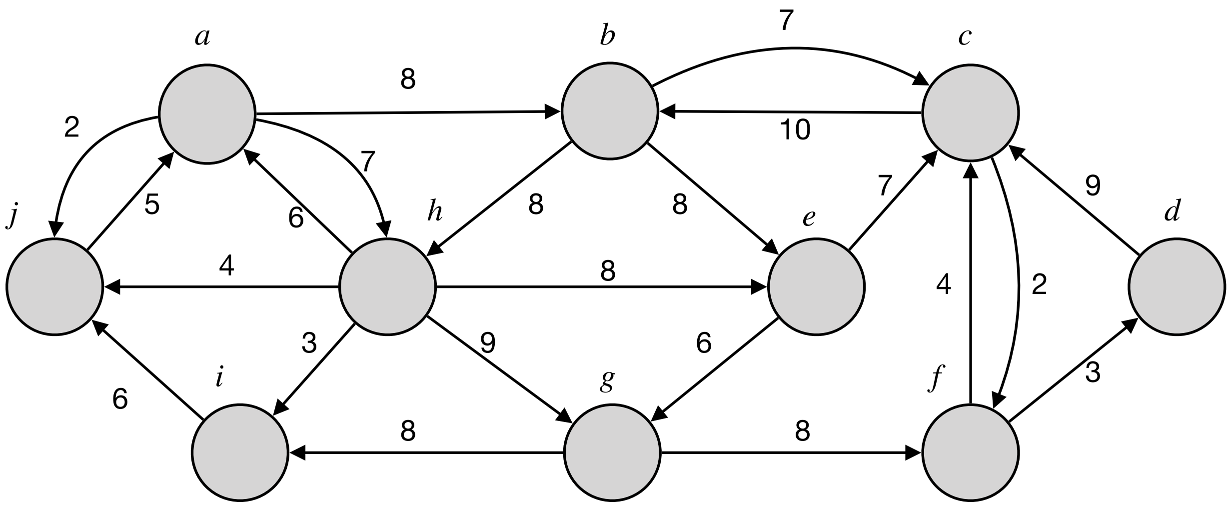
Written Part:

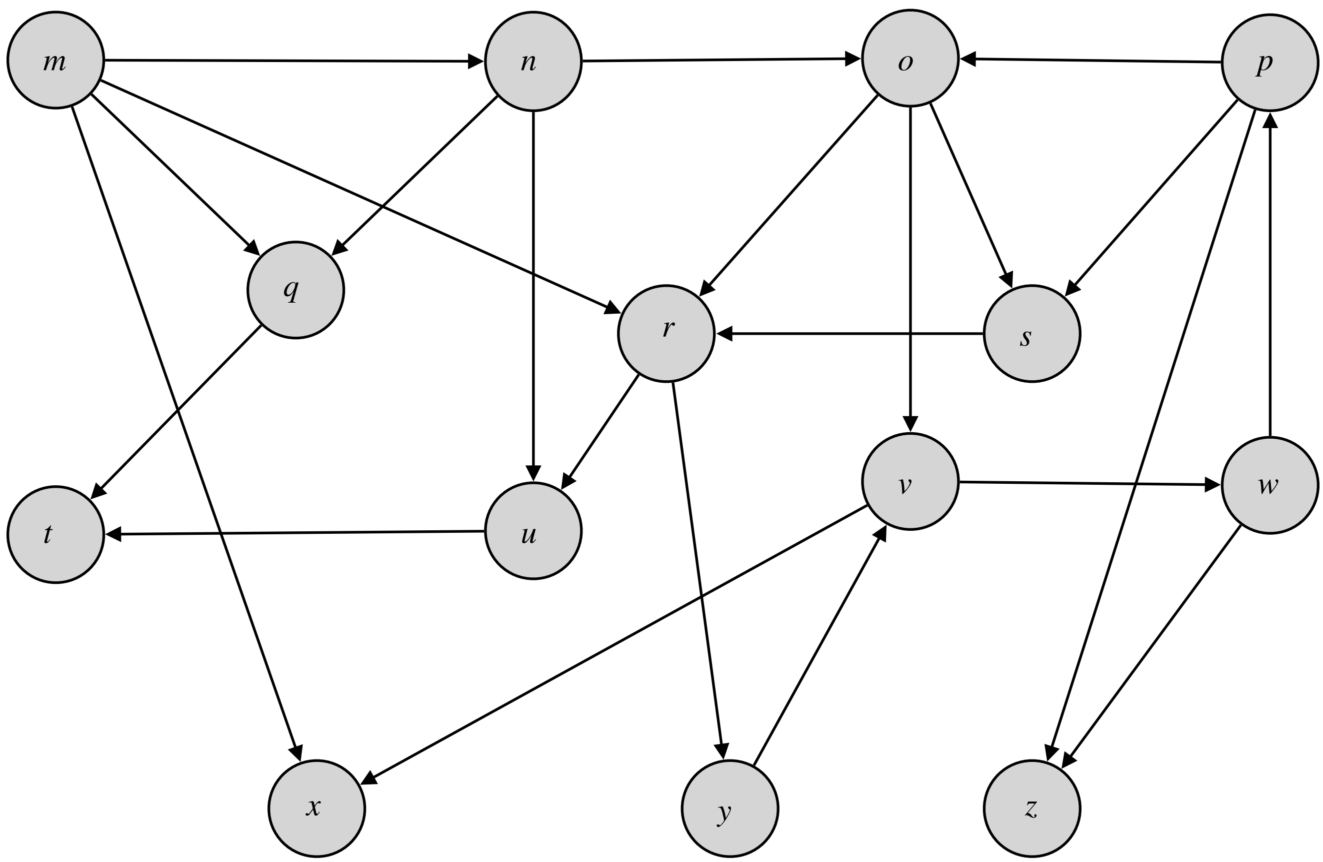
1) (5 points) Show the execution of Radix sort on the following. It is sufficient to show the sorted list at each stage of the algorithm:

QUIZ, JACK, JUMP, JAMB, QURY, JURY, FLAK, NECK, CHIP, ZERO

2) (10 points) Run Dijkstra's algorithm on the following graph starting from vertex A.
Show the relaxation of edges and the open list Q at each step.



3) (15 points) For the following graph:



(a) (4 points) Perform BFS on the following graph starting at vertex m show $v.d$ and $v.\pi$ for each vertex.

(b) (3 points) Draw the Breadth first predecessor tree resulting from running the algorithm in part (a).

(c) (4 points) Perform DFS on the graph and process the vertices alphabetically. Show start and finish times as well as $v.\pi$

(d) (2 points) Draw the DFS predecessor tree resulting from running the algorithm in part (c)

(e) (2 point) Write the parenthesis string resulting from running DFS

5) (5 points) In the algorithm discussed in class to compute the strongly connected components of a directed graph we perform the following:

- Run Depth first search to compute finish times
- Transpose the grab
- Run depth first on the transpose graph processing vertices in decreasing finish time.

Someone suggests the following modification:

- Run Depth first search to compute finish times
- Run depth first on the original graph processing the vertices in increasing order of finish time.

Show a construction of a simple graph to demonstrate that this algorithm doesn't compute the correct strongly connected components. Your construction should have 3 vertices only

Programming Part:

(65 points) Dijkstra's Shortest Path Algorithm

Introduction

In this assignment you will implement Dijkstra's shortest path algorithm. In order to implement this algorithm, you will first have to implement a graph. Your graph should be implemented using an adjacency list representation

Modified Dijkstra's Algorithm

The version of Dijkstra's Algorithm in the book adds all the vertices to the queue Q at the start of the algorithm. However, the optimal time for this algorithm assumes that the queue is a priority queue which can adjust the ordering when a priority changes. The built-in implementations of priority queue in Java and C++ don't implement this (a structure that does implement this is called a Fibonacci heap). We can overcome this difficulty for this assignment by using a less-efficient queue which has some O(n) properties, while keeping the average size of the queue smaller (in the worst case the queue size may still grow to O(V)). This can be accomplished by initializing Q in Dijkstra's algorithm to just the start vertex s. The RELAX function is modified to return true if the destination distance changed, false if unchanged. A vertex is added to Q when RELAX modifies its distance (removing it first if it's already in Q).

```
MODIFIED-DIJKSTRA(G, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    Q = { s }
    while  Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each vertex v ∈ G.Adj[u]
         if RELAX(u, v)
              if ISINQUEUE(Q, v)
                   REMOVEFROMQUEUE(Q, v)
              INSERTINQUEUE(Q, v)
```

The queue can be implemented using the Java PriorityQueue class, which has add (INSERTINQUEUE), poll(EXTRACT-MIN), contains (ISINQUEUE), and remove (REMOVEFROMQUEUE) methods. Some of the methods run in O(1) or O(lg n) time, but others run in O(n) time. The C++ priority_queue implementation does not allow for ISINQUEUE or REMOVEFROMQUEUE equivalent methods. The best strategy is probably to use a vector, which allows equivalents of all the required operations, although mostly in O(n) time. You could also use a multimap with priority as the key, but this would be more complex to implement.

Input

As in the previous assignment, your program should read from standard input a list of data input files, one per line. A data input file contains one line per vertex. The first vertex in the file is always the start vertex for the algorithm. Vertex names are contiguous strings of non-blank characters. Each line contains the source vertex name, followed by pairs of destination vertex names and floating-point edge weights. Each pair indicates a directed edge from the source vertex to the destination vertex. All items are separated by white space (blanks and/or tabs). Represent the weights in your program as type double. Here are two examples. The first example represents Figure 24.6 in the textbook (p. 659):
s t 10 y 5
t x 1 y 2
x z 4
y t 3 x 9 z 2
z x 6 s 7

Output and Deliverables

- Your program should display as output a list of vertices with cost of shortest path for each one
- Your main program should allow user to select a vertex of the graph and should output the shortest path from source to that vertex