Algorithmics 3 Assessed Exercise
Status and Implementation Reports

Ryan Wells
1002253w

November 11, 2012

Status report

Based upon the results found in the empirical results at the end of this document I conclude that both my Dijkstra and Backtrack search algorithms are fully functional and operate in expeted runtimes.

Implementation report

Below I show the implementation of the creation of the graph. Both variations of the algorithm use this creation method.

```
input : An intger value n declaring the number of vertices, Scanner IS of the entire
        adjacency matrix of the graph
output: A directed weighted graph as is described in IS
begin
    numVertices ← n;
    vertices ← Vertex[n];
    for i ← 0 to n do
        vertices[i] ← new Vertex(i);
        for j ← 0 to n do
            vWeight ← next integer from IS;
            if vWeight != 0 then
                Make arc from vertices[i] to vertices[j] with weight vWeight
            end
        end
    end
end
```

Algorithm 1: One constructor of a Graph

(a) I implemented the Dijkstra search algorithm based on the pseudo-code provided in the lecture notes, my exact implementation is below. I decided not to use the boolean Visited inside the Vertex class and opted for a faster method of using an array to store what Vertices have been visited.

input : A file name f contaning data about the Graph and path, assumed to be correct
output: On standard output: Distance from float to sink, path and computation time

begin
    fileScanner ← Scanner for f;
    nodeTotal ← next line of fileScanner;
    dGraph ← `Graph`(nodeTotal,f);
    float ← next line of fileScanner;
    sink ← next line of fileScanner;
    distance ← array containing the distance from root for each node;
    Visited ← array containing whether or not a node has been visited;
    for $i \leftarrow 0$ **to** nodeTotal **do**
        Visited[i] ← 0;
        distance[i] ← distance from float to vertex[i];
        if distance[i]$! = -1$ **then** Predecessor of vertex[i] ← float;
        else Predecessor of vertex[i] ← -1;
    end
    Visited[float] ← 1;
    active ← $false$;
    found ← $false$;
    if vertex[float] *has outward connections in graph* **then** active ← $true$;
    while active **do**
        vertex[i] ← *node with shortest distance to* float *not yet visited*;
        if *no such node exists* **then**
            active ← $false$;
            break;
        end
        Visited[i] ← 1;
        if vertex[i] == vertex[sink] **then**
            active ← $false$;
            found ← $true$;
        end
        for vertex[j] *in* dGraph *not yet visited* **do**
            if vertex[j] *has not been reached yet and can be reached* **then**
                Predecessor of vertex[j] ← vertex[i];
                distance[j] ← vertex[i] + distance from vertex[i] to vertex[j];
            else if vertex[j] *has been reached previously* **then**
                if *distance through* vertex[i] *to* vertex[j] < distance[j] **then**
                    Predecessor of vertex[j] ← vertex[i];
                    distance[j] ← vertex[i] + distance from vertex[i] to vertex[j];
                end
            end
        end
        active = $false$;
        if *further unvisited nodes can be reached* **then**
            active = $true$;
        end
    end
    if found **then**
        Traverse dGraph *from* vertex[sink] *to* vertex[float] *using the Predecessor, adding each* vertex *to a stack to reverse order*;
    end
end

Algorithm 2: Dijkstra Shortest Path

(b) I have implemented the Backtrack search using the pseudo-code given at the end of the assignment handout, however I do some preliminary checks before running the algorithm properly. To get the best path for backtrack, because it is such an exhaustive algorithm, you first call a function called bestPath(float, sink) which will only run the backtrack search if the float and sink nodes are different from the ones relating to the current path or if the current path is null. If this is not qualified then nothing will happen. A get method is needed to access the path, which is stored inside the Graph class. The bestPath is stored inside the Graph class.

I attempted to store the path as a doubly indexed integer array with the x value corresponding to the Vertex ID and the second as the accumulative distance so far so I could add in O(1) and remove in O(1) time. I encountered many problems which may have been solved but I decided that under the time constraints using a LinkedList of AdjListNodes was a more suitable option. Also, I decided to use each Vertex's Visited boolean since the function is recursive. The Backtrack algorithm here is assuming that this is the first time a path has been determined on this Graph, and on the first call of this method the workingPath only contains the Float vertex.

```
input : LinkedList<AdjListNode> workingPath, integer sink
begin
    current ← node referred to by the head of workingPath;
    for vertex in Adjacency List of current do
        if not yet visited then
            Visit and add to head of workingPath ;
            if bestPath == null or workingPath is a shorter path than the bestPath then
                if head of workingPath is sink then
                    bestPath ← deep copy of workingPath ;
                else
                    Recursively call this algorithm on workingPath ;
                end
            end
            Remove and un-visit the head of workingPath
        end
    end
end
To print path, print each Vertex starting from the end of the workingPath;
```

Algorithm 3: Backtrack Search Algorithm

Empirical results

Below are the results from texting the algorithms on my personal 3.4GHz computer running Debain Squeeze.

Test Files	Dijkstra	Backtrack
data6.txt	Shortest distance from vertex 2 to vertex 5 is 11 Shortest path: 2 1 0 5 Elapsed time: 15 milliseconds	Shortest distance from vertex 2 to vertex 5 is 11 Shortest path: 2 1 0 5 Elapsed time: 15 milliseconds
data20.txt	Shortest distance from vertex 3 to vertex 4 is 1199 Shortest path: 3 0 4 Elapsed time: 25 milliseconds	Shortest distance from vertex 3 to vertex 4 is 1199 Shortest path: 3 0 4 Elapsed time: 29 milliseconds
data40.txt	Shortest distance from vertex 3 to vertex 4 is 1157 Shortest path: 3 36 4 Elapsed time: 54 milliseconds	Shortest distance from vertex 3 to vertex 4 is 1157 Shortest path: 3 36 4 Elapsed time: 129 milliseconds
data60.txt	Shortest distance from vertex 3 to vertex 4 is 1152 Shortest path: 3 49 4 Elapsed time: 80 milliseconds	Shortest distance from vertex 3 to vertex 4 is 1152 Shortest path: 3 49 4 Elapsed time: 470 milliseconds
data80.txt	Shortest distance from vertex 4 to vertex 3 is 1152 Shortest path: 4 49 3 Elapsed time: 100 milliseconds	Shortest distance from vertex 4 to vertex 3 is 1152 Shortest path: 4 49 3 Elapsed time: 11472 milliseconds
data1000.txt	Shortest distance from vertex 24 to vertex 152 is 17 Shortest path: 24 582 964 837 152 Elapsed time: 901 milliseconds	Shortest distance from vertex 24 to vertex 152 is 17 Shortest path: 24 582 964 837 152 Elapsed time: 35789 milliseconds