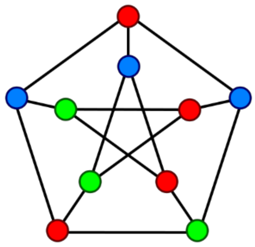
Lecture 21
Algorithm Families



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*

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

Brute-force Counting Change

Try all subsets *S* of coins, *C* to make change totaling *A*.

- Since there are n coins, there are 2^n possible subsets
- Check if sum of subset coins equals *A*
  - Called "feasible solution" set
  - **$O(n)$**
- Pick a feasible subset that minimizes $|S|$
  - Called "objective function"
  - **$O(n)$**

Fewest coins that sum to 30¢?

				# Coins
0	0	0	30	30
0	0	1	25	26
0	0	2	20	12
0	0	3	15	18
0	0	4	10	14
0	0	5	5	10
0	0	6	0	6
0	1	0	20	21
0	1	1	15	17
0	1	2	10	13
0	1	3	5	9
0	1	4	0	5
0	2	0	10	12
0	2	1	5	8
0	2	2	0	4
0	3	0	0	3
1	0	0	5	6
1	0	1	0	2

Brute-Force Counting Change

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

Greedy Algorithms

Definition: Algorithm makes a sequence of decisions (best at each point), 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 an optimal solution

Greedy Counting Change

- 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 already sorted (by value), then the algorithm is $\boldsymbol{O(n)}$

Fewest coins that sum to 30¢?

Greedy: Take the best option at the time

25¢ > 10¢ > 5¢ > 1¢

1. Always pick quarter if possible
2. Pick dimes if possible
3. Pick nickels if possible
4. Pick pennies if possible

Fewest coins that sum to 30¢?

Greedy: Take the best option at the time

25¢ > 10¢ > 5¢ > 1¢

1	0	1	0

\# Coins: 2

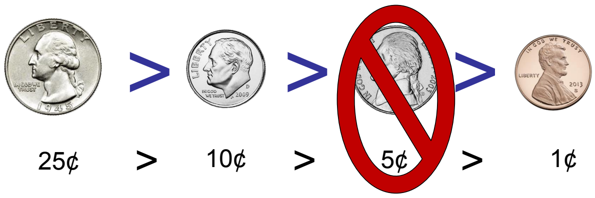
Does Greedy Always Work?

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

A: Pennies, Dimes, Quarters to make 30¢

Fewest coins that sum to 30¢?

Greedy: Take best option at the time

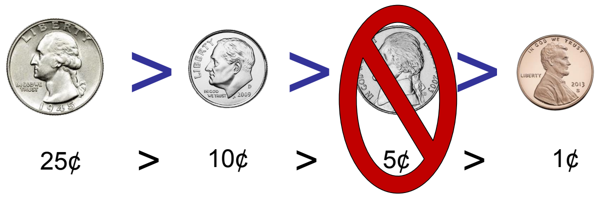


25¢ > 10¢ > 5¢ > 1¢

1. Always pick quarter if possible
2. Pick dimes if possible
3. ~~Pick nickels if possible~~
4. Pick pennies if possible

Fewest coins that sum to 30¢?

Greedy: Take best option at the time



25¢ > 10¢ > 5¢ > 1¢

1	0	-	5

\# Coins: 6

Brute-Force:

Fewest coins that sum to 30¢?

				# Coins
0	0	0	30	30
0	1	0	20	21
0	2	0	10	12
0	3	0	0	3
1	0	0	5	6

Example: Sorting

- Precond: A random array of int called myArr[]
- Postcond: For all i < n – 1, myArr[i] ≤ myArr[i + 1]

Sorting: Brute-Force Approach

- Generate all permutations of array myArr[]
  - ***O*(*n*!)**
- For each permutation, check if all myArr[i] ≤ myArr[i + 1]
  - ***O*(*n*)**

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)

Example: Mountain Climbing

- Brute-Force
  - Lay out a grid in the area around the mountain
  - Visit <u>all</u> possible locations in the grid
  - The highest measured altitude was the top
- Greedy
  - Take a step that increases altitude
  - Iterate until altitude is no longer increasing in any direction

Proving Greedy Optimality

- Need an optimal substructure

  Optimal solution = first "best" action + optimal solution for remaining subproblem
- Need a greedy-choice property

  First action can be chosen greedily without invalidating optimal solution
- Applied recursively though often programmed iteratively

Algorithm Family Summary

- 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

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

- Divide and Conquer
  - Binary Search of sorted list (phonebook)
  - Quicksort
- Combine and Conquer
  - Merge Sort

Dynamic Programming Algorithms

Definition: Remember 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)** -> ***O*(n^c)**)
- Cons
  - Difficult algorithmic approach to grasp

Dynamic Programming: Fibonacci

- Fibonacci Numbers
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
- Try F_{50}

$$
\begin{aligned}
F_{50} &= F_{49} + F_{48} \\
&= F_{48} + F_{47} + F_{47} + F_{46} \\
&= F_{47} + F_{46} + F_{46} + F_{45} + F_{46} + F_{45} + F_{45} + F_{44} \\
&= \vdots
\end{aligned}
$$

Algorithm Family Summary

- Divide and Conquer
  - Divide problem into <u>non-overlapping</u> subspaces
  - Solve within each subspace
  - Most efficient when subspaces divide evenly
- Dynamic Programming
  - Similar to Divide and Conquer, but used for <u>overlapping</u> subspaces
  - Used when partial solutions are needed later
  - Often times looking "nearby" for previously calculated values

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 a satisfying solution is found
    - If one solution is sufficient
- Optimization problems
  - Usually cannot stop early
  - Must develop set of possible solutions
    - Called *feasibility set*
    - Usually just the best complete solution so far, and the current partial solution being developed
  - When done, the best solution seen is the best

Types of Algorithm Problems

- Constraint Satisfaction problems
  - Can rely on *Backtracking algorithms*
- Optimization problems
  - Can rely on *Branch and Bound algorithms*
- For particular problems, there may be much more efficient approaches
  - If greedy works, use that before one of these
- Think of these as a fallback to a more sophisticated version of a brute force approach

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
  - Search space is still large

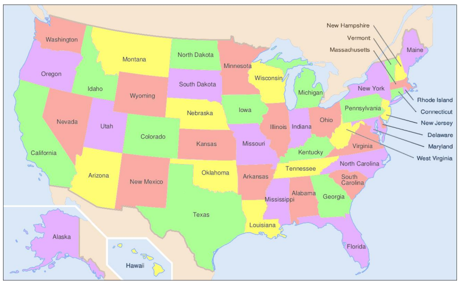
Applied Backtracking: 4 Color

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?

Graph Coloring

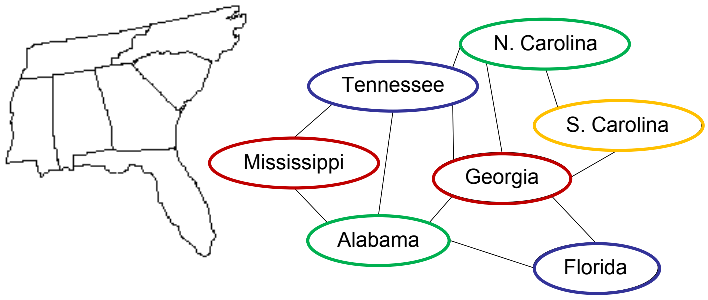
- Ever wonder how to pick colors in a map without coloring adjacent states the same color?



Graph Properties

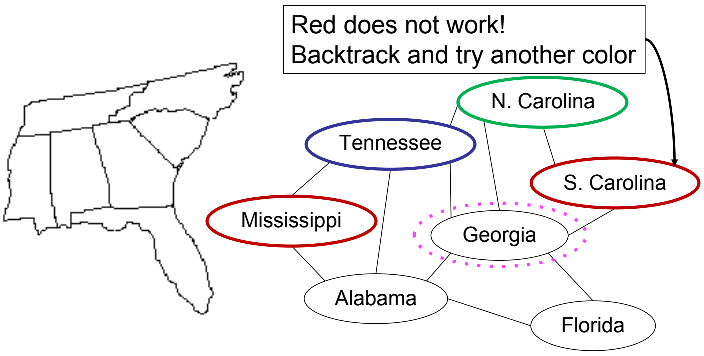
- Cartographic maps can be drawn as *planar* graphs
- **Planar Graph**: a graph that can be drawn with no crossing edges
- Conversion of a map to a planar graph
  - States become nodes
  - Shared borders become edges

Map to Graph conversion



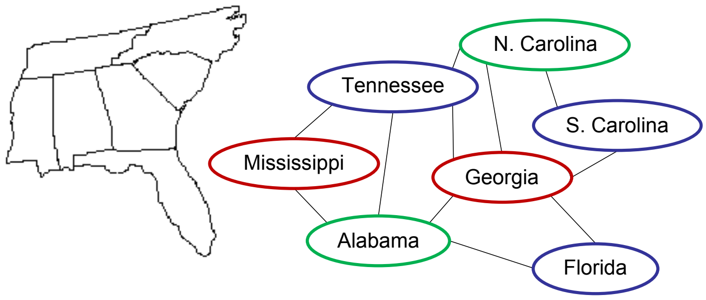
Using 4 colors: R B G O

Map to Graph conversion



Using **3** colors: R B G

Map to Graph conversion



Using **3** colors: R B G

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
- Must maintain the current partial solution being developed
  - Might print *or* maintain all complete 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: n (number of nodes),
       m (number of colors),
       W[0..n)[0..n) (adjacency matrix)
           ( 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[0..n), where
  vcolor[i] is the color associated with
  node i
```


M-Coloring Algorithm

```
Algorithm m_coloring(index i = 0)
   if (i == n)
      print vcolor(0) thru vcolor(n - 1)
      return
   for (color = 0; color < m; color++)
      vcolor[i] = color
      if (promising(i))
         m_coloring(i + 1)
```


M-Coloring Algorithm

```
bool promising(index i)
   for (index j = 0; j < i; ++j)
      if (W[i][j] and vcolor[i] == vcolor[j])
         return false

   return true
```


When is Backtracking Efficient?

- Backtracking avoids looking at large portions of the search space by pruning, but this does not necessarily improve the asymptotic complexity over brute force.
  - e.g. If we prune out 99% of the search space, 0.01 * b^n is still ***O*(*b^n*)**
- However, backtracking works well for constraint satisfaction problems that are either:
  - Highly-constrained: Constraint violations are detected early in partial solutions and lead to MASSIVE amounts of pruning.
  - Under-constrained: Acceptable solutions are densely distributed, so it is quite likely we find one early and can terminate.

Algorithm Family Summary

- Backtracking
  - Used for pruning in *Constraint Satisfaction* problems
  - For problems that require *<u>any</u>* solution
  - Can determine/prune dead ends (choices that break constraints)
- Branch and Bound
  - Used for pruning in *Optimization* problems
  - For problems that require a *<u>best</u>* solution
  - Can determine/prune both dead ends *and* non-promising branches