Greedy Method

Objective:

General approach:

- Given a set of n inputs.
- Find a subset, called feasible solution, of the n inputs subject to some constraints, and satisfying a given objective function.
- If the objective function is maximized or minimized, the feasible solution is optimal.
- It is a locally optimal method.

Algorithm:

Step 1: Choose an input from the input set, based on some criterion. If no more input exit.

Step 2: Check whether the chosen input yields to a feasible solution. If no, discard the input and goto step 1.

Step 3: Include the input into the solution vector and update the objective function. Goto step 1.
Optimal merge patterns

Introduction:

• Merge two files each has n & m elements, respectively:
  \[ \Rightarrow \text{takes } O(n+m). \]

• Given n files
  What's the minimum time needed to merge all n files?

• Example:
  \[ (F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30). \]

  \[ M_1 = F_1 \& F_2 \Rightarrow 20 + 30 = 50 \]
  \[ M_2 = M_1 \& F_3 \Rightarrow 50 + 10 = 60 \]
  \[ M_3 = M_2 \& F_4 \Rightarrow 60 + 5 = 65 \]
  \[ M_4 = M_3 \& F_5 \Rightarrow 65 + 30 = 95 \]

  \[ 270 \]

• **Optimal merge pattern**: Greedy method.

  Sort the list of files:

  \[ (5, 10, 20, 30, 30) = (F_4, F_3, F_1, F_2, F_5) \]
Merge the first two files:

$$ (5, 10, 20, 30, 30) \Rightarrow (15, 20, 30, 30) $$

Merge the next two files:

$$ (15, 20, 30, 30) \Rightarrow (30, 30, 35) $$

Merge the next two files:

$$ (30, 30, 35) \Rightarrow (35, 60) $$

Merge the last two files:

$$ (35, 60) \Rightarrow (95) $$

Total time: $15 + 35 + 60 + 95 = 205$

⇒ This is called a 2-way merge pattern.

- **Problem:**

  - Given n sorted files
  - Merge n files in a minimum amount of time.

- **Algorithm:**

  - We associate with each file a node
**Example:**

Initial

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
</tbody>
</table>

| 5 | 10 |

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>30</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- **Algorithm:**

- Least (L): find a tree in L whose root has the smallest weight.
- Function : Tree (L,n).

  Integer i;
  Begin
  For i=1 to n-1 do
    Get node (T) /* create a node pointed by T */
    Left child (T)= Least (L) /* first smallest */
    Right child (T)= Least (L) /* second smallest */
    Weight (T) = weight (left child (T))
    + weight (right child (T))
    Insert (L,T); /* insert new tree with root T in L */
  End for
  Return (Least (L)) /* tree left in L */
  End.
• **Analysis:**

\[ T = O(n-1) \times \max(O(\text{Least}), O(\text{Insert})). \]

- **Case 1**  
  \( L \) is not sorted.  
  \( O(\text{Least}) = O(n). \)  
  \( O(\text{Insert}) = O(1). \)  
  \[ \Rightarrow T = O(n^2). \]

- **Case 2**  
  \( L \) is sorted.  
  
  **Case 2.1**  
  \( O(\text{Least}) = O(1) \)  
  \( O(\text{Insert}) = O(n) \)  
  \[ \Rightarrow T = O(n^2) \]

  **Case 2.2**  
  \( L \) is represented as a min-heap. Value in the root is \( \leq \) the values of its children.  
  \( O(\text{Least}) = O(1) \)  
  \( O(\text{Insert}) = O(\log n) \)  
  \[ \Rightarrow T = O(n \log n). \]
Knapsack problem

Problem:
- input:
  - n objects.
  - each object i has a weight \( w_i \) and a profit \( p_i \)
  - Knapsack: \( M \)

- output:
  - Fill up the Knapsack s.t. the total profit is maximized.
  - Feasible solution: \((x_1, \ldots, x_n)\).

Formally,
- Let \( x_i \) be the fraction of object i placed in the Knapsack, \( 0 \leq x_i \leq 1 \). For \( 1 \leq i \leq n \).

- Then:
  \[
P = \sum_{1 \leq i \leq n} p_i x_i
  \]
  And \( \sum w_i x_i \leq M \)

Assumptions:
- \( \sum_{i=1}^{n} w_i > M \); not all \( x_i = 1 \).
- \( \sum_{1 \leq i \leq n} w_i x_i = M \)
Example:

- 3 objects (n=3).
- \((w_1,w_2,w_3)=(18,15,10)\)
- \((p_1,p_2,p_3)=(25,24,15)\)
- \(M=20\)

Largest-profit strategy: (Greedy method)

- Pick always the object with largest profit.
- If the weight of the object exceeds the remaining Knapsack capacity, take a fraction of the object to fill up the Knapsack.

Example:

- \(P=0, C=M=20\)  /* remaining capacity */
- Put object 1 in the Knapsack.
  - \(P=25\) Since \(w_1 < M\) then \(x_1=1\)
  - \(C=M-18=20-18=2\)
- Pick object 2
  - Since \(C < w_2\) then \(x_2 = C/w_2 = 2/15\).
  - \(P=25+2/15*24 = 25+3.2 = 28.2\)
- Since the Knapsack is full then \(x_3=0\).
- The feasible solution is \((1, 2/15, 0)\).
Smallest-weight strategy:

✓ be greedy in capacity: do not want to fill the knapsack quickly.

✓ Pick the object with the smallest weight.

✓ If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.

Example:

✓ cu=M=20

✓ Pick object 3
   Since \( w_3 < cu \) then \( x_3=1 \)
   \( \text{P} = 15 \quad \text{cu} = 20 - 10 = 10 \quad x_3 = 1 \)

✓ Pick object 2
   Since \( w_2 > cu \) then \( x_2 = 10/15 = 2/3 \)
   \( \text{P} = 15 + 2/3.24 \)
   \( = 15 + 16 = 31 \quad \text{cu} = 0. \)

✓ Since \( \text{cu}=0 \) then \( x_1=0 \)

✓ Feasible solution: \( (0,2/3,1) \quad \text{p}=31 \).
Largest profit-weight ratio strategy:

- Order profit-weight ratios of all objects.
- \( \frac{p_i}{w_i} \geq \frac{(p_i+1)}{(w_i+1)} \) for \( 1 \leq i \leq n-1 \)
- Pick the object with the largest \( \frac{p}{w} \)
- If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.

Example:

\[
\begin{align*}
\frac{p_1}{w_1} &= \frac{25}{18} = 1.389 \\
\frac{p_2}{w_2} &= \frac{24}{15} = 1.6 \\
\frac{p_3}{w_3} &= \frac{15}{10} = 1.5 \\
\end{align*}
\]

\( \Rightarrow \frac{p_2}{w_2} \geq \frac{p_2}{w_2} \geq \frac{p_3}{w_3} \)

\( C_u = 20; p = 0 \)

- Pick object 2
  - Since \( c_u \geq w_2 \) then \( x_2 = 1 \)
  - \( c_u = 20 - 15 = 5 \) and \( p = 24 \)

- Pick object 3
  - Since \( c_u < w_3 \) then \( x_3 = \frac{c_u}{w_3} = \frac{5}{10} = 0.5 \)
  - \( c_u = 0 \) and \( p = 24 + 1/2.15 = 24 + 7.5 = 31.5 \)

- Feasible solution \((0,1,1/2)\) \( p = 31.5 \)
Minimum Spanning Tree.

Definition:
Let $G=(V,E)$ be an undirected connected graph. $T=(V,E')$ is a spanning tree iff $T$ is a tree.

Example:

Definition:
- If each edge of $E$ has a weight, $G$ is called a weighted graph.

Problem:
- Given an undirected, connected, weighted graph $G=(V,E)$.
- We wish to find an acyclic subset $T \subseteq E$ that connects all the vertices and whose total weight:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

is minimized.

Where $w(u,v)$ is the weight of edge $(u,v)$.
- $T$ is called a minimum spanning tree of $G$. 
Solution:

- Using greedy method.
- Two algorithms:
  - Prim's algorithm.
  - Kruskal's algorithm.

Approach:

- The tree is built edge by edge.
- Let T be the set of edges selected so far.
- Each time a decision is made:
  * Include an edge e to T s.t.:
    Cost (T) + w (e) is minimized, and
    T ∪ {e} does not create a cycle.

Prim's algorithm:

- T forms a single tree.
- The edge e added to T is always least-weight edge connecting the tree, T, to a vertex not in the tree

Implementation:

- To choose the next edge to be included in T, NEAR (i:n) array is used.
\[ V(T) \]

\[ V(G) - V(T) \]

NEAR(I) = 0 
\[ i \in V(T) \]

NEAR(I) = \( v \) s.t. \( i \in V(T) \), \( v \in V(T) \) and cost(i, v) is min among all choices for NEAR(i).

NEAR(I) = \( p \) s.t. cost(I, p) is min cost(I, w) where \( p \leq w \leq m \)
Procedure PRIM (G, Cost, mincost)
    /* Let n be # of vertices */
    Integer NEAR (1:n);
    Integer u,w,p,I;
1. Begin
2. Choose an arbitrary vertex v_o.
3. mincost=o; NEAR (v_o)=o
4. For each vertex w ≠ v_o do
5.    NEAR (w)=v_o;
6.   End for
7. For I=1 to n-1 do /* fin n-1 edges of T */
8.    Choose a vertex w s.t.
9.    cost(w,NEAR(w ))= min (cost (u, NEAR(u)) )
10.   where NEAR (u) ≠ o
11.   mincost = mincost+ cost (w, NEAR(w));
12.   NEAR (w)=o
13.   For each vertex p do
14.      if  NEAR(p) ≠ o & cost (p, NEAR(p) ) > cost (p,w)
15.      then  NEAR (p)= w;
16.      endif
17.   end for
18.   End for
19.  End.

• Analysis:

✓ The for loop between 4 and 6 takes O(n).
✓ Lines between 8 and 10 take O(n)
✓ The For loop between 13 and 17 takes O(n)
✓ Finally, the main For loop that starts at line 7 takes O(n)
✓ the overall algorithm takes O(n²).
Example:

- Let's start form v=1
Kruskal's algorithm

Problem:

- T form a forest.
- The edge e added to T is always least-weight edge in the graph that connects two distinct trees of T.
- At the end of the algorithm T becomes a single tree.

Example:
Procedure kruskal (G, cost).
Begin
T: forest
T= ∅
while |T| ≤ n-1 & E≠∅ do
    choose an edge (v,w)∈ E of least weight
    delete (v,w) from E
    If (v,w) does not create a cycle in T then
        add (v,w) ∈ o T
    else
        discard (v,w);
    endif
end while.

Implementation:

• Choose the edge with the smallest weight:

 ✓ Use min-heap:
    - Get the min & read just the heap takes O (log e).
    - Construct the heap takes O (e).

• Be sure that the chosen edge does not create a cycle in the so far built forest, T:

  ✓ Use union-find:
    Once (u,v) is selected.
    Check if Find (u) ≠ Find (v).
• Summary:
  ✓ Min-heap on edges.
  ✓ Union-find on vertices.

• Time complexity O (e log e).
Single Source Shortest Paths.

Requirements:

- Given a weighted digraph $G = (V, E)$ where the weights are $> 0$.
- A source vertex, $v_o \in V$.
- Find the shortest path from $v_o$ to all other nodes in $G$.
- Shortest paths are generated in increasing order: 1, 2, 3, ....

Algorithm Description: Dijkstra

- $S$: Set of vertices (including $v_o$) whose final shortest paths from the source $v_o$ have already been determined.
- For each node $w \in V - S$,
  - $\text{Dist}(w)$: the length of the shortest path starting from $v_o$ going through only vertices which are in $S$ and ending at $w$.
- The next path is generated as follows:
  - It's the path of a vertex $u$ which has $\text{Dist}(u)$ minimum among all vertices in $V - S$
  - Put $u$ in $S$.
- $\text{Dist}(w)$ for $w$ in $V - S$ may be decreased going through $u$. 
Compare Dist (u) + cost (u,w) with Dist (w).

\textbf{Algorithm:}

Procedure SSSP \((v_o, \text{cost, } n)\)

Array \(S(1:n)\);

Begin

\begin{verbatim}
/* initialization*/
For i=1 to n do
    S(i)=o, Dist (i)= cost (v_o,i)
End for.
S(v_o)=1, Dist (v_o)=0;
For i=1 to n-1 do.
    Choose u s.t. Dist (u)= \( \min_{S(w)=0} \{ \text{Dist (w)} \} \)
    S(u)=1;
    For all w with S(w)=o do.
        Dist (w)= min (Dist (w), Dist (u) + Cost (u,w) )
    End for.
End for.
end.
\end{verbatim}

\checkmark Time complexity: \( O(n^2) \).