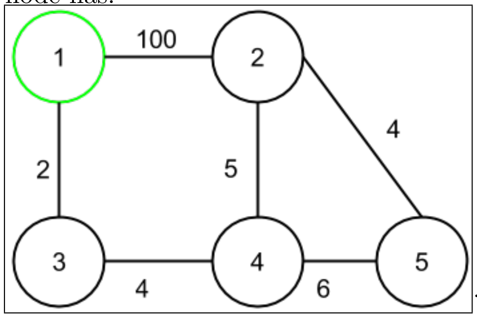
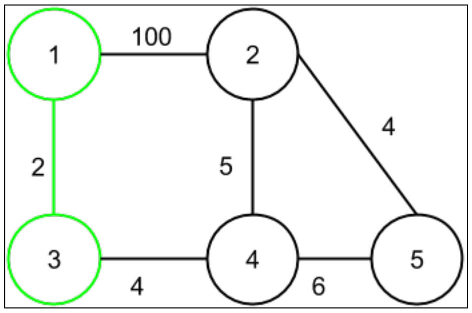
Shortest Paths

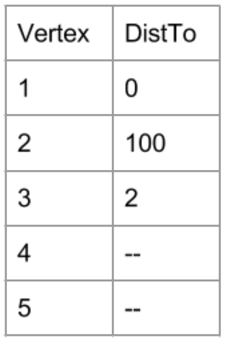
1 Shortest Paths: Dijkstra

As we mentioned earlier, graphs can have "weights" associated with the edge so it would look like the following: Say that we want to approach the problem of finding the shortest path that touches each vertex exactly once.
To find the shortest path, we will use ***Dijkstra's Algorithm*** which helps us find the shortest path tree.

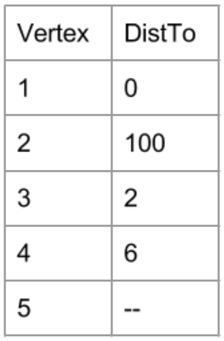
We will walk through a walkthrough of Dijkstra's, the left image in every sequence is the graph at the step we are on and the table next to it is the distTo array which shows, in an array-like format, the priority each node has.



$\longrightarrow$


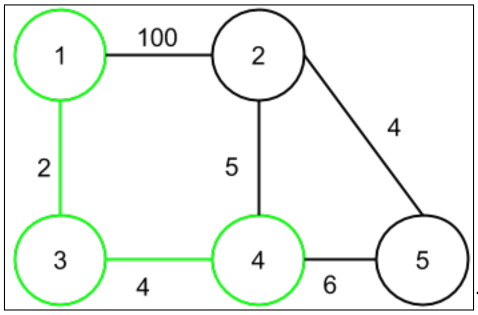


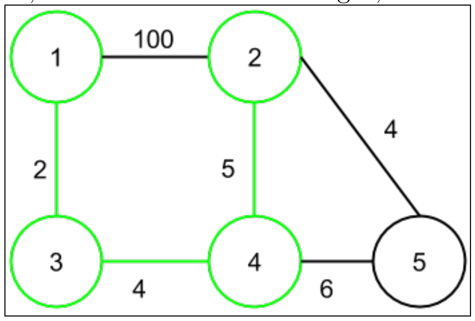
Vertex	DistTo
1	0
2	100
3	2
4	--
5	--

$\longrightarrow$


Vertex	DistTo
1	0
2	100
3	2
4	6
5	--

We start off at the node 1 and we enqueue the vertices nd the weights currently associated with them. Currently, we have two options, to go to node 3 that has a weight of 2 or to node 2 which has a weight of 100. Well since we want the shortest path, we want the lowest weight, sow we would move to 3.



$\longrightarrow$


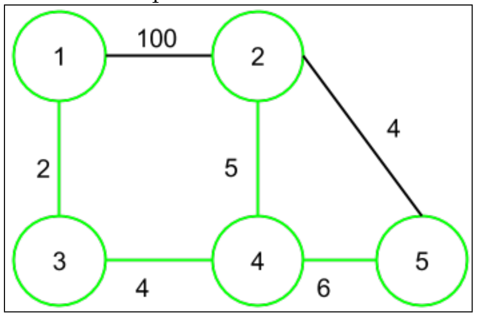
Vertex	DistTo
1	0
2	11
3	2
4	6
5	12

$\rightarrow$

Vertex	DistTo
1	0
2	11
3	2
4	6
5	12

Now we start off at 3 and we enqueue it's neighbor which is 4, and it has a weight of 4. Now we have 2 paths we can choose, to go from 1 to 2 or from 3 to 4. Since we have different start points, we need to take into account the distance traveled so far. As a result, the weight associated with the 4 node would be 6 because we have traveled a distance of 2 so far. 6 is less than 100, so we will travel to 4.

In the next step, we put in 4's neighbors, which are 2 and 5. For 5, we simply add 6 to our distance traveled so far (6) and get a weight of 12 associated with 5. For 2, it is a little bit more complicated. We add 5 to the distance so far and get 11; however, we already have a value associated with 2. To resolve this dispute, we compare the 2 weights, we see that 11 is less that 100 so we replace the value associated with 2 to be 11. Now we compare 11 and 12 to see where we go next, we see 11 is less than 12 so we explore that node.



We now have 1 node left and 2 possibilities. After going to the node 2 we have a new edge, 4, which leads to 5. Now, if we were not careful we would think that 4 is less than 6 so we should go with 4. However, since we had to travel an extra distance of 5 to get to that path, we would not associate a weight of 4 with this edge but rather 15. 15 is greater than 12, the current value associated with 5, so we do not replace the value and travel from 4 to 6.

Since the order in which we enqueue nodes does not matter we will not use a queue or stack like we did for the traversals. Instead, we will use a priority queue where the priority is the distance traveled so far + distance to travel to the next node. The time that this algorithm takes $O(E \log |V|)$ time. When inserting into our priority queue, we have V insertion and V deletion, both which take $O \log |V|$ time, and we have to decrease the priority a maximum of E times. As a result, our runtime is $O((E \log |V| + V \log |V|$. E is always between $V - 1$ and V^2 so we can simplify our expression to be $O(E \log |V|)$.

Let's briefly go over the code for Dijkstra's algorithm.

```
public class DijkstraDemo {
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraDemo(WeightedDiGraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Double.POSITIVE_INFINITY;
        }
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty()) {
            relax(G, pq.delMin());
        }
    }

    private void relax(WeightedDiGraph G, int v) {
        for (DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) {
                    pq.changeKey(w, distTo[w]);
                } else {
                    pq.insert(w, distTo[w]);
                }
            }
        }
    }
}
```

For our purposes, it is more important to understand how Dijkstra's algorithm works rather than the code behind it. This is here just for reference.
A requirement for Dijkstra's Algorithm is that edges cannot be negative. The reasoning behind this is that

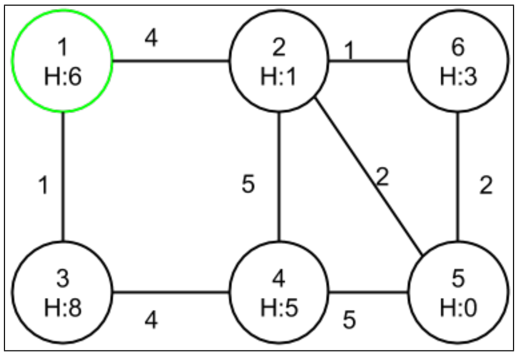
if an edge is negative, there could be a different shortest path to a vertex that we did not take. This is because negative edge weights would take away from the total weight.

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

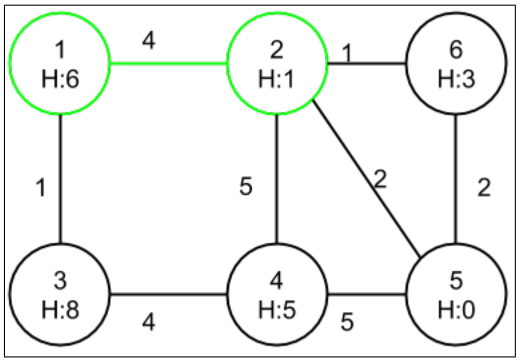
As long as the heuristics are admissible, you will always ind the correct shortest path between two nodes. Once again, we will do a walkthrough of how to run the A* algorithm. The H inside of each node refers to the heuristic. We will be running A* to find the shortest path from 1 to 5.



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

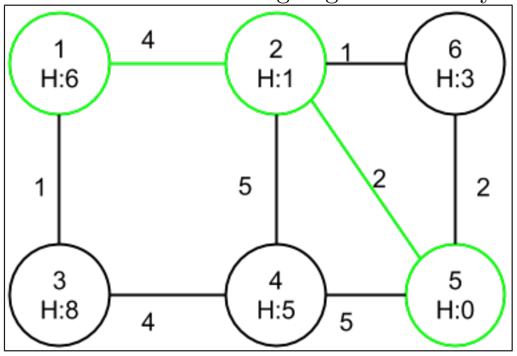
Here we start off at vertex 1. We enqueue the neighbors, 2 and 3, the distance to them, and their heuristic.

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 disstance 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.