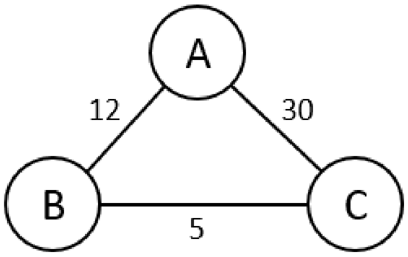
Dijkstra's Algorithm

Step 1

By now, we have encountered two different algorithms that can find a shortest path between the vertices in a graph. However, you may have noticed that all this time we have been operating under the assumption that the graphs being operated on were *unweighted* (i.e., all edge weights were the same). What happens when we try to find the shortest path between two vertices in a *weighted* graph using **BFS**? Note that we define the *shortest weighted path* as the path that has the smallest *total path weight*, where *total path weight* is simply the sum of all edge weights in the path.

For example, lets perform **BFS** on the graph below:



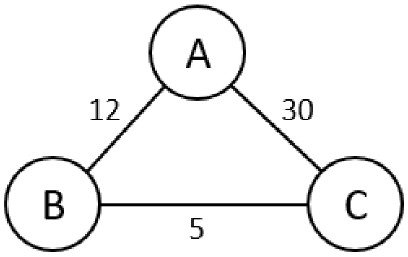
Suppose we are trying to find the shortest path from vertex *A* to *C*. By starting at vertex *A*, **BFS** will immediately traverse the edge with weight 30, thereby discovering vertex *C* and returning the path *A* → *C* to be the shortest path. Yet, if we look closely, we find that counter-intuitively, the *longer* path in terms of number of edges actually produces the *shortest weighted* path (i.e., Path *A* → *B* → *C* has a total path weight of 12 + 5 = 17, which is less than 30).

Consequently, we need to look to a new algorithm to help us find an accurate weighted shortest path: specifically, we will look at **Dijkstra's Algorithm**.

Step 2

The idea behind **Dijkstra's Algorithm** is that we are constantly looking for the lowest-weight paths and seeing which vertices are discovered along the way. As a result, the algorithm explores neighbors similarly to how the **Breadth First Search** algorithm would, except it prioritizes which neighbors it searches for next by calculating which neighbor lies on the path with the lowest overall path weight.

Take the simple example from the previous step:



Suppose we start at vertex *C* and want to find the shortest path from *C* to all other vertices. **Dijkstra's algorithm** would run roughly like this:

1. We would take note that the distance from vertex *C* to vertex *C* is 0 (by definition, the distance from a node to itself must be 0). As a result, we would mark that we have found the shortest path to vertex *C* (which is just *C*).
2. We would traverse the next "shortest" path (i.e., path with the smallest total weight) that we could find from *C*. In this case, our options are a path of weight 5 (*C* → *B*) or a path of weight 30 (*C* → *A*); we choose the path with the smaller weight of 5 (*C* → *B*).

3. We would take note that we have arrived at vertex *B*, which implies that the shortest distance from vertex *C* to vertex *B* is the total weight of the path we took, which is 5. As a result, we would mark that we have found the shortest path to vertex *B* (which is *C* → *B*).
4. We would traverse the next "shortest" path that we could find from *C*. Our options are now the path we forewent earlier (*C* → *A*, with weight 30) and the path extending from *B* (*C* → *B* → *A*, with weight 5 + 12 = 17). We choose the path with the smaller weight of 17 (*C* → *B* → *A*).
5. We would take note that we have arrived at vertex *A*, which implies that the shortest distance from vertex *C* to vertex *A* is the total weight of the path we took, which is 17. As a result, we would mark that we have found the shortest path to vertex *A* (which is *C* → *B* → *A*).
6. We would take note that all vertices have been marked as found, and as a result, all shortest paths have been found!

Step 3

Up until now, we've been saying at a higher level that **Dijkstra's Algorithm** needs to just

1. Search for the next shortest path that exists in the graph, and
2. See which vertex it discovers by taking that shortest path.

However, how do we get an algorithm to just "search" for the next shortest path? Unfortunately, a computer can't just visually scan the graph all at once like we've been doing. To solve this problem, **Dijkstra's Algorithm** takes advantage of the **Priority Queue** ADT to store the possible paths and uses the total weights of the paths to determine priority.

So how do we go about storing the possible paths? The intuitive approach would be to store the paths themselves in the priority queue (e.g. (*A* → *B* → *C*), (*A* → *B* → *D*), etc.). However, when the priority queue tries to order the paths, it would need to repeatedly recompute the total path weights to perform its priority comparisons. Thus, to save time, it might seem easiest to store all the paths with their respective total path costs (e.g. (1, *A* → *B* → *C*), (3, *A* → *B* → *D*), etc.). However, as our graphs grow larger, this can lead to wasting a lot of space (for example, notice how the path *A* → *B* was repeated twice in the example above). To combat this waste of space, we instead make the realization that, when following a path, it is enough to know only the *one* immediate vertex that is to be explored at the end of the path at each step. As a result, rather than inserting *entire paths* with their respective costs in the priority queue, we store tuples consisting of (*cost*, *vertex to explore*).

For our purposes, a vertex object needs to contain at least these 3 fields:

- *Distance*: the shortest path that was discovered to get to this vertex
- *Previous*: the vertex right before it, so we can keep track of the path and overall structure of the graph
- *Done*: a Boolean field to determine if a shortest path has already been found to this vertex

It is important to note that, since we are storing vertices that are immediately reachable from the current vertex in the priority queue, the same vertex can in fact appear in the priority queue more than once if there exists more than one path to reach it.

Step 4

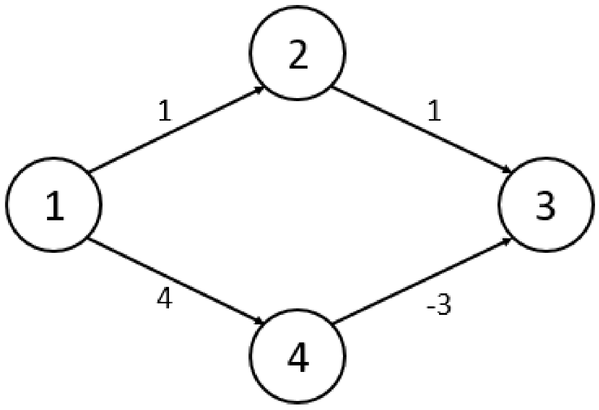
Here is a glimpse at how the pseudocode for **Dijkstra's Algorithm** looks:

```
dijkstra(s):
    // perform initialization step
    pq = empty priority queue
    for each vertex v:
        v.dist = infinity           // the maximum possible distance from s
        v.prev = NULL               // we don't know the optimal previous node yet
        v.done = false              // v has not yet been discovered

    // perform the traversal
    enqueue {0, s} onto pq          // enqueue the starting vertex
    while pq is not empty:
        dequeue {weight, v} from pq
        if v.done == false:         // if the vertex's min path hasn't been discovered yet
            v.done = true
            for each neighbor w of v:
                c = v.dist + weight // c is the total distance to w through v
                if c < w.dist:      // if a smaller-weight path has been found
                                    // (remember, all distances start at infinity!)
                    w.prev = v      // update the node that comes just before w in the path from s
to w
                    w.dist = c      // update the distance of the path from s to w
                    enqueue {c, w} onto
```

As you can see above, **Dijkstra's Algorithm** runs as long as the priority queue is not empty. Moreover, the algorithm never goes back to update a vertex's fields once it is dequeued the first time. This implies that **Dijkstra's Algorithm guarantees that, for each vertex, the first time an instance of it is removed from the priority queue, a shortest path to it has been found.** Why is this true? All other vertices discovered later in the exploration of **Dijkstra's Algorithm** must have at least as great a total path cost to them as the vertex that was just dequeued (if it had a smaller total path cost, then that vertex would have been discovered earlier). Therefore, we couldn't possibly achieve a smaller cost pathway by considering the vertices that appear later in the priority queue. This type of algorithmic approach, where we commit to a decision and never look back, is called a "greedy" approach. Note that the "greedy algorithm" is able to produce an optimal solution each time!

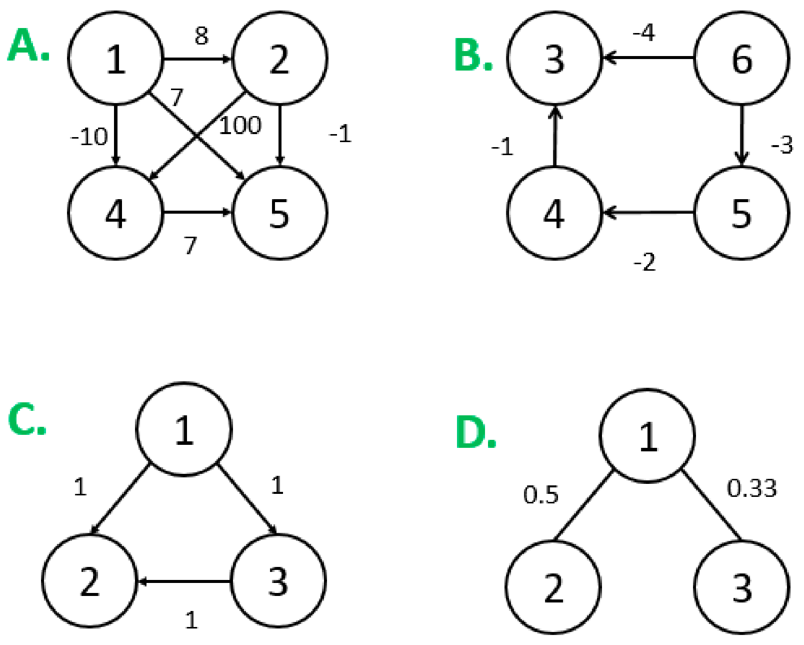
All this time though we have been making one assumption: **no negative weight edges can exist in the graph**! Why? Because if negative weights existed, then we couldn't guarantee that vertices further down a pathway would have at least as great a total path cost to them. By only allowing edges with a weight of 0 or larger, we guarantee that any future edges we encounter in our search can only either increase our total path weight or keep it the same value (if the edge weight is 0). For example, take a look at the graph below:



If **Dijkstra's Algorithm** starts at vertex 1, it would discover vertex 2 first (since it has a total path cost of 1) and then vertex 3 (since it has a total path cost of 1+1 = 2) and lastly vertex 4 (since it has a total path cost of 4). However, the shortest path from vertex 1 to vertex 3 actually has a total path cost of 1! How? If you take the path 1 → 4 → 3, you would end up with a path cost of 4 + (-3) = 1.

Step 5

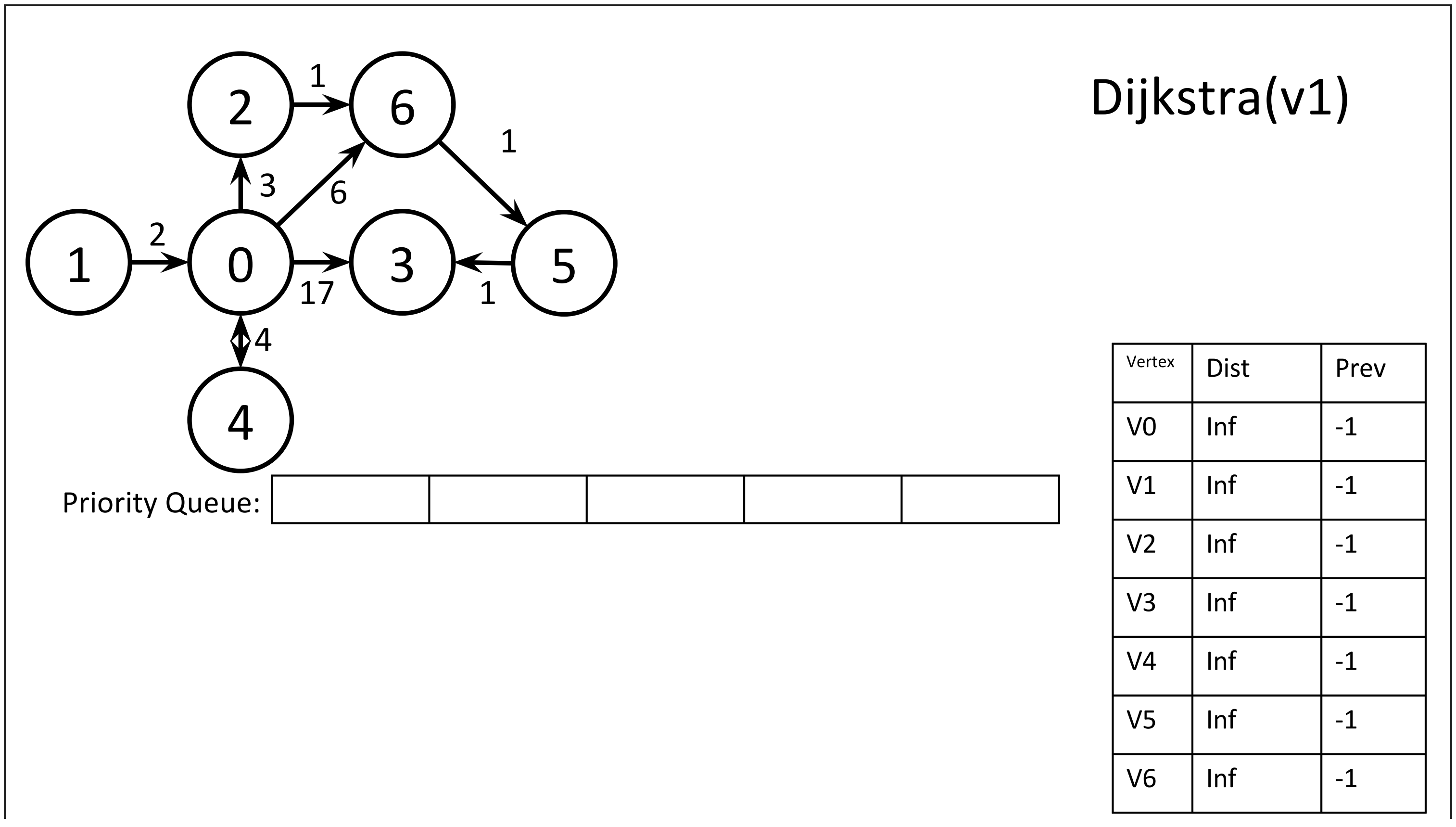
EXERCISE BREAK: Select all valid graphs that **Dijkstra's Algorithm** would be able to successfully find the shortest path starting from any vertex.



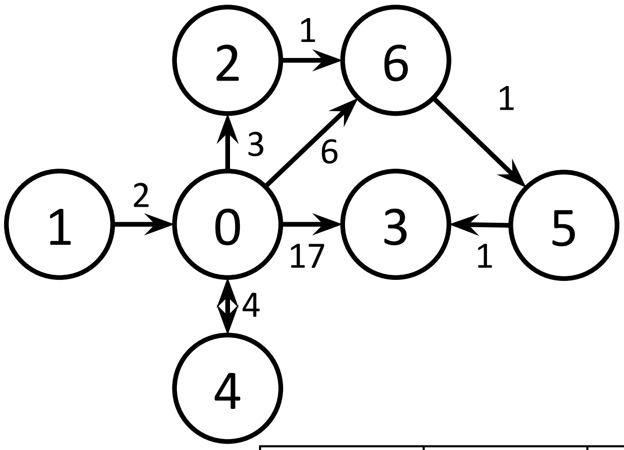
To solve this problem please visit https://stepik.org/lesson/28947/step/5

Step 6

Below is a visual example of how **Dijkstra's Algorithm** takes advantage of the **priority queue** data structure to find the shortest paths in a graph. Use the arrows in the bottom left to walk through the slides.



Dijkstra(v1)



Priority Queue:

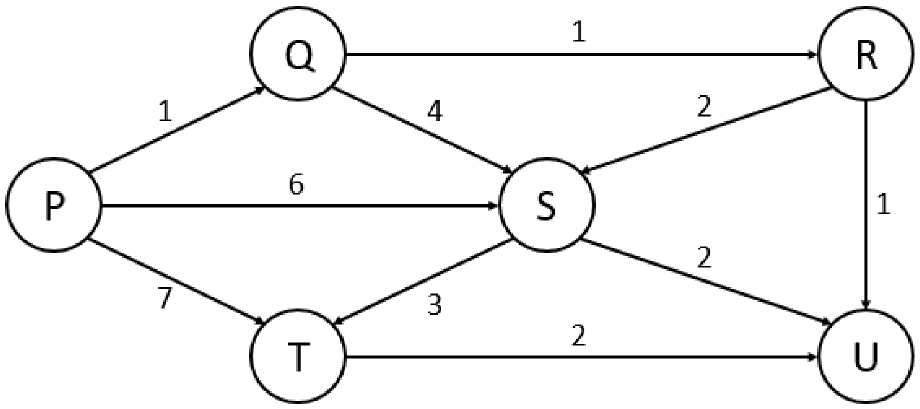
Vertex	Dist	Prev
V0	Inf	-1
V1	Inf	-1
V2	Inf	-1
V3	Inf	-1
V4	Inf	-1
V5	Inf	-1
V6	Inf	-1

| Slide 1 |

Slides

Step 7

EXERCISE BREAK: Sort the vertices from top to bottom (the top most vertex being the first to be explored) in the order that **Dijkstra's Algorithm** would label them as "done" (i.e., the first time **Dijkstra's Algorithm** would dequeue the vertex from the priority queue). Start with vertex ***P***.



To solve this problem please visit https://stepik.org/lesson/28947/step/7

Step 8

As we've seen, **Dijkstra's Algorithm** is able to solve the shortest path problem in weighted graphs, something that **BFS** can't always do. However, what cost do we face when running this more capable algorithm? In other words, what is the worst-case time complexity of **Djikstra's Algorithm**?

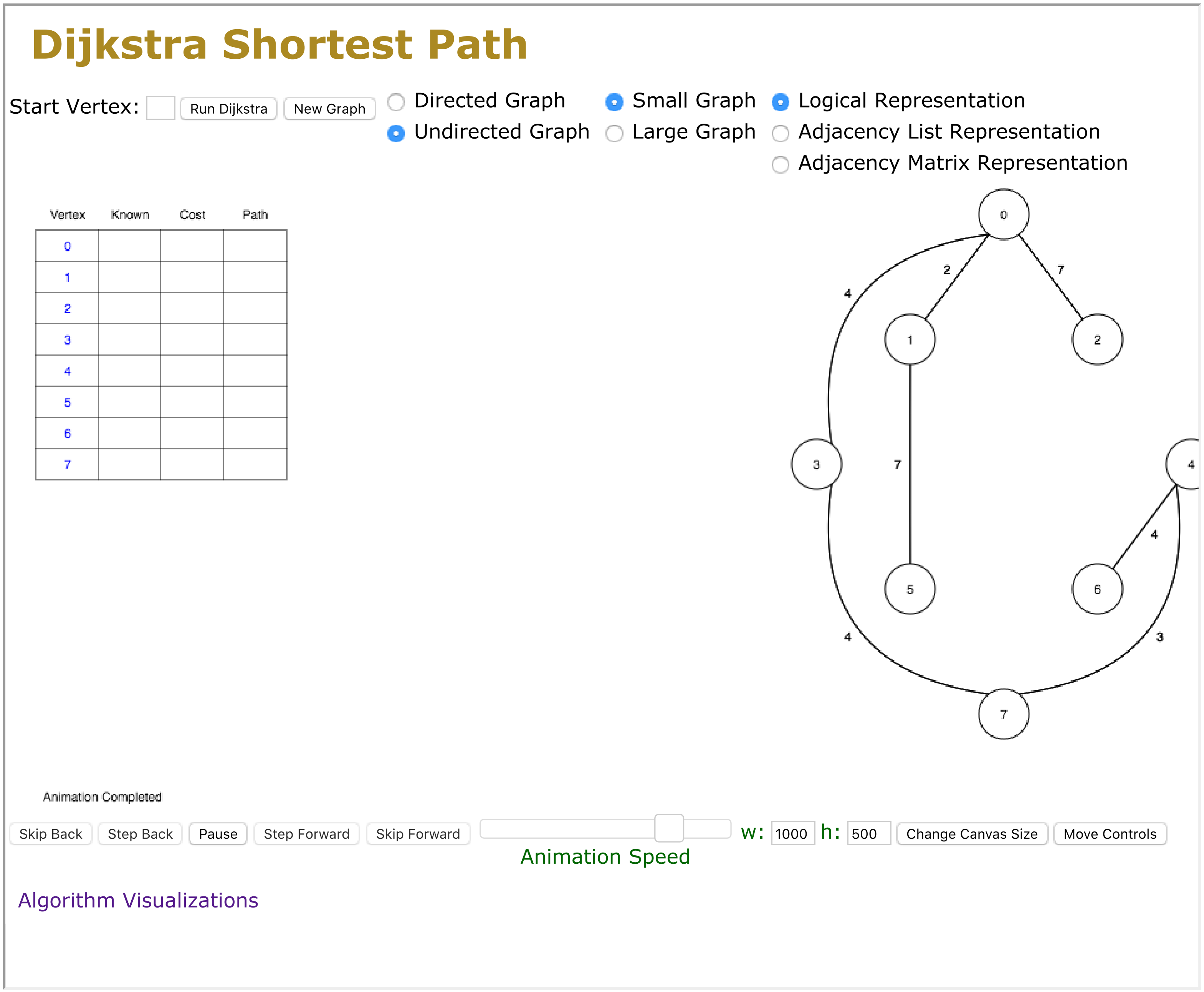
Let's look at the major steps in the algorithm to figure this out:

- Initializing all the fields in a vertex (distance, previous, done): there are |V| such elements, so we have a worst-case time complexity of **O(|*V*|)** here
- Inserting and deleting elements from the Priority Queue:
  - Each edge of the graph (suppose it is implemented as an adjacency list) can cause a single insertion and deletion from the priority queue, so we will be doing at worst **O(|*E*|)** priority queue insertions and deletions
  - The insertion and deletion of a well-implemented priority queue with *N* elements is O(log *N*). In Dijkstra's algorithm, our "*N*" is simply |*E*| in the worst case (because we will have added all |*E*| potential elements into the priority queue), and therefore, *for each element* we insert and delete from the priority queue, we have a worst-case time complexity of **O(log |*E*|)**. Thus, for *all* of the |*E*| elements, we have a worst-case time complexity of **O(|*E*| log |*E*|)**

Taking into account all of the time complexities, the algorithm has an overall worst-case time complexity of **O(|*V*| + |*E*| log |*E*|)**.

Step 9

We've included another visualization tool created by David Galles at the University of San Francisco to let you play around with **Dijkstra's Algorithm** some more. We recommend manually setting the animation speed at the bottom of the screen to low, in order to better see the steps of the algorithm.



Dijkstra Shortest Path

Start Vertex: Run Dijkstra New Graph

Directed Graph [✓ Undirected Graph]

[✓ Small Graph] Large Graph

[✓ Logical Representation] Adjacency List Representation Adjacency Matrix Representation

Vertex	Known	Cost	Path
0			
1			
2			
3			
4			
5			
6			
7			

Animation Completed

Skip Back Step Back Pause Step Forward Skip Forward Animation Speed w: 1000 h: 500 Change Canvas Size Move Controls

Algorithm Visualizations

Step 10

EXERCISE BREAK: Which of the following are valid ways to modify an unweighted graph to have **Dijkstra's Algorithm** be able to accurately produce the shortest paths from one vertex to all other vertices? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/28947/step/10