Lecture 20
Algorithm Families

EECS 281: Data Structures & Algorithms

Outline

- Brute-Force
- Greedy
- Divide and Conquer
- Dynamic Programming
- Backtracking
- Branch and Bound

Brute-force Algorithms

Definition: Solves a problem in the most simple, direct, or obvious way

- Not distinguished by structure or form
- Pros
  - Often simple to implement
- Cons
  - May do more work than necessary
  - May be efficient (but typically is not)
  - *Sometimes, not that obvious*

Greedy Algorithms

Definition: Algorithm that makes sequence of decisions, and never reconsiders decisions that have been made

- Must show that locally optimal decisions lead to globally optimal solution
- Pros
  – May run significantly faster than brute-force
- Cons
  – May not lead to correct/optimal solution

Example: Sorting

- Precond: A random array of ints called myArr[]
- Postcond: For all i, j; i < j implies that myArr[i] <= myArr[j]

Sorting: Brute-Force Approach

- Generate all permutations of array myArr[]
  - $O(n!)$
- For each permutation, check if all myArr[i] <= myArr[j]
  - $O(n^2)$

Sorting: Greedy Approach

- Find smallest item, move to first location
  - n operations
- Find next smallest item, move to second location
  - n - 1 operations
- …
- Leave the largest item in the final location
  - 1 operation (0 ops if you’re clever)

Analogy: Mountain Climbing

- Brute-Force
  - Survey all of the mountains in the world
  - Go to the tallest mountain from the survey
  - Climb it!
- Greedy
  - Take a step that increases my altitude
  - Iterate
  - Until altitude is no longer increasing in any direction

Example: Counting Change

Problem Definition:

- Cashier has collection of ‘coins’ of various denominations
- Goal is to return a specified sum using the smallest number of coins

Example: Counting Change

Mathematical Definition:

- n coins:

  $P = \{p_1, p_2, p_3, \ldots, p_n\}$ with value $D = \{d_1, d_2, d_3, \ldots, d_n\}$

  – Can have repetition (two dimes, three pennies)

  – S is a subset of P

  $S \subseteq P$, such that $s_i = 1$ if $p_i \in S$, $s_i = 0$ if $p_i \notin S$

- A: sum to be returned
- Goal: minimize Σs_i, such that $\Sigma d_i = A$

Brute-force Approach

- Try all subsets of P
  - Since there are n coins, there are 2^n possible subsets
  - Enumerate all possible subsets
  - Check if a subset equals A
    - Called 'feasible solution' set
    - O(n)
  - Pick subset that minimizes Σs_i
    - Called 'objective function'
    - O(n)

Brute-force Approach

- Best Case
  - $\Omega(n\,2^n)$
- Worst Case
  - $O(n\,2^n)$

Greedy Approach

- Go from largest to smallest denomination
  - Return largest coin p_i from P, such that $d_i \leq A$
  - $A = A - d_i$
  - Find next largest coin …

If money is sorted (by value), then algorithm is O(n)

Does Greedy Always Work?

Can you devise a set of coins for which greedy does not yield an optimal solution for some amount?

Divide and Conquer Algorithms

Definition: Divide a problem solution into two (or more) smaller problems, preferably of equal size

- Often recursive
- Often involve log n
  - Why?

Divide and Conquer Algorithms

- Pros
  - Efficiency
  - ‘Elegance’ of recursion
- Cons
  - Recursive calls to small subdomains often expensive
  - Sometimes dependent upon initial state of subdomains
    - Example: binary search requires sorted array

Combine and Conquer Algorithms

Definition: Start with smallest subdomain possible. Then combine increasingly larger subdomains until size = n

Divide and Conquer: Top down

Combine and Conquer: Bottom up

Algorithms You Already Know

- D&Q
  - Search of sorted list (phonebook)
  - Quicksort, best and average case
  - Quicksort, worst case??
- C&Q
  - Mergesort

Dynamic Programming Algorithms

Definition: Remembers partial solutions when smaller instances are related

- Solves small instances first, stores the results, look up when needed
- Pros
  - Can make ‘brutally’ inefficient algorithm very efficient (sometimes $O(2^n) \Rightarrow O(n^c)$)
- Cons
  - Difficult algorithmic approach to grasp

Types of Algorithm Problems

- Constraint Satisfaction problems
  - Can we satisfy all given constraints?
  - If yes, how do we satisfy them?
    (need a specific solution)
  - May have more than one solution
  - Examples: sorting, mazes, spanning tree
- Optimization problems
  - Must satisfy all constraints (can we?) and
  - Must minimize an objective function subject to those constraints
  - Examples: giving change, MST

Types of Algorithm Problems

- Constraint Satisfaction problems
  – Stop when found a satisfying solution
- Optimization problems
  – Can’t always stop early
  – Must develop set of possible solutions
    - Called *feasibility set*
  – Then, must pick ‘best’ solution

Types of Algorithm Problems

- Constraint Satisfaction problems
  - are to *Backtracking*, as
- Optimization problems
  - are to *Branch and Bound*

Backtracking Algorithms

Definition: Systematically consider all possible outcomes of each decision, but *prune* searches that do not satisfy constraint(s)

- Think of as DFS with Pruning
- Pros
  - Eliminates exhaustive search
- Cons

Graph Properties

- Maps can be drawn as *planar* graphs
- Planar definition: a graph that can be drawn with no crossing edges
- Conversion of a map to a planar graph
  - States become nodes
  - Shared borders become edges

Applied Backtracking: 4 Color

Example: *graph coloring* in four colors

- Assign colors to vertices such that no two vertices connected by an edge have the same color
- Some graphs can be 4-colored, and some cannot
  - Give examples
- Given a graph, is it 4-colorable?

From Enumeration to Backtracking

- Enumeration
  - Take vertex v_1, consider 4 branches (colors)
  - Then take vertex v_2, consider 4 branches
  - Then take vertex v_3, consider 4 branches
  - …
- Suppose there is an edge (v_1, v_2)
  - Then among 4 x 4 = 16 branches,
    4 are dead-ends (don't lead to a solution)

Backtracking

- Branch on every possibility
- Maintain one or more "partial solutions"
- Check every partial solution for validity
  - If a partial solution violates some constraint, it makes no sense to extend it (so drop it), i.e., **backtrack**
- Why is this better than enumeration?

M-Coloring Algorithm

Input: integer n (number of nodes), integer m (number of colors), integer adjacency matrix W[1..n][1..n] where W[i][j] is true if there is an edge from node i to node j, and false otherwise

Output: all possible colorings of graph represented by int vcolor[1..n], where vcolor[i] is the color associated with node i

M-Coloring Algorithm

```
Algorithm m_coloring(index i)
   if (promising(i))
      if (i == n)
         print vcolor(1) thru vcolor(n)
      else
         for (color = 1; color <= m; color++)
            vcolor[i + 1] = color
            m_coloring(i + 1)
```

M-Coloring Algorithm

```
bool promising(index i)
   index j = 1
   bool switch = true

   while (j < i and switch)
      if (W[i][j] and vcolor[i] == vcolor[j])
         switch = false
      j++

   return switch
```

Summary: Algorithms

- Brute-force:
  - Solve problem in simplest way
  - Generate entire solution set, pick best
  - Will give optimal solution with (typically) poor efficiency
- Greedy:
  - Make local, best decision, and don't look back
  - May give optimal solution with (typically) 'better' efficiency
  - Depends upon 'greedy-choice property'
    - Global optimum found by series of local optimum choices

Summary: Algorithms

- Divide and Conquer
  - Divide problem into *non-overlapping* subspaces
  - Solve within each subspace
  - Works best (typically) when subspaces divide in half
- Dynamic Programming
  - Similar to D&C, but used for *overlapping* subspaces
  - Used when partial solutions are needed later
  - Often times looking “nearby” for previously calculated values

Summary: Algorithms

- Backtracking
  - Used for pruning in *Constraint Satisfaction* problems
  - For problems that require ***any*** solution
  - Can determine / prune ‘dead-ends’
- Branch and Bound
  - Used for pruning in *Optimization* problems
  - For problems that require a ***best*** solution
  - Can determine / prune non-promising branches