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

decision 1

Possible Answers

decision 2
decision 2
decision 2

All solutions

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

Instances
- A set of n integers: x1..xn
- 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\text{with} = Backtrack(values - \{x\}, subsetSoFar + \{x\}, W)
        result\text{without} = Backtrack(values - \{x\}, subsetSoFar, W)
        if |result\text{with}| > 0 then return result\text{with}
        else return result\text{without}

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

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
    resultwith = {x} U MIS(G - neighbors(x))
    resultwithout = MIS(G - {x})
    if |resultwith| > |resultwithout| then
        return resultwith
    else return resultwithout

Notes:
- Don't need to explicitely return cost, since cost is just |V|

Runtime
T(n) = T(n-1 - d(v)) + T(n-1)
T(n) = 2T(n-1) = 2^n

Maximum Independent Set Better Algorithm

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

Runtime
T(n) = T(n-1) + T(n-2)
T(n) approx (1+sqrt(5)/2)^n = 2.7^n

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 2.6^n

How $2^n$ Grows

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