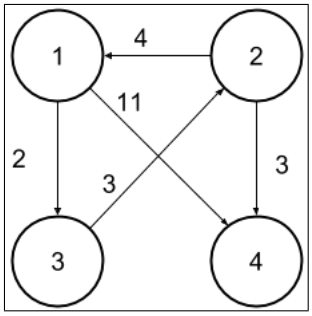
Shortest Paths

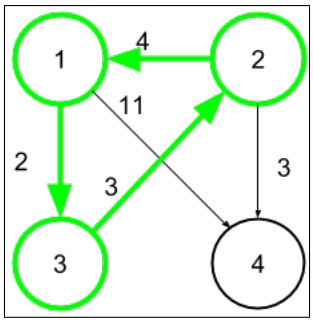
1 Shortest Paths: Dijkstra

As we mentioned earlier, graphs can have "weights" associated with an edge so it would look like the following:



Now let's consider the following problem for the above graph: What is the shortest path (or least weight path) from the node 1 to all the other nodes?

Before we go further with this problem, let's gain some more intuition. Take the following path, is it ever possible that this would be part of our Shortest Path Graph?



The answer is no! The reasoning is that if we have a shortest path to a vertex, there should never be a reason to visit it again because we would have reached it in less distance earlier. If we are finding the shortest distance from 1 to every other node, there is no purpose to visit 3, go to 4, and then back to 1. This leads us to make an interesting observation, the shortest path from a node to any other node will always be a tree- in other words, there are no cycles.

Now that we have some intuition, lets try solving this problem. One interesting approach that we can take is that we can grow our Shortest Paths Tree by one node and one edge at each step . We do this by a a node that fulfills 2 constraints:

1. The node is reachable from our current Shortest Path's Tree. That is, by crossing 1 edge from some vertex in our current Shortest Path Tree we can reach the node.
2. The node is the least distance away from the start (taking into account the distance traveled so far).

The reason that this works is because any node that is in our shortest path tree will be closer than any vertex not in it. This means that we can be sure that when we add a vertex to this shortest path tree that there is no possible way for us to reach it in less distance.

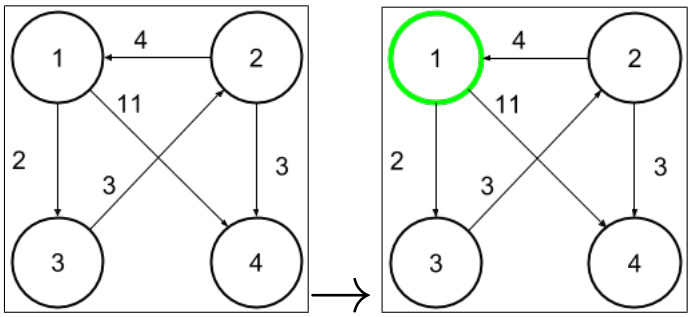
Now that we have an idea of what we want to do, let's think about how we can implement it. We need to keep track of 3 things, the actual shortest path, the minimum distance to each element, and the node that is

the shortest distance away from the source. To keep track of the path, we can have an array of the "parent edge" for each vertex (what vertex leads to it) and backtrack until we reach the source, which would have no parent edge, this array would be called an **edgeTo** array. To find which element has the minimum distance at a certain step, we could use a Minimum Priority Queue that contains all the vertices and perform a deleteMin operation and add the returned node and its "parent edge" to the shortest path graph. However, once we remove all of our vertices from our Priority Queue (when we have our final shortest path tree), we would want to see the actual distance to each vertex, so we would have a second array that keeps track of the minimum distance to each element, we will call this our **distTo** array.

Before going through an example, let's go through the process of running this new algorithm. We would start off with a Priority Queue with our source's priority as 0 and all vertices priorities as infinity. Additionally, the distTo array would have an entry of 0 for the source and the edgeTo array would have an entry of null for all the elements (there is no edge to the source and we have no access to information about any other nodes). We proceed by performing a deleteMin operation. We remove the source and examine its neighbors. We would create an entry in the distTo array for all the neighbors with an entry equal to the weight of their edge + distance to the source from the vertex the edge is "from" (0 in this case as "from" vertex is the source). We would also put an entry in our edgeTo array with the edge from the source to the neighbor. This process continues until all the elements are removed from the Priority Queue.

When checking neighbors, we check to see if the distance to it is less than the current distance we have noted for it (the distTo element for that vertex), this process is called **edge relaxation**. If the distance to the node is greater than the current distance we have noted, then we do not change anything. Otherwise, we change the node's priority in the Priority Queue and the entry in the distTo array to be the smaller distance. We would also change the entry in the edgeTo array to be the edge that creates the smaller distance. To think about edge relaxation, consider some rubber band. At some given moment, we are "stretching" the rubber band across a path that connects 2 vertices. When we find one that is better, we can stop considering the other path and as a result are "relaxed" since we no longer are stretching our rubber band over 2 paths.

This approach is called ***Dijkstra's Algorithm***. We will walk through an example of Dijkstra's, the left image in every sequence is the graph at the step we are on and the table next to it is the distTo array and the edgeTo array. An important note to realize is that entries in the distTo array are the priorities of elements in the Priority Queue.

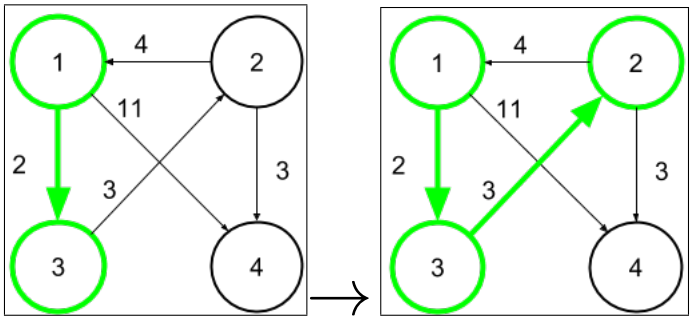


Vertex	DistTo	EdgeTo
1	0	--
2	∞	--
3	∞	--
4	∞	--

→

Vertex	DistTo	EdgeTo
1	0	--
2	∞	--
3	2	1→3
4	11	1→4

We start off at the node 1 and have its priority be 0 while all of the others nodes have a priority of 0. The distTo array would only have an entry, 0, for the vertex 1 while all other entries would be infinity. The edgeTo array would be null for all elements. In the next step, we perform a removeMin operation and delete the vertex 1 from the Priority Queue. When this occurs, we add an entry for all of its neighbors in the distTo array and the edgeTo array. The distTo element would be equal to the edge length from the source (an entry 2 for vertex 3 and entry 11 for vertex 4). The edgeTo element would be the edge between the source and the neighbors.

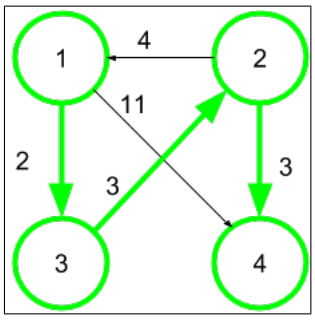


Vertex	DistTo	EdgeTo
1	0	--
2	5	3→2
3	2	1→3
4	11	1→4

Vertex	DistTo	EdgeTo
1	0	--
2	5	3→2
3	2	1→3
4	8	2→4

We analyze perform another deleteMin operation on our Priority Queue. This time, we remove the vertex 3. We explore 3's neighbors and we discover that we can reach a new vertex, 2. We update our distTo array and priority to have the value dist[3]+ e(3,2), as this the the total distance that 2 is from the source. We would also update the edgeTo array for vertex 2 with the entry of the edge between 3 to 2.

Once again we perform a deleteMin operation, this time we remove the vertex 2 as it has a lower priority/distance from the source than the vertex 4 (5 opposed to 11). We explore 2's neighbors and see that we have the vertex 1 and 4 as its neighbors. 1 is already part of our Shortest Path Tree (has been removed from our Priority Queue), so there is no way that we could find a cheaper route to it, so we do not change anything in the Priority Queue, distTo array, or edgeTo array. This can be validated by comparing distTo[1] = 0 with distTo[2] + e(2,1) = 9. Now we will check 2's next neighbor, 4. We compare distTo[2] +e(2,4) = 8 and distTo[4] = 11 and see that distTo[2] +e(2,4) is cheaper. As a result, we will change our priority of the 4 node in the Priority Queue and the distTo value of 4 to be equal to 8. We will also change the edgeTo array to have the edge from 2 to 4 as its value.

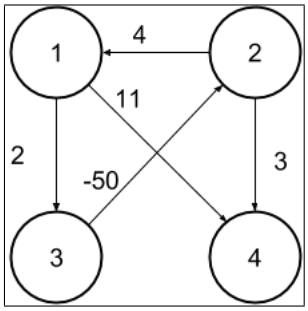


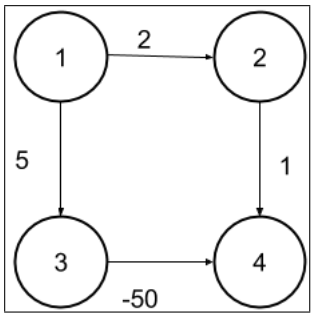
Vertex	DistTo	EdgeTo
1	0	--
2	5	3→2
3	2	1→3
4	8	2→4

We perform 1 more delete min and remove the node 4. We realize that our Priority Queue is completely empty, which is the terminating condition for Dijkstra's algorithm, meaning we are done. In the above picture, there is only 1 graph and 1 distTo array, and they correspond to the same step.

As we learned earlier, we use a Priority Queue to see which element we should add to our Shortest Path's Tree where the priority of an element is the distTo entry for the vertex that its edge is coming from + the weight of the edge between the element we are looking at and the vertex connecting to it. We have $|V|$ insertions, $|V|$ deletions, and O($|E|$) decreasing of priorities. A single one of any of these operations would take O(log($|V|$) time. As a result, our runtime is $O(|E| \log |V| + |V| \log |V| + |V| \log |V|)$. In a connected graph, E is always between $|V| - 1$ and $|V|^2$ so we can simplify our expression to be $O(|E| \log |V|)$.

Dijkstra's is a pretty great algorithm; however, there are some pitfalls. Consider the following graphs and try to identify why running Dijkstra's would not return a valid answer.





In the first graph, if we took the path {(1,3), (3,2), (2,1) , (2,3),(3,2), (2,1)...} we could get into a never-ending cycle because our shortest path would be negative infinity. This is because we could always get a lower path weight by taking the cycle again. In the second graph, we would start at vertex 1, then we would go to vertex 2 because it is closer to the source than vertex 3. We would then travel to vertex 4 because the distTo[4] would be 3, which is less than distTo[3] (5). In the next step, we travel to vertex 3 would get a nasty surprise: since the edge between 3 and 4 is negative, there was a shorter path to the vertex 4 that we did not consider! One that actually has negative weight.

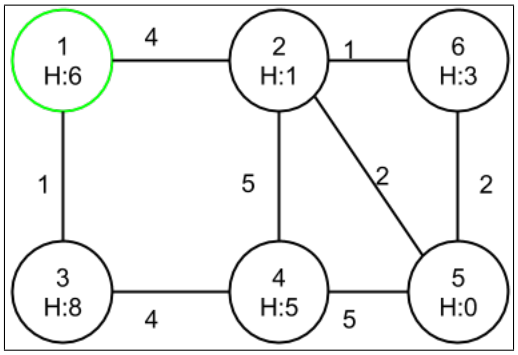
These 2 examples show us a very important requirement for Dijkstra's Algorithm: the graph we are running the algorithm on can have no negative edges. The reasoning behind this is that if an edge is negative, there could be a different shortest path to a vertex that we do not consider since negative edge weights would take away from the total weight of a path.

2 Shortest Paths: A*

Sometimes, we don't want to visit every vertex. Instead, we want to find the shortest path from one vertex to another. Dijkstra's algorithm would not be optimal as we would take the time to explore every vertex instead of the ones that we need. For example, if we used Dijkstra's to go from California to South America, we would go all the way to Africa! That really would be unoptimal. As a result, we would use a algorithm called ***A****. The A* Algorithm works off of the principle of a **heuristic**, which is an estimation to the cost that it will take to travel to the destination from a certain point. For A* to work, you must have:

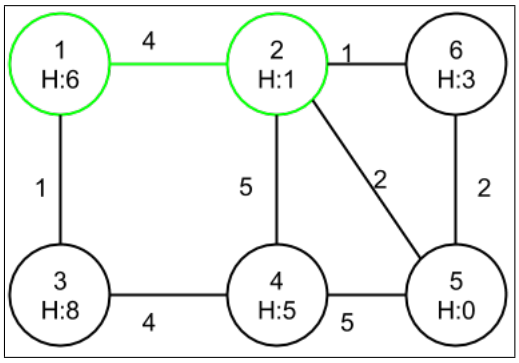
- Heuristics that are not overestimated (they are admissible)
- Consistent Heuristics

As long as the heuristics are admissible, you will always ind the correct shortest path between two nodes. Once again, we will do a walkthrough of how to run the A* algorithm. The H inside of each node refers to the heuristic. We will be running A* to find the shortest path from 1 to 5.



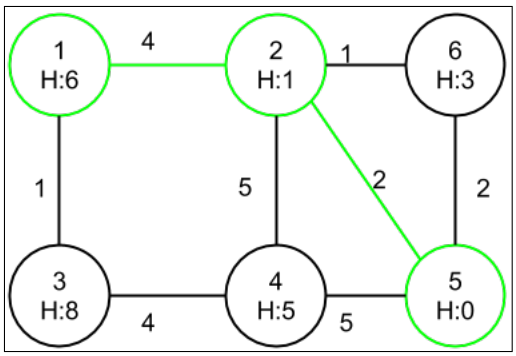
Vertex	DistTo	Heuristic	Sum
1	0	6	6
2	4	1	6
3	1	8	9
4	--	--	--
5	--	--	--
6	--	--	--

Here we start off at vertex 1. We enqueue the neighbors, 2 and 3, the distance to them, and their heuristic. The distance traveled so far is 0 from the vertex.



Vertex	DistTo	Heuristic	Sum
1	0	6	6
2	4	1	6
3	1	8	9
4	9	5	14
5	6	0	6
6	5	3	8

The distance to the node 3 is less than the distance to 2; however, since we are running A* and not Dijkstra's algorithm, we will take a look at the heuristic. We add the heuristic to the distance required to travel to the vertex and we see that going to 2 actually should lead to a shorter path. As a result, we will travel to 2.



In our final step, we take a look at the new neighbors of 2, and we see that 5 has the smallest sum of heuristic + distance traveled so far. As a result, we go directly to 5, and that concludes our A* search. As it can be seen, we avoid going to any unneeded vertices and go straight to our path. This concept is extremely powerful in the world of routing, and it saves us a lot of time.