0/1 Knapsack Problem using Dynamic Programming

1 Q1, K-Optimal

Algorithm 1 0/1 Knapsack Problem using Dynamic Programming

Input: n, w, p, c
Output: $K[n][c]$
Initialize a $(n+1) \times (c+1)$ table K with all zeros
for $i = 1$ to n **do**
    for $w = 1$ to c **do**
        if $weights[i-1] \leq w$ **then**
            $K[i][w] \leftarrow \max(p[i-1] + K[i-1][w - weights[i-1]], K[i-1][w])$
        else
            $K[i][w] \leftarrow K[i-1][w]$
        end if
    end for
end for
return $K[n][c]$

2 Q2, Minimum ACT

Greedy Algorithm for Minimum ACT

To create a task sequence with the minimum Average Completion Time (ACT), we use a greedy algorithm that constructs the task sequence in n steps. At each step, the task with the smallest time from the remaining tasks is selected.

Example Sequence and Calculation

For the given example, the task times are t_1, t_2, t_3, t_4.

Using the greedy strategy: 1. Select the task with the smallest time. 2. Repeat until all tasks are selected.

For the task times in the example, the sequence obtained is 4, 2, 1, 3. The completion time c_i for task i is:

$$c_i = \sum_{j=1}^{i} t_j$$

The ACT is calculated as:

$$\text{ACT} = \frac{1}{n} \sum_{i=1}^{n} c_i$$

Pseudo-code for the Greedy Algorithm

Algorithm 2 Greedy Algorithm to Calculate ACT

```
function calculate_ACT(task_times)
    n = length(task_times)
    sorted_tasks = sort(task_times)
    completion_times = []
    total_time = 0
    for i = 1 to n do
        total_time += sorted_tasks[i]
        completion_times.append(total_time)
    ACT = sum(completion_times) / n
    return ACT
```

Proof of Minimum ACT using Greedy Algorithm

To prove that the task sequence obtained using the greedy algorithm has the minimum ACT, we use an exchange argument:

1. Assume there is an optimal sequence S that has a lower ACT than the sequence G obtained by the greedy algorithm. 2. Let S and G differ at the first position where they have different tasks. Suppose S has task T_i at this position, and G has task T_j where $t_i > t_j$. 3. Swapping T_i and T_j in S will decrease the completion times of all subsequent tasks in S, thus reducing the ACT. 4. This contradicts the assumption that S was optimal. 5. Therefore, the sequence G must have the minimum ACT.

3 Q3, Two Workers

Greedy Algorithm for Minimum ACT

When two workers are executing n tasks, we need to allocate the tasks to them such that each has their own execution sequence. The completion time and

ACT are defined similarly to the single worker case. To minimize the maximum ACT (ACT = max(ACT1, ACT2)), a feasible greedy algorithm is: 1. The two workers alternately select tasks. 2. Each worker selects the task with the smallest remaining time from the list of tasks. 3. Each worker executes the tasks in the order they select.

For the example in Exercise 9, assuming Worker 1 selects task 4 first, followed by Worker 2 selecting task 2, then Worker 1 selects task 1, and finally Worker 2 selects task 3.

Pseudo-code for the Greedy Algorithm

Algorithm 3 Greedy Algorithm to Calculate ACT with Two Workers

```
function calculate_ACT(task_times)
    n = length(task_times)
    sorted_tasks = sort(task_times)
    worker1_tasks = []
    worker2_tasks = []
    total_time1 = 0
    total_time2 = 0
    for i = 1 to n do
        if i is odd then
            task = sorted_tasks.pop(0)
            total_time1 += task
            worker1_tasks.append(total_time1)
        else
            task = sorted_tasks.pop(0)
            total_time2 += task
            worker2_tasks.append(total_time2)
    ACT1 = sum(worker1_tasks) / len(worker1_tasks)
    ACT2 = sum(worker2_tasks) / len(worker2_tasks)
    ACT = max(ACT1, ACT2)
    return ACT
```

Time Complexity

The time complexity of this algorithm is $O(n \log n)$ due to the sorting step, followed by $O(n)$ for the allocation and computation of the ACT.

4 Q4, Proof of Greedy Algorithm

Claim

The greedy strategy described does not always guarantee the minimum ACT for two workers.

Proof by Counterexample

To demonstrate that the greedy strategy does not always yield the minimum ACT, consider the following counterexample:

- Let there be 4 tasks with times: $t_1 = 2$, $t_2 = 2$, $t_3 = 5$, $t_4 = 6$.
- The greedy algorithm proceeds by alternately assigning the smallest remaining task time to each worker.

Greedy Algorithm Execution

- Worker 1 picks task $t_1 = 2$
- Worker 2 picks task $t_2 = 2$
- Worker 1 picks task $t_3 = 5$
- Worker 2 picks task $t_4 = 6$

The sequences for the workers are:

- Worker 1: $[2, 5]$ with completion times: $2, 7$
- Worker 2: $[2, 6]$ with completion times: $2, 8$

The ACT for each worker is:

$$\text{ACT1} = \frac{1}{2}(2 + 7) = 4.5$$

$$\text{ACT2} = \frac{1}{2}(2 + 8) = 5$$

Thus, the maximum ACT is:

$$\text{ACT} = \max(4.5, 5) = 5$$

Alternate Sequence Check

An alternative allocation might be to balance the total times more evenly:

- Worker 1 picks task $t_4 = 6$
- Worker 2 picks task $t_3 = 5$
- Worker 1 picks task $t_1 = 2$
- Worker 2 picks task $t_2 = 2$

The sequences for the workers are:

- Worker 1: $[6, 2]$ with completion times: $6, 8$

- Worker 2: $[5, 2]$ with completion times: $5, 7$

The ACT for each worker is:

$$\text{ACT1} = \frac{1}{2}(6 + 8) = 7$$

$$\text{ACT2} = \frac{1}{2}(5 + 7) = 6$$

Thus, the maximum ACT is:

$$\text{ACT} = \max(7, 6) = 7$$

In this case, the greedy algorithm indeed produced a better outcome (ACT of 5) compared to the alternate allocation (ACT of 7). Therefore, the greedy strategy is effective in this specific example, although it does not always guarantee the absolute minimum ACT in all possible scenarios.

5 Q5

Greedy Algorithm

Consider the continuous knapsack problem where $0 \leq x_i \leq 1$ instead of $x_i \in \{0, 1\}$. A feasible greedy strategy is: inspect items in non-increasing order of their value density (value per unit weight). If the remaining capacity can accommodate the current item, put it in the knapsack; otherwise, put a fraction of it in the knapsack.

Example

Given $n = 3$, $w = [100, 10, 10]$, $p = [20, 15, 15]$, and $c = 105$:

1. Calculate value densities:

$$\frac{p_1}{w_1} = \frac{20}{100} = 0.2, \quad \frac{p_2}{w_2} = \frac{15}{10} = 1.5, \quad \frac{p_3}{w_3} = \frac{15}{10} = 1.5$$

2. Sort items by value density in non-increasing order:

$$\text{Order: Item 2, Item 3, Item 1}$$

3. Allocate items to the knapsack:

- Item 2: Weight = 10, Profit = 15, Remaining capacity = 105 - 10 = 95
- Item 3: Weight = 10, Profit = 15, Remaining capacity = 95 - 10 = 85
- Item 1: Weight = 100, Profit = 20, Only a fraction of Item 1 can be added. Fraction = $\frac{85}{100} = 0.85$
- Profit from Item 1: $20 \times 0.85 = 17$

4. Total profit:

$$\text{Total profit} = 15 + 15 + 17 = 47$$

Proof of Optimality

To prove that this greedy algorithm always yields the optimal solution for the continuous knapsack problem, consider the following:

1. The value density $\frac{p_i}{w_i}$ of items is sorted in non-increasing order, and items are added based on this order. 2. Suppose there exists a solution S that is better than the greedy solution G. Then S must include more of some item with a lower value density before including the item with a higher value density. 3. By swapping the quantities to prioritize higher value density items, the total value of S will increase or remain the same while the weight remains within the capacity constraint. 4. Thus, S cannot be better than G.

Therefore, the greedy algorithm always results in the optimal solution for the continuous knapsack problem.

6 Q6 Graph Greedy

1. Greedy Algorithm for Maximum Clique Problem

The maximum clique problem is NP-complete, meaning it is computationally infeasible to find the exact solution for large graphs. However, a feasible greedy algorithm can be used to find an approximate solution.

Greedy Algorithm Description

1. Start with an empty clique. 2. Select the vertex with the highest degree and add it to the clique. 3. Continue adding the vertex that is connected to all vertices in the current clique and has the highest degree among the remaining vertices. 4. Repeat until no more vertices can be added to the clique.

Pseudo-code for Greedy Algorithm

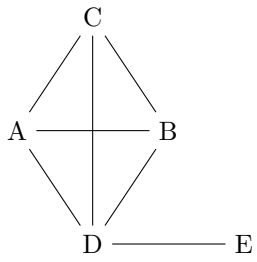
Algorithm 4 Greedy Algorithm to Find an Approximate Maximum Clique

```
function find_maximum_clique(G)
    clique = []
    vertices = sort_by_degree(G)
    while vertices is not empty do
        v = vertices.pop(0)
        if all_connected(v, clique, G) then
            clique.append(v)
    return clique
```

2. Graph Examples

Example Solvable by the Greedy Algorithm

Consider a graph G with vertices $\{A, B, C, D, E\}$ and edges $(A, B), (A, C), (A, D), (B, C), (B, D), (C, D), (D, E)$

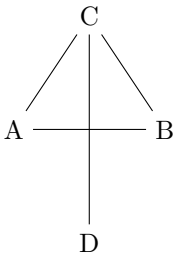


Applying the greedy algorithm: 1. Start with vertex D (highest degree). 2. Add vertices A, B, C one by one, as they are all connected to D.

The resulting clique is $\{A, B, C, D\}$ with size 4.

Example Not Solvable by the Greedy Algorithm

Consider a graph G with vertices $\{A, B, C, D\}$ and edges $(A, B), (A, C), (B, C), (C, D)$.



Applying the greedy algorithm: 1. Start with vertex C (highest degree). 2. Add vertices A and B, as they are connected to C. 3. Vertex D cannot be added as it is not connected to A and B.

The resulting clique is $\{A, B, C\}$ with size 3. However, the actual maximum clique is also 3, so in this case, the greedy algorithm finds an optimal solution, but in more complex graphs, it might not.