Week 7, Lecture 1

Algorithm Analysis and Design

Pratyay Suvarnapathaki, 2020111016

The Shortest Reliable Path Problem

Given a weighted graph G, a `src` node and a `dest` node, the shortest path from `src` to `dest` is to be found, that uses **at most k edges.**
Note: Dijkstra's Algorithm is not useful here as it doesn't keep track of the *number of edges traversed.*

The Dynamic Programming Solution

Since the problem can be broken down into several overlapping subproblems, we define:
Subproblem $dist(v, i)$ (where $i \leq k$) for each vertex v, to be the shortest length for a path from `src` to v.
Base cases: $dist(v, 0) = \infty$, $dist(src, 0) = 0$
The relation thus obtained is:
$dist(v, i) = min_{(u,v)\epsilon E}\{dist(u, i-1) + l(u, v)\}$

The Shortest Path All Vertex Pairs Problem

Given a weighted graph G, the shortest path between every pair of discrete vertices is to be found.

The Dynamic Programming Solution

Defining the Subproblems

Here, we observe that the **optimum substructure property** is followed.
Firstly, we try to find paths between node pairs that do not include *any* intermediate nodes whatsoever. Now, the shortest paths between all vertex pairs is

just the **edge weight**, if there exists an edge between a pair of vertices.
Then, we *gradually* expand our scope and consider intermediate nodes too, and keep expanding this scope till all vertices of the graph are covered.
Hence, we define the subproblem:
$dist(i, j, k)$ = the shortest path from node i to node j with the set of intermediate nodes being limited to $\{1, \cdots, k\}$
Base Cases : $dist(i, j, 0)$ length of edge ij if it exists, else ∞.

Algorithm

Using a two dimensional memoization table, we obtain the following algorithm:

```
# let us conveniently name the memoization table 'dist'. 'Graph' is an
adjacency table (default value = infinity)
for i in range(0,V):
for j in range(0,V):
C[i][j] = Graph.edgeweight(i,j)

for k in range(0,V):
for i in range(0,V):
for j in range(0,V):
if(dist[i][j] > dist[i][k] + dist[k][j]):
dist[i][j] = dist[i][k] + dist[k][j]
return dist
```

The complexity of the above algorithm is naturally $O(n^3)$ owing to the three nested loops.

The Independent Sets In Trees Problem

Given a graph $G = (V, E)$, the largest subset of vertices S is to be found such that there exist no edges between them.

The Dynamic Programming Solution

The layered structure of the tree itself provides the framework to define subproblems, as the **optimum substructure property** is followed since for a tree to represent the optimal solution, *its subtrees need to represent their respective optimal sub-solution.*
Hence, we define the subproblem $I(v)$ = largest independent set in the subtree with root v.
So, for a subtree with root v, the largest independent subset:

- Case 1: Includes v
  Relation: $I_1 = 1 + \sum\{grandchildren\ w\ of\ v\}I(w)$
- Case 2: Does not include v
  Relation: $I_2 = \sum\{children\ w\ of\ v\}I(w)$

and finally, $I(v) = max(I_1, I_2)$
