Backtracking

Week 13

Required Activities

- Check Announcements regularly (every 2-3 days)
- Read FA book: Chapter 5
- Complete and submit Assignment 4 (due Friday Jan 25)

Backtracking Technique

- In Dynamic Programming subsets of a solution are generated while in backtracking, the algorithm decides which subsets need not be generated to make it more efficient
- Efficient for many large instances of a problem (but not all)
- Solves problems in which a sequence of objects is chosen from a set and the sequence satisfies some criterion
- The technique uses modified DFS of a rooted tree (pre-order traversal – root followed by children)
- General-purpose algorithm does not specify order children visited – textbook uses left to right
- Can be used to solve NP-Complete problems such as 0-1 Knapsack more efficiently

Backtracking Process

- After determining a node cannot lead to a solution, backtrack to the node's parent and proceed with the search on the next child
  - Non-promising node: when the node is visited, it is determined the node cannot lead to a solution
  - Promising node: may lead to a solution
- State Space Tree: contains all candidate solutions where path from root to leaf is a candidate solution
- Backtracking
  - DFS of state space tree determining promising and non-promising nodes
  - Pruning state space tree: if a node is determined to be non-promising, back track to its parent
- Pruned State Space Tree: sub-tree consisting of visited nodes

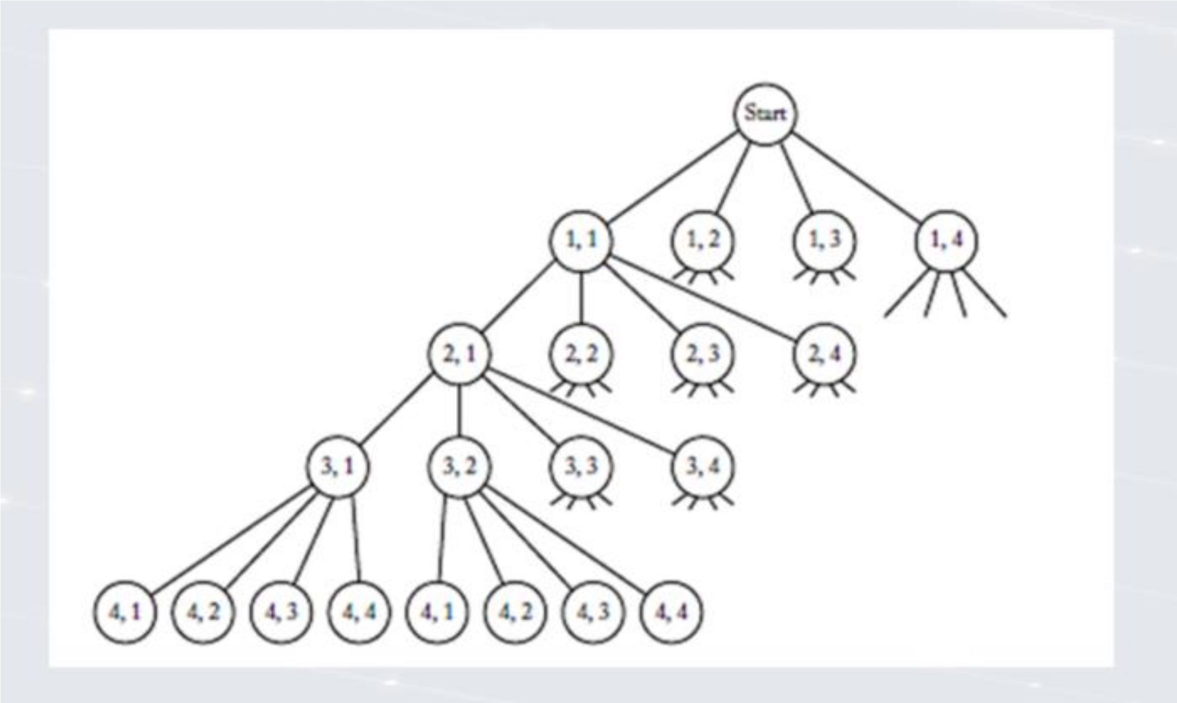
Examples Where Backtracking Used

- N-Queens puzzle
- Knapsack problem
- Subset Sum
- Coloring Problem
- Hamiltonian Cycle
- Permutations of a given string
- Used in logic programming languages such as Prolog

N-Queen Puzzle

- **Goal:** position n Queens on a n x n board such that no two queens threaten each other
  - No two queens may be in the same row, column, or diagonal
- **Sequence:** n positions where queens are placed
- **Set:** n^2 positions on the board
- **Criterion:** no two queens threaten each other

Fig 5.2



4-Queen Problem Solution

	Q		
			Q
Q			
		Q	

4-Queen Problem

- Assign each queen a different row
- Check which column combinations yield solutions
- Each queen can be in any one of four columns: 4 x 4 x 4 x 4 = 256
- **Construct State-Space Tree** – Candidate Solutions
  - Root – start node
  - Column choices for first queen stored at level-1 nodes
  - Column choices for second queen stored at level-2 nodes
  - Etc.
  - Path from root to a leaf is candidate solution
  - Check each candidate solution in sequence starting with left-most path
- Do not check branches that know are non-promising (e.g. no queen in same column so Fig 5.2 don't check <2,1>)
- **Pruning**
  - DFS of state space tree
  - Check to see if each node is promising
  - If a node is non-promising, backtrack to node's parent
  - Pruned state space tree – subtree consisting of visited nodes
  - Promising function n-queen problem: returns false if a node and any of the node's ancestors place queens in the same column or diagonal
- **Promising function:**
  - 2 queens same row? col(i) == col(k)
  - 2 queens same diagonal? col(i) – col(k) == i – k || col(i) – col(k) == k - i

Backtracking Savings

- Algorithm1 – DFS of state space tree without backtracking
- Algorithm2 – checks no two queens same row or same column

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

*Entries indicate numbers of checks required to find all solutions.

†Algorithm 1 does a depth-first search of the state space tree without backtracking.

‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.

0-1 Knapsack Problem

- Set *maxprofit* to 0
- Visit root
  - set *profit*=0 and *weight*=0
  - Calculate totweight, bond, and determine if promising (*weight* < W and *bound* > *maxprofit)*
- Visit left child
  - Compute profit (*profit* + item’s profit) and weight (*weight* + item’s weight)
  - Determine if promising;
    - If promising *maxprofit, profit, weight* are updated
    - if not then *maxprofit* does not change and backtrack to previous node and take right child
- Keep traversing with node being promising if increases maxprofit while staying within W for weight restriction until done

Calculating Bound

- Lets say we have ordered items based on profit/weight:

  Item 1: $40 2 n= 4 W=16
  Item 2: $30 5
  Item 3: $50 10
  Item 4: $10 5
- Initialize *weight* = 0 and *profit* = 0
- We add weights until next item goes over W:

  2 + 5 = 7 ok
  2 + 5 + 10 = 17 but W=16 so over so stop here and we have k=3 (first 3 items)
- Calculate *totweight*: *weight* + summation of items' weight minus last one (k-1)

  totweight = 0 + (2 + 5) = 7
- Calculate *bound*: *profit* plus summation of items' profit minus last one (k-1) + fraction of last one (W-*totweight*) * profit over weight of last item (p_k/w_k)

  bound = 0 + ($40 + $30) + (16-7) * $50/10 = 70 + 9*5 = 115
- When calculating the next node *weight* and *profit* are accumulative (for the nodes that we added because they were promising) so they start at 2 and $40 for this example and we add item's weight and we would have: *profit* = $40 + $30 = $70 and *weight* = 2 + 5 = 7

Questions ?

- Post in the discussions
- Send email to RMcFadden@HarrisburgU.edu
- Respond usually within 48hours