CS403: Algorithm Design and Analysis

Feb-Jun 2017

Indian Institute of Technology Mandi

Assignment 4

Priyansh Saxena

B14118

Computer Science and Engineering

CS 403 – Algorithm Design and Analysis

Assignment 4

Problem 1: Implement Edmond-Karp's Algorithm. The algorithm is given below, in two functions:

```
augment (f , P) {
    Let b = bottleneck (P , f )
    For each edge (u, v) ∈ P
        If e = (u, v) is a forward edge then
            increase f (e) in G by b
        Else ( (u, v) is a backward edge, and let e = (v, u) )
            decrease f (e) in G by b
        Endif
    Endfor
    Return( f )
}
Max-Flow {
    Initially f (e) = 0 for all e in G
    While there is an s - t path in the residual graph $G_f$
        Let P be a simple s - t path in G f
        f’ = augment (f , P)
        Update f to be f’
        Update the residual graph G f to be $G_{f}$
    Endwhile
    Return f
}
```

The proof that this algorithm terminates, and that with the maximum value of flow can be shown by the following arguments.

Assumption: All capacities are integers.

1. The algorithm terminates, because the value of the flow increases by an integers in each iteration, and there is a finite limit on the capacities of the edges.
2. Upon termination, there is no augmenting path from source to sink. This means that at least one edge in each path from source to sink has achieved saturation and it is not possible to add more flow without overgrowing any capacity.

Let C denote the maximum possible flow out of the source. Then C cannot be greater than the sum of the capacities of all the outgoing edges from source, and therefore C is a finite integer. Also, since the first edge in the simple path must come out of the source, and no edge in this path can enter the source (since the path is simple), it is a forward edge. We increase the flow on this edge by the bottleneck, and therefore the flow value is strictly increasing.

Now, the flow increases by at least 1 in every iteration. Every iteration takes O(m) time, since BFS requires O(n+m) time, and in a connected graph, m is at least O(n). So, the overall time complexity of the algorithm is O(mC).

The space-complexity of the solution is O(n^2 + m). This is to ensure that BFS runs in O(n+m) time and residual graph is updated in O(m) time in each iteration.

Problem 2: Implement 0/1 Knapsack Problem, in O(nW) time.

The problem aims to find the maximum value can be accounted for a set of objects, where every object also carries a penalty with it and there is a finite limit to the total penalty that can be tolerated.

The problem can be solved by using dynamic programming, by using the following reasoning: at a point of time when i^{th} object is in consideration, the maximum value that can be carried is the maximum of the values of

1. maximum value obtained upto $(i-1)^{th}$ object.
2. the sum of values of the i^{th} object and the maximum value that can be obtained by dropping the minimum possible weight that allows us to carry the i^{th} object.

Now, the optimal values can be calculated by first including the lightest items first and then adding more weight to increase the value. The <weight,value> pairs are sorted in ascending order of weight. We can remember the values for lighter weights by solving for them and memoising them in an array, for all values of weight from 0 to maxWeight. The algorithm is outlined below:

```
for j from 0 to W do:
    m[0, j] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

The space-complexity of the algorithm is O(nW), to account for the 2D array built to memoise the maximum values obtained at each step. Here W is the maximum weight or penalty that can be tolerated/carried.

The time-complexity of the algorithm is O(nW). This is because a single traversal of a nW-sized array is sufficient to determine the optimal value.

Problem 3: Implement Dijkstra's Algorithm for single-source shortest paths in a graph.

```
Dijkstra's Algorithm (G, l) {
    Let S be the set of explored nodes
        For each $u \in S$ , we store a distance d(u)
    Initially S = {s} and d(s) = 0
    While $S \neq V$
        Select a node v out of  S with at least one edge from S for which
          $d'(v) = \min_{e=(u, v):u \in S} d(u) + l_e$ is as small as possible
        Add v to S and define d(v) = d'(v)
    EndWhile
}
```

The time-complexity of the algorithm is O(m.logn).

1. There are m edges in the graph, and each edge needs to be traversed to determine the shortest path.

2. All operations in a priority-queue based implementation are done in O(log.n) time.

The space-complexity of the algorithm is O(n). This is required to keep track of the distances of all nodes. Also, there can be at most O(n) elements in the priority queue at any time.