Backtracking
Edmonds, Chapter 23-25

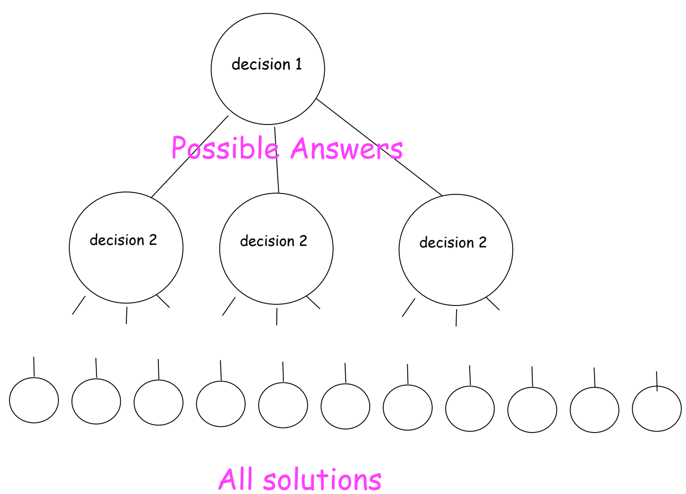
Neil Rhodes
CSE 101
UC San Diego

Backtracking

Search through the solution space by generating a decision tree
- Decisions made so far may be infeasible; no point to go further down the tree
- In worst-case, runtime is exponential (in practice, may do better)
- May get better base for exponent, even if still exponential
- Dynamic Programming is a special subcase
  - Just as Greedy algorithm is a special subcase of Dynamic Programming

Branch and Bound next class
- Search through the decision tree visiting more promising nodes first
- Ignore branches of the tree that can't help

Decision Tree



Optimization Problems

Instances
- The possible inputs to the problem

Solutions for instance
- Each instance has an exponentially large set of solutions

Constraint
- Specifies which of the solutions is valid

Cost of solution
- Each solution has an easy-to-compute cost

Specification of an optimization problem
- Precondition: the input is one instance
- Postcondition: the output is one of the valid solutions with optimal cost

Backtracking Master Algorithm for Optimization Problem

Depth-first search of decision tree

Definition

- Non-promising node: We can't get to a valid solution from here

Basic structure of recursive Backtracking algorithm

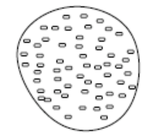
- Backtrack(node v)

```
    if v is not promising then
        return ({}, infinity)
    if there is a solution at v then
        return (solution, cost of solution)
    else
        foreach child u of v
            (optimal subsolution, cost) = Backtrack(u)
            opt-cost_u = optimal subsolution cost + cost of choice u
            opt-sol_u = optimal subsolution + choice u
        umin = u that minimizes opt-cost_u
        return (opt-sol_umin , opt-cost_umin)
```

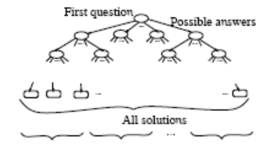

Backtracking Master Method for Optimization Problems

Given one instance

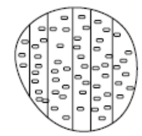
Set of solutions for Instance



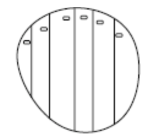
Tree of questions



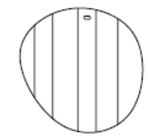
Classification of solutions based on question



Find best solution in each class



Choose best of the best



Sum of Subsets

Sum of Subsets

Instances

- A set of n integers: $x_1..x_n$
- A total W

Solutions for instance

- Subset, S, of the n integers.

Constraint

- The sum of the subset = W

Cost

- |S| (minimize)

Idea

- Like the integer knapsack problem, except all items have the same weight, and the knapsack must be completely filled

Sum of Subsets example

values = {3, 4, 5, 6}
W = 13

Sum of Subsets Algorithm

```
SOS(values, subsetSoFar, W)
  if sum of subsetSoFar > W or sum of subsetSoFar + sum of values < W then
    return {};                    Not promising
  if sum of subsetSoFar = W then
    return subsetSoFar
  else if values = subsetSoFar then
    return {}
  else
    Pick x, the first element from values
    result_with = Backtrack(values - {x}, subsetSoFar + {x}, W)
    result_without = Backtrack(values - {x}, subsetSoFar, W)
    if |result_with| > 0 then return result_with     Don't need to return cost since
    else return result_without                        it's just the size of the subset

SOS(values, V)
  sort values in increasing order          allows us better promising analysis
  return SOS(values, {}, W)
```


Maximum Independent Set

Maximum Independent Set Problem

Instances
- A graph G = (V,E)

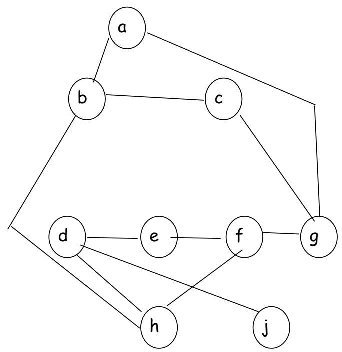
Solutions for instance
- A subset, V', of V

Constraint
- No edge in E connects two nodes in the solution (V' is independent)

Cost
- |V'| (maximize)

Idea
- V is your friends. Edges in E represent two people who dislike each other. You want to invite as many friends as possible to your party, but not people who dislike each other.

Maximum Independent Set Example



Maximum Independent Set Algorithm

```
MIS(V, E)
   if |V| <= 1 then
      return V
   else
      Pick x, an element from V
      result_with = {x} U MIS(G - neighbors(x))
      result_without = MIS(G - {x})
      if |result_with| > |result_without| then
         return result_with
      else return result_without
```

Notes:

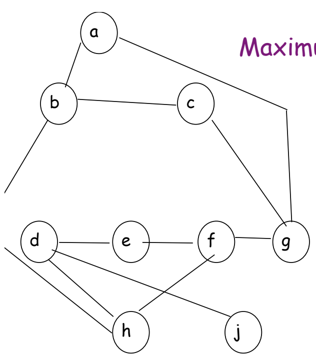
- Don't need to expliclity return cost, since cost is just |V|

Runtime

$T(n) = T(n\text{-}1 - d(v)) + T(n\text{-}1)$

$T(n) = 2T(n\text{-}1) = 2\text{\^{}}n$

Maximum Independent Set Example



Maximum Independent Set Better Algorithm

```
MIS(V, E)
   if |V| <= 1 then
      return V
   else
      Pick x, an element from V
      result_with = {x} U MIS(G - neighbors{x})
      if d(x) > 0 then                v has a neighbor
         result_without = MIS(G - {x})
      else result_without = {}
      if |result_with| > |result_without| then
         return result_with
      else return result_without
```

Runtime

$T(n) = T(n\text{-}1) + T(n\text{-}2)$

$T(n)$ approx $(1+\text{sqrt}(5)/2)\text{\^{}}n = 2^{.7n}$

Tight bound because

line

Maximum Independent Set Even Better Algorithm

```
MIS(G=(V, E))
  if |V| <= 1 then
    return V
  else
    Pick x, an element from V
    result_with = {x} U MIS(G - neighbors{x})
    if d(x) > 1 then        v has more than a single neighbor
      result_without = MIS(G - {x})
    else result_without = {}
    if |result_with| > |result_without| then
      return result_with
    else return result_without
```

x in, y out: MIS(S)

x out, y in: y+MIS(S-...)

x out y out: MIS(S)

Runtime

$T(n) = T(n-1) + T(n-3)$

$T(n)$ approx $2^{.6n}$

How 2^n Grows

N	10	20	30	40	50	60	70
# operations	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}
time (1 op/nanosecond)	1 microsecond	1 millisecond	1 sec	1/2 day	500 days	1000 years	1000 millenia