Algorithm Design Techniques

Hwansoo Han
Algorithm Design

- General techniques to yield effective algorithms
  - Divide-and-Conquer
  - Dynamic programming
  - Greedy techniques
  - Backtracking
  - Local search
Divide-and-Conquer

- Towers of Hanoi

- Move disks on A to C
  - Without putting a larger disk on top of a smaller disk
  - Assume a cycle consists of peg A, B, and C
  - Repeat the following steps
    - On odd-numbered moves, move the smallest disk to one peg clockwise
    - On even-numbered moves, move the only legal move not involving the smallest disk
Divide-and-Conquer

- Hanoi tower (cont’d)
  - Previous solution is concise, but hard to understand why it works

- Solution with recursion
  - Initially, $n$ disks on peg A, we need to move all disks to peg B
  - First, move $n-1$ smallest disks from peg A to peg C, exposing the $n_{th}$ smallest disk on peg A
  - Then, move the $n_{th}$ smallest disk from peg A to peg B
  - Now, we need to move $n-1$ disks from peg C to peg B
    - We can solve this recursively
  - $T(1) = 1$, $T(n) = 2T(n-1) + 1 \implies O(2^n)$
Divide-and-Conquer

- Multiplying long integers (two \( n \)-bit integers)

\[
X := \begin{array}{c}
A \\
B 
\end{array} \quad X = A \cdot 2^{n/2} + B
\]

\[
Y := \begin{array}{c}
C \\
D 
\end{array} \quad Y = C \cdot 2^{n/2} + D
\]

- \( XY = AC \cdot 2^n + (AD + BC) \cdot 2^{n/2} + BD \)
  - 4 multiplications, 3 additions, 2 shifts
  - \( T(1) = 1, T(n) = 4T(n/2) + cn \Rightarrow O(n^2) \)

- \( XY = AC \cdot 2^n + [(A-B)(D-C) + AC + BD] \cdot 2^{n/2} + BD \)
  - 3 multiplications, 6 additions/subtractions, 2 shifts
  - \( T(1) = 1, T(n) = 3T(n/2) + cn \Rightarrow O(n^{\log_2 3}) = O(n^{1.59}) \)
Dynamic Programming

- Use dynamic programming approach
  - When a problem cannot be divided into a small number of subproblems

- Dynamic programming
  - Divide into as many subproblems as necessary
  - Divide each subproblem into smaller subproblems, ...
  - To achieve polynomial-time algorithm, instead of exponential-time one
    - Define a polynomial number of subproblems
    - Keep track of the solution to each subproblem
    - Look up the solution when needed
Dynamic Programming

- Triangulation problem
  - Use triangulation of a polygon to apply shading
- Minimum cost triangulation
  - Selecting a set of chords that minimize the sum of Euclidean lengths of chords
  - Chords - lines between non-adjacent vertexes
Dynamic Programming

- Triangulation

Two subproblems when \((v_0, v_3)\) is chosen as a chord
Dynamic Programming

- Triangulation

Solution for $S_{44}$ can be reused in finding minimum cost triangulation.
Greedy Algorithms

- Making changes with as few as possible coins
  - Coins with values of 25¢, 10¢, 5¢, 1¢

- E.g. make 63¢ in change
  - Two 25¢, one 10¢ and three 1¢
  - Used a greedy algorithm
    1. Use the largest value coin not to exceed the amount
    2. Subtract the coin value from the current amount
    3. Make the current amount be the subtracted amount
    4. Repeat step 1
Greedy Algorithms

- At any individual stage, a greedy algorithm selects a *locally optimal* option

- A greedy algorithm can produce globally optimal solution
  - Making changes
  - Dijkstra’s shortest path algorithm (for non-negative cost graph)
  - Kruskal’s minimum cost spanning tree algorithm

- Not all greedy algorithms are globally optimal
  - E.g., 63¢ : coins with 32¢, 25¢, 15¢, 1¢
Greedy Algorithms as Heuristics

- Greedy algorithms as heuristics for some problems
  - No known greedy algorithm produces an optimal solution
  - Yet, greedy algorithms can be relied upon to produce “good” solutions with high probability
  - A suboptimal solution with a cost of a few percent above optimal is quite satisfactory, if we can find one fast.
Greedy Algorithms as Heuristics

- Travelling salesman problem (TSP)
  - Optimal solution can only be found by exploring “all possibilities” – this leads to exponential time (NP)

- Example
  - Six cities
    - Cost of travel = distance
  - Find a Hamiltonian cycle
    - Include all vertexes
    - Sum of the all costs of edges is a minimum
Greedy Algorithms as Heuristics

- A heuristic for Travelling Salesman Problem (TSP)
  - Similar algorithm to Kruskal’s algorithm

- Try to include the shortest edge in turn
- Accept the edge, if the following condition holds
  - Not causing a vertex to have degree three or more
  - Not forming a cycle with already accepted edges (except the last edge – which completes a cycle)
Backtracking

- A problem has no theory to find the optimum, except exhaustive search

- Systematic exhaustive search
  - Backtracking
  - Alpha-beta pruning
Backtracking – game tree

- Example of backtracking in games
  - Chess, checkers, or tic-tac-toe

- Game tree
  - All possible moves from the starting position

- tic-tac-toe example
  - X playing
  - 1: win, 0: draw, -1: lose
  - Label from the leaves
    - Propagate according to turns
    - X moves prefer 1 (MAX)
    - O moves prefer -1 (MIN)
Backtracking - payoff

Payoff Functions in Game Tree

- Leaf nodes have values (e.g., -1, 0, 1)
- Interior nodes can have payoff values, too

- In general, payoff values at leaves are ambiguous
  - Extending downward for several levels requires a great computing power in complex games (such as chess, go, ...)
  - Payoff function uses board positions to estimate the probability of winning
  - Assuming each side chooses best play, computer chooses the move with the highest payoff – *MiniMax* (*MinMax*) algorithm
Backtracking – recursive search

- Backtrack search
  - Construct a game tree and visit nodes in postorder?
  - Space to store tree can be prohibitively large
  - Build only one path, from the root to some node
    - Postorder recursive search actively builds a node when visits
    - Free the space after return
Backtracking – Alpha-Beta Pruning

- May skip some of children visits
  - Set a *tentative* value for node when at least one child has a *final* value
  - If further visit cannot change the tentative value, prune the subtree
Backtracking – Alphah-Beta Pruning

- α-β pruning example – Postorder visit
Local Search

- General local search algorithm
  - Start with a random solution
  - Apply a transformation to the current solution to improve – the improvement becomes the new “current” solution
  - Repeat until no transformation improves the current solution
Local search – minimal spanning tree

- Add an edge not in MST – which results in a cycle
- Eliminate the highest cost edge in the cycle
Local Search for Minimum Spanning Tree

- Select randomly for an initial MST
- Improve by replacing one edge at a time