Unit 5.1 The Greedy Method

Algorithms

EE3980

Apr. 21, 2020

Knapsack Problem

- Knapsack problem
  - Given n objects, each object i, $1 \le i \le n$, has
    - Weight w_i,
    - Profit $p_i \cdot x_i$, if x_i fraction is placed into the bag $(0 \le x_i \le 1)$.
  - A bag with capacity m.
  - The objective is to maximize the profit.

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i, \tag{5.1.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le m, \tag{5.1.2}$$

$$\text{and} \quad 0 \le x_i \le 1, \qquad 1 \le i \le n. \tag{5.1.3}$$

- A feasible solution is any set $\{x_1, \cdots, x_n\}$ that satisfies Eqs. (5.1.2) and (5.1.3).
- An optimal solution is a feasible solution for which Eq. (5.1.1) is maximized.

Knapsack Problem – Example

- An example of 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

Solution	$\{x_1, x_2, x_3\}$	$\sum w_i x_i$	$\sum p_i x_i$
1	$\{1/2, 1/3, 1/4\}$	16.5	24.25
2	$\{1, 2/15, 0\}$	20	28.2
3	$\{0, 2/3, 1\}$	20	31
4	$\{0, 1, 1/2\}$	20	31.5

- Note that $\sum w_i x_i \le m$ for all 4 feasible solutions.
- Solution 4 yields the maximum profit among these 4 feasible solutions.

Knapsack Problem – Algorithm 1

- A general greedy algorithm for knapsack program is shown below.

Algorithm 5.1.1. Knapsack by Profit

```
   // Solve knapsack problem using max profit greedy method.
   // Input: n, w[1 : n], p[1 : n], m
   // Output: x[1 : n], 0 ≤ x[i] ≤ 1.
 1 Algorithm Knapsack_P(m, n, w, p, x)
 2 {
 3     A[1 : n] := Objects sorted by decreasing p[1 : n]; // p[A[i]] ≥ p[A[j]] if i < j.
 4     for i := 1 to n do x[i] := 0; // Initialize solution vector.
 5     i := 1;
 6     while (i ≤ n and w[A[i]] ≤ m) do { // Selecting max profit object.
 7         x[A[i]] := 1;
 8         m := m − w[A[i]];
 9         i := i + 1;
10     }
11     if (i ≤ n) then x[A[i]] := m/w[A[i]]; // Partial selection.
12 }
```

- Note that line 3 sort A into *decreasing order* by p
- Applying this algorithm we get solution 2 for the example.

Knapsack Problem – Algorithm 2

- The greedy algorithm can be modified as below.

Algorithm 5.1.2. Knapsack by Weight

```
   // Solve knapsack problem using min weight greedy method.
   // Input: n, w[1 : n], p[1 : n], m
   // Output: x[1 : n], 0 ≤ x[i] ≤ 1.
 1 Algorithm Knapsack_W(m, n, w, p, x)
 2 {
 3     A[1 : n] := Objects sorted by increasing w[1 : n]; // w[A[i]] ≤ w[A[j]] if i < j.
 4     for i := 1 to n do x[i] := 0; // Initialize solution vector.
 5     i := 1;
 6     while (i ≤ n and w[A[i]] ≤ m) do { // Selecting min weight object.
 7         x[A[i]] := 1;
 8         m := m − w[A[i]];
 9         i := i + 1;
10     }
11     if (i ≤ n) then x[A[i]] := m/w[A[i]]; // Partial selection.
12 }
```

- Note that line 3 sort A into *increasing order* by w
- Applying this algorithm we get solution 3 for the example.

Knapsack Problem – Algorithm 3

- Another version of greedy algorithm is shown below.

Algorithm 5.1.3. Knapsack

```
   // Solve knapsack problem using max profit/weight ratio greedy method.
   // Input: n, w[1 : n], p[1 : n], m
   // Output: x[1 : n], 0 ≤ x[i] ≤ 1.
 1 Algorithm Knapsack(m, n, w, p, x)
 2 {
 3     A[1 : n] := Objects sorted by decreasing p[i]/w[i];
 4                                  // p[A[i]]/w[A[i]] ≥ p[A[j]]/w[A[j]] if i < j.
 5     for i := 1 to n do x[i] := 0;
 6     i := 1;
 7     while (i ≤ n and w[A[i]] ≤ m) do {
 8         x[A[i]] := 1;
 9         m := m − w[A[i]];
10         i := i + 1;
11     }
12     if (i ≤ n) then x[A[i]] := m/w[A[i]];
13 }
```

- Note that line 3 sort A into *decreasing order* by $p[i]/w[i]$

Knapsack Problem – Complexity and Optimality

- Applying Algorithm (5.1.3) we get solution 4 for the example.
  - This is the optimal solution since p/w is the real objective.
- Knapsack Algorithm (5.1.3) has the time complexity of $\mathcal{O}(n \lg n)$.
  - Dominated by the Sort function on line 3
  - The while loop (lines 7-11) and for (line 5) loop are both $\mathcal{O}(n)$.

Lemma 5.1.4.

In case the sum of all the weights is less than or equal to m, i.e., $\sum_{i=1}^{n} w_i \le m$, then $x_i = 1$, $1 \le i \le n$, is an optimal solution.

Lemma 5.1.5.

In case $\sum_{i=1}^{n} w_i \ge m$, then all optimal solutions will fill the knapsack exactly, i.e., $\sum_{i=1}^{n} w_i x_i = m$.

Knapsack Problem – Solution Optimality

Lemma 5.1.6.

In case that the capacity is smaller than the weight of any object, $m < w_i$, $\forall i$, then the optimal solution is $x_i = m/w_i$, where p_i is the maximum, and $x_j = 0$, $j \ne i$.

Theorem 5.1.7.

If A is sorted by $\{p_i/w_i\}$ in non-increasing order, then the Knapsack algorithm (Algorithm 5.1.3) generates an optimal solution to the instance of the knapsack problem.

Proof. Let the objects be ordered by p_i/w_i.
If $m = w_1$, then $x_1 = 1$, $x_i = 0$, $1 < i < n$, is the optimal solution.
Once object 1 is selected, the capacity is reduced to $m - w_1$ and the process repeated until $m < w_j$ with $x_j = 0$. In that case, from the last lemma the object with the smallest j should be selected and $x_j = m/w_j$. □

- Proof can also be found in textbook [Horowitz], pp. 221-222.

Container Loading

- Container loading problems
  - Input: n containers with $w_i > 0$, $1 \le i \le n$, weight each.
  - A ship with cargo capacity of c.
  - Load the maximum number of containers to the ship without exceeding the cargo capacity.
- Let $x_i \in \{0,1\}$ such that $x_i = 1$ if container i is loaded onto the ship.

$$\begin{aligned} &\text{Constraint:} \quad \sum_{i=1}^{n} x_i \cdot w_i \le c, \\ &\text{Objective:} \quad \max\left(\sum_{i=1}^{n} x_i\right). \end{aligned} \tag{5.1.4}$$

- Example: Suppose there are 8 containers with weights $[w_1, w_2, \cdots w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$ and ship capacity $c = 400$.
  - Then the solution is $[x_1, x_2, \cdots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$.
  - $\sum_{i=1}^{8} w_i x_i = 390$ that satisfies the constraint.
  - $\sum_{i=1}^{8} x_i = 6$ is the maximum number of containers loaded.

Container Loading – Algorithm

Algorithm 5.1.8. Container Loading

```
// Load maximum containers (weights w[1 : n]) with capacity c.
// Input: c, n, w[1 : n]
// Output: solution vector x[1 : n], x[i] = 0 or 1.
Algorithm ContainerLoading(c, n, w, x)
{
    A[1 : n] := Containers sorted by non-decreasing w[1 : n] ;
        // w[A[i]] ≤ w[A[j]] if i < j.
    for i := 1 to n do x[i] := 0 ;
    i := 1 ;
    while (i ≤ n and w[A[i]] ≤ c) do {
        x[A[i]] := 1 ;
        c := c − w[A[i]] ;
        i := i + 1 ;
    }
}
```

- Note that $w[A[i]]$ is sorted into increasing order.
  - Using the last example, $w[1:8] = \{100, 200, 50, 90, 150, 50, 20, 80\}$, then $A[1:8] = \{7, 3, 6, 8, 4, 1, 5, 2\}$ such that $w[A[i]]$ is in non-decreasing order.

Container Loading – Complexity and Optimality

- The time complexity of the `ContainerLoading` algorithm is dominated by the `Sort` function (line 3), which is $\mathcal{O}(n \lg n)$.
- The `while` loop (lines 7-11) is $\mathcal{O}(n)$.
- Overall complexity $\mathcal{O}(n \lg n)$.

Theorem 5.1.9.

The Container Loading Algorithm (Algorithm 5.1.8) generates optimal loading.

Proof. Let $A = \{x_i\}$ be the set found by the algorithm, and $|A| = k$. It can be shown that $\sum_{i=1}^{k} w(x_i)$ is the minimum for any subset with k containers.
Suppose the optimal solution is $B = \{y_j\}$, $|B| = h$. One can prove that $h = k$.
If $h > k$, since $\sum_{i=1}^{k} w(x_i) \le \sum_{i=1}^{k} w(y_i)$. Thus, for any object $y_j \in B$ and $y_j \notin A$, $1 \le j \le k$, there is an x_i, $1 \le i \le k$ such that $w(x_i) \le w(y_j)$. Replacing y_j by x_i in set B to form a B', then B' is an optimal solution. Repeating this process, we found that $A \cup \{y_j | k < j \le h\}$ is an optimal solution. But, the algorithm states that no such y_j exists, hence $h \le k$. □

- An alternative proof can also be found in textbook [Horowitz], pp. 215-217.

Subset Optimization Problems

- A special class of problems that has n inputs,
  - Arrange the inputs to satisfy some constraints – feasible solutions
  - Find feasible solution that minimize or maximize an objective function – optimal solution
- The greedy method is a algorithm that takes one input at a time
  - If a particular input results in infeasible solution, then it is rejected; otherwise it is included.
  - The input is selected according to some measure
  - The selection measure can be the objective functions or other functions that approximate the optimality
  - However, this method usually generates a suboptimal solution.

Greedy Method

- The following is an abstraction of the greedy method in subset paradigm

Algorithm 5.1.10. Greedy Method

```
// Given n-element set A, find a subset that is an optimal solution.
// Input: A[1 : n], int n
// Output: solution ⊂ A.
Algorithm Greedy(A, n)
{
    solution := ∅ ;
    for i := 1 to n do {
        x := Select(A) ;
        A := A − {x} ;
        if Feasible(solution ∪ x) then solution := solution ∪ x;
    }
    return solution;
}
```

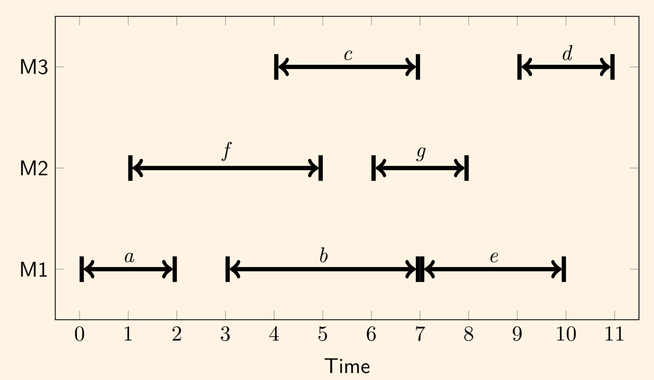
- In this subset paradigm the `Select` function selects an input from A and removes it.
- The `Feasible` function determines if it can be included into the solution vector.

- A variation of the greedy method is the ordering paradigm.
  - The inputs are ordered first and thus the `Select` function is not needed.

Machine Scheduling Problem

- Machine schedule problem
  - Input: n tasks and infinite number of machines
  - Each task has a start time $s[1 : n]$ and finish time, $f[1 : n]$, $s[i] < f[i]$.
  - Two tasks i and j overlap if and only if their processing intervals overlap at a point other than the interval start or end times.
  - A feasible task-to-machine assignment is that no machine is assigned with overlapping tasks.
  - An optimal assignment is a feasible assignment that utilizes the fewest number of machines.
- Example

Task	a	b	c	d	e	f	g
Start time	0	3	4	9	7	1	6
Finish time	2	7	7	11	10	5	8



Machine Scheduling Problem – Algorithm

Algorithm 5.1.11. Machine Scheduling

```
   // Schedule n tasks with minimum number of machines, m.
   // Input: n, start s[1 : n], finish f[1 : n]
   // Output: m, assignment: M[1 : n].
 1 Algorithm MachineSchedule(n, s, t, m, M)
 2 {
 3     A[1 : n] := sorted by increasing s[1 : n]; // s[A[i]] ≤ s[A[j]], if i < j.
 4     m := 1; M[A[1]] := m; MF[m] := f[A[1]]; // MF[] is machine finish time.
 5     for i := 2 to n do {
 6         j := {j | MF[j] = min_{1≤k≤m} MF[k]}; // Min finish time of all machines.
 7         if (MF[j] ≤ s[A[i]]) then { // Machine j can process A[i]
 8             M[A[i]] := j; // Assign task A[i] to machine j
 9             MF[j] := f[A[i]]; // update machine finish time.
10         }
11         else { // Need more machines, assign and update machine finish time
12             m := m + 1; M[A[i]] := m; MF[m] := f[A[i]];
13         }
14     }
15 }
```


Machine Scheduling Problem – Algorithm Execution

- Example

Task	a	b	c	d	e	f	g
Start time	0	3	4	9	7	1	6
Finish time	2	7	7	11	10	5	8

- After executing line 3, we have

$$\begin{array}{rcccccccl} A[1:n] = \{ & a, & f, & b, & c, & g, & e, & d & \} \\ s[A[1:n]] = \{ & 0, & 1, & 3, & 4, & 6, & 7, & 9 & \} \\ f[A[1:n]] = \{ & 2, & 5, & 7, & 7, & 8, & 10, & 11 & \} \end{array}$$

- And at line 4: $m = 1$, $M[A[1]] = 1$, $MF[1] = 2$.
  and the iteration is shown below.

i=2	j=1	MF[1]=2	s[A[2]]=1	m=2	M[A[2]]=2	MF[1]=2 MF[2]=5
i=3	j=1	MF[1]=2	s[A[3]]=3		M[A[3]]=1	MF[1]=7 MF[2]=5
i=4	j=2	MF[2]=5	s[A[4]]=4	m=3	M[A[4]]=3	MF[1]=7 MF[2]=5 MF[3]=7
i=5	j=2	MF[2]=5	s[A[5]]=6		M[A[5]]=2	MF[1]=7 MF[2]=8 MF[3]=7
i=6	j=1	MF[1]=7	s[A[6]]=7		M[A[6]]=1	MF[1]=10 MF[2]=8 MF[3]=7
i=7	j=3	MF[3]=7	s[A[7]]=9		M[A[7]]=3	MF[1]=10 MF[2]=8 MF[3]=11

Machine Scheduling Problem – Complexity

- In Algorithm (5.1.11), the time complexity is dominated by
  - `Sort` function on line 3: $\mathcal{O}(n \lg n)$
  - `Min` function on line 6: $\mathcal{O}(\lg n)$
    - MF can be kept as a min-heap.
    - In a **for** loop and thus $\mathcal{O}(n \lg n)$
  - Total complexity: $\mathcal{O}(n \lg n)$.

Theorem 5.1.12.

The Machine Scheduling Algorithm (Algorithm 5.1.11) generates an optimal assignment.

Summary

- Knapsack problem
- Container loading problem
- Greedy method
- Machine scheduling problem