable caveat to the shortest path problem that pertains to *negatively weighted edges*. If a graph contains a *negative cycle* reachable from the source node v_s, then the shortest path is negatively infinite and the problem has no solution. For instance, consider the following graph, where the cycle $B \to C \to D \to B$ has a total weight of 5 + 3 + (-9) = -1.



What is the shortest path from $A \to E$? It should be quickly apparent that the answer is negatively infinite, since you can repeatedly traverse the cycle $B \to C \to D \to B$ to infinitely decrease your total path weight.

It is important to note the distinction between a graph with a negative *cycle* and a graph with negative *edge weights*. If a graph has a negative cycle reachable from the source vertex, then the shortest path solution is negatively infinite. However, a graph with negative edges still has a well-defined shortest path if those edges do not produce a negative cycle. If we change the weight of $\overline{BD}$ from -9 to -7, the weight of the cycle $B \to C \to D \to B$ becomes $3 + 5 + (-7) = +1$. Therefore, it is no longer advantageous to continuously traverse the cycle, and the graph has a well-defined shortest path of $A \to C \to B \to D \to E$.



Nonetheless, the mere existence of negative edges in a graph — regardless of whether a negative cycle exists — restricts the types of shortest path algorithms that we can use. As we will see later on, graphs with negative edges invalidate the greedy-choice property, and thus cannot be solved using algorithms that rely on a greedy approach (such as Dijkstra's algorithm).

25.2 Dijkstra's Algorithm

※ 25.2.1 *Implementing Dijkstra's Algorithm*

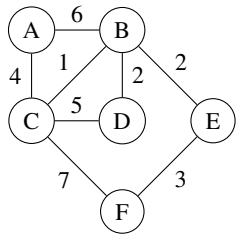
One of the simplest and most efficient shortest path algorithms is **Dijkstra's algorithm**, which utilizes a greedy approach to find the shortest path from a source vertex v_s to *every* other vertex of a weighted graph, as long as the graph has no negative edges. The core idea behind Dijkstra's algorithm is to greedily explore vertices in order of increasing distance from the source vertex. To run Dijkstra's algorithm, we need to keep track of three things for each vertex:

- *Whether the vertex has been visited.* A vertex is visited if all of its neighbors have been fully processed (this will make more sense as we go over an example). Dijkstra's algorithm makes the assumption that, once we fully visit a vertex, its best known distance *must* be optimal, and we do not have to visit it ever again.
- *The best known distance to the vertex from the source vertex.* Throughout the algorithm, we keep track of the best distance known so far to get to each vertex. This value will be used by the algorithm to determine which unvisited vertex we should visit next.
- *The predecessor of the vertex.* This is the vertex that directly precedes the current vertex along the path with the best known distance. This value can be used to reconstruct the shortest path at the end of the algorithm.

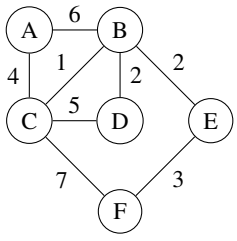
Using this information, Dijkstra's algorithm is implemented as follows:

1. Given a graph, split its vertices into two groups: ones that are visited and ones that are unvisited. At the start of the algorithm, all vertices are marked as unvisited.
2. Assign the source vertex with a best known distance of 0, and assign all other vertices with a best known distance of ∞.
3. Mark the starting vertex as the "current" vertex.
4. Mark the current vertex as visited.
5. Iterate over all unvisited neighbors that are reachable from the current vertex and calculate each neighbor's distance to the source vertex if you pass *through* the current vertex. If this distance is ever better than the best known distance for a neighbor, that neighbor's best known distance is updated to this better solution.
6. Select the unvisited vertex with the smallest best known distance and set it as the new current vertex.
7. Repeat steps 4-6 until either the destination vertex is visited, or the best known distances of the remaining unvisited vertices are all ∞ (which indicates there is no connection between the source vertex and the remaining unvisited vertices). Once you reach this step, the algorithm is complete, and the best known distance of each vertex v is also the weight of the shortest path from the source vertex to v.

To illustrate this process, let's use an example. Consider the following weighted graph, for which we want to find the shortest path from vertex A to vertex F.

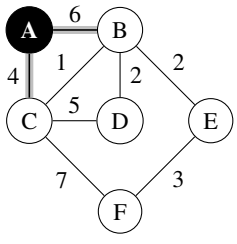


As mentioned, we need to keep track of three things for each vertex: whether it has been visited, its best known distance, and its predecessor. Notice that this is very similar to the table we used when running Prim's algorithm to find a graph's MST! In fact, we can use this same table structure to help us implement Dijkstra's algorithm — in this case, we will define k_v as whether a vertex has been visited, d_v as its best known distance, and p_v as its predecessor. Since vertex A is our starting vertex, we assign d_A to 0 and all other best known distances to ∞.



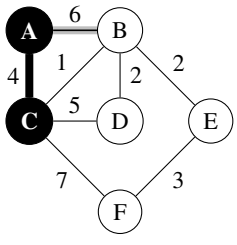
v	k_v	d_v	p_v
A	F	0	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-
F	F	∞	-

First, we consider all of the unvisited neighbors of our starting vertex A and compute the optimal distance required to reach each of them. The unvisited neighbors of A are B and C, which can be reached with distances 6 and 4, respectively. This is better than the current best known distances for these two vertices (∞), so we update their values and set their predecessors to A. We also mark vertex A as visited. *Important note: the best known distances of B and C are being set to 6 and 4 not because the edge weights of $\overline{AB}$ and $\overline{AC}$ are 6 and 4 — instead, it's because the best known distances to B and C from the source vertex are 6 and 4. This will make a bigger difference in later iterations as we move past the starting vertex.*



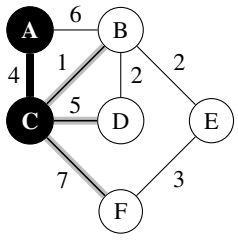
v	k_v	d_v	p_v
A	**T**	0	-
B	F	**6**	A
C	F	**4**	A
D	F	∞	-
E	F	∞	-
F	F	∞	-

Now that vertex A has been visited, we will pick our next vertex by finding the unvisited vertex with the smallest best known distance. Here, vertex B has a best known distance of 6, vertex C has a best known distance of 4, and all remaining unvisited vertices have a best known distance of ∞. The best known distance of vertex C is smallest, so we set C as our new current vertex and mark it as visited.



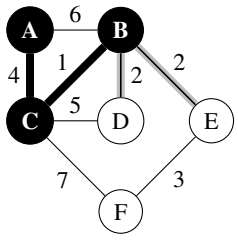
v	k_v	d_v	p_v
A	T	0	-
B	F	6	A
C	**T**	4	A
D	F	∞	-
E	F	∞	-
F	F	∞	-

We will now repeat what we did earlier and iterate over all the unvisited neighbors of vertex C (which are B, D, and F) and compute the best distance to reach these vertices while going through vertex C. Here, the best distance to get from the source vertex to B while passing through vertex C is equal to $d_C + \overline{CB} = 4 + 1 = 5$. This is better than the best known distance of 6 for B so far, so we update B's best known distance to 5 and set its predecessor to C. Similarly, the best distance to get to vertex D through vertex C is $d_C + \overline{CD} = 4 + 5 = 9$ (which is better than D's current best known distance of ∞), and the best distance to get to vertex F through vertex C is $d_C + \overline{CF} = 4 + 7 = 11$ (which is better than F's best known distance of ∞). Thus, we also update D and F's best known distances and predecessors accordingly.



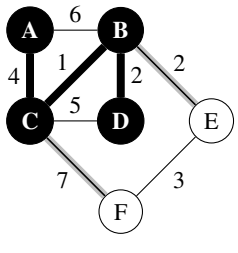
v	k_v	d_v	p_v
A	T	0	-
B	F	**5**	C
C	T	4	A
D	F	**9**	C
E	F	∞	-
F	F	**11**	C

The unvisited vertex with the smallest best known distance is now vertex B, so we set it as our current vertex. We then iterate over its unvisited neighbors, D and E, and update our table accordingly.



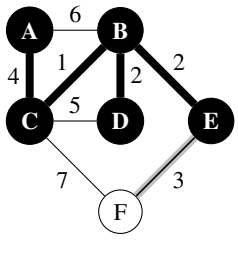
v	k_v	d_v	p_v
A	T	0	-
B	**T**	5	C
C	T	4	A
D	F	**7**	B
E	F	**7**	B
F	F	11	C

Both vertex D and E now have the smallest unvisited tentative distance, so we can choose either of them for our next vertex. Here, we will choose vertex D first; however, since D has no unvisited neighbors, we only need to mark D as visited.



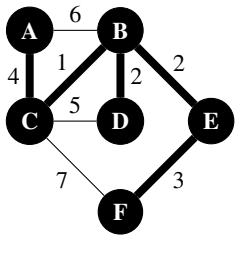
v	k_v	d_v	p_v
A	T	0	-
B	T	5	C
C	T	4	A
D	**T**	7	B
E	F	7	B
F	F	11	C

We then choose E as our next vertex to visit. Vertex E has one unvisited vertex (F), and the best distance to get to vertex F through vertex E is $d_E + \overline{EF} = 7 + 3 = 10$. Therefore, we update the best known distance of F to 10 and its predecessor to E.



v	k_v	d_v	p_v
A	T	0	-
B	T	5	C
C	T	4	A
D	T	7	B
E	**T**	7	B
F	F	**10**	E

Vertex F is the only vertex that remains unvisited, and its best known distance is not ∞, so we mark it as visited and our algorithm is complete. This is the final result of running Dijkstra's algorithm on the graph:



v	k_v	d_v	p_v
A	T	0	-
B	T	5	C
C	T	4	A
D	T	7	B
E	T	7	B
F	**T**	10	E

At this point, the d_v column of the table now gives us the shortest distance from the starting vertex A to every other vertex in the graph (for instance, $d_F = 10$, so the shortest path from A to F has a total weight of 10). We can also use the predecessors to determine the path we took from A to any other node: for instance, F's predecessor is E, E's predecessor is B, B's predecessor is C, and C's predecessor is A, so the shortest path from A to F must be $A \to C \to B \to E \to F$.

Remark: Even though we won't go over an example here, Dijkstra's algorithm on a *directed* graph works the same way. When iterating over the neighbors of a vertex, make sure to consider edge direction and visit only the neighbors that are directly reachable along a directed edge.

Similar to Prim's algorithm, there are two ways to implement Dijkstra's algorithm. These two implementations differ in how the smallest unvisited d_v is found. One implementation relies on a *linear search* of all the vertices to find the unvisited vertex with the smallest d_v. The code for this approach is shown below:

```
1   struct DijkstraData {
2     double d;
3     int32_t p;
4     bool k;
5     DijkstraData()
6       : d{ std::numeric_limits<double>::infinity() }, p{ -1 }, k{ false } {}
7   };
8
9   // graph is in the form of an adjacency matrix
10  int32_t dijkstra(const std::vector<std::vector<int32_t>>& graph, int32_t src, int32_t dest) {
11    std::vector<DijkstraData> dijkstra_table(graph.size());
12    dijkstra_table[src].d = 0;
13    for (size_t count = 0; count < graph.size(); ++count) {
14      // set current vertex as unvisited vertex with smallest tentative distance
15      int32_t min_dist = std::numeric_limits<int32_t>::max();
16      size_t idx = 0;
17      for (size_t i = 0; i < dijkstra_table.size(); ++i) {
18        if (!dijkstra_table[i].k && dijkstra_table[i].d < min_dist) {
19          min_dist = dijkstra_table[i].d;
20          idx = i;
21        } // if
22      } // for i
23
24      if (min_dist == std::numeric_limits<int32_t>::max()) {
25        return -1; // no solution
26      } // if
27
28      // mark idx as visited
29      dijkstra_table[idx].k = true;
30
31      // update table if distance is better
32      for (size_t neighbor = 0; neighbor < graph.size(); ++neighbor) {
33        int32_t new_dist =  dijkstra_table[idx].d + graph[idx][neighbor];
34        if (!dijkstra_table[neighbor].k && new_dist < dijkstra_table[neighbor].d) {
35          dijkstra_table[neighbor].d = new_dist;
36          dijkstra_table[neighbor].p = idx;
37        } // if
38      } // for neighbor
39    } // for count
40
41    return dijkstra_table[dest].d;
42  } // dijkstra()
```

The time complexity of the linear search approach is $\Theta(|V|^2)$, where $|V|$ is the number of vertices in the graph; this is because a linear pass is completed $|V|$ times to find the minimum d_v for every vertex in the graph (lines 17-22). This is ideal if the given graph is dense, since each vertex would have many connecting edges, and iterating over most of the neighbors would be necessary. However, if you are given a sparse graph, each vertex would not have as many connections: this makes a linear traversal over all vertices a wasteful endeavor. In this case, much like with Prim's algorithm, it would be better to use a *min-heap* instead of a linear search to identify the unvisited vertex with the smallest d_v.

The heap-implementation of Dijkstra's algorithm is shown below:

```
1   struct DijkstraData {
2     double d;
3     int32_t p;
4     bool k;
5     DijkstraData()
6       : d{ std::numeric_limits<double>::infinity() }, p{ -1 }, k{ false } {}
7   };
8
9   using AdjList = std::vector<std::vector<std::pair<int32_t, int32_t>>>; // <neighbor, weight>
10  using DistPair = std::pair<int32_t, int32_t>;  // <d_v, v>
11
12  int32_t dijkstra(const AdjList& graph, int32_t src, int32_t dest) {
13    std::vector<DijkstraData> dijkstra_table(graph.size());
14    std::priority_queue<DistPair, std::vector<DistPair>, std::greater<DistPair>> pq;
15    dijkstra_table[src].d = 0;
16    pq.emplace(0, src);
17
18    while (!pq.empty()) {
19      auto [min_dist, idx] = pq.top();
20      pq.pop();
21
22      if (min_dist == std::numeric_limits<int32_t>::max()) {
23        return -1; // no solution
24      } // if
25
26      if (!dijkstra_table[idx].k) {
27        dijkstra_table[idx].k = true;
28        for (auto& neighbor_dist_pair : graph[idx]) {
29          auto [neighbor, dist] = neighbor_dist_pair;
30          int32_t new_dist = dijkstra_table[idx].d + dist;
31          if (new_dist < dijkstra_table[neighbor].d) {
32            dijkstra_table[neighbor].d = new_dist;
33            dijkstra_table[neighbor].p = idx;
34            pq.emplace(new_dist, neighbor);
35          } // if
36        } // for neighbor_dist_pair
37      } // if
38    } // while
39
40    return dijkstra_table[dest].d;
41  } // dijkstra()
```

What is the time complexity of the heap implementation, defined in terms of the number of edges $|E|$ and vertices $|V|$ in the graph? If a binary heap is used as the underlying implementation, it takes $\Theta(\log(|V|))$ time to pop a vertex from the heap, which is done at most once for each of the $|V|$ vertices in the graph. In addition, we iterate over all the neighbors of the current vertex (which takes at most $\Theta(|E|)$ time) and update their best known distances and push them into the heap if needed (where each push takes $\Theta(\log(|V|))$ time). Combining these together, the worst-case time complexity of Dijkstra's algorithm using a min-binary heap is $\Theta(|V|\log(|V|)+|E|\log(|V|))$. Since $|E|$ is significantly larger than $|V|$ in the worst case, we can also express this complexity class as $\Theta(|E|\log(|V|))$ by removing lower order terms.

Remark: Similar to Prim's algorithm, if you use a Fibonacci heap as the underlying implementation of the priority queue (which allows an element's priority to be decreased in amortized $\Theta(1)$ time), the worst-case time complexity of Dijkstra's algorithm drops from $\Theta(|V|\log(|V|)+|E|\log(|V|))$ to $\Theta(|V|\log(|V|)+|E|)$, as the cost of updating a vertex's priority would no longer take $\Theta(\log(|V|))$ time. This is why you might see the time complexity of Dijkstra's as $\Theta(|E|+|V|\log(|V|))$ in other resources; however, we will assume that a binary heap is used unless otherwise specified.

Similar to Prim's algorithm, the heap approach is less performant than the linear search approach for dense graphs. This is because $|E|$ is on the order of $\Theta(|V|^2)$ in a dense graph, so the time complexity of $\Theta(|E|\log(|V|))$ essentially becomes $\Theta(|V|^2\log(|V|))$. This is worse than the linear search time complexity of $\Theta(|V|^2)$.

In summary, you should use the linear search approach if you want to run Dijkstra's algorithm on a dense graph, and the min-heap approach if you want to run it on a sparse graph. The worst-case time complexity of the linear search approach is $\Theta(|V|^2)$, and the worst-case time complexity of the min-heap approach is $\Theta(|E|\log(|V|))$. Notice that this result directly mirrors our analysis of Prim's algorithm! It comes as no surprise that these two algorithms share the same time complexities, as they apply a similar greedy strategy to a graph, even if the work done at each step is slightly different.

Summary of Dijkstra's Algorithm Time Complexities

Implementation Method	Time Complexity
Adjacency matrix, linear search	$\Theta(\|V\|^2)$
Adjacency list, binary heap	$\Theta(\|E\|\log(\|V\|))$
Adjacency list, Fibonacci heap	$\Theta(\|E\|+\|V\|\log(\|V\|))$

※ 25.2.2 *Proving Dijkstra's Correctness (✻)*

Now that we have discussed *how* Dijkstra's algorithm works, it is also worthwhile to take a look at *why* it works. How do we know that Dijkstra's algorithm always returns an optimal solution? To understand why, it is important to understand the core assumption that Dijkstra's makes: *once a vertex is marked as visited, its best known distance from the source vertex must be optimal.* We will use a proof by contradiction to show that this assumption always holds, as long as there are no negative edges.

Let us denote $\delta(v_1, v_n)$ as the length of the actual optimal shortest path from v_1 to v_n. For Dijkstra's to fail, we claim that there must exist some vertex v_k reachable from v_1 that is the first to be marked as visited with a *suboptimal* best known distance from the source vertex v_1: i.e., $d_{v_k} > \delta(v_1, v_k)$. Since a path must exist between v_1 and this first incorrectly marked vertex v_k, we will define a shortest path from v_1 to v_k as $P_{v_1 \to v_k} = \langle v_1, v_2, v_3, \ldots, v_k \rangle$, where k is the number of vertices in this shortest path and v_k is the k^{th} vertex along $P_{v_1 \to v_k}$. There are two key observations we can make here:

1. Since v_k is the *first* vertex to be marked as visited with a suboptimal best known distance, all vertices that precede it in its shortest path (i.e., $v_1, v_2, \ldots, v_{k-1}$) must have a best known distance that is optimal. In other words, $d_{v_1} = \delta(v_1, v_1)$, $d_{v_2} = \delta(v_1, v_2)$, …, and $d_{v_{k-1}} = \delta(v_1, v_{k-1})$.[1]
2. Any subpath $P_{v_i \to v_j} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ of $P_{v_1 \to v_k}$ $(1 \le i \le j \le k)$ must itself be optimal, or we would be able to replace this subpath to get a shorter path from v_1 to v_k (i.e., since there would be a shorter way to get from P_{v_i} to P_{v_j}).

Right before we mark vertex v_k as visited, there are two scenarios that are possible:

1. *Vertex v_k is the first vertex along $P_{v_1 \to v_k}$ that is unvisited.*

In this case, all vertices from v_1 to v_{k-1} have been visited. From our initial claim that v_k is the first vertex to be visited without an optimal best known distance, we know that the best known distance of vertex v_{k-1} (the vertex directly preceding v_k) must be optimal, i.e., $d_{v_{k-1}} = \delta(v_1, v_{k-1})$. Thus, when we update v_k's best known distance while processing vertex v_{k-1}, d_{v_k} is set to a value no larger than $[\delta(v_1, v_{k-1}) + W(v_{k-1}, v_k)]$ (where $W(x, y)$ is defined as the weight of the edge connecting vertices x and y). However, since the edge connecting v_{k-1} and v_k is on a shortest path from v_1 to v_k, the value of $[\delta(v_1, v_{k-1}) + W(v_{k-1}, v_k)]$ must be equal to $\delta(v_1, v_k)$. Therefore, if we explore v_{k-1} before v_k, the value of d_{v_k} cannot be greater than $\delta(v_1, v_k)$. We initially claimed that $d_{v_k} \neq \delta(v_1, v_l)$, so this results in a contradiction.

2. *There exists a vertex v_c, $1 < c < k$, that is the first vertex along $P_{v_1 \to v_k}$ that is unvisited.*

If some other vertex v_c is the first vertex along $P_{v_1 \to v_k}$ that remains unvisited, we know that the predecessor of this vertex v_{c-1} must be visited and that $d_{v_{c-1}} = \delta(v_1, v_{c-1})$. Thus, after exploring vertex v_{c-1}, the best known distance of vertex v_c must be no greater than $d_{v_{c-1}} + W(v_{c-1}, v_c)$. Therefore,

$$d_{v_c} \le d_{v_{c-1}} + W(v_{c-1}, v_c)$$

Since $d_{v_{c-1}} = \delta(v_1, v_{c-1})$, we can rewrite this inequality as

$$d_{v_c} \le \delta(v_1, v_{c-1}) + W(v_{c-1}, v_c)$$

Notice here that $\delta(v_1, v_{c-1}) + W(v_{c-1}, v_c)$ is the distance of the path $P_{v_1 \to v_c}$, which is a subpath of $P_{v_1 \to v_k}$. Because we defined $P_{v_1 \to v_k}$ as an optimal path from v_1 to v_k at the beginning of the proof, we can use our second observation to conclude that the distance of this subpath must itself be optimal between v_1 and v_c. In other words, $\delta(v_1, v_{c-1}) + W(v_{c-1}, v_c) = \delta(v_1, v_c)$.

$$d_{v_c} \le \delta(v_1, v_{c-1}) + W(v_{c-1}, v_c) \quad \textit{and} \quad \delta(v_1, v_{c-1}) + W(v_{c-1}, v_c) = \delta(v_1, v_c) \quad \textit{implies} \quad d_{v_c} \le \delta(v_1, v_c)$$

Under the assumption that there are *no negative edges in the graph*, $\delta(v_1, v_c)$ cannot be greater than $\delta(v_1, v_k)$, since v_c precedes v_k in our shortest path (and thus there must be at least one more non-negative edge in the graph after v_c). Using our initial claim that d_{v_k} is not optimal, we would get the following inequality.

$$d_{v_c} \le \delta(v_1, v_c) \quad \textit{and} \quad \delta(v_1, v_c) < \delta(v_1, v_k) \quad \textit{and} \quad \delta(v_1, v_k) < d_{v_k} \quad \textit{implies} \quad d_{v_c} < d_{v_k}$$

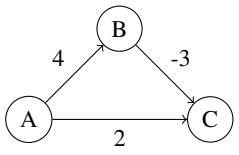
From here, we can see that $d_{v_c} < d_{v_k}$ if v_k were marked as visited with a suboptimal d_{v_k}. However, this is a problem, since Dijkstra's algorithm would never mark v_k as visited before v_c if $d_{v_c} < d_{v_k}$. Therefore, it is impossible for v_k to be marked as visited before any preceding vertex v_c if d_{v_k} were not optimal, which contradicts our claim that v_k is marked as visited with a suboptimal best known distance. From this proof, we have shown that vertex v_k — or a vertex whose best known distance is suboptimal when marked as visited — cannot possibly exist. Thus, we have proved the correctness of Dijkstra's algorithm.

※ 25.2.3 *Why Dijkstra's Algorithm Fails on Negative Edges*

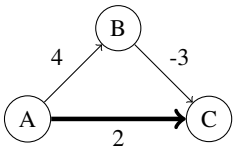
However, once we add negative edges into the equation, the proof falls apart, and Dijkstra's algorithm can no longer guarantee an optimal solution. Previously, when we marked a vertex as visited, we were sure that no other paths could yield a better solution — this is because any additional paths not yet encountered by the algorithm would add a non-negative amount of weight, and thus cannot be optimal. When we allow negative edges, this assumption is no longer true; even after marking a vertex as visited, there is still the off chance that another unencountered path has a negative edge that could give us a better solution.

[1] For this observation to hold, we must also show that d_v cannot be *better* than $\delta(s, v)$ for any vertex v. A quick proof would show that, since the best known distance of any vertex is computed using the best known distances of earlier vertices (all the way back to the source vertex), it would be impossible to set any vertex's best known distance to a value better than the actual distance required to reach it.

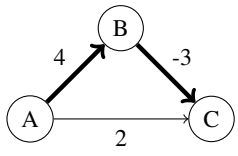
For example, consider the following graph, where we want to find the shortest path from A to C:



Dijkstra's algorithm would greedily select edge $\overline{AC}$ for a distance of 2, since it is better than the distance of 4 from edge $\overline{AB}$. As a result, the algorithm would mark vertex C as visited with a best known distance of 2… however, this is not the optimal solution! If we had gone through vertex B instead, we would have ended up with a better distance of $4 + (-3) = 1$. However, Dijkstra's algorithm failed to find this optimal path because it did not know about edge $\overline{BC}$ and that its weight was negative enough to overcome the seemingly suboptimal choice of edge $\overline{AB}$.



Dijkstra Solution



Optimal Solution

In general, if there are multiple ways to reach a vertex in a graph, and at least one of those ways involves a negative edge, then Dijkstra's algorithm could potentially mark that vertex as visited before encountering the negative edge. Since Dijkstra's uses a greedy approach, it never reconsiders this decision and assumes that the solution it found is optimal, using it to construct the solutions of subsequent vertices that are encountered later on. By the time the algorithm discovers a negative edge that improves the solution of a previously visited vertex, it is too late! This is why Dijkstra's algorithm is avoided if a graph has negative edges. If you do have a graph with negative edges, it is better to use a different algorithmic approach that can handle this scenario (specifically dynamic programming, which will be discussed in the next section).

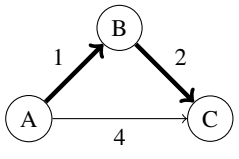
※ 25.2.4 *Changing Edge Weights in the Shortest Path*

Example 25.1 Let P be the shortest path from vertex s to vertex t of a given graph. If the weight of every edge in the graph is *multiplied* by the same positive constant k (i.e., the weight of each edge is changed from x to kx), is P still guaranteed to be the shortest path in the graph? Or could the shortest path change?

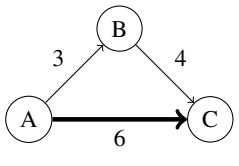
The shortest path does not change if you multiply every edge by a constant factor. We can show this using a proof by contradiction: suppose there exists a different path P' from s to t that is shorter than P after all the edges are multiplied by k. However, if we divide all the edges in P' by k, we would see that P' must have been shorter before all the edges were multiplied. Therefore, P would not have been the shortest path, which results in a contradiction. In general, multiplying all edges by a constant factor does not change the shortest path because multiplication is distributive (unlike addition, which we will see in the next example).

Example 25.2 Let P be the shortest path from vertex s to vertex t of a given graph. If the weight of every edge in the graph is *incremented* by the same positive constant k (i.e., the weight of each edge is changed from x to $k + x$), is P still guaranteed to be the shortest path in the graph? Or could the shortest path change?

Unlike multiplication, addition does not guarantee that the shortest path will remain the same. This can be shown using a quick example: in the following graph, the shortest path from A to C is to go through B for a total weight of $1 + 2 = 3$.



However, if we add 2 to every edge, the shortest path now travels from A to C directly for a total weight of 6.



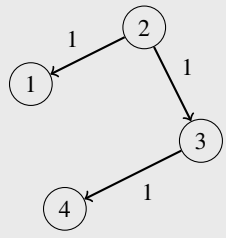
The reason for this is that addition, unlike multiplication, does not distribute weight proportionally across all edges in the graph. This ends up punishing paths with more edges significantly more than paths with fewer edges, as longer paths have more edges on which additional weight can be applied. As a result, a path that is initially optimal may be overtaken by a path with fewer edges if the same weight is applied equally to every edge of the graph.

※ 25.2.5 *Solving Problems Using Dijkstra's Algorithm*

Example 25.3 You are given a network of n nodes, labeled 1 to n, and a vector of travel times `times` as **directed** edges, such that the value of `times[i] = (u, v, w)`, where `u` is the source node, `v` is the target node, and `w` is the time it takes for a signal to travel from source to target. Write a function that takes in a vector of delay times, the number of nodes n, and a starting node *src*, and returns the time it would take for all nodes to receive a signal that is sent from node k. If it is not possible for all nodes to receive the signal, return `-1`.

```
int32_t network_delay_time(const std::vector<std::vector<int32_t>>& times, int32_t n, int32_t src);
```

Example: Given `times = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]`, `n = 4`, and `src = 2`, you would return `2`. This is because it takes two units of time for all nodes in the network to receive the signal (where node 2 to 4 takes the longest).



This is a shortest-path problem on a weighted graph, so Dijkstra's algorithm can be used to solve it. First, we will need to convert our input into either an adjacency list or adjacency matrix so that Dijkstra's can be applied. Whether a list or matrix is chosen depends on the input (e.g., whether the graph is dense or sparse, time complexity constraints, etc.), but since we are not given any additional information in the example, we will implement our first solution using an adjacency list. After converting the input into an adjacency list, we can run the binary heap implementation of Dijkstra's algorithm covered previously, as shown below. Then, when the algorithm completes, we can look through each of the vertices to identify which one requires the most time to receive a message from the source vertex (using the final value of d_v).

```
1  struct DijkstraData {
2    int32_t d;
3    int32_t p;
4    bool k;
5    DijkstraData()
6      : d{ std::numeric_limits<int32_t>::max() }, p{ -1 }, k{ false } {}
7  };
8
9  using DistPair = std::pair<int32_t, int32_t>;  // <d_v, v>
10
11 int32_t network_delay_time(const std::vector<std::vector<int32_t>>& times, int32_t n, int32_t src) {
12   // convert times vector into an adjacency list
13   std::unordered_map<int32_t, std::vector<std::pair<int32_t, int32_t>>> graph;
14   for (auto& time_vec : times) {
15     graph[time_vec[0]].emplace_back(time_vec[1], time_vec[2]);
16   } // for time_vec
17
18   std::vector<DijkstraData> dijkstra_table(n + 1);
19   std::priority_queue<DistPair, std::vector<DistPair>, std::greater<DistPair>> pq;
20   dijkstra_table[src].d = 0;
21   pq.emplace(0, src);
22
23   while (!pq.empty()) {
24     auto [min_dist, idx] = pq.top();
25     pq.pop();
26     if (!dijkstra_table[idx].k) {
27       dijkstra_table[idx].k = true;
28       for (auto& neighbor_dist_pair : graph[idx]) {
29         auto [neighbor, dist] = neighbor_dist_pair;
30         int32_t new_dist = dijkstra_table[idx].d + dist;
31         if (new_dist < dijkstra_table[neighbor].d) {
32           dijkstra_table[neighbor].d = new_dist;
33           dijkstra_table[neighbor].p = idx;
34           pq.emplace(new_dist, neighbor);
35         } // if
36       } // for neighbor_dist_pair
37     } // if
38   } // while
39
40   // return largest distance in the final Dijkstra table
41   int32_t max_dist = std::numeric_limits<int32_t>::min();
42   for (auto it = dijkstra_table.begin() + 1; it != dijkstra_table.end(); ++it) {
43     max_dist = std::max(max_dist, it->d);
44   } // for data
45   return max_dist == std::numeric_limits<int32_t>::max() ? -1 : max_dist;
46 } // network_delay_time()
```

We can write a similar solution using an adjacency matrix and linear search, as shown (this code is nearly identical to the linear search solution introduced in our initial discussion of Dijkstra's algorithm, where we perform a linear search over the vertices to determine which unvisited vertex has the minimal distance, rather than using a heap).

```
struct DijkstraData {
  int32_t d;
  int32_t p;
  bool k;
  DijkstraData()
    : d{ std::numeric_limits<int32_t>::max() }, p{ -1 }, k{ false } {}
};

int32_t network_delay_time(const std::vector<std::vector<int32_t>>& times, int32_t n, int32_t src) {
  // convert times vector into an adjacency matrix
  std::vector<std::vector<int32_t>> graph(n + 1, std::vector<int32_t>(n + 1, -1));
  for (auto& time_vec : times) {
    graph[time_vec[0]][time_vec[1]] = time_vec[2];
  } // for time_vec

  std::vector<DijkstraData> dijkstra_table(n + 1);
  dijkstra_table[src].d = 0;
  for (size_t count = 0; count < graph.size(); ++count) {
    int32_t min_dist = std::numeric_limits<int32_t>::max();
    size_t idx = 0;
    for (size_t i = 0; i < dijkstra_table.size(); ++i) {
      if (!dijkstra_table[i].k && dijkstra_table[i].d < min_dist) {
        min_dist = dijkstra_table[i].d;
        idx = i;
      } // if
    } // for i
    dijkstra_table[idx].k = true;
    for (size_t neighbor = 0; neighbor < graph.size(); ++neighbor) {
      int32_t new_dist = dijkstra_table[idx].d + graph[idx][neighbor];
      if (!dijkstra_table[neighbor].k && graph[idx][neighbor] != -1
          && new_dist < dijkstra_table[neighbor].d) {
        dijkstra_table[neighbor].d = new_dist;
        dijkstra_table[neighbor].p = idx;
      } // if
    } // for neighbor
  } // for count

  // return largest distance in the final Dijkstra table
  int32_t max_dist = std::numeric_limits<int32_t>::min();
  for (auto it = dijkstra_table.begin() + 1; it != dijkstra_table.end(); ++it) {
    max_dist = std::max(max_dist, it->d);
  } // for data
  return max_dist == std::numeric_limits<int32_t>::max() ? -1 : max_dist;
} // network_delay_time()
```

The time complexity of the adjacency list solution is $\Theta(|V|+|E|)$, and the time complexity of the adjacency matrix solution is $\Theta(|V|^2)$.

25.3 Bellman-Ford Algorithm (*)

If negative edges exist in a graph, Dijkstra's algorithm cannot be used, since there is no guarantee that the greedy choice will always lead to an optimal solution. Instead, we will have to rely on a different algorithm to find a shortest path. One such algorithm is the **Bellman-Ford algorithm**, which uses dynamic programming to solve the single-source shortest path problem.

The Bellman-Ford algorithm uses the following recurrence relation, where $d(v,k)$ represents the weight of the shortest path from the source vertex s to v using at most k edges. If $k = 0$ and $v = s$ (source vertex is also the destination), then the weight of the shortest path is trivially 0 since no edges are required to get from a vertex to itself. Similarly, if $k = 0$ and $v \neq s$, then the weight of the shortest path is ∞, since there is no way to travel from one vertex to another using 0 edges. If neither of these cases apply, then the algorithm would take the *smaller* of the following to compute $d(v,k)$:

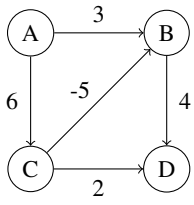
1. *The weight of the shortest path from s to v using at most k - 1 edges, i.e., d(v, k - 1).* This is the smaller value of the two if allowing a k^{th} edge in the path from s to v does not improve the best known solution.
2. *The weight of the shortest path from s to v using exactly k edges.* This is equal to the minimum weight of the shortest path from s to u using at most $k-1$ edges, plus the weight of u to v, for all vertices u such that $u \rightarrow v$ exists as an edge in the graph, (i.e., $\min_{(u \rightarrow v) \in E}\{d(u,k-1)+W(u,v)\}$). This is the smaller value if allowing a k^{th} edge in the path from s to v improves the solution.

$$d(v,k) = \begin{cases} 0, & \text{if } k = 0 \text{ and } v = s \\ \infty, & \text{if } k = 0 \text{ and } v \neq s \\ \min(d(v,k-1), \ \min_{(u \rightarrow v) \in E}\{d(u,k-1)+W(u,v)\}), & \text{otherwise} \end{cases}$$

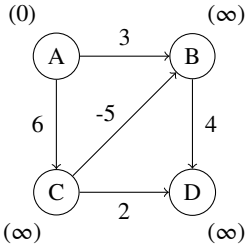
The Bellman-Ford algorithm provides an optimized implementation of this recurrence relation by completing the following steps:

1. Given $|V|$ vertices, initialize a table of size $|V|$ that stores
   - the *best known distance* of each vertex from the given source vertex
   - the *predecessor vertex*, which precedes the current vertex along the path with the best known distance
2. Iterate over *all* edges of the graph and compute the distance to reach the edge's terminal vertex along a path that passes through that edge (updating any best known distances if necessary). *Note that this is different from Dijkstra's algorithm, which only updates the best known distances of vertices that are locally reachable from the current vertex. We need to consider* ***all*** *edges with each iteration to handle the possibility of negative edges later on that may improve our solution.*
3. Repeat step 2 a total of $|V| - 1$ times. After doing this, the table now stores the optimal distance from the source vertex to every other vertex in the graph.

To visualize this process, consider the following graph, for which we want to find the shortest path from A to D.



First, we will instantiate a table that keeps track of each vertex's best known distance (d_v) as well as its predecessor (p_v). From the base case, the source vertex has its best known distance set to 0, while all others have theirs set to ∞.



v	d_v	p_v
A	0	-
B	∞	-
C	∞	-
D	∞	-

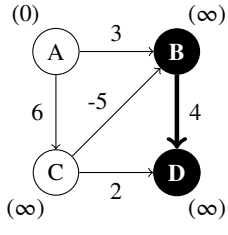
Next, we will iterate over all the edges of the graph $|V| - 1$ times and determine if each edge yields a better solution for its terminal vertex. In other words, for each edge $u \to v$, we calculate if the best known distance to u, plus the weight of the edge $u \to v$, improves the best known distance to v. The edges can be processed in any order, so for our example, we will process the edges in the following arbitrary ordering:

$$\overrightarrow{BD}, \overrightarrow{AB}, \overrightarrow{CD}, \overrightarrow{CB}, \overrightarrow{AC}$$

We will now begin iterating over the edges of the graph. Since the number of vertices is $|V| = 4$, we will perform $|V| - 1 = 3$ iterations.

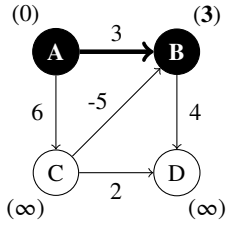
Iteration 1:

First, we will consider edge $\overrightarrow{BD}$. The current best known distance to B is ∞, and the weight of the edge is 4. Thus, the best known distance to D using edge $\overrightarrow{BD}$ is "$\infty + 4$" (or just ∞), which is not better than the current best known distance to D of ∞. Therefore, nothing in the table gets updated at this step.



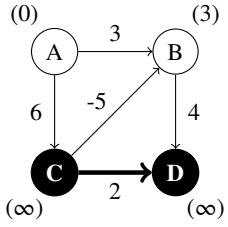
v	d_v	p_v
A	0	-
B	∞	-
C	∞	-
D	∞	-

Next, we will consider edge $\overrightarrow{AB}$. The current best known distance to A is 0, and the weight of the edge is 3. Thus, the best known distance to B using edge $\overrightarrow{AB}$ is $0 + 3$, which is better than the current best known distance to B of ∞. The best known distance to B is thereby updated to 3, and its predecessor is set to A.



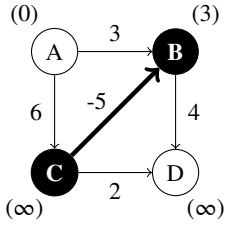
v	d_v	p_v
A	0	-
B	**3**	A
C	∞	-
D	∞	-

Next, we will consider edge $\overrightarrow{CD}$. The current best known distance to C is ∞, and the weight of the edge is 2. Thus, the best known distance to B using edge $\overrightarrow{CD}$ is "$\infty + 2$" (or just ∞), which is not better than the current best known distance to D of ∞. Therefore, nothing in the table gets updated at this step.



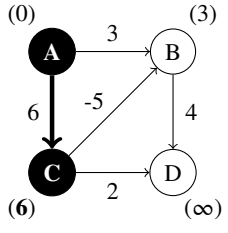
v	d_v	p_v
A	0	-
B	3	A
C	∞	-
D	∞	-

Next, we will consider edge $\overrightarrow{CB}$. The current best known distance to B is ∞, and the weight of the edge is -5. Thus, the best known distance to B using edge $\overrightarrow{CB}$ is "$\infty - 5$" (or just ∞), which is not better than the best known distance to B of 3. Therefore, nothing in the table gets updated at this step.



v	d_v	p_v
A	0	-
B	3	A
C	∞	-
D	∞	-

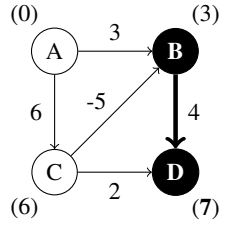
Lastly, we will consider edge $\overrightarrow{AC}$. The current best known distance to A is 0, and the weight of the edge is 6. Thus, the best known distance to C using edge $\overrightarrow{AC}$ is $0 + 6$, which is better than the best known distance to C of ∞. The best known distance to C is thereby updated to 6, and its predecessor is set to A.



v	d_v	p_v
A	0	-
B	3	A
C	**6**	***A***
D	∞	-

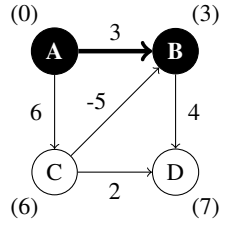
Iteration 2:

We will repeat the same process, this time using the values obtained from iteration 1. First, we consider edge $\overrightarrow{BD}$. The current best known distance to B is 3, and the weight of the edge is 4. Thus, the best known distance to D using edge $\overrightarrow{BD}$ is 7, which is better than the current best known distance to D of ∞. The best known distance to D is updated to 7, and its predecessor set to B.



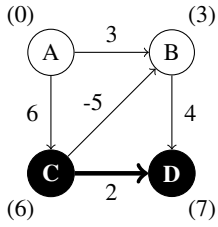
v	d_v	p_v
A	0	-
B	3	A
C	6	A
D	**7**	***B***

Next, we will consider edge $\overrightarrow{AB}$. The current best known distance to A is 0, and the weight of the edge is 3. Thus, the best known distance to B using edge $\overrightarrow{AB}$ is still 3, so nothing in the table gets updated at this step.



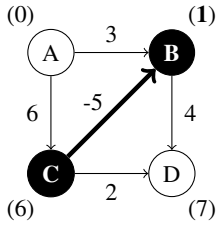
v	d_v	p_v
A	0	-
B	3	A
C	6	A
D	7	B

Next up is edge $\overrightarrow{CD}$. The current best known distance to C is 6, and the weight of the edge is 2. Thus, the best known distance to D using edge $\overrightarrow{CD}$ is 8. This is not better than the current best known distance to D of 7, so nothing gets updated at this step.



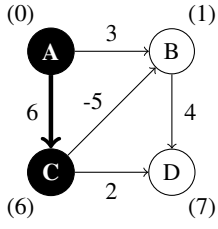
v	d_v	p_v
A	0	-
B	3	A
C	6	A
D	7	B

Next, we will consider edge $\overrightarrow{CB}$. The current best known distance to C is 6, and the weight of the edge is -5. Thus, the best known distance to B using edge $\overrightarrow{CB}$ is 1. This is better than the current best known distance to B of 3, so B's best known distance is updated to 1, and its predecessor is updated to C.



v	d_v	p_v
A	0	-
B	**1**	C
C	6	A
D	7	B

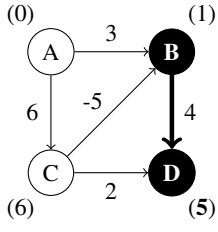
Lastly, we will consider edge $\overrightarrow{AC}$. The current best known distance to A is 0, and the weight of the edge is 6. Thus, the best known distance to C using edge $\overrightarrow{AC}$ is still 6, so nothing gets updated at this step.



v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	7	B

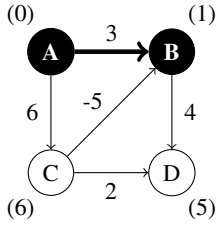
Iteration 3:

We will now complete one final iteration of the edges of the graph, using the values we obtained from iteration 2. First, we will look at edge $\overrightarrow{BD}$. The current best known distance to B is 1, and the weight of the edge is 4. Thus, the best known distance to D using edge $\overrightarrow{BD}$ is 5, which is better than the current best known distance to D of 7. The best known distance to D is thereby updated to 5, and its predecessor is set to B.



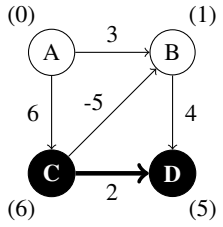
v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	**5**	B

Next, we will consider edge $\overrightarrow{AB}$. The current best known distance to A is 0, and the weight of the edge is 3. Thus, the best known distance to B using edge $\overrightarrow{AB}$ is still 3, so nothing gets updated at this step.



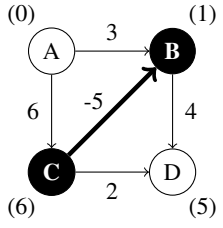
v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	5	B

Next, we will consider edge $\overrightarrow{CD}$. The current best known distance to C is 6, and the weight of the edge is 2. Thus, the best known distance to D using edge $\overrightarrow{CD}$ is 8. This is not better than the current known distance to D of 5, so nothing gets updated at this step.



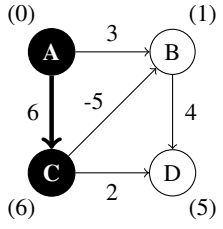
v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	5	B

Next, we will consider edge $\overrightarrow{CB}$. The current best known distance to C is 6, and the weight of the edge is -5. Thus, the best known distance to B using edge $\overrightarrow{CB}$ is still 1, so nothing gets updated at this step.



v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	5	B

Lastly, we will consider edge $\overrightarrow{AC}$. The current best known distance to A is 0, and the weight of the edge is 6. Thus, the best known distance to C using edge $\overrightarrow{AC}$ is still 6, so nothing gets updated at this step.



v	d_v	p_v
A	0	-
B	1	C
C	6	A
D	5	B

After the third iteration, the values in our table have converged to the optimal solution, and we can conclude from the table that the shortest path from A to D has a total weight of 5, the shortest path from A to B has weight 1, and the shortest path from A to C has weight 6. How do we know that the table must be optimal after the third iteration? Notice that, after the k^{th} iteration, we end up knowing the minimum distance to any vertex *when restricted to paths of length at most k*. For instance, the best known distance to D is 7 after the second iteration — this indicates that the shortest path to D has a weight of 7 if we are limited to paths with a length of at most 2. This is because, for any shortest path $v_0 \to v_1 \to \ldots \to v_k$, the distance to v_1 is correctly computed after the first iteration, the distance to v_2 is correctly computed after the second iteration, and so on. Since the maximum possible length of a shortest path is $|V|-1$, our solution must be optimal after $|V|-1$ iterations, which in this example is 4 - 1 = 3.

Just because three iterations are needed to guarantee an optimal solution does not mean that all three iterations actually improve our solution. In fact, the order of edges processed in our example was actually a worst-case ordering, as all three iterations were required before the optimal distance to our destination vertex could be known. In the best case, only a single iteration of the edges is needed to discover the optimal solution, which happens if we iterate over the edges in a way that maintains the order of edges in the optimal path (i.e., $\overrightarrow{AC}$, then $\overrightarrow{CB}$, then $\overrightarrow{BD}$). However, this requires us to know what the optimal path is in the first place, and finding an optimal ordering beforehand is not feasible in practice.[2]

There is also another reason why all $|V|-1$ iterations may still be necessary, regardless of the order of edges chosen: the detection of negative cycles. Since Bellman-Ford can be used on a graph with negative edges, it needs to handle the case where a negative cycle exists in a given graph. To detect the presence of a negative cycle, simply perform an additional iteration after the $(|V|-1)^{\text{th}}$ iteration — if any best known distance is improved during this additional iteration, that means we have found a shortest path that is longer than $|V|-1$, which can only happen if a negative cycle exists in the graph.

[2]The exception, however, is if you are given a directed, acyclic graph (DAG). In such a case, you can find the shortest path in $\Theta(|V|+|E|)$ time by topologically sorting the graph, and then iterating over the vertices (and processing their edges) in topological order.

An implementation of Bellman-Ford is shown below:

```
1   struct BFData {
2     double d;
3     int32_t p;
4     BFData()
5       : d{ std::numeric_limits<double>::infinity() }, p{ -1 } {}
6   };
7
8   using AdjList = std::vector<std::vector<std::pair<int32_t, int32_t>>>; // <neighbor, weight>
9
10  int32_t bellman_ford(const AdjList& graph, int32_t src, int32_t dest) {
11    std::vector<BFData> weight_table(graph.size());
12    weight_table[src].d = 0;
13
14    // iterate over all edges |V|-1 times
15    for (int32_t iteration = 1; iteration < graph.size(); ++iteration) {
16      for (size_t idx = 0; idx < graph.size(); ++idx) {
17        for (const auto& [neighbor, weight] : graph[idx]) {
18          if (weight_table[idx].d + weight < weight_table[neighbor].d) {
19            weight_table[neighbor].d = weight_table[idx].d + weight;
20            weight_table[neighbor].p = idx;
21          } // if
22        } // for neighbor, weight
23      } // for idx
24    } // for iteration
25
26    // check for negative weight cycles by completing one more iteration
27    // if any weight improves, then we have a negative weight cycle
28    for (size_t idx = 0; idx < graph.size(); ++idx) {
29      for (const auto& [neighbor, weight] : graph[idx]) {
30        if (weight_table[idx].d + weight < weight_table[neighbor].d) {
31          throw std::invalid_argument("Graph contains negative weight cycle");
32        } // if
33      } // for neighbor, weight
34    } // for idx
35
36    return weight_table[dest].d;
37  } // bellman_ford()
```

The Bellman-Ford algorithm iterates over all the edges of a graph $\Theta(|V|)$ times and, for each edge, computes whether it improves the best known distance to a vertex of our graph. Each of these computations (which happen on lines 16-20) takes a constant amount of time and is done a total of $\Theta(|E|)$ times per iteration, where $|E|$ is the number of edges in the graph. Therefore, the overall time complexity of Bellman-Ford is $\Theta(|V||E|)$. In addition, since the algorithm initializes a table of size $\Theta(|V|)$ to keep track of the distances to each vertex, the auxiliary space used by Bellman-Ford is $\Theta(|V|)$.

Overall, from an asymptotic runtime perspective, the Bellman-Ford algorithm is not as efficient as Dijkstra's algorithm. However, that is to be expected, since Dijkstra's operates using a greedy approach and only considers locally optimal edges, while Bellman-Ford considers all edges during a single iteration. The main benefit of Bellman-Ford is its versatility: you can use it to solve the single-source shortest path problem for any graph without negative cycles, something that Dijkstra's cannot guarantee.

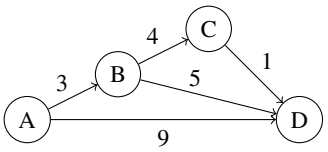
Remark: You may have noticed that the example we used for Bellman-Ford involves a directed graph. Can Bellman-Ford also be used for an undirected graph? It depends. Bellman-Ford can only be used on an undirected graph *if there are no negative edges present*. However, this is true for all shortest-path algorithms that can handle negative edges. This is because a negative edge in an undirected graph is inherently a negative cycle! For example, if there is an undirected edge from vertex A to B with a weight of -1, you could continuously travel from A to B and then back to A to infinitely reduce your total weight.

However, if an undirected graph does have no negative edges, it also means that Dijkstra's algorithm would work as well. Since Dijkstra's algorithm is asymptotically faster, Bellman-Ford is typically an inferior choice in these situations. That being said, there are cases where Bellman-Ford is a preferable choice, such as in environments that require distributed computation (which is why you will see Bellman-Ford again if you ever take a networking class, but that is a topic for another day). In addition, even if a graph does have a negative cycle, Bellman-Ford can be flexibly adapted to identify where these negative cycles exist, as well as the shortest paths to vertices that are not affected by these negative cycles.

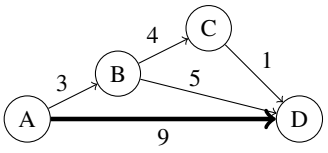
25.4 Floyd-Warshall Algorithm (⁎)

Both Dijkstra's algorithm and Bellman-Ford are designed to solve the single-source shortest path problem, in that they find the shortest path from a given source vertex to every other vertex of a graph. However, there are situations where finding the shortest path to a *single* vertex is not enough. This leads us a problem known as the **all-pairs shortest path problem**, which seeks to find the shortest path between *all* pairs of vertices in a graph. One obvious approach to this problem is to run a single-source algorithm like Dijkstra's algorithm once for every vertex. However, this approach does not work in all situations, such as if a given graph contains negative edges. In the next two sections, we will discuss two additional algorithms that can be used to solve the all-pairs shortest path problem, each with its own ideal use case.

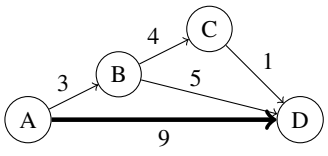
The first of these algorithms is the **Floyd-Warshall algorithm**, which applies dynamic programming to find the shortest path between all pairs of vertices in a weighted graph. Floyd-Warshall is best suited for dense graphs, and it does support graphs with negative edges (as long as there are no negative cycles). The key idea behind Floyd-Warshall is to use *intermediate vertices* to build up the optimal paths between all pairs of vertices. To illustrate how this works, consider the following graph:

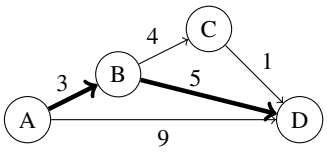


Let's suppose we want to find the shortest path from vertex A to D, under the constraint that we can use zero intermediate vertices. In this case, the answer is trivial: if we cannot travel through any intermediate vertices from A to D, then the shortest path must be the weight of edge $\overrightarrow{AD}$, or 9. We will denote this path as $P_0(A, D)$, which represents the optimal path that uses at most zero intermediate vertices to get from A to D.



Now, let's loosen our constraint and allow the shortest path to go through at most one intermediate vertex, B. With this change, we now have two choices: we can either route through vertex B to get to D, or we could travel to D directly. The optimal path would therefore be the smaller of these two choices.

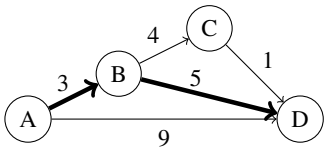


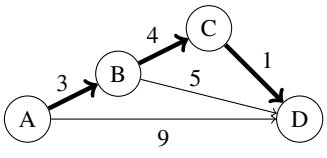


$$\underbrace{P_1(A,D)}_{\text{shortest path from } A \text{ to } D \text{ allowing one intermediate vertex } \{B\}} = \min(\underbrace{P_0(A,D)}_{\text{shortest path from } A \text{ to } D \text{ using zero intermediate vertices}}, \underbrace{P_0(A,B) + P_0(B,D)}_{\text{shortest path from } A \text{ to } D \text{ routing through vertex } B})$$

If we continued this strategy and allowed the shortest path to go through at most two intermediate vertices, B and C, we would be able to directly compute this new shortest path using the solution we computed previously. If we add C to our list of allowable vertices, our best solution would be the minimum of:

- the shortest path when our path is routed through vertex C
- the shortest path when we were only allowed to route through vertex B (this is smaller if routing through C does not produce a better solution)



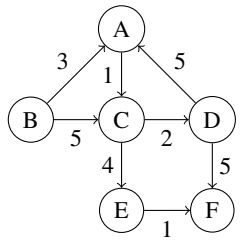


$$\underbrace{P_2(A,D)}_{\text{shortest path from } A \text{ to } D \text{ allowing two intermediate vertices } \{B,C\}} = \min(\underbrace{P_1(A,D)}_{\text{shortest path from } A \text{ to } D \text{ using one intermediate vertex } \{B\}}, \underbrace{P_1(A,C) + P_1(C,D)}_{\text{shortest path from } A \text{ to } D \text{ routing through vertex } C})$$

As you can see, we could continue this process for all other vertices that exist in our graph, adding them one-by-one to a set of allowable intermediate vertices, and then computing if a shorter path can be obtained by routing through the newest vertex in our allowable set. This forms the core recurrence relation for the Floyd-Warshall algorithm. Given a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$, we can use the following recurrence to compute $P_k(i,j)$, which is the shortest path from i to j that is only allowed to go through intermediate vertices in the subset $V_k = \{v_1, v_2, \ldots, v_k\}$ for $0 \leq k \leq n$, for all pairs (i,j):

$$P_k(i,j) = \begin{cases} W(i,j), & \text{if } k = 0 \\ \min(P_{k-1}(i,j), \quad P_{k-1}(i,k) + P_{k-1}(k,j)), & \text{otherwise} \end{cases}$$

Let's look at the Floyd-Warshall algorithm in action, using the following graph. We will represent the graph in the form of a adjacency matrix[3], where ∞ is used to denote that no direct path exists between two vertices. We will also keep track of a matrix of predecessors so that we can reconstruct the shortest path between any two vertices at the end of the algorithm.



$V_0 = \{\}$

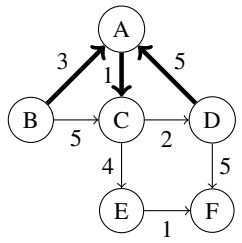
	A	B	C	D	E	F
A	0	∞	1	∞	∞	∞
B	3	0	5	∞	∞	∞
C	∞	∞	0	2	4	∞
D	5	∞	∞	0	∞	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	B	C	D	E	F
A	–	–	*A*	–	–	–
B	*B*	–	*B*	–	–	–
C	–	–	–	*C*	*C*	–
D	*D*	–	–	–	–	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

First, we will add vertex A to our set of allowable intermediate vertices. We will then iterate over all pairs of vertices in our graph and compute if routing through vertex A produces a solution better than what we know so far for each pair. In this case, the shortest paths between for the following two pairs are improved if we are allowed to visit A as an intermediate vertex:

- (B,C): previously, the best known path between B and C had a weight of $P_0(B,C) = 5$, but now the pair has a better path with weight $P_0(B,A) + P_0(A,C) = 3 + 1 = 4$.
- (D,C): previously, the best known path between D and C had a weight of $P_0(D,C) = \infty$, but now the pair has a better path with weight $P_0(D,A) + P_0(A,C) = 5 + 1 = 6$.

The values of (B,C) and (D,C) are thereby updated in the matrix to reflect these newly improved paths.

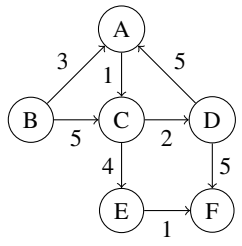


$V_1 = \{A\}$

	A	B	C	D	E	F
A	0	∞	1	∞	∞	∞
B	3	0	**4**	∞	∞	∞
C	∞	∞	0	2	4	∞
D	5	∞	**6**	0	∞	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	B	C	D	E	F
A	–	–	*A*	–	–	–
B	*B*	–	***A***	–	–	–
C	–	–	–	*C*	*C*	–
D	*D*	–	***A***	–	–	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

Next, we will add vertex B to our set of allowable intermediate vertices. We then iterate over all pairs of vertices in our graph and compute if routing through vertex B produces a solution better than what we know so far for each pair. Here, there exists no pair that has their path improved if B is allowed as an intermediate vertex, so nothing gets updated during this iteration.



$V_2 = \{A, B\}$

	A	B	C	D	E	F
A	0	∞	1	∞	∞	∞
B	3	0	4	∞	∞	∞
C	∞	∞	0	2	4	∞
D	5	∞	6	0	∞	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

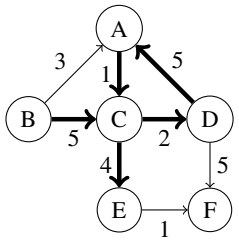
	A	B	C	D	E	F
A	–	–	*A*	–	–	–
B	*B*	–	*A*	–	–	–
C	–	–	–	*C*	*C*	–
D	*D*	–	*A*	–	–	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

[3]Even though Floyd-Warshall is best designed for dense graphs, a sparse graph is used in this example just to make things easier to follow. This is why we are still using an adjacency matrix to represent the graph, despite the graph not being dense.

Next, we will add vertex C to our set of allowable intermediate vertices. Again, we will iterate over all pairs of vertices in our graph to see if any pair has their shortest known path improved when C is permitted as an intermediate vertex. During this iteration, there are five pairs that have a better solution:

- (A, D): previously, the best known path between A and D had a weight of $P_2(A, D) = \infty$, but now the pair has a better path with weight $P_2(A, C) + P_2(C, D) = 1 + 2 = 3$.
- (A, E): previously, the best known path between A and E had a weight of $P_2(A, E) = \infty$, but now the pair has a better path with weight $P_2(A, C) + P_2(C, E) = 1 + 4 = 5$.
- (B, D): previously, the best known path between B and D had a weight of $P_2(B, D) = \infty$, but now the pair has a better path with weight $P_2(B, C) + P_2(C, D) = 4 + 2 = 6$.
- (B, E): previously, the best known path between B and E had a weight of $P_2(B, E) = \infty$, but now the pair has a better path with weight $P_2(B, C) + P_2(C, E) = 4 + 4 = 8$.
- (D, E): previously, the best known path between D and E had a weight of $P_2(D, E) = \infty$, but now the pair has a better path with weight $P_2(D, C) + P_2(C, E) = 6 + 4 = 10$.

The values of these five pairs are thereby updated in the matrix to reflect these newly improved paths.



$V_3 = \{A, B, C\}$

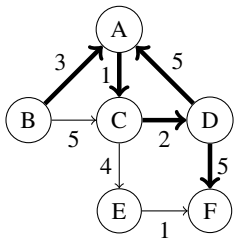
	A	**B**	**C**	**D**	**E**	**F**
A	0	∞	1	**3**	**5**	∞
B	3	0	4	**6**	**8**	∞
C	∞	∞	0	2	4	∞
D	5	∞	6	0	**10**	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	**B**	**C**	**D**	**E**	**F**
A	–	–	*A*	***C***	***C***	–
B	*B*	–	*A*	***C***	***C***	–
C	–	–	–	*C*	*C*	–
D	*D*	–	***A***	–	***C***	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

Next, we will add vertex D to our set of allowable intermediate vertices. Using the same process as before, we discover four new pairs of edges that have their shortest paths improved with D permitted as an intermediate vertex:

- (A, F): previously, the best known path between A and F had a weight of $P_3(A, F) = \infty$, but now the pair has a better path with weight $P_3(A, D) + P_3(D, F) = 3 + 5 = 8$.
- (B, F): previously, the best known path between B and F had a weight of $P_3(B, F) = \infty$, but now the pair has a better path with weight $P_3(B, D) + P_3(D, F) = 6 + 5 = 11$.
- (C, A): previously, the best known path between C and A had a weight of $P_3(C, A) = \infty$, but now the pair has a better path with weight $P_3(C, D) + P_3(D, A) = 2 + 5 = 7$.
- (C, F): previously, the best known path between C and F had a weight of $P_3(C, F) = \infty$, but now the pair has a better path with weight $P_3(C, D) + P_3(D, F) = 2 + 5 = 7$.

The values of these four vertices are updated in the matrix as follows:



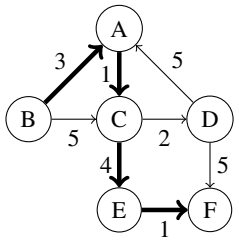
$V_4 = \{A, B, C, D\}$

	A	**B**	**C**	**D**	**E**	**F**
A	0	∞	1	3	5	**8**
B	3	0	4	6	8	**11**
C	**7**	∞	0	2	4	**7**
D	5	∞	6	0	10	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	**B**	**C**	**D**	**E**	**F**
A	–	–	*A*	*C*	*C*	***D***
B	*B*	–	*A*	*C*	*C*	***D***
C	***D***	–	–	*C*	*C*	***D***
D	*D*	–	*A*	–	*C*	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

Continuing this process, we then add vertex E to our set of allowable intermediate vertices. This improves the shortest paths between these pairs of vertices:

- (A, F): previously, the best known path between A and F had a weight of $P_4(A, F) = 8$, but now the pair has a better path with weight $P_4(A, E) + P_4(E, F) = 5 + 1 = 6$.
- (B, F): previously, the best known path between B and F had a weight of $P_4(B, F) = 11$, but now the pair has a better path with weight $P_4(B, E) + P_4(E, F) = 8 + 1 = 9$.
- (C, F): previously, the best known path between C and F had a weight of $P_4(C, F) = 7$, but now the pair has a better path with weight $P_4(C, E) + P_4(E, F) = 4 + 1 = 5$.

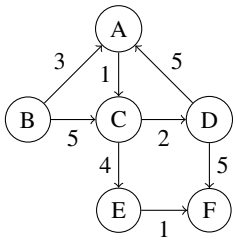


$V_5 = \{A, B, C, D, E\}$

	A	**B**	**C**	**D**	**E**	**F**
A	0	∞	1	3	5	**6**
B	3	0	4	6	8	**9**
C	7	∞	0	2	4	**5**
D	5	∞	6	0	10	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	**B**	**C**	**D**	**E**	**F**
A	–	–	*A*	*C*	*C*	***E***
B	*B*	–	*A*	*C*	*C*	***E***
C	*D*	–	–	*C*	*C*	***E***
D	*D*	–	*A*	–	*C*	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

Vertex F is the final vertex we add to our set of allowable intermediate vertices. However, the inclusion of vertex F does not improve the shortest path between any pair of vertices, so our matrices stay the same as before. Since we have added all vertices to the allowable set, we are now done, and this is the final result of the algorithm.



$V_6 = \{A, B, C, D, E, F\}$

	A	B	C	D	E	F
A	0	∞	1	3	5	6
B	3	0	4	6	8	9
C	7	∞	0	2	4	5
D	5	∞	6	0	10	5
E	∞	∞	∞	∞	0	1
F	∞	∞	∞	∞	∞	0

	A	B	C	D	E	F
A	–	–	*A*	*C*	*C*	*E*
B	*B*	–	*A*	*C*	*C*	*E*
C	*D*	–	–	*C*	*C*	*E*
D	*D*	–	*A*	–	*C*	*D*
E	–	–	–	–	–	*E*
F	–	–	–	–	–	–

At this point, the values in the distance matrix store the shortest paths between any two pairs of vertices. For instance, the value of weight_matrix[A][F] is 6, which indicates that the shortest path from A to F has a total weight of 6. We can also backtrack through the predecessor matrix to reconstruct the actual vertices along our shortest path. For example, we know that the predecessor of F along the shortest path from A to F is predecessor_matrix[A][F] = E. The predecessor of E along the shortest path from A to E is C, and the predecessor of C from A to C is A. Therefore, the shortest path from A to F is $A \rightarrow C \rightarrow E \rightarrow F$.

Floyd-Warshall can also be adapted to detect negative cycles. Since negative cycles can be used to infinitely reduce the weight of a path, a negative cycle exists if the distance from a vertex to itself ever becomes negative during the algorithm (which indicates there exists a negative-weighted path from a vertex back to itself). This check can be added in while computing the distances of all pairs of vertices during each iteration of the algorithm. An implementation is shown below:

```
1   struct FWData {
2     double d;
3     int32_t p;
4     FWData()
5       : d{ std::numeric_limits<double>::infinity() }, p{ -1 } {}
6   };
7
8   using AdjMat = std::vector<std::vector<FWData>>;
9
10  AdjMat floyd_warshall(const AdjMat& graph) {
11    AdjMat output{graph};
12    for (size_t k = 0; k < graph.size(); ++k) {
13      for (size_t i = 0; i < graph.size(); ++i) {
14        for (size_t j = 0; j < graph.size(); ++j) {
15          if (output[i][k].d + output[k][j].d < output[i][j].d) {
16            output[i][j].d = output[i][k].d + output[k][j].d;
17            output[i][j].p = output[k][j].p;
18            if (i == j && output[i][j].d < 0)
19              throw std::invalid_argument("Graph contains negative weight cycle");
20          } // if
21        } // for j
22      } // for i
23    } // for k
24    return output;
25  } // floyd_warshall()
```

From the triple `for` loop, we can see that the worst-case time complexity Floyd-Warshall is $\Theta(|V|^3)$. This is because the outer loop (line 12) executes $|V|$ times to add each vertex to the allowable set, and the inner nested loop (lines 13-14) performs $|V|^2$ computations to identify improvements between all pairs of vertices, where each computation (lines 15-20) takes constant time.

How does this compare to running Dijkstra's algorithm $|V|$ times? For a dense graph, each run of Dijkstra's algorithm would take $\Theta(|V|^2)$ time, so running Dijkstra's $|V|$ times would result in an overall time complexity of $\Theta(|V|^3)$ — the same as Floyd-Warshall! From a pure complexity standpoint, either approach would be efficient for solving the all-pairs shortest path problem when given a dense graph. However, since Floyd-Warshall involves less overhead than running Dijkstra's $|V|$ times, is simpler to implement, and also supports negative edges, it is typically the preferred choice out of the two.

25.5 Johnson's Algorithm (*)

Floyd-Warshall's $\Theta(|V|^3)$ time complexity makes it a good candidate for the all-pairs shortest path problem on dense graphs, but what if you are given a sparse graph? Recall that the time complexity of Dijkstra's on a sparse graph is $\Theta(|E|\log(|V|))$ using the min-heap implementation. This means that running Dijkstra's $|V|$ times results in an overall time complexity of $\Theta(|V||E|\log(|V|))$ for a sparse graph, which is better than the $\Theta(|V|^3)$ time complexity of Floyd-Warshall. Therefore, if you are given a sparse graph, running Dijkstra's multiple times seems to be a better choice for solving the all-pairs shortest path problem.

However, there is a catch: Dijkstra's algorithm cannot support negative edges. Thus, if you want to solve the all-pairs shortest path problem for a sparse graph with negative edges in less than $\Theta(|V|^3)$ time, you will need to rely on another shortest path algorithm. One such algorithm is **Johnson's algorithm**, which can be used to efficiently solve the all-pairs shortest path problem for sparse graphs with negative edge weights.

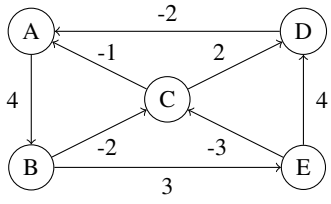
Johnson's algorithm uses techniques from two shortest-path algorithms we have discussed already: Dijkstra's algorithm and the Bellman-Ford algorithm. The aim of Johnson's algorithm is twofold: first, it uses the Bellman-Ford algorithm to transform a graph *with* negative edges into one *without* negative edges; then, it invokes Dijkstra's algorithm $|V|$ times to explore this newly transformed non-negative graph to find all pairs' shortest paths. The implementation of Johnson's algorithm can be summarized using the following steps:

1. Insert a new vertex s into the graph and connect it to every other vertex using directed edges of weight 0. This ensures that s is able to reach every vertex of the graph, while at the same time keeping the other vertices unaware of its existence (this allows it to be safely added without changing the solution).
2. Run the Bellman-Ford algorithm once with s as the source vertex (terminating if a negative cycle is detected). This discovers the shortest path from s to every other vertex of the original graph.
3. Remove vertex s, leaving behind just the remaining vertices and their corresponding weights computed by Bellman-Ford.
4. Reweight all the edges in the graph using the following equation, where d_v represents the lowest cost from s to v discovered using the Bellman-Ford algorithm. This transformation ensures that all edges become non-negative.

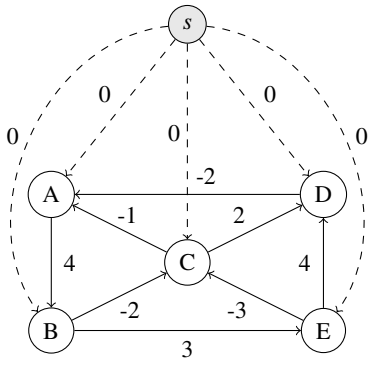
$$\text{new\_weight}(\overrightarrow{uv}) = \text{old\_weight}(\overrightarrow{uv}) + d_u \text{ - } d_v$$

5. Run Dijkstra's algorithm $|V|$ times to discover the shortest path from each vertex to every other vertex of the reweighted graph. For each shortest distance between any two vertices (u, v) returned by Dijkstra's algorithm, undo the transformation by adding $d_v - d_u$ to get the actual best distance between u and v.

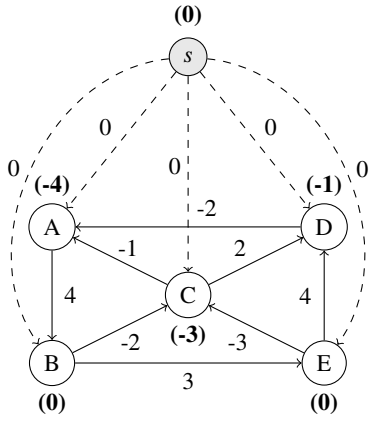
To illustrate how Johnson's algorithm works, consider the following graph for which we want to find all-pairs shortest paths.



First, we will add a vertex s to the graph that is connected to all other vertices with a directed edge of cost 0. This ensures that all the vertices in the graph are reachable from s, which is required for the Bellman-Ford step of the algorithm (since it needs to discover the shortest path to every vertex of the graph).

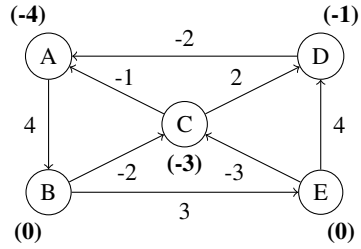


We then run the Bellman-Ford algorithm on this new graph with a source vertex of s. This gives us the following result.



v	d_v	p_v
s	0	-
A	-4	C
B	0	s
C	-3	E
D	-1	C
E	0	s

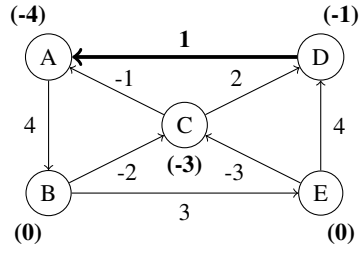
Vertex s is then removed, leaving us with the original graph, but each vertex is now associated with a Bellman-Ford weight d_v. This is the key insight used by Johnson's algorithm: we can now use these weights to transform our graph into one without negative edges without changing the shortest path between any pair of vertices.



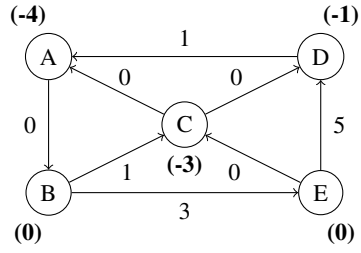
The edges in the graph are all reweighted using the following equation:

$$\text{new\_weight}(\overrightarrow{uv}) = \text{old\_weight}(\overrightarrow{uv}) + d_u - d_v$$

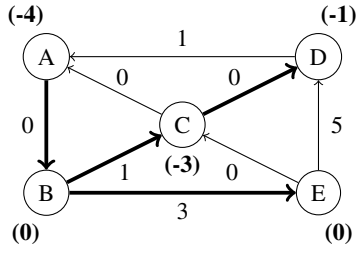
For example, edge $\overrightarrow{DA}$ would be updated to old_weight($\overrightarrow{DA}$) + d_D - d_A = -2 + (-1) - (-4) = 1.



Applying this equation to the remaining edges gives us the following reweighted graph:

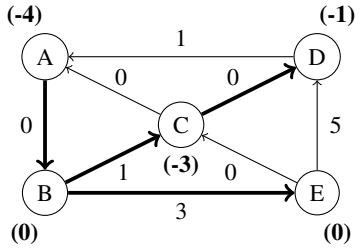


We then run Dijkstra's algorithm $|V|$ times, once with each vertex as the source, to discover the shortest path between all pairs of vertices. Each shortest path we discover is then rescaled back to its original weight by adding $d_v - d_u$, where d_v is the Bellman-Ford distance of the destination vertex and d_u is the Bellman-Ford distance of the source vertex. Running Dijkstra's algorithm with A as the source vertex gives us the following:



v	d_v	p_v
A	0	-
B	0	A
C	1	B
D	1	C
E	3	B

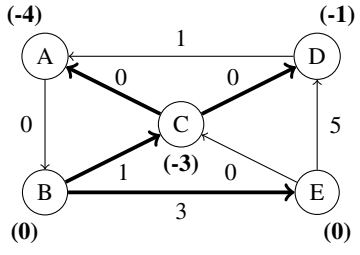
This gives us the shortest paths from A to every other vertex in our reweighted graph. To determine the shortest paths for our *original* graph (before reweighting), we can simply add back the difference between the Bellman-Ford weights of each destination vertex and the source vertex A. For example, the shortest path from A to D has weight 1 in our reweighted graph. Since the Bellman-Ford weight of vertex A is -4 and the Bellman-Ford weight of vertex D is -1, the shortest path from A to D in our original graph has a total weight of 1 + (-1) - (-4) = 4. The remaining paths are adjusted as follows:



v	d_v	p_v		v	d_v	p_v
A	0	-		A	0	-
B	0	A	$\longrightarrow$ + 0 - (-4) $\longrightarrow$	B	**4**	A
C	1	B	$\longrightarrow$ + (-3) - (-4) $\longrightarrow$	C	**2**	B
D	1	C	$\longrightarrow$ + (-1) - (-4) $\longrightarrow$	D	**4**	C
E	3	B	$\longrightarrow$ + 0 - (-4) $\longrightarrow$	E	**7**	B

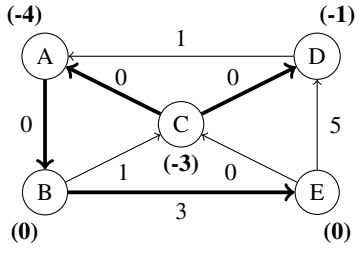
To find the shortest paths between all other pairs of vertices, Dijkstra's is run with each of the remaining vertices as the source vertex using the same process. The results of each vertex, after rescaling back to the graph's original weights, are shown below:

Dijkstra with B as the source vertex:



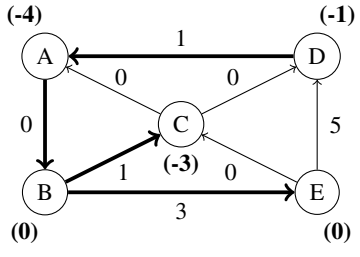
v	d_v	p_v		v	d_v	p_v
A	1	C	$\longrightarrow$ + (-4) - 0 $\longrightarrow$	A	**-3**	C
B	0	-		B	0	-
C	1	B	$\longrightarrow$ + (-3) - 0 $\longrightarrow$	C	**-2**	B
D	1	C	$\longrightarrow$ + (-1) - 0 $\longrightarrow$	D	**0**	C
E	3	B	$\longrightarrow$ + 0 - 0 $\longrightarrow$	E	**3**	B

Dijkstra with C as the source vertex:



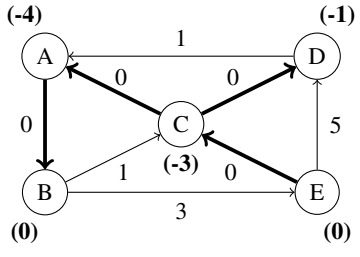
v	d_v	p_v		v	d_v	p_v
A	0	C	$\longrightarrow$ + (-4) - (-3) $\longrightarrow$	A	**-1**	C
B	0	A	$\longrightarrow$ + 0 - (-3) $\longrightarrow$	B	**3**	A
C	0	-		C	0	-
D	0	C	$\longrightarrow$ + (-1) - (-3) $\longrightarrow$	D	**-2**	C
E	3	B	$\longrightarrow$ + 0 - (-3) $\longrightarrow$	E	**6**	B

Dijkstra with D as the source vertex:



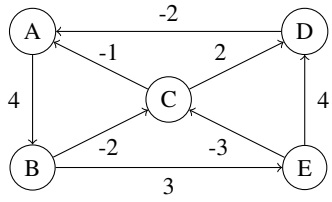
v	d_v	p_v		v	d_v	p_v
A	1	D	$\longrightarrow$ + (-4) - (-1) $\longrightarrow$	A	**-2**	D
B	1	A	$\longrightarrow$ + 0 - (-1) $\longrightarrow$	B	**2**	A
C	2	B	$\longrightarrow$ + (-3) - (-1) $\longrightarrow$	C	**0**	B
D	0	-		D	0	-
E	4	B	$\longrightarrow$ + 0 - (-1) $\longrightarrow$	E	**5**	B

Dijkstra with E as the source vertex:



v	d_v	p_v		v	d_v	p_v
A	0	C	$\longrightarrow$ + (-4) - 0 $\longrightarrow$	A	**-4**	C
B	0	A	$\longrightarrow$ + 0 - 0 $\longrightarrow$	B	**0**	A
C	0	E	$\longrightarrow$ + (-3) - 0 $\longrightarrow$	C	**-3**	E
D	0	C	$\longrightarrow$ + (-1) - 0 $\longrightarrow$	D	**-1**	C
E	0	-		E	0	-

Putting this all together, this is the final result of Johnson's algorithm for our graph. Each value of d_v in the tables below represents the weights of the shortest path between the source vertex as v.



Source: A

v	d_v	p_v
A	0	-
B	4	A
C	2	B
D	4	C
E	7	B

Source: B

v	d_v	p_v
A	-3	C
B	0	-
C	-2	B
D	0	C
E	3	B

Source: C

v	d_v	p_v
A	-1	C
B	3	A
C	0	-
D	-2	C
E	6	B

Source: D

v	d_v	p_v
A	-2	D
B	2	A
C	0	B
D	0	-
E	5	B

Source: E

v	d_v	p_v
A	-4	C
B	0	A
C	-3	E
D	-1	C
E	0	-

The code for Johnson's algorithm is shown on the following page. As you can see, Johnson's algorithm is an combination of two shortest-path algorithms that are run sequentially: Bellman-Ford, which is run once for a time complexity of $\Theta(|V||E|)$; and Dijkstra's, which is run $|V|$ times for a time complexity of $|V| \times \Theta(|E|\log(|V|)) = \Theta(|V||E|\log(|V|))$. Because the time complexity of running Dijkstra $|V|$ times contributes to a higher order term, the overall time complexity of Johnson's algorithm is also $\Theta(|V||E|\log(|V|))$, assuming you use a binary min-heap as your priority queue implementation.[4]

At this point, you may be wondering: how does Johnson's algorithm even work? The idea that Bellman-Ford can be used to transform a graph with negative edges into one without can seem counterintuitive, but we can prove its correctness using a bit of arithmetic. Let's define $W(x, y)$ as the original weight of an edge between vertices x and y, and let $P_{v_1 \to v_n} = \langle v_1, v_2, v_3, \ldots, v_n \rangle$ represent a path from vertex v_1 to vertex v_n. After reweighting, the weight of the path $P_{v_1 \to v_n}$ becomes

$$|P_{v_1 \to v_n}| = (W(v_1, v_2) + d_{v_1} - d_{v_2}) + (W(v_2, v_3) + d_{v_2} - d_{v_3}) + \ldots + (W(v_{n-1}, v_n) + d_{v_{n-1}} - d_{v_n})$$

This is a telescoping series, so all the Bellman-Ford distances of intermediate vertices cancel out. This leaves us with

$$|P_{v_1 \to v_n}| = W(v_1, v_2) + W(v_2, v_3) + \ldots + W(v_{n-1}, v_n) + d_{v_1} - d_{v_n}$$

This implies that all possible paths between vertices v_1 and v_n have their total path weight increased by the *same amount* of $d_{v_1} - d_{v_n}$. Because of this, the shortest path between v_1 and v_n before reweighting *must remain the shortest path after reweighting*, since all alternative paths between v_1 and v_n have their weights incremented by the same amount. Therefore, the reweighting process does not change the shortest path between any two vertices in our graph.

However, proving that reweighting does not change our solution alone is not enough to prove the correctness of Johnson's algorithm. We must also prove that the Dijkstra step also returns the shortest path between any two pairs of vertices in our reweighted graph. To do so, all we need to show is that the reweighted graph does not contain any negative edges, which guarantees the correctness of Dijkstra's algorithm (which we proved earlier).

Recall that our Bellman-Ford weights were computed using a starting vertex s, which was connected to all other vertices with an edge weight of 0. If we define d_u as the optimal distance from the artificial vertex s to vertex u (i.e., the Bellman-Ford weight that we calculate), then we know that the value of d_v for some other vertex v connected to u cannot be worse than d_u plus the cost to get from u to v (otherwise, d_v wouldn't be the shortest distance from s to v, which is a contradiction). In other words,

$$d_u + W(u, v) \geq d_v$$

Rearranging this equation by subtracting d_v from both sides, we get

$$W(u, v) + d_u - d_v \geq 0$$

$W(u, v) + d_u - d_v$ is simply the cost of the reweighted edge between u and v. Therefore, any edge between two vertices in our reweighted graph must have an edge weight that is at least zero. Since we have proven that reweighting does not change the optimal path, and that the reweighted graph cannot have negative edges, we have thereby proven that Dijkstra's always returns the optimal path between any two vertices of our reweighted graph. This, in turn, proves that Johnson's algorithm is correct.

[4] If you use a Fibonacci heap to implement Dijkstra's algorithm, each iteration of Dijkstra's would take $\Theta(|E| + |V|\log(|V|))$, leading to an overall time complexity of $|V| \times \Theta(|E| + |V|\log(|V|)) = \Theta(|V||E| + |V|^2\log(|V|))$ for Johnson's algorithm.

An implementation of Johnson's algorithm is shown below:

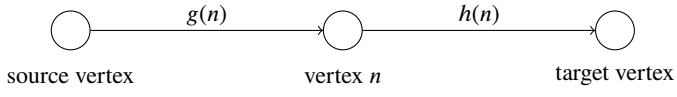
```
using AdjList = std::vector<std::vector<std::pair<int32_t, int32_t>>>;

struct JohnsonData {
  double d;
  int32_t p;
  JohnsonData()
    : d{ std::numeric_limits<double>::infinity() }, p{ -1 } {}
};

// returns Bellman-Ford weights for a graph starting from src
// e.g., vec[5] stores shortest cost from vertex src to vertex 5
std::vector<int32_t> bellman_ford(const AdjList& graph, int32_t src);

// returns Dijkstra weights/predecessors for a graph starting from src
std::vector<JohnsonData> dijkstra(const AdjList& graph, int32_t src);

// adds a new vertex 0 (i.e., s) with zero-weighted edges to all other vertices
AdjList add_zero_vertex(const AdjList& graph);

std::vector<std::vector<JohnsonData>> johnson(const AdjList& graph) {
  AdjList s_graph = add_zero_vertex(graph);
  std::vector<int32_t> bf_weights;

  try {
    bf_weights = bellman_ford(s_graph, 0);
  } // try
  catch (const std::exception&) {
    std::cerr << "Graph contains negative weight cycle, cannot compute shortest paths!" << std::endl;
    throw;
  } // catch

  // reweight each edge (u, v) by adding bf_weights[u] - bf_weights[v]
  AdjList reweighted_graph{graph};
  for (size_t u = 0; u < graph.size(); ++u) {
    for (auto& [v, weight] : reweighted_graph[u]) {
      weight += (bf_weights[u] - bf_weights[v]);
    } // for v, weight
  } // for u

  // invoke Dijkstra |V| times to find all pairs shortest paths
  std::vector<std::vector<JohnsonData>> result(graph.size());
  for (size_t i = 0; i < graph.size(); ++i) {
    result[i] = dijkstra(reweighted_graph, i);
  } // for i

  // reweight the shortest path for each pair (u, v) by adding
  // bf_weights[v] - bf_weights[u]
  for (size_t u = 0; u < graph.size(); ++u) {
    for (size_t v = 0; v < graph.size(); ++v) {
      if (result[u][v].d != std::numeric_limits<double>::infinity()) {
        result[u][v].d += (bf_weights[v] - bf_weights[u]);
      } // if
    } // for v
  } // for u

  return result;
} // johnson()
```

25.6 A* Search (⁎)

A* search (pronounced "A-star" search) is an optimized algorithm that can be used to find the shortest path between two vertices in a graph. The A* algorithm can be seen as an extension of Dijkstra's algorithm, using the steps of Dijkstra's as a foundation for how it explores a graph. However, unlike Dijkstra's algorithm, A* uses a *heuristic function* to estimate the remaining cost required to reach the target vertex from the current position along the search. This information is then used by the A* algorithm to determine the order in which vertices are explored. More formally, when trying to finding a shortest path from a source vertex to a target vertex in a graph, the A* algorithm applies the following function $f(n)$ to determine which vertex to explore next along the search path:

$$f(n) = g(n) + h(n)$$

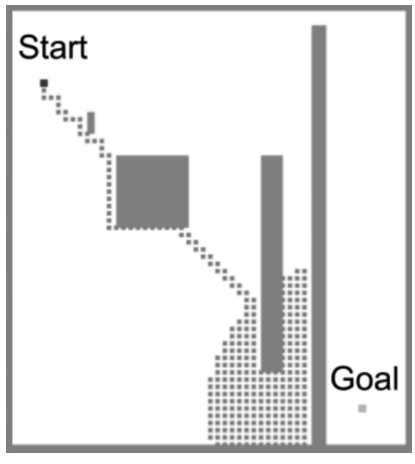
where $g(n)$ is the cost from the source vertex to vertex n, and $h(n)$ is a heuristic function that estimates the cheapest cost required to reach the target vertex from vertex n. At each step of the search, the A* algorithm looks through each of the unvisited neighbors of the current vertex and extends the search path by visiting the one that minimizes $f(n)$. Note that this differs from Dijkstra's algorithm, which simply chooses the vertex with the smallest known distance from the source; in fact, you can think of Dijkstra's as a special case of A* where $h(n) = 0$ for all vertices n.



There are a few details to note about the A* algorithm. First, unlike Dijkstra's algorithm, A* does not find the shortest path from the source vertex to *all* other vertices in the graph (i.e., it does not solve the single-source shortest path problem). This is the trade-off of applying a heuristic to optimize the search, since an effective heuristic is used to guide the search toward a specific target vertex. Second, the correctness of A* search depends on the heuristic that is used to estimate the remaining cost to reach the target vertex. For A* search to guarantee the shortest path between two vertices, its heuristic must be *admissible* — that is, it cannot overestimate the actual cost required to get to the desired goal.

Because of its optimized performance, the A* search algorithm has several common applications related to pathfinding problems. A* can be used in robotics to guide a robot along the shortest path from one location to another while avoiding obstacles, in artificial intelligence settings to facilitate natural language processing, and in game design to guide non-playable characters toward a certain goal, just to name a few.

An example of the A* algorithm in action is shown in the image below. This screenshot depicts the progress of a search that is being performed on a two-dimensional grid, where you are allowed to move in four directions: up, down, left, and right. The darker shaded objects represent obstacles that you cannot pass through, and the dots represent coordinates of the grid that have been visited along the search. The heuristic used by A* in this example is a simple Euclidean distance calculation between the current coordinate and the goal.



As you can see, the use of a heuristic to estimate additional cost helps guide the search in the direction of the desired goal, which is something that Dijkstra's alone does not provide. Because A* applies a heuristic to identify the most promising search path, it is often called a *best-first search* or *informed search* algorithm. If you choose a good heuristic, the A* search algorithm can help you find the shortest path between two points in a graph faster than the non-informed search algorithms discussed earlier!

25.7 Summary of Shortest Path Algorithms

In this chapter, we mainly concerned ourselves with two different types of shortest path problems: the *single-source* shortest path problem and the *all-pairs* shortest path problem. The first of these variants, the *single-source* shortest path problem, seeks to find the shortest path from a single source vertex to every other vertex in a graph. When working with single-source shortest path problems, there are two primary algorithms we can use: Dijkstra's algorithm and the Bellman-Ford algorithm.

Dijkstra's algorithm employs a greedy approach to explore a graph, and it can be implemented using either a linear search or a min-heap to determine which vertices to explore. The linear search works best for dense graphs and has an overall time complexity of $\Theta(|V|^2)$, where $|V|$ is the number of vertices in the graph. On the other hand, the min-heap approach works best for sparse graphs and has an overall time complexity of $\Theta(|E|\log(|V|))$ (assuming a binary heap is used), where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices.

One major pitfall of Dijkstra's algorithm is its inability to handle graphs with negative edge weights. This is because the greedy nature of the algorithm prevents it from ever going back to previous states, even if it discovers a negative edge that improves the solution of a previously visited vertex. To address graphs with negative edge weights, we can instead use the Bellman-Ford algorithm to solve the single-source shortest path problem. Bellman-Ford utilizes dynamic programming, allowing it to consider subproblems that Dijkstra's was unable to consider. Because Bellman-Ford needs to consider all subproblems rather than just the greedy choice, its overall time complexity of $\Theta(|V||E|)$ is asymptotically less efficient than that of Dijkstra's.

Single-Source Shortest Path Algorithms

	Dijkstra's Algorithm (Linear Search)	**Dijkstra's Algorithm (Min Binary Heap)**	**Bellman-Ford Algorithm**
Time Complexity	$\Theta(\|V\|^2)$	$\Theta(\|E\|\log(\|V\|))$	$\Theta(\|V\|\|E\|)$
Recommended Graph Type	Dense	Sparse	Either
Supports Negative Edge Weights	No	No	Yes
Can Detect Negative Cycles	No	No	Yes

On the other hand, the *all-pairs* shortest path problem seeks to find the shortest path between all pairs of vertices in a graph. We have discussed three primary techniques for solving the all-pairs shortest path problem: running Dijkstra's once for each vertex, the Floyd-Warshall algorithm, and Johnson's algorithm.

Running Dijkstra's algorithm $|V|$ times is the best strategy if a graph is sparse and has no negative edges. Recall that each invocation of Dijkstra's algorithm takes $\Theta(|V|^2)$ time if a linear search is used and $\Theta(|E|\log(|V|))$ time if a min-binary heap is used. However, $|E|$ is on the order of $\Theta(|V|^2)$ in dense graphs — this means that running Dijkstra's $|V|$ times would take $\Theta(|V|^3)$ time if a linear search is used, and $\Theta(|V||E|\log(|V|)) = \Theta(|V|^3\log(|V|))$ time if a min-binary heap is used. As a result, Floyd-Warshall would be a better algorithm to use if you are given a dense graph, since it also runs in $\Theta(|V|^3)$ time (which is the best you can do with repeated Dijkstra's), is simpler to implement, and also supports graphs with negative edge weights.

If the underlying graph is sparse, but negative edge weights are involved, then Johnson's algorithm would be a preferable choice. Much like Floyd-Warshall, Johnson's algorithm supports graphs with negative edge weights, as long as there are no negative cycles. However, the time complexity of Johnson's algorithm is $\Theta(|V||E|\log(|V|))$, which is better than Floyd-Warshall's $\Theta(|V|^3)$ time complexity if a graph is sparse (since $|E|$ would be on the order of $\Theta(|V|)$ in a sparse graph, which implies that $\Theta(|V||E|\log(|V|))$ is asymptotically equal to $\Theta(|V|^2\log(|V|))$).

All-Pairs Shortest Path Algorithms

	Dijkstra's $\|V\|$ Times (Min Binary Heap)	**Floyd-Warshall Algorithm**	**Johnson's Algorithm (Min Binary Heap)**
Time Complexity	$\Theta(\|V\|\|E\|\log(\|V\|))$	$\Theta(\|V\|^3)$	$\Theta(\|V\|\|E\|\log(\|V\|))$
Recommended Graph Type	Sparse	Dense	Sparse
Supports Negative Edge Weights	No	Yes	Yes
Can Detect Negative Cycles	No	Yes	Yes

Lastly, it is important to recognize that all of these shortest path algorithms are designed for *weighted* graphs. If you want to find the shortest path between two vertices in an *unweighted* graph, then your goal would simply be to find the path between the two vertices that traverses the fewest number of edges. In this scenario, a straightforward breadth-first search would suffice.

Shortest path algorithms play a critical role in much of the technology we rely on today, from determining the optimal route to take on a road trip to routing data packets efficiently in a communication network. Additional shortest path algorithms build upon these concepts to perform an *informed search* that optimizes the performance of finding the shortest path between two vertices of a graph. As discussed, one such algorithm is the *A* search algorithm*, which is one of the most popular algorithms for solving pathfinding problems due to its efficiency. The A* algorithm improves upon the standard Dijkstra's implementation by applying a heuristic that estimates the cost of the cheapest path from a vertex to the destination; this heuristic value is then combined with the best known distance to make a more informed decision on which vertex to visit next.

Remark: In addition to the single-source and all-pairs shortest path problems, you may also encounter the *single-destination shortest path problem*, which seeks to find the shortest path *to a single destination vertex* from every other vertex in a graph. However, solving this type of problem does not require anything new: by simply reversing the direction of each edge in the graph, the single-destination shortest path problem essentially reduces to a single-*source* shortest path problem, which we can solve using an algorithm like Dijkstra's or Bellman-Ford.