Greedy Algorithms

Josh Wilcox (jw14g24)

March 17, 2025

Table of Contents

Intro

The greedy strategy is a **design** paradigm

- They make **locally** optimal choices at each steap
- The *hope* is that such choices lead to a **globally optimal** solution
  - This is not always the case (in fact it rarely is)
  - But when it works, it is **extremely efficient**
- Dijkstra's, Prim's, and Kruskals are examples of greedy algorithms we have already seen

Job Scheduling

- Assuming we have n jobs, each with weight w_i and length l_i
- Each job shares the same resource
  - Must be run **sequentially**
- If we run the jobs in order $1, 2, 3, \ldots$ the completion time would be:

$$c_i = \sum_{k=1}^{i} l_k$$

- Our goal is to **minimize**:

$$f = \sum_{k=1}^{n} w_k \cdot c_k$$

Special Cases

2 Job Scheduling
- Special Cases
- Shortest Job First
- Equal Weight
- General Case

Shortest Job First

2. Job Scheduling
   - Special Cases
   - Shortest Job First
   - Equal Weight
   - General Case

Say we have:

$$f = w(l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + \cdots + l_n))$$
$$= w(n \cdot l_1 + (n-1) \cdot l_2 + (n-2) \cdot l_3 + \ldots \cdot + 1 \cdot l_n)$$

- f will be minimized if we choose $l_1 \leq l_2 \leq \ldots \leq l_n$
- Known as **shortest job first**

Equal Weight

2. Job Scheduling
   - Special Cases
   - Shortest Job First
   - Equal Weight
   - General Case

If all jobs have equal weights (i.e. $w_1 = w_2 = \cdots = w_n = w$), the cost function becomes

$$\ell = w \cdot \Big(l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \cdots + (l_1 + l_2 + \cdots + l_n)\Big)$$

or equivalently,

$$\ell = w \cdot \big(1 \cdot l_1 + 2 \cdot l_2 + 3 \cdot l_3 + \cdots + n \cdot l_n\big).$$

In this special case the objective is to minimize the sum of the completion times by scheduling jobs with shorter execution times first, which is exactly the **Shortest Job First** (SJF) strategy.

By arranging the jobs so that $l_1 \leq l_2 \leq \ldots \leq l_n$, we ensure that each job waits the shortest possible time for its predecessors to complete. Hence, with equal weights the scheduling problem reduces to minimizing the processing delays, which is the core idea behind the SJF approach.

Equal Length

- Consider the case when all jobs have the same length, i.e. $l_1 = l_2 = \cdots = l_n = l$.
- The cost function becomes:

$$f = l \cdot \Big(w_1 + (w_1 + w_2) + \cdots + (w_1 + w_2 + \cdots + w_n)\Big),$$

  which simplifies to:

$$f = l \cdot \big(n \cdot w_1 + (n-1) \cdot w_2 + \cdots + 1 \cdot w_n\big).$$

- To minimize f, jobs should be scheduled in decreasing order of weight (i.e., heaviest first).
- This ensures that heavier jobs contribute less to the cumulative waiting time.

General Case

It can be shown that choosing, among the remaining tasks, the one with the largest ratio $\frac{w_i}{l_i}$ first leads to the optimal solution. This approach follows a greedy strategy, where we iteratively select the best local option to reach a globally optimal solution.

We are given an array A of n pairs (w_i, l_i), where:

- w_i represents the weight (or benefit) of task i.
- l_i represents the length (or cost) of task i.

The goal is to process the tasks in an optimal order based on their benefit-to-cost ratio.

Require: Array A of n pairs (w_i, l_i)
Ensure: Tasks are executed in optimal order
  Initialize an empty list B
  for $i \leftarrow 1$ to n **do**
    $B[i] \leftarrow \left(\frac{w_i}{l_i}, i\right)$ ▷ Compute benefit-to-cost ratio
  end for
  Sort B in descending order by the first component (ratio) ▷ $O(n \log n)$ sorting step
  for all $(w/l, k) \in B$ **do**
    run(k) ▷ Execute task in optimal order
  end for

- Computing the ratios takes $O(n)$ time.
- Sorting the tasks by their ratio takes $O(n \log n)$ time.
- Executing the tasks takes $O(n)$ time.

Thus, the overall complexity of the algorithm is $O(n \log n)$. This makes it efficient for large values of n compared to an exhaustive search.

Interval Scheduling

- Involves choosing a non-overlapping subset of intervals such that the total number of intervals is maximum
- For a greedy solution to this problem we can use the following options
  - Shortest interval first
  - Interval with the smallest starting time
  - Interval with smallest number of overlaps
  - Etc...

Greedy Solution

- Consists of choosing the next compatible interval with **the shortest finishing time**
- Build a min-heap based on *finishing times*

Let I be the set of intervals and T the desired solution

Algorithm Hikmat Farhat Greedy Strategy / COMP 1201 Algorithmics 11/27

$Q \leftarrow I$
$T \leftarrow \varnothing$
$last \leftarrow -1$
while $Q \neq \varnothing$ **do**
    $(s, f) \leftarrow \text{EXTRACT-MIN}(Q)$
    if $s \geq last$ **then**
        $T \leftarrow T \cup \{(s, f)\}$
        $last \leftarrow f$
    end if
end while

Fractional Knapsack

- We have a "bag" with a limited total capacity C
- Let x be a fraction of item i that is used
- We want to maximze:

$$\sum_{i=1}^{n} x_i \cdot w_i \leq C$$

Greedy Solution

4. Fractional Knapsack
   - Greedy Solution

- First, sort all items in descending order based on their **value-to-weight ratio** $\frac{value}{weight}$. Alternatively, use a *max-heap* for this purpose.
- Iteratively extract the item with the highest ratio from the sorted list or max-heap.
- Check if the entire selected item can fit in the remaining capacity of the knapsack:
  - If it fits, add the whole item and decrease the remaining capacity accordingly.
  - If it does not fit, add only a fraction of the item such that the knapsack is filled exactly.
- Continue this process until the knapsack is full or there are no items left.
- This greedy approach maximizes the total value by always choosing the current best option.
- It ensures an optimal solution for the Fractional Knapsack problem while operating in $O(n \log n)$ time due to the sorting or heap operations.

Algorithm Greedy algorithm for solving the Fractional Knapsack problem

$W \leftarrow C$ ▷ Remaining capacity in the knapsack
while $W > 0$ **and** $Q \neq \varnothing$ **do**
  $(i, v_i, w_i) \leftarrow \textsc{Extract-Max}(Q)$ ▷ Item with the highest value-to-weight ratio
  if $w_i \leq W$ **then** ▷ The entire item fits
    $x_i \leftarrow 1$ ▷ Select the whole item
    $W \leftarrow W - w_i$
  else ▷ Only a fraction can be taken
    $x_i \leftarrow \frac{W}{w_i}$ ▷ Select the fraction that fits
    $W \leftarrow 0$
  end if
end while

Huffman Coding

Trivial Symbol Encoding

- Suppose we have alphabet $S = \{s_1, \ldots s_k\}$ of size k and we want to encode a message M with n symbols from S
- A trivial encoding is to use $n \cdot \lceil \log_2 k \rceil$ bits to encode the message.
- For example, $S = \{a, b, c\}$ and $M = \text{abaabc}$
  - we can assign $a = 000$, $b = 001$ and $c = 010$
  - thus $M = 000\,001\,000\,000\,001\,010$
  - For a total of 18 bits.

Better Symbol Encoding

- We can use **variable length encoding**
- This uses shorter codes for the **most frequent symbols**
- The same method as **MORSE CODE**
- Problem - we lose the "boundary" between symbols
  - To solve this we use prefix code
  - No code for one character could not be the prefix for another code
  - I,e $a = 0$ and $b = 01$ is not allowed as b starts with a in this case
- Prefix code can be represented using a **Binary Tree**

Using Huffman Coding

- Using a tree representation of prefixes, the average number of bits is given by:

$$A = \sum_{i-1}^{k} f_i \cdot d(s_i)$$

  Where $d(s_i)$ is the depth of the leaf corresponding to s_i
- The **greedy algorithm** constructs this prefix tree "bottom up"

Greedy Algorithm for Constructing Prefixes

- A prefix code is equivalent to a binary tree. Once we build the optimal binary tree, we can "read off" the encoding.
- The basic idea of Huffman Coding (HC) is to build the optimal tree recursively in a greedy manner.
- The optimal tree T for k symbols is obtained by constructing the optimal tree T' for $k-1$ symbols.
- T' is the same as T except replacing the two nodes with the smallest frequencies in T, x and y, by a single node having the sum of the frequencies: $f_w = f_x + f_y$.
  - Start with a list of all symbols and their frequencies.
  - Find the two symbols with the smallest frequencies.
  - Create a new node that combines these two symbols, with a frequency equal to the sum of their frequencies.
  - Replace the two symbols in the list with this new combined node.
  - Repeat the process until there is only one node left in the list. This node represents the root of the Huffman tree.
- $T' = T - \{x, y\} \cup \{w\}$.

Pseudocode for Building Huffman Coding

Algorithm Build Huffman Tree

Require: Array of frequency, symbol pairs, F
Ensure: Huffman tree

```
function BuildTree(F)
    for all (f, s) ∈ F do
        Insert(Q, new Node(f, s))
    end for
    while |Q| > 1 do
        x ← Extract-Min(Q)
        y ← Extract-Min(Q)
        z ← new node
        z.left, z.right ← x, y
        z.f ← x.f + y.f
        Insert(Q, z)
    end while
    return Extract-Min(Q)
end function
```

Psuedocode for getting huffman codes for each symbol

Algorithm Get Huffman Codes

Require: Root of the Huffman tree
Ensure: The code for each symbol

```
function WalkTree(node, prefix)
    if node is a leaf then
        code[node.symbol] ← prefix
    else
        WalkTree(node.left, prefix + "0")
        WalkTree(node.right, prefix + "1")
    end if
end function
WalkTree(root, "")
```