Chapter 4: Greedy Algorithm

4.1 Introduction

- Optimization problem:
  - solution requires to satisfy some constraints
  - many possible solutions, each with a value (feasible solutions)
  - goal: to find a feasible solution with the optimal value (either max. or min.)
  - may have few optimal solutions

- Greedy Algorithms

  - Given an optimization problem. If the problem exhibits the following properties, then we may use greedy algorithm strategy to solve it:
    
    **greedy choice property**: local greedy choice gives globally optimal solution
    
    - usually, arrange elements in certain order
    
    - consider one input at a time, determine whether or not a particular input is in optimal solution (could be partially)

  **optimal sub-structure**: the optimal solution to the problem contains optimal solution to sub problem

  - after considering an input, simplify the remaining set to obtain a feasible sub-problem

  - then find optimal solution for the remaining elements by repeating steps in "greedy choice property"
4.2 Activity-selection problem

- Problem: Given a set of n activities, i.e. \( \{1,2,\ldots, n\} \)
  Let \( s_i \) the start time and \( f_i \) is the finishing time of activity \( i \)

- Goal: Find the maximum size subset of compatible activities
  (i.e. intervals do not overlap)

- Example: an instance of activity-selection problem

<table>
<thead>
<tr>
<th>( i )</th>
<th>( s_i )</th>
<th>( f_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>17</td>
</tr>
</tbody>
</table>

Possible feasible solutions: \( \{2,6\} \) \( \{1,3,6\} \) \( \{1,3,5\} \) etc

- Algorithm X: /* assume \( f_1 \leq f_2 \leq \ldots \leq f_n */

\[
A = \{ 1 \}; \quad j = 1
\]

\[
\text{for } \ i = 2 \text{ to } n \ {\}
\]
\[
\quad \text{if } s_i \geq f_j \text{ then } \{ \quad /* otherwise, overlap intervals */
\]
\[
\quad \quad A = A \cup \{i\}; \quad j = i;
\]
\[
\}
\]

\[
\}
\]

\[
\text{return } A;\]

- For the above example: \( A = \{1,3,5\} \)
Theorem: Algorithm X is an optimal algorithm
Proof:

/* Greedy choice property : \{1\} is always in optimal solution */

Let \( S = \{1,2,...,n\} \) assume \( f_1 \leq f_2 \leq ... \leq f_n \)

Show that there is an optimal solution \( A \) containing \{1\}

Let \( B \) be an optimal solution, let order activities in \( B \) by their finishing time, \( f_{b_1} \leq f_{b_2} \leq ... \leq f_{b_k} \)

If \( b_1 = \) activity 1, then done
If \( b_1 \neq \) activity 1, then let \( A = B \setminus \{b_1\} \cup \{1\} \)

\( |A| = |B| \) and \( f_1 \leq f_{b_1} \) since \( f_1 \) has the least finishing time \( \rightarrow \)
\( A \) is a feasible solution \( \rightarrow \) \( A \) is also an optimal solution

/* Optimal substructure */

Let \( S^* = \{ \text{activity } i \in S \text{ and } s_i \geq f_1 \} \)

Let \( C = A \setminus \{1\} \) where \( A \) is an optimal solution with activity 1

Claim: \( C \) is an optimal solution for \( S^* \)

If this is not true, assume \( D \) is an optimal solution for \( S^* \), i.e. \( |D| > |C| \)
\( \rightarrow \) \( D \cup \{1\} \) contains a set of compatible activities for \( S \)
\( \rightarrow \) \( |D \cup \{1\}| > |A| \)
\( \rightarrow \) \( A \) cannot be optimal \( \star \) contradiction \( \star \)
4.3 Knapsack problem

- Problem: Given \( n \) objects \( \{o_1, o_2, \ldots, o_n\} \), each has weight \( w_i \) and profit \( p_i \), and a knapsack has a capacity \( M \). If a fraction \( x_i \), \( 0 \leq x_i \leq 1 \) of \( o_i \) is placed in knapsack then profit of \( p_i \times x_i \) is earned.

- Objective: To obtain a filling of the knapsack that maximize the total profit earned, i.e. maximize \( \sum p_i x_i \) such that \( \sum w_i x_i \leq M \)

- Example: an instance of knapsack problem: \( M = 20 \)

<table>
<thead>
<tr>
<th>( o_i )</th>
<th>( w_i )</th>
<th>( p_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_1 )</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>( o_2 )</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>( o_3 )</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

some feasible solutions:

- order objects by weight, i.e. \( <w_1,w_2,w_3> \Rightarrow x_i = <18/18, 2/15, 0/10> \Rightarrow \text{total profit} : 25 + 3.466 + 0 = 28.466

- order objects by profit, i.e. \( <p_2,p_1,p_3> \Rightarrow x_i = <15/15, 5/18, 0/10> \Rightarrow \text{total profit} : 26 + 6.944 + 0 = 32.944

- order objects by profit/weight, i.e. profit per unit, i.e. \( <p_2/w_2,p_3/w_3,p_1/w_1> \Rightarrow x_i = <15/15, 5/10, 0/18> \Rightarrow \text{total profit} : 26 + 7.5 + 0 = 33.50

Algorithm Y: /* assume total weight of all objects > \( M \); otherwise, all objects are included. running time: \( O(n \log n) \) */

- sort elements in nondecreasing order of \( p_i/w_i \)

- re-label objects such that \( p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n \)

- use objects in sequence \( o_1, o_2, \ldots, o_n \), put as much weight into knapsack as possible until knapsack is filled
Theorem: Algorithm Y is an optimal algorithm
Proof:

/* Greedy choice property */

Let \( \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n} \)

Show that there is an optimal solution such that \( x_1 = \min \{ \frac{M}{w_1}, 1 \} \)

i.e. if \( M < w_1 \), then knapsack contains only \( o_1 \) and \( x_1 = M/w_1 \);
otherwise \( x_1 = 1 \)

Let \( Y = (y_1, y_2, \ldots, y_n) \) be an optimal solution such that \( y_i \) is the fraction of \( o_i \) included in knapsack

If \( y_1 = x_1 \) then done!

If \( y_1 \neq x_1 \)
→ increase weight of \( o_1 \) in knapsack so that \( y_1 \) becomes \( x_1 \) and decrease same
amount of weight from other objects from knapsack
→ same total weight; does not decrease the total profit of \( Y \) since \( o_1 \) has
largest profit per unit weight.

Therefore, there is an optimal solution such that \( x_1 = \min \{ \frac{M}{w_1}, 1 \} \).
Let the solution be \( X = (x_1, x_2, \ldots, x_n) \)

/* Optimal substructure */

New subproblem: Let there be \( n-1 \) remaining objects, \( \{ o_2, o_3, \ldots, o_n \} \) and
remaining capacity of knapsack be \( M^* = M - x_1 w_1 \)
Then \( X^* = (x_2, x_3, \ldots, x_n) \) is an optimal solution for the new subproblem.

If this is not true, say, \( Z = (z_2, z_3, \ldots, z_n) \) is an optimal solution for the sub
problem, i.e. \( \text{total-profit}(Z) > \text{total-profit}(X^*) \)
→ \( \text{total-profit} (x_1, z_2, \ldots, z_n) > \text{total-profit of} (x_1, x_2, \ldots, x_n) \)
→ \( X \) is not an optimal solution  ** contradiction
4.4 Exercises

- **Total time in the system**: Given \(n\) jobs, \(\{J_1, J_2, \ldots, J_n\}\), available at time \(= 0\) in the system. Each job \(J_i\) requires a service time \(T_i > 0\). The objective of the problem is to service all \(n\) jobs such that the total time (waiting time + service time) in the system is minimized.

- **Coin changing problem**: The objective is to change \(n\) cents using the fewest number of coins.
  
  (a) Let \(A=\{32, 16, 8, 4, 2, 1\}\) be a set of distinct coin types and assume each type is available in unlimited quantity. Give a greedy algorithm to solve the problem.

  (b) Let \(A=\{25, 10, 1\}\). Does your algorithm still yield optimal solution?

- **0/1 knapsack problem**, i.e. \(x_i = 1\) or \(0\) only. How can you solve this problem?
Chapter 5 : Dynamic Programming

5.1 Introduction

- Usually, this technique covers all possible cases to produce a set of results. Optimal values are included in this set.

- Optimal solutions may be computed from the result set.

- The solution is based on recursive property and obtained by solving the problem in bottom-up fashion.

- Once the recursive property is proved, the correctness of the solution is obvious.

- General approach :

  Step 1. characterize the structure of an optimal solution (optimal substructure)

  Step 2. recursively define the value of an optimal solution

  Step 3. compute the value of an optimal solution in a bottom-up fashion
          /* optimal value */

  Step 4. construct an optimal solution for computed information
          /* use computed values from Step 3 to get an optimal solution */
5.2 Longest Common Subsequence (LCS)

- Problem: given two sequences $X[1..m]$ and $Y[1..n]$, find a longest subsequence common to both.

- Example: $X = A\ B\ C\ B\ D\ A\ B$
  $Y = B\ D\ C\ A\ B\ A$
  
  LCS: $<B\ C\ B\ A>$, $<B\ D\ A\ B>$ and etc
  Optimal value is the size of LCS, i.e. 4

- Algorithm #1: Brute force approach
  
  For every subsequence $A$ of $X$, check if $A$ a subsequence of $Y$
  
  Running time:
  
  Assume $|X| = m$, then $2^m$ subsequences for $X$
  
  Each subsequence check with $Y$. if $|Y| = n$, it takes $O(n)$ to check
  
  Total time: $O(n\ 2^m)$
• Dynamic programming algorithm

Step 1: characterize the optimal substructure

Let \( X = (x_1, x_2, \ldots, x_m), Y = (y_1, y_2, \ldots, y_n) \) and
\( Z = (z_1, z_2, \ldots, z_k) \) be any LCS of \( X \) and \( Y \)

/* Assume \( S_i = (s_1, s_2, \ldots, s_i) \), e.g. \( X_m = X^* \) */

- if \( x_m = y_n \) then
  \( (z_k = x_m = y_n) \) and \( (Z_{k-1} \) is a LCS of \( X_{m-1} \) and \( Y_{n-1} \) )

- if \( x_m \neq y_n \) then
  \( (z_k \neq x_m) \) \( \Rightarrow \) \( Z \) is an LCS of \( X_{m-1} \) and \( Y \)

- if \( x_m \neq y_n \) then
  \( (z_k \neq y_n) \) \( \Rightarrow \) \( Z \) is an LCS of \( X \) and \( Y_{n-1} \)

Step 2: define the recursive equation to obtain optimal value

Definition:

Let \( c[i,j] = \) length of LCS of \( X[1,\ldots,i] \) and \( Y[1,\ldots,j] \)
then \( c[m,n] = \) length of LCS of \( X \) and \( Y \)

Based on Step 1, define recursive function :

\[
c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\text{if } i,j > 0 \text{ and } x_i = y_j \\
\text{max}(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \ne y_j 
\end{cases}
\]
Recursive Function (From recursive equation above)

LCS(X, i, Y, j) // exponential running time
{
    if (i == 0) or (j == 0) return 0

    if (X[i] == Y[j]) result = 1 + LCS(X,i-1,Y,j-1)
    else result = max(LCS(X,i-1,Y,j), LCS(X,i,Y,j-1))

    return result;
}

Example of recursive tree:

- Total depth O(n+m), Each node has 3 branches, so total work O(3^{n+m})
- For dynamic programming - do not recalculate the repeated sub-problem (memorize the solution)
- For divide and conquer - recalculate all subproblems, e.g. partition
- How many non-overlapping subproblems in LCS?
Improved Recursive Function (From recursive equation above)

// assume there is a global table C[i,j], all entries are initialized to “unknown”

LCS(X, i, Y, j) // running time: ??
{
    if (i == 0) or (j == 0) return 0
    if (C[i,j] != unknown) return C[i,j] // add this line
    if (X[i] == Y[j]) result = 1 + LCS(X,i-1,Y,j-1)
       else result = max(LCS(X,i-1,Y,j), LCS(X,i,Y,j-1))
    C[i,j]=result; // add this line
    return result;
}

Non-Recursive Solution (From recursive equation above)

Compute value of c[i,j] by using a table, store results of all subproblems in the table. Algorithm LCS_Length(X,Y):

- Let m = length(X), n= length(Y)
- Set C[i,0] = 0 and C[0,j] = 0 for 0 <= i <= m and 0<= j <= n
- for i = 1 to m
    for j = 1 to n
        if x_i = y_j then C[i,j] = C[i-1,j-1]+1
        else C[i,j] = max(C[i-1,j], C[i,j-1]);

Running time O(mn), since the size of table is (m+1) by (n+1)
Example: A table C[] for X=ABCBDAB and Y=BDCABA

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X</strong></td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
C[i-1,j-1] & C[i-1,j] \\
\hline
C[i,j-1] & C[i,j]
\end{array}
\]

Step 4: Use table C[] above, to compute a LCS

Algorithm Print_LCS(C,X,Y,i,j)
Input: Any C[i,j] entry with optimal value; e.g. C[7,5] or C[6,6]

- if (i=0) or (j=0) then return
- if (C[i,j] = C[i-1,j-1]+1) and (x_i = y_j) then
  Print_LCS(C,X,Y,i-1,j-1); output x_i
- else if (C[i-1,j] >= C[i,j-1]) then Print_LCS(C,X,Y,i,j-1)
- else Print_LCS(C,X,Y,i-1,j)

Example: C[7,5] \rightarrow C[6,4] \rightarrow C[5,3] \rightarrow C[5,2] \rightarrow C[4,1] \rightarrow C[3,0]

B A D B

Running time ??
5.3 0/1 Knapsack Problem

- Problem: Given n objects \(\{o_1, o_2, \ldots, o_n\}\), each has integer weight \(w_i\) and profit \(p_i\), and a knapsack has a capacity \(M\). If \(o_i\) is placed in knapsack then \(x_i=1\), otherwise \(x_i=0\).

- Objective: To obtain a filling of the knapsack that maximize the total profit earned, i.e. maximize \(\sum p_i x_i\) such that \(\sum w_i x_i \leq M\).

- Optimal Substructure:

  \(Z\) – set of objects in optimal solution for \(\{o_1, o_2, \ldots, o_n\}\) with total weight \(\leq M\).

  - If \(o_n\) in \(Z\) then \(Z - \{o_n\}\) is an optimal solution for \(\{o_1, o_2, \ldots, o_{n-1}\}\) with total weight \(\leq M - w_n\).
  - If \(o_n\) is not in \(Z\) then \(Z\) is an optimal solution for \(\{o_1, o_2, \ldots, o_{n-1}\}\) with total weight \(\leq M\).

- The recursive equation to obtain optimal value

\[
f(j,Y) = \max \{ f(j-1,Y), f(j-1, Y-w_j) + p_j \} \quad \text{for } j > 0 \text{ and } Y > 0
\]
\[
0 \quad \text{for } j = 0 \text{ or } Y = 0
\]
\[
-\infty \quad \text{for } Y < 0
\]

then \(f(n,M)\) is the desired optimal profit.
Recursive Function (From recursive equation above)

Knapsack(j, Y)               // exponential running time
{
    if (j == 0)    return 0
    if (w_j > Y) result = Knapsack(j-1, Y)  // cannot use this item j
    else result = max(p_j + Knapsack(j-1, Y - w_j), Knapsack(j-1, Y));
    return result;
}

Improved Recursive Function (From recursive equation above)
// assume there is a global table T[j,Y], all entries are initialized to “unknown”

Knapsack(j, Y)               // running time??
{
    if (j == 0)    return 0
    if (T[j,Y] != unknown) return C[j,Y]             // add this line
    if (w_j > Y) result = Knapsack(j-1, Y)  // cannot use this item j
    else result = max(p_j + Knapsack(j-1, Y - w_j), Knapsack(j-1, Y));

    T[j,Y]=result;                  // add this line
    return result;
}

Non-recursive Solution
Goal: To compute a table T of (n+1) by M entries

Initialize T(0,k) = 0 for all k and assume T(j,k) = -∞ for all k < 0.

for j = 1 to n
    for k = 0 to M
        T(j,k) = max { T(j-1,k), T(j-1, k-w_j) + p_j }
• Example: \( M = 6 \)

<table>
<thead>
<tr>
<th>Object</th>
<th>( o_1 )</th>
<th>( o_2 )</th>
<th>( o_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Profit</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( j \downarrow k \rightarrow )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• Running time \( O(nM) \)

5.5 Exercises

Multistage Graphs Problem: Given \( G=(V,E) \) is a directed graph where
- \( V \) is divided into \( k \geq 2 \) disjoint sets \( \{V_1, V_2, \ldots, V_k\} \); \( V_1 = \{s\} \) and \( V_k = \{t\} \).
  /* \( s \) is source and \( t \) is sink */
- Each edge \((u,v) \in E\) has a positive cost \( c(u,v) \geq 1 \); \( u \in V_i \) and \( v \in V_{i+1} \)

Objective: Find a minimum cost path from \( s \) to \( t \).

Example:
Chapter 6: Backtracking & Branch-and-bound

6.1 Introduction

- Breadth-first search and depth-first search algorithm for a tree

\[
\text{BFS(tree } T\text{)} \\
\{ \\
\quad \text{initialize}(Q) \\
\quad \text{insert}(Q,r) \\
\quad \text{while } Q \neq \text{empty } \{ \\
\quad \quad \text{u = delete}(Q) \\
\quad \quad \text{for each child } v \text{ of } u \{ \\
\quad \quad \quad \text{visit } v \\
\quad \quad \quad \text{insert}(Q,v) \\
\quad \quad \} \\
\quad \} \\
\}
\]

\[
\text{DFS(tree-node } u\text{)} \\
\{ \\
\quad \text{for each child } v \text{ of } u \\
\quad \text{DFS}_\text{Visit}[v] \\
\}
\]

- Example:

BFS Tree

DFS Tree
Many problems have solutions which can be expressed as a sequence of objects, i.e. 
\((x_1, x_2, \ldots, x_n)\), where \(x_i\) are chosen from some finite set.

- Example: 0/1 Knapsack problem

- By using brute force approach, we can evaluate all possible candidates to obtain all solutions.

- Frequently, this method involves evaluating large number of possible solutions such as \(2^n\) (all possible subsets), \(n!\) (all possible permutations) etc.

- We can also use BFS and DFS to obtain all solutions.

- Example 1:

  Sum of subsets problem: Given \(n\) positive numbers \(S = (w_1, w_2, \ldots, w_n)\) and a positive number \(m\). Find all possible subsets of \(S\) whose sums are \(m\).

  \(S = (11, 13, 24, 7), \ m = 31\)

  Two possible ways to obtain solutions (state space tree)

Each node is a subset!
S=(11, 13, 24, 7), m = 31

- Use DFS (recursive version) to obtain solutions for Sum of Subset
  - Use a global array x[1..n] to keep track of solution, initialize with 0
  - To start the computation, call Sum_of_subsets(1)
  - Sum_of_subsets (i) recursive function:

    ```java
    if (i <= n) {
        x[i] = 1 // include w_i
        Sum_of_subsets(i+1)
        x[i] = 0 // exclude w_i
        Sum_of_subsets(i+1);
    } else // i > n
        Check sum of values in array x[1…n], compare with m
    ```

A path to a leaf is a subset!
Example 2:

n-queens problem: Place n queens on an n by n chessboard so that no two queens threaten each other. That is, no two queens may be in the same row, column or diagonal.

A possible solution for 8-queens problem:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Use DFS to obtain solutions for 4-queens problem (state space tree)

- Use a global array x[1..n] to keep track of solution, initialize with 0
- To start the computation, call Nqueens(1,n)
- Recursive algorithm Nqueens(k,n):

```plaintext
for i = 1 to n do {
    x[k]=i   // place a queen at position [k,i]
    if (k=n) print or check a possible solution in array x
    else Nqueens(k+1,n)
}
```
A path to a leaf is a position of 4 queens.

- Backtracking technique and branch-and-bound technique modify DFS/BFS to obtain solution with far fewer trials.
6.2 Backtracking

- Use modified DFS strategy

- Main idea
  
  - if a node cannot lead to a solution, we call the node nonpromising; otherwise, it is promising.
  
  - if a node is nonpromising, backtracking to the node’s parent and proceed the search on next child. This process is called pruning the state space tree.

- The subtree consisting of visited node is called pruned state space tree.

- The backtracking algorithm for n-queens problem

  - Each row has exactly one queen; No two queens in the same column; No two queens in the same diagonal

  - Algorithm promising(k,i) {
      // return true if k\textsuperscript{th} queen can be placed in k\textsuperscript{th} row and i\textsuperscript{th} column
      // otherwise, return false
      // x[] is a global array with column values have been set for k-1
      // queens in k-1 rows, i.e. x[a]=b \rightarrow a\textsuperscript{th} queen is in position [a,b]

      for j= 1 to k-1
        if ((x[j] = i) or (|x[j]-i| = |j-k|))
          return false
      return true
  }

  - Algorithm Nqueens(k,n) { // start with (1,n)
      for i = 1 to n do
        if promising(k,i) then {
          x[k]=i
          if (k=n) print a solution in array x
          else Nqueens(k+1,n)
        }
    }
A portion of the pruned state space tree for 4-queens problem:

- The total number of nodes in state space tree is $O(n^n)$

- Usually, we only count the number of nodes checked by the backtracking algorithm

For 8-queens problems
- Total number of nodes: 19,173,961
- Check rows & columns only: 40,320
- Check using promising(): 15,721
• The backtracking algorithm for sum of subset problem

• Assume input numbers are sorted, i.e. \( w_0 \leq w_1 \leq \ldots \leq w_{n-1} \)

• Let total\_so\_far be the total weight of a node after considering \( w_0, w_1, \ldots, w_{i-1} \)

A node is promising if

\[
( \text{total\_so\_far} + w_i \leq m \quad \text{OR} \quad \text{total\_so\_far} = m ) \quad \text{and} \\
\text{total\_so\_far} + \text{remain\_total} \geq m
\]

where remain\_total = \( w_i + w_{i+1} + \ldots + w_{n-1} \)

• Algorithm promising(i) {
    return ( (total\_so\_far + remain\_total) \geq m \quad \text{AND} \\
            (total\_so\_far = m \quad \text{OR} \quad \text{total\_so\_far} + w_i \leq m) )
    }

• Sum\_of\_subsets (i, total\_so\_far, remain\_total) {
    // initialize all entries in array x to 0 
    if (promising(i)) {
        if (total\_so\_far = m) print a solution in array x 
        else {
            x[i] = 1 // include \( w_i \) 
            Sum\_of\_subsets(i+1, total\_so\_far + w_i, remain\_total – w_i);
            x[i] = 0 // exclude \( w_i \) 
            Sum\_of\_subsets(i+1, total\_so\_far, remain\_total – w_i);
        }
    }
}

• To start the computation, call Sum\_of\_subsets(0, 0, \( \sum w_i \))
• The backtracking algorithm for sum of subset problem
  • Assume input numbers are sorted, i.e. \( w_1 \leq w_2 \leq \ldots \leq w_n \)
  • assume items i=1….n, global array \( x[1\ldots i] \) contains 0/1 values
  • already done with 1…i-1, check for i
  • total_so_far = can be computer from array \( x[1\ldots i-1] \)
    remain_total = \( w_i + w_{i+1} + \ldots + w_n \)

    promising(i) {
      return (total_so_far + remain_total \geq m \text{ AND}
              (total_so_far = m \text{ OR total_so_far} + w_i \leq m));
    }

• Start with Sum_of_subsets(1),
  • Sum_of_subsets (i) recursive function :

    if (i <=n) {
      if (promising(i)) {
        if (total_so_far = m) print a solution in array x
        else {
          x[i] = 1 // include \( w_i \)
          Sum_of_subsets(i+1)
          x[i] = 0 // exclude \( w_i \)
          Sum_of_subsets(i+1);
        }
      }
    } else // x[] contain values from 1..n
    Check sum of values in array \( x[1\ldots n] \), compare with m
- A portion of the pruned state space tree for Sum_of_subsets algorithm:

\[ n=6 \ \{5, 10, 12, 13, 15, 18\} \ m=30; \]

node value: (total so far, remain weight)

\[
\begin{align*}
0.73 & \quad x_1 \\
5,68 & \quad 0,68 \\
5,58 & \quad 10,58 \\
15,58 & \quad 15,46 \\
17,46 & \quad 27,46 \\
10,46 & \quad 15,33 \\
12,46 & \quad 5,33 \\
0,46 & \quad 15,33 \\
13,33 & \quad 0,33 \\
13,18 & \quad 30,0 \\
& \quad 30,18 \\
& \quad 20,18
\end{align*}
\]

\[ x_i \text{ level edges} = i-1^{th} \text{ object}!! \]
6.3 Branch-and-Bound

- Use modified DFS or BFS strategy
- Use to solve optimization problems
- Main idea
  - A bound at a node is the best value that could be obtained by expanding beyond the node
  - Compute a bound at node to determine whether a node is promising. If the bound value is no better than the current best solution, the node is nonpromising; otherwise, it is promising
  - If a node is not promising, then do not expand beyond the node
- BFS (DFS) is also called FIFO (LIFO)
- In general, BFS (DFS) Branch-and-Bound is also called FIFO (LIFO) Branch-and-bound.
- Improved BFS method is called Best-First Search
- Best-First Search Branch-and-Bound is also called Least-Cost Branch-and-Bound.
The FIFO Branch-and-Bound algorithm for 0/1 Knapsack problem

Recall: Given n objects \{o_1, o_2, \ldots, o_n\}, each has integer weight \(w_i\) and profit \(p_i\), and a knapsack has a capacity \(M\). If \(o_i\) is placed in knapsack then \(x_i=1\), otherwise \(x_i=0\)

Objective: To obtain a filling of the knapsack that maximize the total profit earned, i.e. maximize \(\sum p_i x_i\) such that \(\sum w_i x_i \leq M\)

- Arrange items in nonincreasing order of \(p_i/w_i\)

- Each node has:
  - level – the node’s level in the tree (at most n levels for n input objects)
  - weight – total weight of the items that have been included up to a node
  - profit – total profit of the items that have been included up to a node
  - \textbf{max profit} – current maximum profit among all visited nodes
  - \textbf{bound} – an upper bound of profit we could obtain by expanding node in the next level. i.e. \text{bound} = \text{current node profit} + \text{maximum profit from remaining objects such that the total weight is } M

Example:

Assume current profit = 20; current weight = 15; \(M=20\) and there are 3 remaining objects where \((w_i, p_i) = (4, 12) (3, 6) (2, 2)\)

Then, for total weight \(M = 15 + 4 + 1 = 20\), \text{bound} = 20 + 12 + 2 = 34

- A node is nonpromising if
  - weight > \(M\)  \(//\) set \text{bound} = 0
  - bound < \text{current max profit}
• Example: \( M = 16 \)

<table>
<thead>
<tr>
<th>( o_i )</th>
<th>( o_1 )</th>
<th>( o_2 )</th>
<th>( o_3 )</th>
<th>( o_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[i] )</td>
<td>40</td>
<td>30</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>( w[i] )</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>( p[i]/w[i] )</td>
<td>20</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

• The pruned state space tree produced using BFS Branch-and-Bound algorithm

Node value: profit, weight, bound
Algorithm

Input : an instance of 0/1 knapsack problem
Output : optimal maxprofit value

initialize(Q);
    v.level = 0; v.profit=0; v.weight=0;
    maxprofit = 0;
    insert(Q,v);

while ( !empty(Q)) {
    v = delete(Q);
    u.level = v.level++;

    // case 1 : try to include next item
    u.weight = v.weight + w[u.level];
    u.profit = v.profit + p[u.level];
    if (u.weight <= M && u.profit > maxprofit) maxprofit=u.profit;
    if(bound(u) > maxprofit) insert(Q,u);

    // case 2: do not include the next item
    u.weight = v.weight;
    u.profit = v.profit;
    if(bound(u) > maxprofit) insert(Q,u);
}

bound(u) { // return a boundProfit
    if (u.weight >= M) return 0;
    boundProfit = u.profit;
    totalWeight = u.weight;
    // grab as many remaining objects as possible
    // i.e. object u.level +1, u.level+2, ..., n
    // update boundProfit and totalWeight
    // until all objects have been included or
    // the totalWeight = M
    // note : last object may be partially included
}
- The Best-First Search Branch-and-Bound algorithm for 0/1 Knapsack problem

- Improve the FIFO by first expanding the promising & unexpanded node with the best bound.

- The pruned state space tree produced using Best-First Search Branch-and-Bound algorithm

Node value : profit, weight, bound
Algorithm

Input : an instance of 0/1 knapsack problem
Output : optimal maxprofit value

Note: need to add “bound” field into a node structure
Use priority queue PQ instead of FIFO Q

initialize(PQ);
v.level = 0; v.profit=0; v.weight=0;
maxprofit = 0;
v.bound = bound(v);
insert(PQ,v);

while ( !empty(PQ)) {

    v = delete_max(PQ);

    if (v.bound > maxprofit) {

        u.level = v.level++;

        // case 1 : try to include next item
        u.weight = v.weight + w[u.level];
        u.profit = v.profit + p[u.level];
        if (u.weight <= M && u.profit > maxprofit) maxprofit=u.profit;
        u.bound = bound(u);
        if(u.bound > maxprofit) insert(PQ,u);

        // case 2: do not include the next item
        u.weight = v.weight;
        u.profit = v.profit;
        u.bound = bound(u);
        if (u.bound > maxprofit) insert(PQ,u);

    }

}
• Branch-and-Bound Exercise : The Traveling Salesperson problem

• Backtracking Exercise : m-Coloring problem

Determine all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color

• Research problem : The 15-puzzle problem

Problem : Given an initial arrangement of 15 numbered titles on a 4 by 4 square frame.

Objective : Transform the initial arrangement into the goal arrangement (see below) through a series of legal moves.

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>4</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

→

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>