Overview of Materials After Midterm

Dynamic Programming (DP)

- Often DP is used for optimization or counting problems
- DP is just a way to solve recurrences by memorization
- Main idea of DP
  - Subproblem definition
  - Analyze the structure of an optimal solution and write out the recurrence
  - Compute the value of an optimal solution (usually bottom-up)
- Usual strategy
  - Try 1D first; if not working, try 2D
  - Some problems (two strings, intervals) naturally require 2D DP
  - Start with the most natural definition; if not working, ask why and change the definition
  - After changing the definition, need to make sure it still solves the original problem.

Solution Format

- Definition of subproblem
  - If the last subproblem does not solve the original problem, say how all subproblems together can solve the original problem
- Recurrence relation
- Bottom-up computation
  - Can just say how to do it, or use pseudo code
  - Remember the base case
  - Running time usually easy
- Construction of the optimal solution
  - Not always required
  - Use a “pred” array to remember the best choice made by DP for each subproblem
  - Construct the solution by tracing back the pred array

Shortest symmetric supersequence

Recall that one of the homework questions is to find the longest symmetric subsequence of a given string. Here your job is to design an algorithm to find the shortest symmetric supersequence. For example, for the string ACBAC, the shortest symmetric supersequence is CACBCAC or ACBABCA, both of which have length 7. Your algorithm only needs to output the optimal length, not the actual symmetric supersequence.

Solution

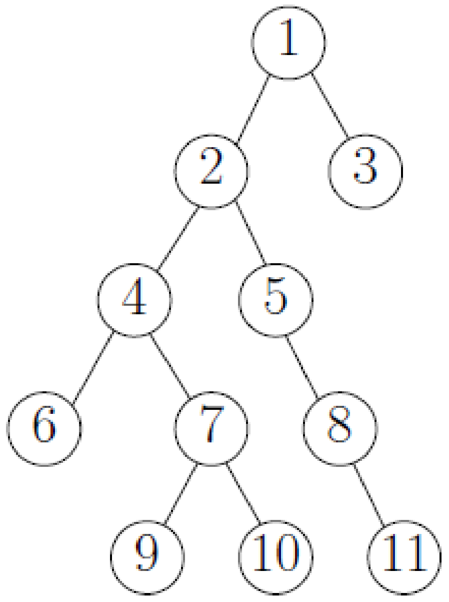
- Definition of subproblem:
  For a given string *s*, denote by *S*[*i, j*] the length of shortest symmetric supersequence of *s*[*i..j*].
- Recurrence:
  base case *S*[*i, i*] = 1 and *S*[*i, i* − 1] = 0

$$S[i,j] = \begin{cases} S[i+1, j-1] + 2, & \text{if } i < j \;\&\; s[i] = s[j]; \\ min\{S[i, j-1], S[i+1, j]\} + 2, & \text{if } i < j \;\&\; s[i] \neq s[j]. \end{cases}$$

- Bottom-up computation:
  We compute all the *S*[*i, j*]'s from small intervals to larger intervals. The running time is $O(n^2)$.

Longest Path in a Tree

Design an algorithm that, given a binary tree, finds the longest path in the tree. For example, in the tree below, the longest path is 9-7-4-2-5-8-11. Your algorithm just needs to output the two endpoints of the path, i.e., 9 and 11 in this example. For full credits, your algorithm should run in O(n) time, where n is the number of nodes in the tree.



Solution

- Subproblem definition
  Let d(u) be the length of the longest path from u to one of the leaves below u. This does not directly solve the original problem, but we can find the length of the longest path with u as the highest node as follows:

$$path\_length(u) = \begin{cases} d(u.left) + d(u.right) + 2 & \text{if u.left != null and u.right != null,} \\ d(u.left) + 1 & \text{if u.left != null and u.right = null,} \\ d(u.right) + 1 & \text{if u.left = null and u.right != null,} \\ 0 & \text{if u is a leaf.} \end{cases}$$

Then we just find the u with the maximum path length(u).

Solution (continued)

(2) Recurrence for $d(u)$:

$$d(u) = \begin{cases} \max\{d(u.left), d(u.right)\} + 1 & \text{if u.left != null and u.right != null,} \\ d(u.left) + 1 & \text{if u.left != null and u.right = null,} \\ d(u.right) + 1 & \text{if u.left = null and u.right != null,} \\ 0 & \text{if u is a leaf.} \end{cases}$$

- Bottom-up computation:
  We can compute all the d(u)’s from leaves to the root. The order can be obtained by a post-order traversal of the binary tree. The running time is O(n).
- For each u, let c(u) be the child of u that the DP has chosen during the bottom-up computation of d(u). After we have found the u with the maximum path length(u), we trace down the two paths starting from u.left and u.right (if they exist) following the c() array.

Graph Algorithms

- Basic graph theory
  - Not required: Euler path
- Graph representation
  - Adjacency list (default) vs adjacency matrix
- Algorithms learned:
  - BFS / DFS
  - Topological sorting for a DAG
  - MST (Prim and Kruskal)
  - Shortest paths (Dijkstra and various DPs)
    - Not required: The improved versions of Bellman-Ford and Floyd-Warshall
  - Maximum flow (Ford-Fulkerson)
  - Need to know algorithms and correctness proof

Types of Possible Questions

- Show steps of a learned algorithm on an example
- Variants of learned algorithms
- Proving certain graph properties, MST, shortest path
- Design a new algorithm

Minimum spanning tree and the cut lemma

Let G be a connected undirected graph with distinct weights on the edges, and let T be the MST of G. Recall that the cut lemma states: For any cut (S; V-S) of the graph, the minimum-weight edge e crossing the cut must belong to T. Prove or disprove (i.e., give a counter example) the following “reverse cut lemmas”.

(a) Every edge e of T must be the minimum-weight edge crossing any cut of G.

(b) Every edge e of T must be the minimum-weight edge crossing some cut of G.

Proof of (b)

For any edge e in T. Removing e breaks T into two parts S and $V - S$. We claim that e is the minimum-weight edge crossing the cut $(S, V - S)$. Indeed, if there is another e' crossing the cut, we can replace e with e' to improve the MST, which contradicts with the fact that T is an MST.

Shortest Path on an Chessboard

In the game of chess, the Knight can move two squares horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter “L”. You are given two squares on an $n \times n$ chessboard. Design an algorithm to determine the shortest sequence of knight moves from one square to the other. For full credits, your algorithm should run in $O(n^2)$ time.

Solution

We build an undirected graph G = {V,E}, where V consists of n^2 nodes corresponding to the n^2 squares in the chessboard. For two nodes $v_{i,j}$ and $v_{p,q}$, they are connected by an edge if and only if the Knight can directly move from [i, j] to [p, q]. Then we just need to find the shortest path from the source square to the destination square.

Since the graph is unweighted, we can use BFS which takes time O(V+E). In this graph, we have n^2 vertices. Each vertex has 8 neighbors so there are at most $8n^2 = O(n^2)$ edges.

So the running time is $O(n^2)$.

Survey Design

Survey design.

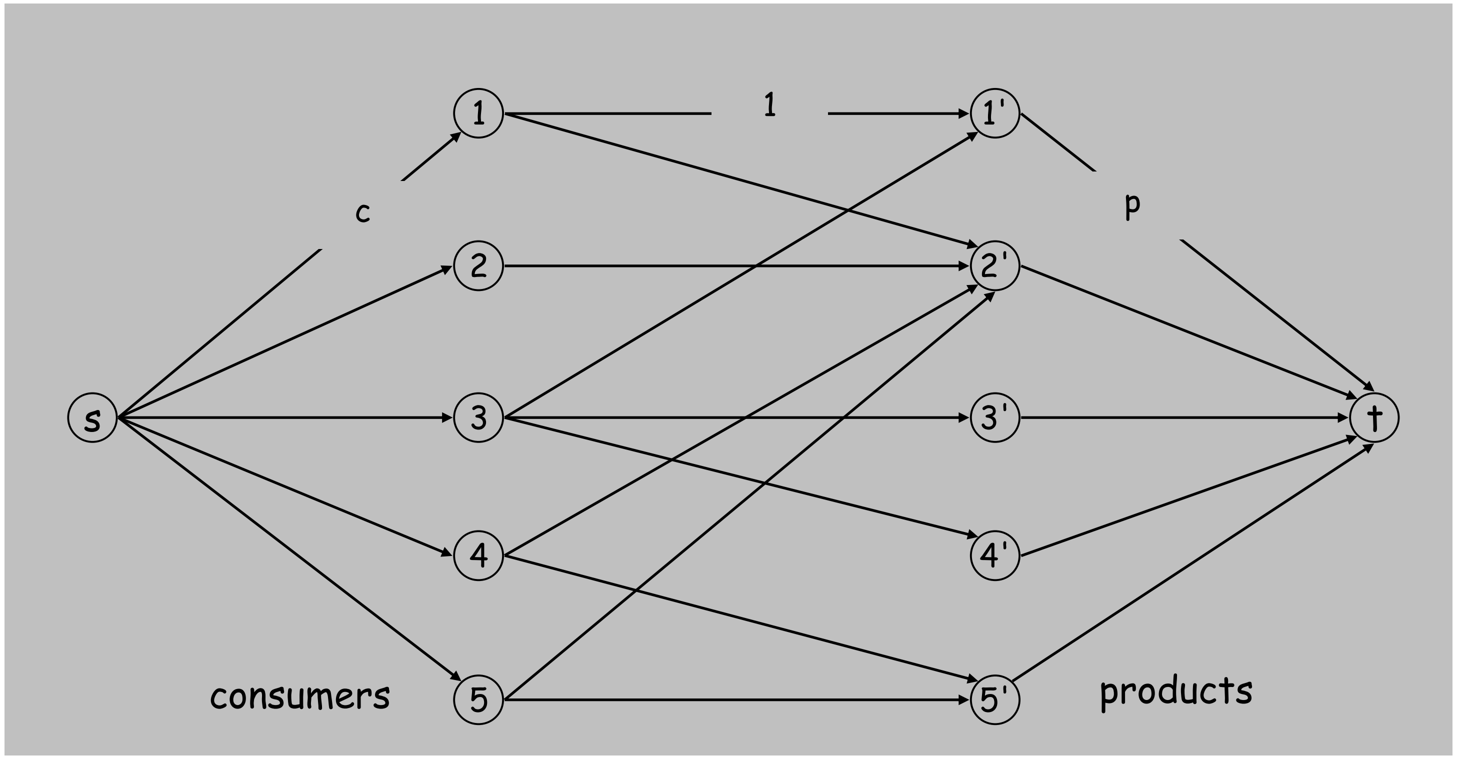
- Design survey asking n consumers about m products.
- Can only survey consumer i about a product j if they own it, and you are given this information.
- Ask a consumer at most c questions.
- Ask at most p questions for any product.

Goal. Design a survey that meets these requirement, while maximizing number of questions asked in total.

Survey Design

Algorithm. Formulate as a maximum flow problem.

- Include an edge (i, j) if customer own product i.
- Add a source s, and connect to each consumer with capacity c.
- Add a target t, and connect from each product with capacity p.



About the Final Exam

- The final is on: 16-Dec-2015 (Wed) 12:30pm-3:30pm, LG1 (table tennis room)
- It will be with closed books and notes, and will cover all materials for the entire semester, but with a focus on those after the midterm.
- All necessary mathematical background will be provided.
- You can use pencils (so that you can easily correct possible mistakes), but calculators or other electronic means are not allowed (or needed).
- Office hours will be announced later.