Data Structures and Algorithms Study Guide

Josepeh Sepich Nov 6

1 Shortest Path

1.1 Dijkstra's Algorithm

Use a priority queues and update path distances starting from minimum unfinalized vertex.

1.1.1 Algorithm

```
Dijkstra(G, l, s) {
  Input: Graph g, weights l, source vertex s
  Output: distance to each vertex from s

  dist = new int[|V|]
  for each v in V {
    dist[v] = inf
  }
  dist[s] = 0
  R = {s}
  # create queue
  # get min value of q aka s
  for (i = 1 to |V|) {
    pop min distance vertex not in R # use getMin from PQ
    foreach (v, z) in E {
      if (dist[z] >= dist[v] + l(v, z)) {
        dist[z] = dist[v] + l(v, z)
        # update keys in queue (insert if inf)
      }
    }
  }

  return dist
}
```

1.1.2 Running Time

$O(|V|^2)$ when sorting list, but we are going to use a priority queue implementation. The following table is how running time is affected by different queue implementations where there are |V| inserts, |V| getMins, and |E| decreaseKey.

Implementation of PQ	GetMin	Insert/ DecreaseKey	Dijkstra's
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary Heap	$O(\log\lvert V\rvert)$	$O(\log\lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert)\log\lvert V\rvert)$.
d-ary Heap	$O\left(\frac{d\log\lvert V\rvert}{\log d}\right)$	$O\left(\frac{\log\lvert V\rvert}{\log d}\right)$	$O\left((\lvert V\rvert\cdot d+\lvert E\rvert)\frac{\log\lvert V\rvert}{\log d}\right)$.
Fibonacci Heap	$O(\log\lvert V\rvert)$	$O(1)$ (amortized time)	$O(\lvert V\rvert\log V+\lvert E\rvert)$.

1.1.3 Priority Queue Implementations

Priority Queue has three functions: insert, getMin, and decreaseKey. Binary heap is a heap with two leaves, whereas d-ary the d will affect the runnning time. Amortized time from a finbonacci heap references how the running time changes after each run.

1.2 Bellman-Ford Algorithm

Bellman-Ford helps to solve shortest path when there are negative edges involved. Incrementing edges does not work, because if one path to a node has 1 edge and another has 3 edges, then the path will be incremented by a total of 1 and 3 respectively, so you cannot merely perform a linear transformation to find shortest paths. The Bellman-Ford algorithm works by iteratively solving distance in a BFS fashion, where each edge is updated every time.

```
Bellman-Ford(G, l, s) {
  Input: same as Dijkstra
  Output: same as Dijkstra
  for all w in V {
    dist[w] = inf
  }
  dist[s] = 0
  for (i to |V|-1) {
    for all e in E {
      update(e)
    }
  }
}

update(e = (v, w)) {
 if (dist[w] >= dist[v] + l(v,w)) {
  dist[w] = dist[v] + l(v,w)
 }
}
```

In order to check for a negative cycle run update on every edge an additional time. If any distances are updated then a negative cycle exists.

1.2.1 Running Time

The running time of this algorithm is $O(|V||E|)$, because you update each edge for |V| - 1 times.

1.2.2 Shorest Path in DAG

Run Bellman-Ford by starting at source verticies. This requires to sort by topological order. This ensures you do not have to run more than |V|-1 times.

```
DAG-Shorest-Path(G, l, s) {
  for every u in V {
    dist[u] = inf
  }
  dist[s] = 0
  topologically sort G # run DFS and sort by decreasing post number
  for each w in V  (in topo order) {
    for each (w, v) in E {
      update(w, v)
    }
  }
}
```

Running time of $O(|V| + |E|)$

2 Maximum Cardinality Matching

3 Max Flow Problem

3.1 Ford-Fulkerson Algorithm

3.1.1 Running Time

3.1.2 Proof of Correctness

4 Greedy Choice Algorithms

4.1 Minimal Spanning Tree

4.1.1 Data Structures

4.2 Kruskal Algorithm

4.2.1 Running Time

4.3 Prim's Algorithm

5 Huffman Encoding

5.1 Huffman's Algorithm

5.2 Encoding Cost

6 Matroids

6.1 Definition

6.2 Properties

6.3 Task Scheduling