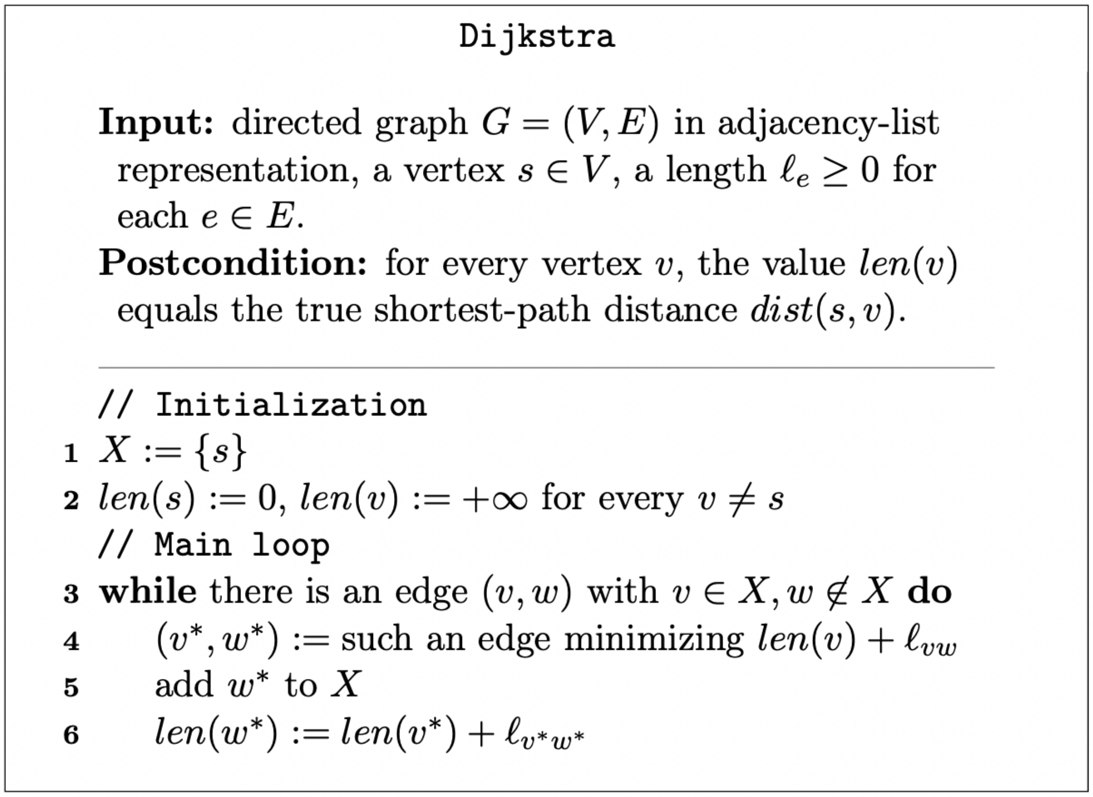
The goal for project one is to implement Dijkstra's algorithm using two different techniques, as given in Sections 9.2 and 10.4 of the textbook. The first technique is a straightforward implementation, and the second one is the application of heap. Algorithms are used to solve the single-source shortest path problem in different directed graphs. A directed graph is given with the equation $G = (V, E)$, which has V as the number of vertices, E as the number of edges, starting vertex s, and nonnegative edge lengths. In this report, I am going to describe the algorithms, particularly what I have done in coding, predict the time complexity, and compare the empirical performances of algorithms with each other and with predicted performance.

The first implementation is the straightforward algorithm with the following pseudocode.



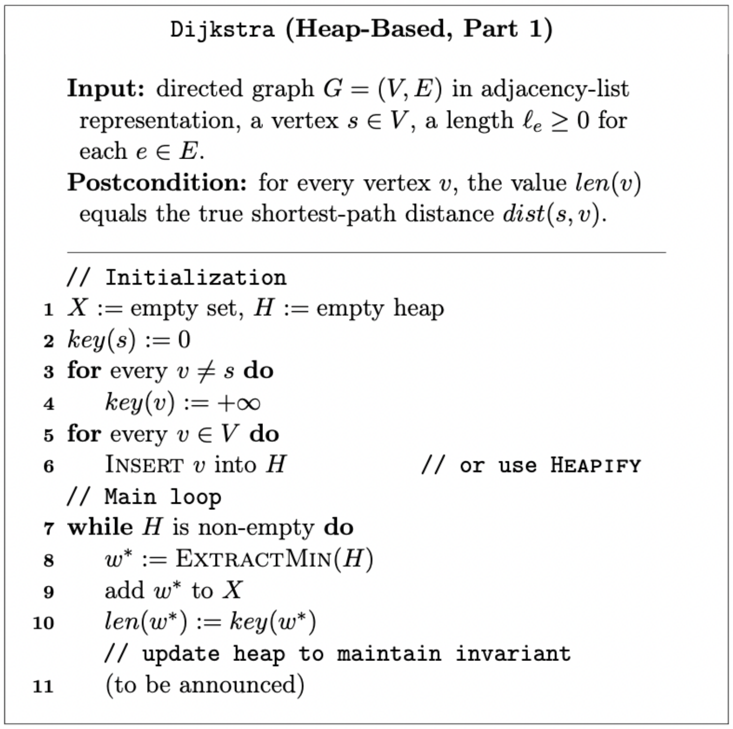
Dijkstra

Input: directed graph $G = (V, E)$ in adjacency-list representation, a vertex $s \in V$, a length $\ell_e \geq 0$ for each $e \in E$.
Postcondition: for every vertex v, the value $len(v)$ equals the true shortest-path distance $dist(s, v)$.

```
  // Initialization
1 $X := \{s\}$
2 $len(s) := 0$, $len(v) := +\infty$ for every $v \neq s$
  // Main loop
3 while there is an edge $(v, w)$ with $v \in X, w \notin X$ do
4     $(v^*, w^*) :=$ such an edge minimizing $len(v) + \ell_{vw}$
5     add $w^*$ to $X$
6     $len(w^*) := len(v^*) + \ell_{v^*w^*}$
```

In the beginning, the distance of the starting vertex $len(s)$ is set as 0 and other distances are infinity. Each main loop iteration processes one unvisited vertex, dealing with every reachable vertices from that vertex. If there is such an edge minimizing the Dijkstra score (distance from starting vertex to reachable vertex), which is defined as $len(v) + l_{vw}$, it updates the current distance to the sum of the distance from the starting vertex and the weight of reachable vertex. When the algorithm has already dealt with a vertex, it marks the vertex as visited. The algorithm continues doing it until there is non unvisited vertex. The algorithm returns distances of every vertex from the starting vertex. The outer loops takes linear running time $O(n)$, where n is the number of vertices. I have two inner loops: one for finding a vertex with current minimum distance and one for updating the distances of reachable vertices. The first inner loop goes through every vertex and check if one is the one we need - unvisited and has the minimum distance - or not, which gives $O(n)$ running time. The second inner loop goes through every reachable edge from the current vertex, which gives $O(n)$ with n is total edges. The predicted running time for two inner loops is $2n$, which results in $O(n)$. At the end, the program gives $O(mn)$ time complexity, where m is the total number of vertices and n is the total number of edges.

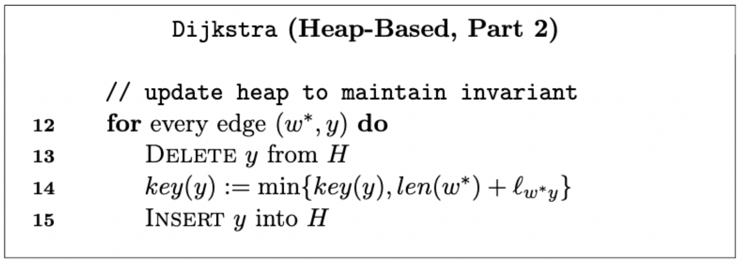
The second implementation is the application of heaps for Dijkstra's algorithm. In every iteration, the algorithm identifies the distance from starting vertex to connecting vertex with the minimum Dijkstra score. It minimizes the number of computations from linear time to logarithmic time by only adding the vertices with minimum scores. The following pseudocode is the implementation of Dijkstra's algorithm using heaps.



Dijkstra (Heap-Based, Part 1)

Input: directed graph $G = (V, E)$ in adjacency-list representation, a vertex $s \in V$, a length $\ell_e \geq 0$ for each $e \in E$.
Postcondition: for every vertex v, the value $len(v)$ equals the true shortest-path distance $dist(s, v)$.

```
// Initialization
X := empty set, H := empty heap
$key(s) := 0$
for every $v \neq s$ do
    $key(v) := +\infty$
for every $v \in V$ do
    INSERT $v$ into $H$                // or use HEAPIFY
// Main loop
while $H$ is non-empty do
    $w^* :=$ EXTRACTMIN($H$)
    add $w^*$ to $X$
    $len(w^*) := key(w^*)$
    // update heap to maintain invariant
    (to be announced)
```

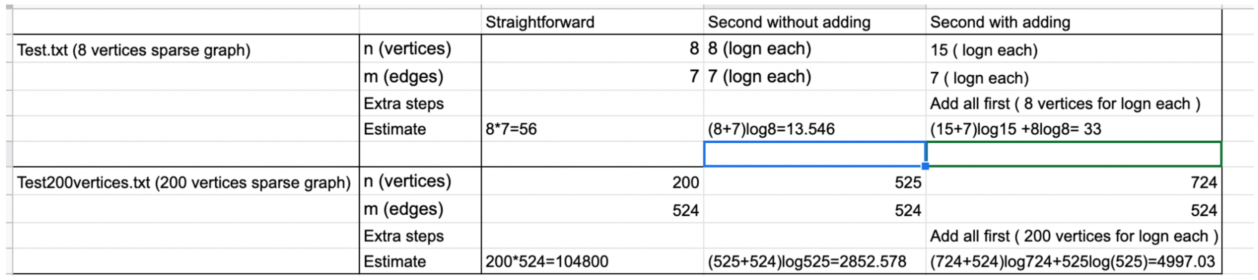


Dijkstra (Heap-Based, Part 2)

```
    // update heap to maintain invariant
    for every edge $(w^*, y)$ do
        DELETE $y$ from $H$
        $key(y) := \min\{key(y), len(w^*) + \ell_{w^*y}\}$
        INSERT $y$ into $H$
```

In the beginning, the distance of the starting vertex $len(s)$ is set as 0 and other distances are infinity. I use priority queue as the implementation of heap with the comparator of weight: less weight has the higher priority. The next step is to insert every vertices to heap. While the heap is not empty, poll the first vertex, which is the vertex with minimum key (distance from the starting vertex), and mark that to be visited. The key of that vertex is updated to be the corresponding distance. Inside the while loop, there is another for loop. With every edge, it removes the connecting vertex from the heap, updates the key to be the Dijkstra's score $len(w^{*}) + l_{yw^{*}}$ if it is smaller than the current key, and reinserts the connecting vertex with the new key. The running time for add() method in heap is $O(logn)$. By inserting every vertex into the heap, the running time is $O(nlogn)$, where n is the total number of vertices. The next while loop gives a total of $2m$ operations and the running time of $O(mlogn)$, where m is the total number of edges times the running time of add() method. Adding up those two running time, we have the predicted time complexity of $O((m + n)logn)$. I provided a second implementation of heap for this logic. The only difference is I added only the first vertex to the queue and started doing the second while loop immediately. I suspected this would give the running time of $O(nlogn)$.

For the small test case, the number of operations in the first implementation is 56, and that of the second implementation is 13.5 with adding all the vertices and 33 without adding all the vertices. The first place comes for the second implementation without adding all the vertices, followed by other second implementation and the first implementation. With the test case of 200 vertices, 104800 is the number of operations in the first implementation, 2852.5 is that of the second implementation with adding all the vertices, and 4997 is that of second implementation without adding all the vertices. The results illuminate my analyze for the running time order (fastest to slowest): the second implementation with adding all the vertices, the other second implementation, and the first implementation. For the small test case, there would be little difference between the straightforward and heap implementations. However, the greater the data set, the bigger the difference. With sparse graphs, the heap's running time would be a lot smaller than the straightforward implementation. With dense graphs, the gap between would be less significant. The second implementation without adding all vertices in the beginning will be the most efficient among those three.



		Straightforward	Second without adding	Second with adding
Test.txt (8 vertices sparse graph)	n (vertices)	8	8 (logn each)	15 (logn each)
	m (edges)	7	7 (logn each)	7 (logn each)
	Extra steps			Add all first (8 vertices for logn each)
	Estimate	8*7=56	(8+7)log8=13.546	(15+7)log15 +8log8= 33
Test200vertices.txt (200 vertices sparse graph)	n (vertices)	200	525	724
	m (edges)	524	524	524
	Extra steps			Add all first (200 vertices for logn each)
	Estimate	200*524=104800	(525+524)log525=2852.578	(724+524)log724+525log(525)=4997.03