Problem Set 12 - Solution

Graphs: Traversals, Topological Sorting, Dijkstra's Algorithm

1. You are given a directed acyclic graph:

```
class Neighbor {
   public int vertex;
   public Neighbor next;
   ...
}

class Vertex {
   String name;
   Neighbor neighbors; // adjacency linked lists for all vertices
}

public class Graph {
   Vertex[] vertices;

   // returns an array with the names of vertices in topological sequence
   public String[] topsort() {
     // FILL IN THIS METHOD
     ...
   }
   ...
}
```

Complete the `topsort` method implementation to toplogically sort the vertices using **using BFS (breadth-first search)**. Assume that the graph has already been read in. Implement helper methods as needed. You may use the following `Queue` class:

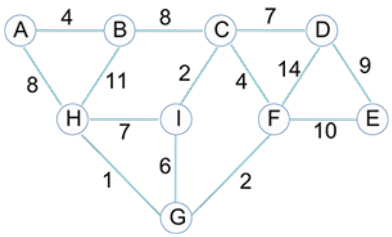
```
public class Queue<T> {
   ...
   public Queue() {...}
   public void enqueue(T item) {...}
   public T dequeue() throws NoSuchElementException {...}
   public boolean isEmpty() {...}
   ...
}
```

SOLUTION

```
   // returns an array with the names of vertices in topological sequence
   public String[] topsort()
   throws Exception {
      int[] indeg = indegrees();
      int topnum=0;
      String[] tops = new String[vertices.length];
      Queue queue = new Queue();

      // find all vertices with indegree zero, assign them topological numbers, and enqueue
      for (int i=0; i < indeg.length; i++) {
          if (indeg[i] == 0) {
             tops[topnum++] = vertices[i].name;
             queue.enqueue(i);
          }
      }

      // loop until queue is empty
      while (!queue.isEmpty()) {
          int v = queue.dequeue();
          for (Neighbor nbr=vertices[v].neighbors; nbr != null; nbr=nbr.next) {
              indegrees[nbr.vertex]--;
              if (indegrees[nbr] == 0) {
                 tops[topnum++] = vertices[nbr.vertex].name;
                 queue.enqueue(nbr.vertex);
              }
          }
      }

      return tops;
   }

   // computes indegrees for all vertices
   private int[] indegrees() {
      int[] indeg = new int[vertices.length];
      for (int i=0; i < vertices.length; i++) {
         indeg[i] = 0;
      }
      for (int i=0; i < vertices.length; i++) {
          for (Neighbor nbr=vertices[i].neighbors; nbr != null; nbr=nbr.next) {
              indeg[nbr.vertex]++;
          }
      }
      return indeg;
   }
```

2. **DONE IN CLASS**

Suppose you are given this undirected graph in which the vertices are towns, and the edges are toll roads between them. The weight of an edge is the dollar amount of toll.



Use Dijsktra's shortest paths algorithm to determine the minimum toll route from A to all other cities.

- Show each step of the algorithm in tabular form. Here's the table after the initial step:

Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]	D[H]	D[I]
A	4,A	8,A	∞	∞	∞	∞	∞	∞

  Note that along with the distance, the "previous" vertex is also shown.

- Draw the shortest path tree induced on the graph.

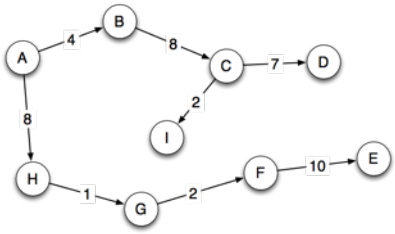
SOLUTION

-

Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]	D[H]	D[I]
A	4,A	∞	∞	∞	∞	∞	8,A	∞
B		12,B	∞	∞	∞	∞	8,A	∞
H		12,B	∞	∞	∞	9,H		15,H
G		12,B	∞	∞	11,G			15,H
F		12,B	25,F	21,F			15,H	
C			19,C	21,F			14,C	
I			19,C	21,F				
D				21,F				
E								

  Note that along with the distance, the "previous" vertex is also shown.

- The shortest path tree induced on the graph:



Problems 3-8 that follow are on analyzing the worst case big O running time of Dijkstra's shortest paths algorithm. Assume the graph has *n* vertices, and *e* edges with positive weights. And, except for problem #8 the graph is stored in adjacency linked lists.

The first step in the analysis is to identify the primary activities that are to be counted towards the running time. These are:

- (A)dding a vertex to the fringe
- (P)icking the minimum distance vertex from the fringe
- (C)hecking the neighbors of a vertex, for possible distance update
- (U)pdating (lowering) the distance of a vertex that is in the fringe

We will ignore the time to set the initial distance of a vertex. The reason is it takes unit time to set the distance in the distance array, but is done done as a part of the (A)dding a vertex to the fringe, which will take at least unit time. So as far as big O is concerned, determining the time to (A)dd fringe vertices will also account for distance initialization.

We will also ignore the time to set the previous vertex. Using a similar argument as above, it takes unit time to set the previous vertex in the previous array. But this is done as a part of (A)dding a vertex to the fringe, or (U)pdating the distance of a vertex, and the (A)dd and (U)pdate will take at least unit time themselves. So accounting for the (A)dd and (U)pdate times will suffice as far as the big O is concerned.

The fringe is the variable part as far the data structures are concerned (since distance and previous vertex are recorded in arrays), and the worst case depends on exactly how the fringe is implemented. You will compute the running times for different versions of the fringe in problems 5-7.

---

3. The running time has to do with how many vertices are in the fringe, since the time to (A)dd, (P)ick, and (U)pdate all depend on the fringe size. To get started:
   1. What is the *best* case scenario, for which these operations can be performed in the fastest time possible? The scenario is both the structure of the graph, and the sequence in which vertices and added and picked out of the fringe.
   2. What is the *worst* case scenario?

   SOLUTION

   1. In the best case, the graph is a single chain of vertices. Starting with the neighbor of the source, only one vertex is added to the fringe in each iteration of the algorithm, and that vertex is then picked out in the next iteration. This results in O(1) time for (A)dd, (P)ick and (U)pdate.
   2. In the worst case, the source is connected to the all the other *n-1* vertices in the graph, which results in the largest possible fringe from the very first step, when the neighbors of the source are (A)dded to the fringe. Since the neighbors are added one at a time, every time a neighbor is added, the fringe is as large as it can possibly be. (P)icking the minimum distance vertex, and (U)pdating distances of vertices in the fringe are also done every time on the largest possible fringe.

---

4. What is the total running time, through the entire run of the algorithm, for (C)hecking the neighbors of each vertex for possible distance update? Does this depend on the fringe?

   SOLUTION

The running time for (C)hecking the neighbors of each vertex does not depend on the fringe, since the neighbors and edge weights are obtained from the adjacency linked lists, and the current distance is obtained from the distance array.

The total number of neighbors of all vertices is *e*, if the graph is directed, and *2e*, if the graph is undirected. Each neighbor contributes one unit of time toward the distance check. And since every vertex has to be located in the adjacency linked lists array first, this adds *n* to the running time. The total time then is *O(n+e)*.

---

5. If the fringe was implemented as an unordered linked list (not arranged in any specific order of distances), what would be the big O worst case running time of each of the A, P, and U components of the algorithm, through the entire run from start to end? What would be the total worst case big O running time of the algorithm?

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since fringe is an unordered linked list, adds can be done at the front of the linked list in O(1) time each, so the total time is *(n-1)*, which is *O(n)*
   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Identifying the minimum distance vertex in an unordered linked list of size *k* will take *(k-1)* time units. So the series of minimum picks will add up as follows:

     $$(n-2) + (n-3) + \ldots + 2 + 1 = (n-1)*(n-2)/2$$

     which is $O(n^2)$
   - (U): Updating distance of vertex in fringe.
     The updates are done in the distance array, the structure of the fringe itself is not relevant. Each update can be done in O(1) time. In the worst case, every neighbor distance check results in an update. Since there are *e* neighbor distance checks in all, this gives a total time of *O(e)*

   Adding these times, and factoring in the time of *O(n+e)* for the (C) component worked out in the previous problem, gives us a total time of:

   $$O(n) + O(n^2) + O(e) + O(n+e) = O(n^2)$$

---

6. Repeat the analysis for a fringe implemented as a linked list maintained in ascending order of distances.

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since fringe is an ordered linked list, each add needs to first find the correct spot, which will take *O(k)* in the worst case for a linked list of length *k*. So the total time is:

     $$1 + 2 + \ldots + (n-3) + (n-2) = (n-1)*(n-2)/2 = O(n^2)$$

   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Since the linked list is arranged in ascending order of distances, the minimum distance vertex is the first item, and can be accessed in unit time. So the total time is *O(n)*
   - (U): Updating distance of vertex in fringe.
     While the actual updates are done in the distance array, because the fringe is maintained in ascending order of distances, an update would in general require a repositioning of the vertex in the linked list.

     The new distance would need to be compared with the distance of the vertex before it in the linked list. If the new distance is less, then another comparison would need to made with the vertex two spots before this vertex, etc. In the worst case, this sequence of comparisons would go all the way to the front of the list, so that if the list is of length *k*, then *k* comparisons would need to be made.

     For the worst case for all updates, we assume that all *e* updates are done on the maximum possible fringe size, which is *n-1*. The time for a single update on this fringe size would be O(*e*), as outlined in the previous paragraph, so time for all *e* updates would be O(*en*)

   Adding these times, and factoring in the time of *O(n+e)* for the (C) component, gives us a total time of:

   $$O(n^2) + O(n) + O(en) + O(n+e) = O(n^2+en)$$

---

7. Repeat the analysis for a fringe implemented as a min-heap that supports inserts, deletes, and priority updates in *O(log n)* time.

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since the fringe is a min-heap, adding (which includes sifting up) in a heap of size *k* would take *O(log k)* time. So the total time is:

     $$\log 1 + \log 2 + \log 3 \ldots + \log (n-1) = \log (n-1)! = O(n\log n)$$

   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Again, since deleting from a heap of size *k* would take *O(log k)* time, the total time is:

     $$\log (n-1) + \log (n-2) + \ldots + \log 2 + \log 1 = \log (n-1)! = O(n\log n)$$

   - U: Updating distance of vertex in fringe.
     Updating the priority (which here means decreasing the distance) of an item in a heap of size *k* takes *O(log k)* time. For the worst case for all updates, we assume that all *e* updates are done on the maximum possible fringe size, which is *n-1*, so time for all *e* updates would be O(*elogn*)

   The total running time, adding the time for all components, and bringing in the time for (C) is:

   $$O(n\log n) + O(n\log n) + O(e\log n) + O(n+e) = O((n+e)\log n)$$

---

8. Repeat the analysis for a fringe implemented as a min-heap that supports inserts, deletes, and priority updates in *O(log n)* time, for a graph stored in an adjacency *matrix* format instead of adjacency linked lists.

   SOLUTION

   In #7 above, we analyzed the running time for an updatable-heap based fringe, and it turned out to be O((*n*+*e*)*logn*). However this assumed that the graph was stored in adjacency linked format, which resulted in an O(*(n+e)*) running time for the (C) component. But if the graph is stored in an adjacency matrix, the running time of

the (C) component would change to $O(n^2)$. All other components would have the same running time since they all depend only on the fringe.

Since the (C) component was not the dominant one in the original analysis, we can simply replace the old running time of (C) with $O(n^2)$, which changes the running time to:

$$O(nlogn) + O(elogn) + O(n^2) = O(n^2+elogn)$$

(Since n^2 is of a higher order than *nlogn*, the *nlogn* term is taken out)