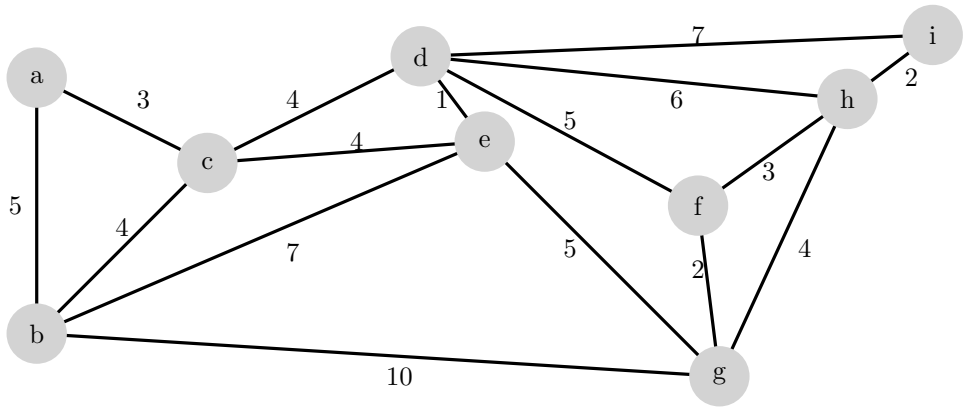
The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework05.html`



0.

(a) Illustrate the execution of breadth-first search on the graph above, treating it as an unweighted graph (all edges have a weight of 1), and with vertex e as the source.

Solution. Starting at e, breadth first stores the following contents in its (FIFO) queue and records the following distances

step	queue	a	b	c	d	e	f	g	h	i
init	e	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$e \to g$	g	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	1	$+\infty$	$+\infty$
$e \to d$	g, d	$+\infty$	$+\infty$	$+\infty$	1	0	$+\infty$	1	$+\infty$	$+\infty$
$e \to b$	g, d, b	$+\infty$	1	$+\infty$	1	0	$+\infty$	1	$+\infty$	$+\infty$
$e \to c$	g, d, b, c	$+\infty$	1	1	1	0	$+\infty$	1	$+\infty$	$+\infty$
$g \to f$	d, b, c, f	$+\infty$	1	1	1	0	2	1	$+\infty$	$+\infty$
$g \to h$	d, b, c, f, h	$+\infty$	1	1	1	0	2	1	2	$+\infty$
$d \to i$	b, c, f, h, i	$+\infty$	1	1	1	0	2	1	2	2
$b \to a$	c, f, h, i, a	2	1	1	1	0	2	1	2	2

In all subsequent iterations, each vertex being processed will not result in any new updates to distances, and all have at this point been computed. •

(b) Illustrate the execution of Dijkstras algorithm on the graph above, together with e as the source.

Solution. In the following, we show the contents of the priority queue in no particular order. The priority of the elements stored is specified by the distances listed afterwards.

step	pqueue	a	b	c	d	e	f	g	h	i
init	e	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
relax $e \to g$	g	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	5	$+\infty$	$+\infty$
relax $e \to d$	d, g	$+\infty$	$+\infty$	$+\infty$	1	0	$+\infty$	5	$+\infty$	$+\infty$
relax $e \to b$	b, d, g	$+\infty$	7	$+\infty$	1	0	$+\infty$	5	$+\infty$	$+\infty$
relax $e \to c$	b, c, d, g	$+\infty$	7	4	1	0	$+\infty$	5	$+\infty$	$+\infty$
relax $d \to c$	b, c, g	$+\infty$	7	4	1	0	$+\infty$	5	$+\infty$	$+\infty$
relax $d \to e$	b, c, g	$+\infty$	7	4	1	0	$+\infty$	5	$+\infty$	$+\infty$
relax $d \to f$	b, c, f, g	$+\infty$	7	4	1	0	6	5	$+\infty$	$+\infty$
relax $d \to h$	b, c, f, g, h	$+\infty$	7	4	1	0	6	5	7	$+\infty$
relax $d \to i$	b, c, f, g, h, i	$+\infty$	7	4	1	0	6	5	7	8
relax $c \to a$	a, b, f, g, h, i	**7**	7	4	1	0	6	5	7	8
relax $c \to b$	a, b, f, g, h, i	7	7	4	1	0	6	5	7	8
relax $c \to e$	a, b, f, g, h, i	7	7	4	1	0	6	5	7	8
relax $c \to d$	a, b, f, g, h, i	7	7	4	1	0	6	5	7	8
relax $g \to b$	a, b, f, h, i	7	7	4	1	0	6	5	7	8
relax $g \to e$	a, b, f, h, i	7	7	4	1	0	6	5	7	8
relax $g \to f$	a, b, f, h, i	7	7	4	1	0	6	5	7	8
relax $g \to h$	a, b, f, h, i	7	7	4	1	0	6	5	7	8
relax $f \to d$	a, b, h, i	7	7	4	1	0	6	5	7	8
relax $f \to h$	a, b, h, i	7	7	4	1	0	6	5	7	8
relax $f \to g$	a, b, h, i	7	7	4	1	0	6	5	7	8
relax $a \to b$	b, h, i	7	7	4	1	0	6	5	7	8
relax $a \to c$	b, h, i	7	7	4	1	0	6	5	7	8
relax $b \to a$	h, i	7	7	4	1	0	6	5	7	8
relax $b \to c$	h, i	7	7	4	1	0	6	5	7	8
relax $b \to e$	h, i	7	7	4	1	0	6	5	7	8
relax $b \to g$	h, i	7	7	4	1	0	6	5	7	8
relax $h \to d$	i	7	7	4	1	0	6	5	7	8
relax $h \to i$	i	7	7	4	1	0	6	5	7	8
relax $h \to f$	i	7	7	4	1	0	6	5	7	8
relax $h \to g$	i	7	7	4	1	0	6	5	7	8
relax $i \to d$	–	7	7	4	1	0	6	5	7	8
relax $i \to h$	–	7	7	4	1	0	6	5	7	8

(c) Illustrate the execution of the Prim Dijkstra-Jarnik algorithm on the graph above.

Solution. This algorithm can start at any vertex. Let's assume it starts with a (as the only node in the tree). We add edges to this tree in the following order:
a—c: connects c to tree,
b—c: connects b to tree (or c—e, or c—d)
c—e: connects e to tree
d—e: connects d to tree
d—f: connects f to tree (or e—g)
f—g: connects g to tree

f—-h: connects h to tree
h—i: connects i to tree. •

(d) Illustrate the execution of Kruskal's algorithm on the graph above.

Solution. The algorithm starts with a forest of individual vertices, and adds edges in the following order:
d—e
h—i (or f—g)
f—g
f—h (or a—c)
a—c
c—b (or c—d or c—-e)
c—d (or c—e)
d—f (or e—g) •

(e) Implement Dijkstra's algorithm in Java.

Solution. The following is a possible implementation of Dijkstra's algorithm in Java. In particular it is the variant that returns the shortest path tree.

There are several issues in translating the algorithm discussed into code. The first is how to represent a weighted graph. If we choose an adjacency matrix as a starting point, we can replace the "1" entries with actual weights, and leaving "0" entries to represent the lack of an edge. In the following solution, we chose to start with the adjacency list representation, but replaced the lists of neighbors with maps of neighbors to edge weights. For example, if v has exactly two neighbors, w and x, such that $v \to w$ has weight a and $v \to x$ has weight b, this is represented as an entry in the map, where the key is v and the value is the map mapping w to a and x to b.

The next issue is how to notify the priority queue that there has been a change to the distance of a vertex, given that it has no decreaseKey() method, which is pretty typical of priority queues. The solution (although perhaps less efficient) is to remove the vertex, update its distance, and reinsert it with its new priority (ie distance).

Finally, in order to delete the vertex from the priority queue, we need to be able to locate it within the priority queue. Since it is actually being stored as a map entry (which is necessary to have both the vertex, and its priority), we use an auxiliary map data structure, which we will call locator, and which maps vertices to the map entry or key-value pair consisting of the vertex itself and its priority.

To get the most out of this solution, it is recommended that you read through the code with the goal to understand all the main ideas, and then to reimplement it entirely yourself without looking.

Without further ado, the code.

```
import java.util.*;

public class Dijkstra {

   // Representing a weighted graph as something similar to an adjacency list, but instead of mapping each vertex
   // to a list of neighbors, map each vertex v to a *map* of its neighbors w to edge weight on v->w.
   public static <T> Map<T, T> shortestPaths(Map<T, Map<T, Integer>> weightedGraph, T source) {
      Map<T, Integer> dist = new HashMap<>();
      Map<T, T> prev = new HashMap<>();
      PriorityQueue<Map.Entry<T, Integer>> priorityQueue = new PriorityQueue<>(Comparator.comparing(Map.Entry::getValue));
      Map<T, Map.Entry<T, Integer>> locator = new HashMap<>(); // used to locate priority queue elements by their T-value.

      Integer infinity = sumOfWeights(weightedGraph);
      for (T v : weightedGraph.keySet()) {
         dist.put(v, infinity);
         prev.put(v, null);
      }
      dist.put(source, 0);
      priorityQueue.addAll(dist.entrySet());
      for (Map.Entry<T, Integer> entry : dist.entrySet())
         locator.put(entry.getKey(), entry);

      while (!priorityQueue.isEmpty()) {
         Map.Entry<T, Integer> vEntry = priorityQueue.peek();
         priorityQueue.remove(vEntry);
         T v = vEntry.getKey();

         for (Map.Entry<T, Integer> edge : weightedGraph.get(v).entrySet()) {
            T w = edge.getKey();
            int weight = edge.getValue(); // weight of edge v -> w
            if (dist.get(v) + weight < dist.get(w)) {
               // update dist of w in the dist map and in the priority queue
               Map.Entry<T, Integer> distWEntry = locator.get(w); // get the map entry: (w, dist.get(w))
               priorityQueue.remove(distWEntry); // (Java has no decreaseKey, instead remove w...
               distWEntry.setValue(dist.get(v) + weight); // equivalent to dist.put(w, dist.get(v) + weight)
               priorityQueue.add(distWEntry); // ... after update to dist.get(w), re-add w)
               // update prev of w to v
               prev.put(w, v);
            }
         }
      }
      return prev;
   }

   private static <T> Integer sumOfWeights(Map<T, Map<T, Integer>> weightedGraph) {
      Integer sum = 0;
      for (Map<T, Integer> neighbors : weightedGraph.values())
         for (Integer weight : neighbors.values())
            sum += weight;
      return sum;
   }

   public static void main(String[] args) {
      Map<Integer, Integer> m0 = new HashMap<>();
      m0.put(1, 6);
```

```
        m0.put(3, 2);
        Map<Integer, Integer> m1 = new HashMap<>();
        m1.put(0, 6);
        m1.put(2, 4);
        m1.put(3, 3);
        m1.put(4, 1);
        Map<Integer, Integer> m2 = new HashMap<>();
        m2.put(1, 4);
        m2.put(4, 6);
        Map<Integer, Integer> m3 = new HashMap<>();
        m3.put(0, 2);
        m3.put(1, 3);
        m3.put(4, 1);
        Map<Integer, Integer> m4 = new HashMap<>();
        m4.put(1, 1);
        m4.put(2, 6);
        m4.put(3, 1);
        Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();
        graph.put(0, m0);
        graph.put(1, m1);
        graph.put(2, m2);
        graph.put(3, m3);
        graph.put(4, m4);
        System.out.println(shortestPaths(graph, 0));
    }
}
```

•

1. (a) What is the running time of this algorithm, using O-notation, as a function of the number n of vertices in the input graph?

   Solution. Dijkstra's algorithm runs in

   $$O(1 \cdot \mathrm{T}(\text{make-queue})(n) + n \cdot \mathrm{T}(\text{extract-min})(n) + m \cdot \mathrm{T}(\text{decrease-key})(n)).$$

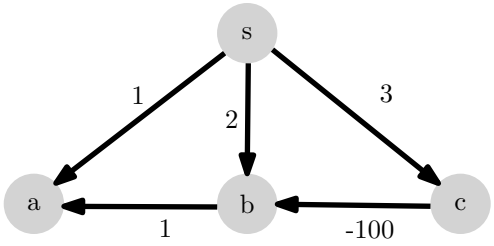
   Using an unsorted list, which takes $O(n)$ to make-queue with n items, $O(n)$ time per extract-min operation and constant time per decrease-key operation, that comes out to $O(n^2 + m)$. Using the fact that the number of edges is $m = O(n^2)$ in a "normal" graph (one that does not allow multiple edges going in the same direction between the same two vertices), this simplifies to $O(n^2)$. •

   (b) Would this algorithm be a better or worse choice than the more usual form of Dijkstra's algorithm using a binary heap, for this type of graph? Explain your answer.

   Solution. Better since the running time using a binary heap is $O(n \log n + m \log n)$, which simplifies to $O(m \log n)$, assuming that $m \geq n$, a reasonable assumption for a graph that is an input into Dijkstra's algorithm. (If there were fewer edges than vertices (technically, $m < n - 1$), the graph would be disconnected, and certain vertices would not be reachable from the source.) •

2. Give an example of an weighted directed graph, G, with negative-weight edges but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex v.

   Solution. Consider the following example.



   In this example, we first pull s out of the priority queue, and relax the three edges ($s \to a$, $s \to b$ and $s \to c$) coming out of s in no particular order. This sets

   - dist[a] = 1
   - dist[b] = 2
   - dist[c] = 3

   Since a is the closest vertex to s by dist-value, it is pulled out of the priority queue. a has no outgoing edges, so no edges are relaxed.

   Next we pull b out of the priority queue and relax its only outgoing edge $b \to a$. This does not produce a better path for a, which remains at a distance of 1 from the source.

   Finally, we pull c out of the priority queue and relax its edge $c \to b$, which resets the distance of b to -97.

   So the final distances produced by Dijkstra's algorithm are

   - dist[a] = 1
   - dist[b] = -97
   - dist[c] = 3

   However, the shortest path to a is actually $s \to c \to b \to a$, which means the distance from s to a is actually 3 – 100 + 1 = -96 and not 1. In other words, Dijkstra did not find the correct distance to a.