CS 2500: Algorithms

Lecture 19: Greedy Algorithms: Introduction and Scheduling Problem

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 22, 2024

Introduction to the Greedy Method

- The greedy method is a straightforward design technique.
- It is applied to optimization problems.
- Most problems have n inputs and require obtaining a subset that satisfies some constraints.
- Any subset that satisfies the constraints is called a **feasible solution**.
- A feasible solution that maximizes or minimizes a given objective function is called an **optimal solution**.

Introduction to the Greedy Method

What is an Optimization Problem? An optimization problem is associated with:

- An objective function called a value.
- A predicate P representing feasibility criteria.
- A solution space of all possible solutions U.
- An extremum (maximum or minimum) requirement.

Introduction to the Greedy Method

- Most problems have n inputs and require obtaining a subset that satisfies some constraints.
- Any subset that satisfies the constraints is called a **feasible solution**.
- A feasible solution that maximizes or minimizes a given objective function is called an **optimal solution**.

Greedy algorithms solve optimization problems exercising locally optimal decisions at every step that can lead to a globally optimum solution.

Greedy Strategy

- The greedy method involves making a sequence of decisions, considering one input at a time.
- At each step, a decision is made regarding whether an input is part of the optimal solution.
- Selection is based on the optimization measure.
- The method operates in **stages**, with each stage aiming to build towards an optimal solution.

Making Change Problem

Problem Statement:

- Given the following coins:
  - Dollars (100 cents), Quarters (25 cents), Dimes (10 cents), Nickels (5 cents), and Pennies (1 cent)
- Devise an algorithm to pay a given amount using the **smallest possible number of coins**.

Example:

- To pay 2.89(289 cents), the best solution is to give:
  - 2 Dollars, 3 Quarters, 1 Dime, and 4 Pennies (total: 10 coins)

Approach: Greedy Algorithm

Intuitive Solution:

- Use a **greedy algorithm** that starts with the largest denomination and works its way down.
- At each step, choose the largest coin that does not exceed the remaining amount.

Why Greedy?

- Simple and intuitive approach: We unconsciously use this method every day.
- Always takes the largest available coin that does not exceed the amount left.

Algorithm: Making Change

Algorithm MakeChange(n)

1: $C \leftarrow \{100, 25, 10, 5, 1\}$ ▷ Available coin denominations
2: $S \leftarrow \emptyset$ ▷ Initialize the solution set
3: $s \leftarrow 0$ ▷ Sum of the coins in the solution
4: **while** $s \neq n$ **do**
5: $x \leftarrow$ Largest item in C such that $s + x \leq n$
6: **if** x does not exist **then**
7: **return** "no solution found"
8: **end if**
9: $S \leftarrow S \cup \{x\}$ ▷ Add the selected coin to the solution
10: $s \leftarrow s + x$ ▷ Update the sum
11: **end while**
12: **return** S

Example Walkthrough: 2.89(289 cents)

1. Start with 289 cents.
2. Choose 2 Dollars (200 cents): Remaining 89 cents.
3. Choose 3 Quarters (75 cents): Remaining 14 cents.
4. Choose 1 Dime (10 cents): Remaining 4 cents.
5. Choose 4 Pennies (4 cents): Remaining 0 cents.

Total Coins Used: 2 Dollars, 3 Quarters, 1 Dime, 4 Pennies = 10 Coins

Characteristics of the Greedy Algorithm

Key Points:

- The algorithm always chooses the largest possible coin without exceeding the amount left.
- **Optimal Solution:** The greedy approach works when coins are multiples of each other (e.g., 25, 10, 5, 1).
- **Challenges:** With other coin systems, the greedy approach might fail to find the optimal solution.

Why "Greedy"?

- Never changes its decision once a coin is chosen.
- Always selects what appears to be the best immediate option.

Limitations and Considerations

When the Greedy Approach Might Fail:

- With a different set of coins, it might not produce an optimal solution.
- Example: If coins are 1, 3, and 4 cents, to make 6 cents:
  - Greedy: $4 + 1 + 1$ (3 coins)
  - Optimal: $3 + 3$ (2 coins)

Proof of Correctness:

- The greedy algorithm can be shown to be optimal for the given set of coins when we have an adequate supply of each denomination.
- Hard to prove formally, but intuitive for standard coins.

Greedy Problems in Daily Life

- **Making coin change:** Let's say a friend asks you to lend him 67 cents, assuming the available denominations of coins are 1, 5, 10, 25, 50. Would you give them 67 one-cent coins?
- **Finding the shortest path:** Find the shortest path between Rolla and St. Louis. The objective here is to find the shortest path to reduce the effort.
- **Optimal loading of a suitcase with items of different weights:** The objective is to maximize the number of items.
- **The arrangement of items on a shelf:** The objective is to maximize the number of items and optimize space.

Key Concepts

- **Feasible Solution**: A subset of inputs that meets the problem's constraints.
- **Optimal Solution**: A feasible solution that maximizes or minimizes the objective function.
- **Subset Paradigm**: The greedy method works by incrementally constructing a subset solution.
- **Ordering Paradigm**: An alternative, where inputs are processed based on a predefined order.

Greedy Algorithm: Control Abstraction

Algorithm Greedy(a, n)

```
1: solution ← ∅                          ▷ Initialize the solution
2: for i = 1 to n do
3:     x ← Select(a)
4:     if Feasible(solution, x) then
5:         solution ← Union(solution, x)
6:     end if
7: end for
8: return solution
```

Explanation of Algorithm Components

- **Select**: Function that selects an input from a and removes it.
- **Feasible**: Boolean function that checks if x can be added to the *solution*.
- **Union**: Combines x with the current *solution* and updates the objective function.
- This abstraction helps in designing specific greedy algorithms based on different problems.

Subset Paradigm vs. Ordering Paradigm

- **Subset Paradigm**:
  - Focuses on constructing a subset that satisfies constraints.
  - Decisions are made based on whether adding an element maintains feasibility.
- **Ordering Paradigm**:
  - Processes inputs based on a predetermined order.
  - Optimizes using decisions already made.

Greedy Algorithms: Examples

- Subset Paradigm:
  1. Job Scheduling Problem (with and without deadlines).
  2. Fractional Knapsack Problem
  3. Minimum Cost Spanning Tree Problem (Prim's and Kruskal's Algorithms)
  4. Tree Vertex Splitting Problem
  5. Single Source Shortest Path Problem (Dijkstra's Algorithm)
- Ordering Paradigm:
  1. Huffman Coding

Scheduling Problems

- In the shop, there is only one clerk.
- Various customers want to get the bills of the items they have purchased.
- The customers who have bought one item require less time for processing than those who bought a large number of items.
- The clerk feels that perhaps a little reordering of the queue might help satisfy all the customers. If so, what would be the best possible arrangement?

Job Scheduling Without Deadlines

Problem Statement:

- Given: A set S of n jobs.
- Each job $j \in S$ has an associated processing time $t_j > 0$.
- Objective: Minimize the total turnaround time, which is the sum of the waiting times plus the processing times for all jobs.

How do we solve this?

- An optimal solution involves sorting the jobs by their processing times in non-decreasing order, ensuring that shorter jobs are completed first, thereby reducing the total waiting time.
- This approach fits the principle of the shortest job first (SJF) scheduling.

Job Scheduling Without Deadlines: Example 1

- Consider three tasks with the following service times:
  - $t_1 = 2$ units
  - $t_2 = 7$ units
  - $t_3 = 4$ units
- Our goal is to find the optimal order of scheduling these jobs.

Job Scheduling Without Deadlines: Example 1

Turnaround Time Calculation

Order	Total Time in System	Average Time
t_1, t_2, t_3	$2 + (2 + 7) + (2 + 7 + 4) = 24$	$24/3 = 8.00$
t_1, t_3, t_2	$2 + (2 + 4) + (2 + 4 + 7) = 21$	$21/3 = 7.00$
t_2, t_1, t_3	$7 + (7 + 2) + (7 + 2 + 4) = 29$	$29/3 \approx 9.67$
t_2, t_3, t_1	$7 + (7 + 4) + (7 + 4 + 2) = 31$	$31/3 \approx 10.33$
t_3, t_1, t_2	$4 + (4 + 2) + (4 + 2 + 7) = 23$	$23/3 \approx 7.67$
t_3, t_2, t_1	$4 + (4 + 7) + (4 + 7 + 2) = 28$	$28/3 \approx 9.33$

Table: Turnaround Time Calculation for Different Schedules

- From the table, we can observe that the optimal order is t_1, t_3, t_2.
- Scheduling jobs in ascending order of service time minimizes the average turnaround time.

Job Scheduling without Deadlines: Example 2

Problem:

- Given jobs without deadlines, find the optimal scheduling order to minimize the total completion time.
- Use the Shortest Job First (SJF) strategy.

Jobs:

- $J_1 = 4$
- $J_2 = 5$
- $J_3 = 10$

Job Scheduling without Deadlines: Example 2

Solution:

1. Sort jobs by processing time (Shortest Job First):

$$\text{Optimal Order: } J_1, J_2, J_3$$

2. Calculate completion times:
   - $C(J_1) = 4$
   - $C(J_2) = 4 + 5 = 9$
   - $C(J_3) = 9 + 10 = 19$
3. **Total Completion Time:**

$$C = 4 + 9 + 19 = 32$$

Conclusion: The optimal scheduling order is J_1, J_2, J_3, with a total completion time of **32**.

Job Scheduling Without Deadlines

Greedy Approach for SJF:

- Sort the jobs by service time in a non-decreasing order.
- Select the next job from the sorted list and include it in the solution set.
- Continue until all jobs are scheduled.

Job Scheduling Without Deadlines

Algorithm SJFSchedule(J)

1: $S \leftarrow$ sorted array of jobs in J based on service time
2: $i \leftarrow 1$
3: $solution \leftarrow \emptyset$
4: **while** $i \leq n$ **do**
5: Select the next job i from S
6: $solution \leftarrow solution \cup job\ i$
7: $i \leftarrow i + 1$
8: **end while**
9: **return** *solution*

Time Complexity:

- Core part: Sorting (Needs $O(n \log n)$ time).
- All remaining tasks: $O(n)$ time.
- Result: $O(n \log n)$

Job Scheduling with Deadlines

- Given a set of n jobs, each job i has:
  - A deadline $d_i \geq 0$
  - A profit $p_i > 0$
- Profit p_i is earned only if the job is completed by its deadline.
- The goal is to find a subset of jobs that maximizes the total profit.
- Only one machine is available to process the jobs, and each job takes one unit of time.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J or $\sum_{i \in J} p_i$.
- An optimal solution is a feasible solution with maximum value.
- This problem fits the **subset paradigm** because the solution involves selecting a subset of jobs.

Job Scheduling with Deadlines: Example 1

- Number of jobs: $n = 4$
- Profits: $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$
- Deadlines: $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$
- Goal: Maximize the profit while completing each job before its deadline.

Job Scheduling with Deadlines: Example 1

Feasible Solutions and Their Value

Feasible Solution	Processing Sequence	Total Profit
$(1, 2)$	$2, 1$	110
$(1, 3)$	$1, 3$ or $3, 1$	115
$(1, 4)$	$4, 1$	127
$(2, 3)$	$2, 3$	25
$(3, 4)$	$4, 3$	42
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27

Table: Feasible Solutions and Their Total Profit

- Solution $(1, 4)$ is optimal with a total profit of 127.
- Jobs 1 and 4 must be processed in the order: Job 4 first, followed by Job 1.

Job Scheduling with Deadlines: Example 1

Greedy Approach:

- To formulate a greedy algorithm, we need an optimization measure.
- Choose jobs in non-increasing order of profits p_i.
- Ensure that each selected job can be scheduled before its deadline.
- Objective: Maximize $\sum_{i \in J} p_i$, where J is the set of selected jobs.

Job Scheduling with Deadlines: Example 1

Greedy Algorithm Explanation

- Start with an empty set $J = \emptyset$.
- Add Job 1 (with the highest profit) to J: $J = \{1\}$.
- Next, consider Job 4. Adding Job 4 results in a feasible solution: $J = \{1, 4\}$.
- Consider Job 3. Adding Job 3 makes $J = \{1, 3, 4\}$, which is not feasible, so discard it.
- Consider Job 2. Adding Job 2 makes $J = \{1, 2, 4\}$, which is not feasible, so discard it.
- Final solution: $J = \{1, 4\}$, with a total profit of 127.

Job Scheduling with Deadlines: Example 1

Why is $J = \{1, 3, 4\}$ Not Feasible?

- Job 1 must be completed by the end of time unit 2.
- Job 3 must also be completed by the end of time unit 2.
- Job 4 must be completed by the end of time unit 1.
- When scheduling $J = \{1, 3, 4\}$:
  - If Job 4 is scheduled first (deadline $= 1$), it occupies time unit 1.
  - Jobs 1 and 3 both need to finish by time unit 2.
  - Only one time unit (time unit 2) is left, but two jobs (1 and 3) need to be completed.
- Therefore, it is impossible to finish both Jobs 1 and 3 within their deadlines.

Job Scheduling with Deadlines

Algorithm GreedyJob(d, J, n)

1: $J \leftarrow 1$
2: $i \leftarrow 1$
3: **while** $i \leq n$ **do**
4: **if** (all jobs in $J \cup \{i\}$ can be completed by their deadlines) **then**
5: $J \leftarrow J \cup \{i\}$;
6: **end if**
7: $i \leftarrow i + 1$
8: **end while**

Job Scheduling with Deadlines: Example 2

Problem:

- Given jobs with deadlines and profits, schedule them to maximize profit.
- Each job takes 1 unit of time, and no two jobs can be scheduled simultaneously.

Jobs:

Job	Deadline	Profit
1	2	60
2	1	30
3	2	40
4	1	80

Job Scheduling with Deadlines: Example 2

Solution:

1. Sort jobs by profit in descending order:

$$\text{Jobs: } (4, 1, 3, 2)$$

2. The maximum deadline here is 2, so we have two slots: `[empty, empty]`
3. Place jobs:
   - Job 4 (Deadline 1, Profit 80) $\rightarrow$ Slot 1: `[4, empty]`
   - Job 1 (Deadline 2, Profit 60) $\rightarrow$ Slot 2: `[4, 1]`
   - Job 3 (Deadline 2, Profit 40) $\rightarrow$ Cannot be scheduled
   - Job 2 (Deadline 1, Profit 30) $\rightarrow$ Cannot be scheduled
4. Scheduled Jobs: Job 4 and Job 1
5. **Total Profit:** $80 + 60 = 140$

Conclusion: The maximum profit is **140**.