Unit 2: Greedy Strategy

Reena Pagare

June 2016
Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible Rs note and coins.

- At each step, take the largest possible note or coin that does not overshoot.
  - Example: To make Rs 61, you can choose:
    - a Rs 50 note
    - a Rs 1 coin, to make Rs 51
    - a Rs 10 note or Rs10 coin, to make Rs 61
Greedy Algorithm Paradigm

- Characteristics of greedy algorithms:
  - make a sequence of choices
  - each choice is the one that seems best so far, only depends on what's been done so far
  - choice produces a smaller problem to be solved
- In order for greedy heuristic to solve the problem, it must be that the optimal solution to the big problem contains optimal solutions to sub problems
Elements of Greedy Strategy

- Determine the optimal substructure of the problem
- Develop a recursive solution
- Show that if we make the greedy choice then only one subproblem remains
- Prove that it is always sake o make the greedy choice
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative algorithm
Designing a Greedy Algorithm

- Cast the problem so that we make a greedy (locally optimal) choice and are left with one subproblem.
- Prove there is always a (globally) optimal solution to the original problem that makes the greedy choice.
- Show that the choice together with an optimal solution to the subproblem gives an optimal solution to the original problem.
An optimization problem is one in which you want to find, not just a solution, but the best solution.

A “greedy algorithm” sometimes works well for optimization problems.

A greedy algorithm works in phases. At each phase:

- You take the best you can get right now, without regard for future consequences.
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.
A failure of the greedy algorithm

- In some (fictional) monetary system, “coins” come in Rs 1 coin, Rs 7 coin, and Rs 10 coins.

- Using a greedy algorithm to count out Rs 15, you would get:
  - A Rs 10 coin
  - Five Rs 1 coins, for a total of Rs 15
  - This requires six coins

- A better solution would be to use two Rs 7 coin pieces and one Rs 1 coin piece
  - This only requires three coins

- The greedy algorithm results in a solution, but not in an optimal solution.
Terms in Greedy Methods

- **N inputs** – \{1, 7, 10\} rs or coins
- **Subset** that satisfies some constraints \{10,1\} or \{7,7,1\}
- This is **feasible solution**
- Minimizes or Maximizes – **Objective function** \{Total is Rs 15 ; min no of coins or notes\}
- A feasible solution that does satisfies the objective function is called an **optimal solution**. --- \{7,7,1\}
- **Feasible solution need not be optimal.**
procedure $\text{GREEDY}(A, n)$

// $A(1:n)$ contains the $n$ inputs/

solution $\leftarrow \phi$  //initialize the solution to empty/

for $i \leftarrow 1$ to $n$ do

$x \leftarrow \text{SELECT}(A)$

if $\text{FEASIBLE}(\text{solution}, x)$

then $\text{solution} \leftarrow \text{UNION}(\text{solution}, x)$

endif

repeat

return (solution)

end $\text{GREEDY}$

Algorithm 4.1  Greedy method control abstraction
Greedy Strategy : Control Abstraction

- **SELECT**: Selects an input from A, removes it and assigns value to `x`.
- **FEASIBLE**: Boolean-valued function which determines if it can be included into the solution vector.
- **UNION**: Combines `x`th solution and updates the objective function.
- **Procedure GREEDY**: Algorithm.
Some Greedy Algorithms

- fractional knapsack algorithm
- Huffman codes
- Kruskal's MST algorithm
- Prim's MST algorithm
- Dijkstra's SSSP algorithm
- Activity Selection
- Scheduling Jobs/Tasks
- ....
Example

- Given: A set $S$ of $n$ items, with each item $i$ having
  - $b_i$ - a positive benefit
  - $w_i$ - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most $W$.

<table>
<thead>
<tr>
<th>Items</th>
<th>Weight</th>
<th>Benefit</th>
<th>Value ($ per ml)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 ml</td>
<td>$12</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8 ml</td>
<td>$32</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2 ml</td>
<td>$40</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>6 ml</td>
<td>$30</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1 ml</td>
<td>$50</td>
<td>50</td>
</tr>
</tbody>
</table>

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

“knapsack”
The Fractional Knapsack Problem

- Given: A set $S$ of $n$ items, with each item $i$ having
  - $b_i$ - a positive benefit
  - $w_i$ - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most $W$.
- If we are allowed to take fractional amounts, then this is the fractional knapsack problem.
  - In this case, we let $x_i$ denote the amount we take of item $i$

- Objective: maximize $\sum_{i \in S} b_i \left( \frac{x_i}{w_i} \right)$
- Constraint: $\sum_{i \in S} x_i \leq W$
Knapsack Problem

Now, let us try to apply the greedy method to solve a more complex problem. This problem is the knapsack problem. We are given $n$ objects and a knapsack. Object $i$ has a weight $w_i$ and the knapsack has a capacity $M$. If a fraction $x_i$, $0 \leq x_i \leq 1$, of object $i$ is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is $M$, we require the total weight of all chosen objects to be at most $M$. Formally, the problem may be stated as:

$$\text{maximize } \sum_{1 \leq i \leq n} p_i x_i \quad (4.1)$$

$$\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq M \quad (4.2)$$

$$\text{and } 0 \leq x_i \leq 1, \ 1 \leq i \leq n \quad (4.3)$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximum.
Greedy Algorithm

procedure GREEDY__KNAPSACK(P, W, M, X, n)
    //P(1:n) and W(1:n) contain the profits and weights respectively of the n//
    //objects ordered so that P(i)/W(i) ≥ P(i + 1)/W(i + 1). M is the//
    //knapsack size and X(1:n) is the solution vector//
    real P(1:n), W(1:n), X(1:n), M. cu;
    integer i, n;
    X ← 0    //initialize solution to zero//
    cu ← M    //cu = remaining knapsack capacity//
    for i ← 1 to n do
        if W(i) > cu then exit endif
        X(i) ← 1
        cu ← cu − W(i)
    repeat
    if i ≤ n then X(i) ← cu/W(i) endif
end GREEDY__KNAPSACK

Algorithm 4.3 Algorithm for greedy strategies for the knapsack problem
The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time: $O(n \log n)$. Why?

- Correctness: Suppose there is a better solution
  - there is an item $i$ with higher value than a chosen item $j$, but $x_i < w_i$, $x_j > 0$ and $v_i < v_j$
  - If we substitute some $i$ with $j$, we get a better solution
  - How much of $i$: $\min\{w_i - x_i, x_j\}$
  - Thus, there is no better solution than the greedy one

Algorithm $\text{fractionalKnapsack}(S, W)$

**Input:** set $S$ of items w/ benefit $b_i$ and weight $w_i$; max. weight $W$

**Output:** amount $x_i$ of each item $i$ to maximize benefit w/ weight at most $W$

```plaintext
for each item $i$ in $S$
    $x_i \leftarrow 0$
    $v_i \leftarrow b_i / w_i$ (value)
    $w \leftarrow 0$ (total weight)
while $w < W$
    remove item $i$ w/ highest $v_i$
    $x_i \leftarrow \min\{w_i, W - w\}$
    $w \leftarrow w + \min\{w_i, W - w\}$
```
Example 4.2 Consider the following instance of the knapsack problem: 
\( n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15) \) and \( (w_1, w_2, w_3) = (18, 15, 10) \). Four feasible solutions are:

<table>
<thead>
<tr>
<th>((x_1, x_2, x_3))</th>
<th>(\Sigma w_i x_i)</th>
<th>(\Sigma p_i x_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i) ((1/2, 1/3, 1/4))</td>
<td>16.5</td>
<td>24.25</td>
</tr>
<tr>
<td>ii) ((1, 2/15, 0))</td>
<td>20</td>
<td>28.2</td>
</tr>
<tr>
<td>iii) ((0, 2/3, 1))</td>
<td>20</td>
<td>31</td>
</tr>
<tr>
<td>iv) ((0, 1, 1/2))</td>
<td>20</td>
<td>31.5</td>
</tr>
</tbody>
</table>
Different Methods

- Method 1: Max Profit (solution 2)
- Method 2: Minimum weight (solution 3)
- Method 3: Min profit/weight ratio (solution 4)
Task Scheduling

- Given: a set \( T \) of \( n \) tasks, each having:
  - A start time, \( s_i \)
  - A finish time, \( f_i \) (where \( s_i < f_i \))
- Goal: Perform all the tasks using a minimum number of “machines.”
Task Scheduling Algorithm

- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: $O(n \log n)$. Why?
- Correctness: Suppose there is a better schedule.
  - We can use $k - 1$ machines
  - The algorithm uses $k$
  - Let $i$ be first task scheduled on machine $k$
  - Machine $i$ must conflict with $k - 1$ other tasks
  - But that means there is no non-conflicting schedule using $k - 1$ machines

Algorithm taskSchedule($T$)

Input: set $T$ of tasks w/ start time $s_i$ and finish time $f_i$

Output: non-conflicting schedule with minimum number of machines

$m \leftarrow 0$ {no. of machines}

while $T$ is not empty
  remove task $i$ w/ smallest $s_i$
  if there’s a machine $j$ for $i$ then
    schedule $i$ on machine $j$
  else
    $m \leftarrow m + 1$
    schedule $i$ on machine $m$
Example

- Given: a set $T$ of $n$ tasks, each having:
  - A start time, $s_i$
  - A finish time, $f_i$ (where $s_i < f_i$)
- $[1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8]$ (ordered by start)
- Goal: Perform all tasks on min. number of machines

[Diagram showing machines and task scheduling]
A scheduling problem

- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available

Time to completion: $18 + 11 + 6 = 35$ minutes

This solution isn’t bad, but we might be able to do better
Another approach

- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

<table>
<thead>
<tr>
<th>P1</th>
<th>3</th>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>5</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>P3</td>
<td>6</td>
<td>14</td>
<td>20</td>
</tr>
</tbody>
</table>

- That wasn’t such a good idea; time to completion is now $6 + 14 + 20 = 40$ minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum
An optimum solution

- Better solutions do exist:

  - This solution is clearly optimal (why?)
  - Clearly, there are other optimal solutions (why?)
  - How do we find such a solution?
    - One way: Try all possible assignments of jobs to processors
    - Unfortunately, this approach can take exponential time
Activity-selection Problem

- **Input:** Set $S$ of $n$ activities, $a_1, a_2, \ldots, a_n$.
  - $s_i =$ start time of activity $i$.
  - $f_i =$ finish time of activity $i$.

- **Output:** Subset $A$ of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don’t overlap.

Example:

Activities in each line are compatible.
Suppose an optimal solution includes activity $a_k$.

- This generates two subproblems.
- Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before $a_k$ starts (compatible with $a_k$).

Assume activities are sorted by finishing times.

- $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after $a_k$ finishes.
- The solutions to the two subproblems must be optimal.
Recursive Solution

- Let $S_{ij}$ = subset of activities in $S$ that start after $a_i$ finishes and finish before $a_j$ starts.
- **Subproblems**: Selecting maximum number of mutually compatible activities from $S_{ij}$.
- Let $c[i, j] = $ size of maximum-size subset of mutually compatible activities in $S_{ij}$.

**Recursive Solution:**

$$c[i, j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset \\
\max \{c[i, k] + c[k, j] + 1 \} & \text{if } S_{ij} \neq \emptyset 
\end{cases}$$
The problem also exhibits the **greedy-choice property**.

- There is an optimal solution to the subproblem $S_{ij}$, that includes the activity with the smallest finish time in set $S_{ij}$.
- Hence, there is an optimal solution to $S$ that includes $a_1$.

Therefore, **make this greedy choice** without solving subproblems first and evaluating them.

Solve the subproblem that ensues as a result of making this greedy choice.

Combine the greedy choice and the solution to the subproblem.
Recursive Algorithm

Recursive-Activity-Selector \((s, f, i, j)\)

1. \(m \leftarrow i+1\)
2. while \(m < j\) and \(s_m < f_i\)
3. do \(m \leftarrow m+1\)
4. if \(m < j\)
5. then return \(\{a_m\} \cup \) Recursive-Activity-Selector\((s, f, m, j)\)
6. else return \(\emptyset\)

Initial Call: Recursive-Activity-Selector \((s, f, 0, n+1)\)

Complexity: \(\Theta(n)\)

Straightforward to convert the algorithm to an iterative one.
Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there’s always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.

- **Example:** Sorting activities by finish time.
Elements of Greedy Algorithms

- **Greedy-choice Property.**
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- **Optimal Substructure.**
For this example the subset \{a3, a9, a11\} consists of mutually compatible activities. It is not a maximum subset, however, since the subset \{a1, a4, a8, a11\} is larger. In fact \{a1, a4, a8, a11\} is a largest subset of mutually compatible activities; another largest subset is \{a2, a4, a9, a11\}.
Iterative Greedy Activity Selection

\textsc{Greedy-Activity-Selector}(s, f)

1. \hspace{1em} n = s.\text{length}
2. \hspace{1em} A = \{a_1\}
3. \hspace{1em} k = 1
4. \hspace{1em} \textbf{for} m = 2 \textbf{to} n
5. \hspace{1em} \hspace{1em} \textbf{if} s[m] \geq f[k]
6. \hspace{1em} \hspace{1em} \hspace{1em} A = A \cup \{a_m\}
7. \hspace{1em} \hspace{1em} k = m
8. \hspace{1em} \hspace{1em} \textbf{return} A
Huffman encoding

- The Huffman encoding algorithm is a greedy algorithm
- You always pick the two smallest numbers to combine

Average bits/char:
0.22*2 + 0.12*3 + 0.24*2 + 0.06*4 + 0.27*2 + 0.09*4 = 2.42

The Huffman algorithm finds an optimal solution
Minimum spanning tree

- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
  - Start by picking any node and adding it to the tree
  - Repeatedly: Pick any least-cost edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
  - Stop when all nodes have been added to the tree

- The result is a least-cost (3+3+2+2+2=12) spanning tree
- If you think some other edge should be in the spanning tree:
  - Try adding that edge
  - Note that the edge is part of a cycle
  - To break the cycle, you must remove the edge with the greatest cost
    - This will be the edge you just added
Traveling salesman

- A salesman must visit every city (starting from city A), and wants to cover the least possible distance
  - He can revisit a city (and reuse a road) if necessary
- He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is

From A he goes to B
- From B he goes to D
- This is not going to result in a shortest path!
- The best result he can get now will be ABDBCE, at a cost of 16
- An actual least-cost path from A is ADBCE, at a cost of 14
A greedy algorithm typically makes (approximately) \( n \) choices for a problem of size \( n \)

(The first or last choice may be forced)

Hence the expected running time is:
\[ O(n \times O(\text{choice}(n))) \], where \( \text{choice}(n) \) is making a choice among \( n \) objects

- Counting: Must find largest useable coin from among \( k \) sizes of coin (\( k \) is a constant), an \( O(k) = O(1) \) operation;
  - Therefore, coin counting is \( n \)

- Huffman: Must sort \( n \) values before making \( n \) choices
  - Therefore, Huffman is \( O(n \log n) + O(n) = O(n \log n) \)

- Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is \( O(n \log n) \) overall
  - Therefore, MST is \( O(n \log n) + O(n) = O(n \log n) \)
Other greedy algorithms

- Dijkstra’s algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node

- Kruskal’s algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge

- Prim’s algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree
Dijkstra’s shortest-path algorithm

- Dijkstra’s algorithm finds the shortest paths from a given node to all other nodes in a graph
  
  - Initially,
    
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  
  - Repeatedly (until all nodes are known),
    
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
      
      - If so, also reset the predecessor of the new node
Analysis of Dijkstra’s algorithm I

- Assume that the *average* out-degree of a node is some constant $k$
  - Initially,
    - Mark the given node as *known* (path length is zero)
      - This takes $O(1)$ (constant) time
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
      - If each node refers to a list of $k$ adjacent node/edge pairs, this takes $O(k) = O(1)$ time, that is, constant time
      - Notice that this operation takes *longer* if we have to extract a list of names from a hash table
Analysis of Dijkstra’s algorithm II

- Repeatedly (until all nodes are known), (n times)
  - Find an unknown node containing the smallest distance
    - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes $k \times O(\log n)$ time each time a new node is marked “known” (and this happens n times)
  - Mark the new node as known -- $O(1)$ time
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - If so, also reset the predecessor of the new node
    - There are $k$ adjacent nodes (on average), operation requires constant time at each, therefore $O(k)$ (constant) time
  - Combining all the parts, we get:
    \[ O(1) + n \times (k \times O(\log n) + O(k)) \]
    that is, $O(nk \log n)$ time
There are $n$ white dots and $n$ black dots, equally spaced, in a line.
You want to connect each white dot with some one black dot, with a minimum total length of “wire”.

Example:

Total wire length above is $1 + 1 + 1 + 5 = 8$

Do you see a greedy algorithm for doing this?
Does the algorithm guarantee an optimal solution?

- Can you prove it?
- Can you find a counterexample?
Collecting coins

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect all the coins using the minimum number of robots

Example:

Do you see a greedy algorithm for doing this?

Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?
References

Slides adapted from
- Len/Devi Fall 2003
- Prof. Evdokia Nikolova, Spring 2013
- Goodrich Tamassia 2004

Books
- Horowitz Sahani, “Fundamentals of Algorithms”
- Cormen, “Introduction to Algorithms”