Analysis of Algorithms – HW2

Li-Hen Chen

Department of Computer Science & Engineering, Texas A&M University

1. Design algorithms for Min(H), Insert(H,a), and Delete(H,i), where the set H is stored in a heap, a is the element to be inserted into the heap H, and i is the index of the element in the heap H to be deleted. Analyze the complexity of your algorithms.

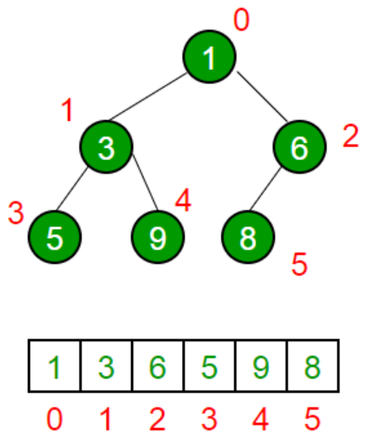
Psuedo-code for Min-heap.

```
class MinHeap
    int[] harr                  // array
    int capacity
    int heap_size
public:
    int parent(i);              // get parent
        return (i-1)/2;
    int left(i);              //left child
        return (2*i+1);
    int right(i);               //right child
        return (2*i+2)
    int Min(H);             //get min value
        return harr[0]
    int popMin(H)           //pop min
        int root = harr[0];
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
        return root;
    void Delete(H, i);
        heapify(i,int_min);
        popMin();
    void Insert(H, a);
        heap_size++;
```

```
          int i = heap_size - 1;
          harr[i] = a;                 // keep the heap not being violated
          heapify(i)
     void heapify(i,new_value);      // or decrease_key
          harr[i] = new_value
          while (i != 0 && harr[parent(i)] > harr[i])
          {
               swap(harr[i], harr[parent(i)]);
               i = parent(i);
          }
     void MinHeapify(int i)
     {
          int l = left(i);
          int r = right(i);
          int smallest = i;
          if (l < heap_size && harr[l] < harr[i])
               smallest = l;
          if (r < heap_size && harr[r] < harr[smallest])
               smallest = r;
          if (smallest != i)
          {
               swap(harr[i], harr[smallest]);
               MinHeapify(smallest);
          }
```

Time Complexity Analysis:

The heap is constructed by an array and the access is O(1) Search is O(n) Insertion is O(n). The heap's structure is like a tree as following.



- Min: It returns the root element of the Minheap. Its time complexity of this operation is O(1).
- popMin: It removes the minimum of the heap which is array[0]. When the min value is removed, the heap should maintain its heap property. The worst case it would traverse from root to leaves so the time complexity is O(log n).
- Delete: We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- Insert: We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property. Since we way traverse from leaves to root, the time complexity is O(log n).

2. Writethepsuedo-codefortheDijstra’salgorithmthatsolvestheSingle-SourceShortest Path problem. Analyze the complexity of the algorithm (you can assume that the algorithm uses a heap for fringes and you can use your results in Question 1 directly). Give a formal proof that the algorithm works correctly when the edge weights are all non-negative.using the Q1 result and min-Heap.

Psuedo-code:

```
Dijkstra(Graph, start, end)
    dist[start] = 0 // Initialization
    create vertex set Q // min-Heap
        for each vertex v in Graph:
            if v ≠ start
                dist[v] = INFINITY        // Unknown distance from start to v
            prev[v] = UNDEFINED           // Predecessor of v
            Q.Inset ((v,dist[v]))
        while Q is not empty:
            u = Q.popMin ()               // Remove and return best vertex
            for each neighbor v of u:              // only v that are still in Q
                alt = dist[u] + length(u, v)
                if alt < dist[v]
                    dist[v] = alt
                    prev[v] = u
                    heapify(v,alt)
    return dist prev
```

Time Complexity Analysis:

Give V vertices and E edges. For sparse graph, Dijkstra's algorithm can be implemented more efficiently by using min-Heap. The time complexity of the above algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times. The inner loop has heapify (decrease_key) operation which takes O(log V) time. So overall time complexity is O(E+V)*O(log V) which is O((E+V)* log V) = O(E log V) because V is O(E) ,assuming a connected graph.

Proof of Correctness:

Invariant hypothesis: For each visited node v, dist[v] is considered the shortest distance from source to v; and for each unvisited node u, dist[u] is assumed the shortest distance when traveling via visited nodes only, from source to u. This assumption is only considered if a path exists, otherwise the distance is set to infinity. (Note : we do not assume dist[u] is the actual shortest distance for unvisited nodes)

The base case is when there is just one visited node, namely the initial node source, in which case the hypothesis is trivial.

Otherwise, assume the hypothesis for n-1 visited nodes. In which case, we choose an edge vu where u has the least dist[u] of any unvisited nodes and the edge vu is such that dist[u] = dist[v] + length[v,u]. dist[u] is considered to be the shortest distance from source to u because if there were a shorter path, and if w was the first unvisited node on that path then by the original hypothesis dist[w] > dist[u] which creates a contradiction. Similarly if there was a shorter path to u without using unvisited nodes, and if the last but one node on that path were w, then we would have had dist[u] = dist[w] + length[w,u], also a contradiction.

3. 3. Developalinear-time(i.e.,O(m)-time)algorithmthatsolvestheSingle-SourceShortest Path problem for graphs whose edge weights are positive integers bounded by 10. (Hint. You can either modify Dijstra's algorithm or consider using Breath-First-Search.)

If we have a Single-Source Shortest Path problem with edge weights are positive integers bounded by 10. We can modify Djikstra algorithm using different data structure, buckets. A bucket structure is a data structure that uses key values as the indices of the buckets, and store items of the same key value in corresponding bucket. Time complexity of buckets is as following:

- Insert: O(1)
- Removal: O(1)
- Decrease Key: O(1)
- Find Minimum: O(c) where c is the maximum key value.

Algorithm Procedure

1. Maintain some buckets with number 1 to 10.
2. Bucket k contains all temporarily labels nodes with distance equal to k.
3. Node in each buckets are represented by list of vertices.
4. One by one, these nodes with minimum labels are permanently labeled and deleted from bucket during the scanning process.
5. Thus operations involving vertex include:
    - Checking if a bucket is empty
    - Adding a vertex to a bucket
    - Deleting a vertex from a bucket
6. The position of a temporarily labeled vertex in the buckets is updated accordingly when distance label of a vertex changes.
8. Process repeated until all vertices are permanently labeled.

Psuedo-code

```
Distance Table (vertices) [Inf, Inf, ..... ,Inf]
BucketList [10*vertex+1]      // keep vertex of distance label i
while BuckList is not empty:
    vertex u = pop vertex with minimum distance
    for every adjacent vertex v:
        if D[v] > D[u] + edge(u,v):
            D[v] = D[u] + edge(u,v)
            update distance D[v], BucketList[D[v]]
```

Time Complexity:

Assume we have V vertices and E edges. According to the above procedures in worst case we have to deal with every vertices which is V and every edges E. Besides, at the worst case we may update a vertex 10 times at this case because 10 is the bounded weights. As a result the algorithm takes O(E+10V) which is linear-time.

4. Consider an extended version of the Shortest-Path problem. Suppose that you want to traverse from city s to city t. In addition, for some reason, you also need to pass through cities x, y, and z (in any order) during your trip. Your objective is to minimize the cost of the trip. The problem can be formulated as a graph problem as follows: Given a positively weighted directed graph G and five vertices s, t, x, y, z, find a path from s to t that contains the vertices x, y, z such that the path is the shortest over all paths from s to t that contain x, y, z, assuming that these paths are allowed to contain repeated vertices and edges. Develop an O(m log n)-time algorithm for this problem.

We are required to find a shortest path from s to t passing cities x, y, z (in any order). We have discussed Dijkstra algorithm extensively above. Then we could utilize the above results to solve the problem. As we know, the time complexity of Dijkstra algorithm is O((E+V)* log V) = O(E log V). If we would have to find a shortest path from s to t passing cities x, y, z we simply perform Dijkstra algorithm to each points.

Pseudo-code:

```
For permu in Permutation([x,y,z]):
    for i in range(0,len(permu)-1):
        Dijkstra(G,permu[i],permu[i+1])
    Dijkstra(s,permu[0])
    Dijkstra(permu[-1],t)

return min(result)
```

Time Complexity:

We perform Dijkstra to every permutation[x,y,z]. Since there are just 3 nodes to pass, so the permutation of [x,y,z] = 3! = 6. We have to perform Dijkstra to every 2 nodes. As a result, we totally have to perform $6 \times 4 = 24$ Dijkstra in this particular question. Time complexity of this alogrithm is $24 \times O(E \log V) = O(E \log V)$.
That is to say the time complexity of the algorithm is O(m log n) due to it's less vertices to pass by. However, if we have n vertices to pass by, the time complexity would become (n!)(n+1)O(m log n), which is extremely high.