Greedy Algorithm

KOMS120403 / KOMS119602 - Design and Analysis of Algorithm (2021/2022)

09 - Greedy Algorithm

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 4-8 April 2022

Table of contents

Optimization problem

An optimization problem is the problem of finding the best solution from all feasible solutions.

Types of optimization problems

- Maximization
  - Example: Integer knapsack problem
- Minimization
  - Example: Graph coloring problem, TSP

Standard form:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \ i = 1, \ldots, m \\ & h_j(x) = 0, \ j = 1, \ldots, p \end{aligned}$$

Greedy algorithm

Definition

Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

This is the most popular and the simplest algorithm to solve optimization problems (maximization and minimization).

- Greedy principal: **Take what you can get now!**
- Greedy algorithm builds the solution step-by-step.
- In each step, there are many possible choices. We take the best decision in each step, i.e. we choose the local optimum, in order to reach the global optimum.

Greedy algorithm

Example (Coin exchange problem)

We have a check of 42 in hand. There are coins of nominal 1, 5, 10, and 25. We want to exchange the money with the coins. What is the minimum number of coins needed in the exchange.

There are many possible combinations of coins. With brute-force algorithm, we can simply list all possibilities:

- $42 = 1 + 1 + \cdots + 1 \rightarrow 42$ coins
- $42 = 5 + 1 + 1 + \cdots + 1 \rightarrow 38$ coins
- ... etc.

With greedy algorithm, at each step, we take a coin with maximum value as many as possible.

$$42 = 25 + 10 + 5 + 1 + 1$$

So 5 coins are enough.

Scheme of greedy algorithm

Components of greedy algorithm:

- A candidate set - consists of candidate of solution that is created from the set.
- A selection function - used to choose the best candidate to be added to the solution.
- A feasibility function - used to determine whether a candidate can be included in the solution (feasible/unfeasible).
- An objective function - used to assign a value to a solution or a partial solution (maximizing or minimizing).
- A solution function - used to indicate whether a complete solution has been reached.

Scheme of greedy algorithm

Components analysis of the coin exchange problem:

- Candidate set: the set of coins $\{1, 5, 10, 25\}$.
- Selection function: choose the coin that has maximum value.
- Feasibility function: check if the sum of coins after taking a new coin does not exceed the amount of money.
- Objective function: minimizing the number of coins used.
- Solution function: the set of selected coins $\{1, 10, 25\}$.

Scheme of greedy algorithm

Algorithm 1 General scheme of greedy algorithm

```
1:  procedure GREEDY(C: candidate set)
2:      S ← {}                                   ▷ S is the solution function
3:      while (not SOLUTION(S)) and (C ≠ {}) do
4:          x ← SELECTION(C)
5:          C ← C − {x}
6:          if FEASIBLE(S ∪ {x}) then
7:              S ← S ∪ {x}
8:          end if
9:      end while
10:     if SOLUTION(S) then  return S
11:     else print('No solution exists')
12:     end if
13: end procedure
```

- At the end of iteration ("if condition"), we have a local optimum solution.
- At the end of the while loop, we obtain the global optimum solution (if any)

Solving problems with greedy algorithm

Some examples:

1. Coin exchange problem
2. Activity selection problem
3. Time minimization in the system
4. Integer knapsack problem
5. Fractional knapsack problem
6. Huffman coding
7. Traveling Salesman Problem

1. Coin exchange problem

1. Coin exchange problem (1)

Problem formulation:

- The amount of money want to be exchanged: M
- The set of available coins: $\{c_1, c_2, \ldots, c_n\}$
- The solution set: $X = \{x_1, x_2, \ldots, x_n\}$, where $x_i = 1$ if a_i is chosen and $x_i = 0$ otherwise

The objective function:

$$\text{Minimize } F = \sum_{i=1}^{n} x_i$$

$$\text{subject to } \sum_{i=1}^{n} c_i x_i = M$$

1. Coin exchange problem (2)

Solution by exhaustive search

- Since $X = \{x_1, x_2, \ldots, x_n\}$ and $x_i \in \{01\}$, then there are 2^n possible solutions.
- To evaluate the objective function for each solution candidate, we need $\mathcal{O}(n)$-time
- So, the time complexity of the exhaustive search is: $\mathcal{O}(n \cdot 2^n)$.

1. Coin exchange problem (3)

Algorithm 2 General scheme of greedy algorithm

```
1:  procedure CoinExchange(C: coin set, M: integer)
2:      S ← {}
3:      while (∑(all coins in S) ≠ M) and (C ≠ {}) do
4:          x ← coin of maximum value
5:          C ← C − {x}
6:          if ∑(all coins in S) + value(x) ≤ M then
7:              S ← S ∪ {x}
8:          end if
9:      end while
10:     if ∑(all coins in S) = M then
11:         return S
12:     else
13:         print('No feasible solution')
14:     end if
15: end procedure
```

1. Coin exchange problem (4)

Time complexity:

- Choosing a coin with maximum value: $\mathcal{O}(n)$ (using brute-force to get max).
- The while loop is repeated n times (maximum), so the overall complexity is: $\mathcal{O}(n^2)$.
- If the coin is ordered in descending order, the complexity becomes $\mathcal{O}(n)$, because choosing a coin with max value only takes $\mathcal{O}(1)$-time.

2. Activity selection problem

2. Activity selection problem (1)

Problem: given n activities $S = \{1, 2, \ldots, n\}$, that will use a resource (for instance, meeting room, studio, processor, etc.).

Suppose that a resource can only be used to do one activity at one time. Each time an activity occupies a resource, then the other activities cannot use it until the first activity is finished.

Each activity i starts at time s_i and ends at time f_i, where $s_i \leq f_i$. Two activities i and j is called *compatible* if the interval $[s_i, f_i]$ and $[s_j, f_j]$ do not intersect.

Our goal is to choose as many activities as possible that can be served by a resource.

2. Activity selection problem (2)

Example (Activity selection problem)

Given $n = 11$ activities with the starting-ending time as given in the following table:

i	s_i	f_i
1	1	4
2	3	5
3	4	6
4	5	7
5	3	8
6	7	9
7	10	11
8	8	12
9	8	13
10	2	14
11	13	15

2. Activity selection problem (3)

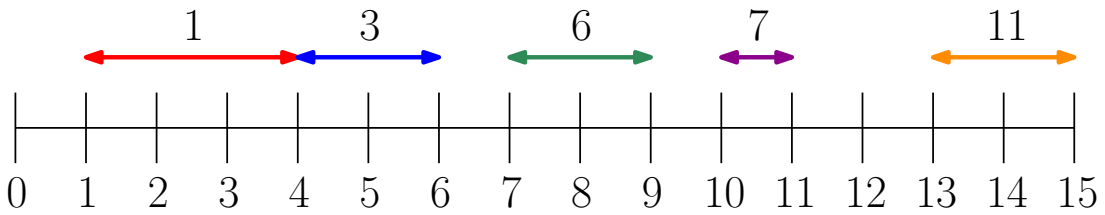
Solution with exhaustive search:

1. List all subsets of the set of n activities.
2. Evaluate each subset, check if the solution is compatible.
3. If yes, then the subset is a solution candidate.
4. Choose the solution candidate with the maximum number of activities
5. The time complexity of the algorithm is $\mathcal{O}(n \cdot 2^n)$. Why?

2. Activity selection problem (4)

The greedy approach:

1. Order the activities based on the end-time in ascending order.
2. At each step, choose the activity whose starting time is greater than or equal to the end-time of the activity that was chosen before



Strategy: at each step, we choose the activity of the smallest index that can be done in the available time-slot.

Set of solution: $\{1, 3, 6, 7, 11\}$

2. Activity selection problem (5)

Algorithm 3 Greedy Activity Selector

```
1:  procedure ActvtySlctr((s_1, ..., s_n), (f_1, ..., f_n))
2:      n ← length(s)                          ▷ s = (s_1, ..., s_n)
3:      A ← {1}                                ▷ A is the solution function
4:      j ← 1
5:      for i ← 2 to n do
6:          if s_i ≥ f_j then                  ▷ s_i: starting time of activity i, f(j): finishing time of activity j
7:              A ← A ∪ {i}
8:              j ← i
9:          end if
10:     end for
11: end procedure
```

Time complexity: $\mathcal{O}(n)$. Why?

2. Activity selection problem (6)

Theorem

The greedy algorithm gives an optimal solution for the activity selection problem.

Proof idea.

- We wanted to show that the schedule, A, chosen by greedy was optimal.
- To do this, we showed that the number of activities in A was at least as large as the number of activities in any other non-overlapping set of activities.
- To show this, we considered any arbitrary, non-overlapping set of activities, B. We showed that we could replace each activity in B with an activity in A.

Proof.

Let $S = \{1, 2, \ldots, n\}$ be the set of activities that are ordered based on the finishing time, A be the optimal solution, and B be the output of the greedy algorithm. Moreover, they are ordered based on the finishing time.

Let a_x be the first activity in A that is different than an activity in B. So:

- $A = a_1, a_2, \ldots, a_{x-1}, a_x, a_{x+1}, \ldots$
- $B = a_1, a_2, \ldots, a_{x-1}, b_x, b_{x+1}, \ldots$

Since B was chosen by the Greedy algorithm, b_x must have a finishing time earlier than the finishing time of a_x.

So, $A' = A - \{a_x\} \cup \{b_x\} = a_1, a_2, \ldots, a_{x-1}, b_x, b_{x+1}, \ldots$ (i.e. replacing a_x with b_x in A) is also a feasible solution.

Continuing this process, we see that we can replace each activity in A with an activity in B. □

Another proof alternatives (*as explained in class*)

Proof.

Let $S = \{1, 2, \ldots, n\}$ be the set of activities that are ordered based on the finishing time.

Suppose that $A \subseteq S$ be an optimal solution, where the elements in A are also ordered based on the finishing time, and the first element of A is k_1.

If $k_1 = 1$, then A is begun with a greedy choice (as in the algorithm). Otherwise, we show that $B = (A - \{k_1\}) \cup \{1\}$ (a solution begin with the greedy choice 1 is another optimal solution).

Since $f_1 \leq f_{k_1}$ (bcs 1 is the first element of S), and the activities in A are compatible, then the activities in B are also compatible. Since $|A| = |B|$, then B is an optimal solution. Hence, there is also an optimal solution that is begun with a greedy choice. □

3. Time minimization in the system

3. Time minimization in the system (1)

Problem: a server (processor, cashier, customer service, etc.) has n clients that must be served. The time to serve client i is t_i. How to minimize the total time in the system (including the waiting time)?

$$T = \sum_{i=1}^{n} \text{time spent in the system}$$

Remark. This problem is equivalent to minimizing the average time of clients in the system.

3. Time minimization in the system (2)

Example

There are three clients with serving time: $t_1 = 5$, $t_2 = 10$, and $t_3 = 3$.

Solution

The possible order of the clients:

- **1,2,3:** $5 + (5 + 10) + (5 + 10 + 3) = 38$
- **1,3,2:** $5 + (5 + 3) + (5 + 3 + 10) = 31$
- **2,1,3:** $10 + (10 + 5) + (10 + 5 + 3) = 43$
- **2,3,1:** $10 + (10 + 3) + (10 + 3 + 5) = 41$
- **3,1,2:** $3 + (3 + 5) + (3 + 5 + 10) = 29$
- **3,2,1:** $3 + (3 + 10) + (3 + 10 + 5) = 34$

3. Time minimization in the system (3)

- Other problems similar to minimizing time in the system is optimal storage on tapes or music storage in a cassette tape (analog system with sequential storage system).
- The programs/musics are saved in the tape sequentially. The length of every song i is t_i (in second/minute). To retrieve and play a song, the tape is initially placed in the beginning.
- If the songs in the tape are saved in the order $X = \{x_1, x_2, \ldots, x_n\}$, then the time needed to play the song x_j to the end is $T_j = \sum_{1 \le k \le j} t_{x_k}$.
- If all songs are often played, the *mean retrieval time / MRT* is $\frac{1}{n} \sum_{1 \le j \le n} T_j$.
- Here we are asked to find a permutation of n songs s.t. if the songs are saved in the tape, then the MRT is minimum. Minimizing MRT is equivalent to minimizing the following:

$$d(X) = \frac{1}{n} \sum_{1 \le j \le n} \sum_{1 \le k \le j} t_{x_k} \quad \text{or} \quad d(X) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} t_{x_k}$$

3. Time minimization in the system (4)

Solution with the exhaustive search

- The order of clients in the system is a permutation. The number of permutation of n elements is $n!$.
- To evaluate the objective function of a permutation needs $\mathcal{O}(n)$-time
- The time complexity of exhaustive search is $\mathcal{O}(n \cdot n!)$.

3. Time minimization in the system (5)

Solution with the greedy algorithm

Strategy: At each step, choose the client that needs the minimum serving time among all clients that have not been served.

Algorithm 4 Clients Scheduling

```
1: procedure ClientsScheduling(n)
2:     S ← {}                                  ▷ S is the solution function
3:     while C ≠ {} do                         ▷ C is the solution candidate
4:         i ← client with minimum t[i] in C   ▷ t_i is the serving time of client i
5:         C ← C − {i}
6:         S ← S ∪ {i}
7:     end while
8:     return S
9: end procedure
```

Time complexity: $\mathcal{O}(n^2)$. Why?

3. Time minimization in the system (6)

If the clients are ordered based on the serving time (in ascending order), then the complexity of the greedy algorithm is $\mathcal{O}(n)$.

Algorithm 5 Clients Scheduling

```
1: procedure ClientsScheduling2(n)
2:     input: clients (1, 2, ..., n) ordered ascendingly based on t_i
3:     for i ← 1 to n do
4:         print(i)
5:     end for
6: end procedure
```

Remark. The greedy algorithm for clients scheduling based on the serving time with ascending order always produces an optimal solution.

3. Time minimization in the system (7)

Theorem

If $t_1 \leq t_2 \leq \cdots \leq t_n$, *then the order* $i_j = j$, $1 \leq j \leq n$ *minimizes:*

$$T = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j}$$

for all possible permutations of i_j, $1 \leq j \leq n$.

Proof available in Ellis Horrowitz & Sartaj Sahni, Computer Algorithms, 1998). See next slide.

3. Time minimization in the system (8)

Proof: Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j} = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$$

If there exist a and b such that $a < b$ and $t_{i_a} > t_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1) t_{i_k} \right] + (n - a + 1) t_{i_b} + (n - b + 1) t_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(t_{i_a} - t_{i_b}) + (n - b + 1)(t_{i_b} - t_{i_a}) \\ &= (b - a)(t_{i_a} - t_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the t_i's can have minimum d. It is easy to see that all permutations in nondecreasing order of the t_i's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. □

to be continued...