Greedy Algorithm

Annu Malik, Anju Sharma, Mr. Vinod Saroha (Guide)

CSE(Network Security), SES BPSMV University, Khanpur Kalan, Sonepat, Haryana

Abstract- This paper presents a survey on Greedy Algorithm. This discussion is centered on overview of Activity Selection Problem and Task Scheduling Problem . A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.Greedy algorithms determine the minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}. The coin of the highest value, less than the remaining change owed, is the local optimum. (Note that in general the change-making problem requires dynamic programming or integer programming to find an optimal solution; However, most currency systems, including the Euro and US Dollar, are special cases where the greedy strategy does find an optimum solution.)

Index Terms- Greedy, scheduling, activity, optimal, algorithm etc

I. INTRODUCTION

Greedy Algorithm solves problem by making the choice that seems best at the particular moment. Many Optimization problems can be solved using a greedy algorithm. Some problems have no efficient solution, but a greedy algorithm may provide an efficient solution that is close to optimal. A greedy algorithm works if a problem exhibit the following two properties:

1) Greedy Choice Property: A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making "greedy" choices.
2) Optimal Substructure: Optimal solutions contains optimal sub solutions. In other words, solutions to sub problems of an optimal solution are optimal.

II. TYPES OF GREEDY ALGORITHM

Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that the greedy algorithm can be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm. There are a few variations to the greedy algorithm:

- Pure greedy algorithms
- Orthogonal greedy algorithms
- Relaxed greedy algorithms

III. AN ACTIVITY SELECTION PROBLEM

Our first example is the problem of scheduling a resource among several competing activities. We shall find that the greedy algorithm provides a well-designed and simple method for selecting a maximum-size of mutually compatible activities.

Suppose S={1,2,.....,n} is the set of proposed activities. The activities share a resource, which can be used by only one activity at a time e.g., a Tennis Court, a Lecture Hall etc. Each activity i has a **start time s_i** and a **finish time f_i,** where $s_i \leq f_i$. If selected, activity i takes place during the half-open time interval$[s_i, f_i\}$. Activities i and j are compatible if the intervals $[s_j, f_j)$ and $[s_i, f_i)$ do not overlap (i.e, I and j are compatible if $s_i \geq$ fj or sj $\geq$ fj).

The activity selection problem selects the maximum-size set of mutually compatible activities.

In this strategy we first select the activity with minimum duration (f_i-s_i) and schedule it. Then, we skip all activities that are not compatible to this one, which means we have to select compatible activities having minimum duration and then we have to schedule it. This process is repeated until all the activities are considered. It can be observed that the process of selecting the activity becomes faster if we assume that the input activities are in order by increasing finishing time:

$f_1 \leq f_2 \leq f_3 \leq ..... \leq f_n$

The running time of an algorithm GREEDY-ACTIVITY-SELECTOR is Θ(nlogn), as sorting can be done in O(nlogn). There are O(1) operations per activity, thus total time is

O(nlogn) + n.O(1) = O(nlogn)

The pseudo code of GREEDY-ACTIVITY-SELECTOR is as follows:

```
GREEDY-ACTIVITY-SELECTOR(s, f)
[1]  n ← length[s]
[2]  A ← {1}
[3]  j ← 1
[4]  for I ← 2 to n
[5]  do if s_i ≥ f_j
[6]  then A ← a Union {1}
[7]  j ← i
```

[8] return A

The GREEDY-ACTIVITY-SELECTOR algorithm gives an optimal solution to the activity selection problem.

Example. Given 10 activities along with their start and finish time as
$S = (A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10})$

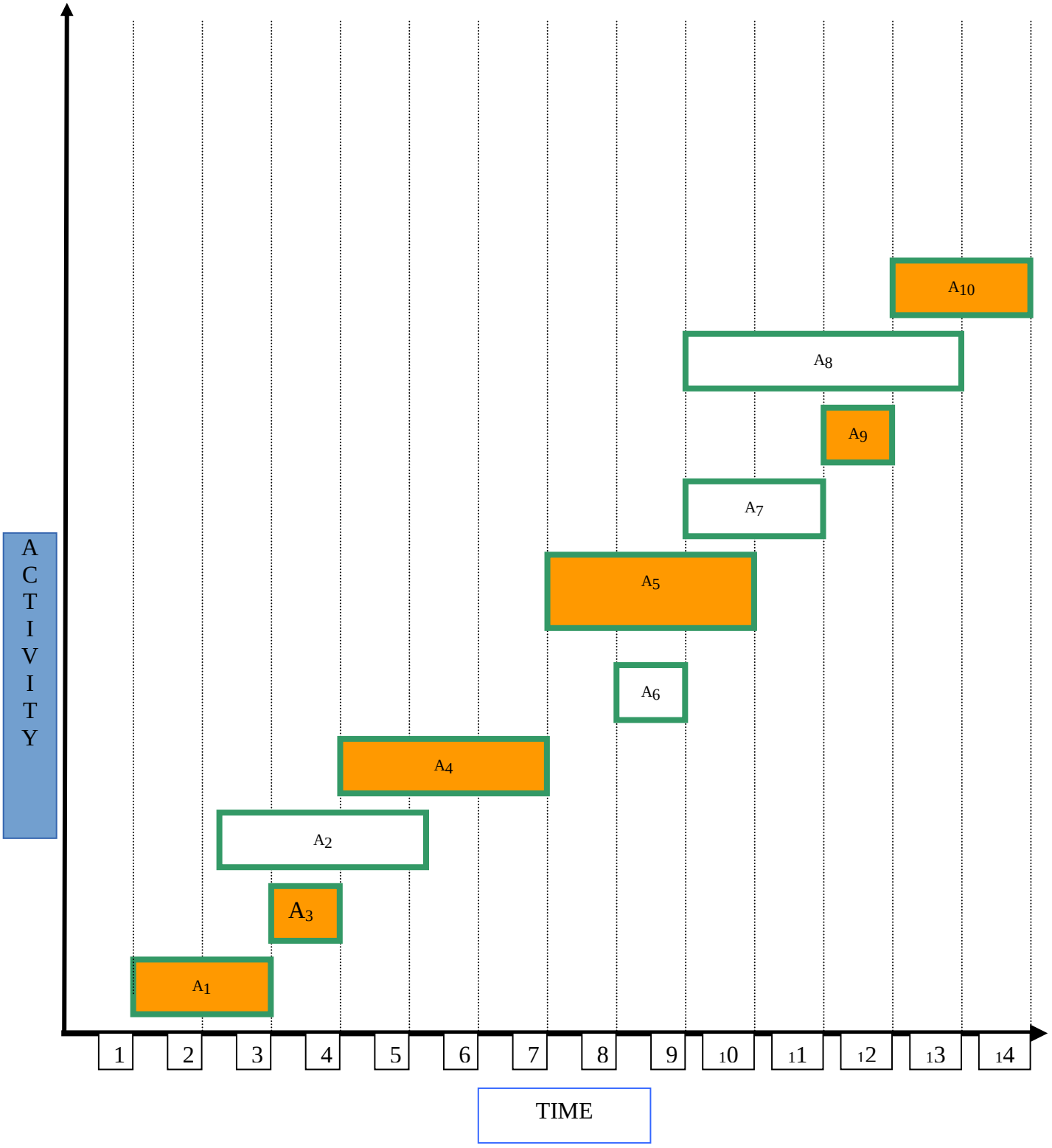
$S_i = (1, 2, 3, 4, 7, 8, 9, 9, 11, 12)$
$F_i = (3, 5, 4, 7, 10, 9, 11, 13, 12, 14)$

Compute a schedule where the largest number of activities takes place.

Solution. The solution for the above activity scheduling problem using greedy strategy is illustrated below.

Arranging the activities in increasing order of finish time.

Activity	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14


ACTIVITY
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
4
5
6
7
8
9
10
11
12
13
14
TIME

Next schedule A_3 , as $A_1$and A_3 are non-interferimg. Next , schedule A_4 as A_1, A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1, A_3, A_4 and A_6 and are not interfering.
Skip A_5 as it is interfering.

Next, schedule A_7 as A_1, A_3, A_4 , A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1, A_3, A_4 , A_6, A_7 and A_9 are non-interfering.
Skip A_8, as it is interfering.

Next, schedule A_{10} as A_1, A_3, A_4 , A_6, A_7, A_9 and A_{10} are non-interfering.
Thus, the final activity schedule is
(A_1, A_3, A_4 , A_6, A_7, A_9, A_{10})

IV. ACTIVITY OR TASK SCHEDULING PROBLEM

This is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a dead line and a penalty that must be paid if the dead line is missed.

A unit time task is a job such as a program to be run on a computer that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2 and so on.

The problem of scheduling unit time tasks with dead lines and penalties for a single processor has the following inputs:

- a set S = {1, 2, 3, ….., n} of n-unit-time tasks.
- A set of n integer dead lines d_1, d_2, d_3, …..d_n such that d_i satisfies $1 \le d_i \le n$ and task I is supposed to finish by time d_i and
- a set of n non-negative weights or penalties w_1, w_2, w_3, ……, w_n such that a penalty w_i, is incurred if task I is not finished by time di and no penalty is incurred if a task i is not finished by time d_i and no penalty is incurred if a task finishes by its dead lines.

Here we find a schedule for S that minimizes the total penalty incurred for missed dead lines.

A task is late in this schedule if it is finished after its dead line. Otherwise, the task is early in the schedule. An arbitrary schedule can always be put into early-first form, in which the early tasks precede the late tasks, i.e., if some early task x follows some late task y, then we can switch the positions of x and y without effecting x being early or y being late.

An arbitrary schedule can always be put into canonical form, in which the early tasks precede the late tasks and the early tasks are scheduled in order of non-decreasing dead lines.

The search for an optimal schedule reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A determined, we can create the actual schedule by listing the elements of A in order of non-decreasing dead line, then listing the late tasks(i.e., S-A) in any order, producing a canonical ordering of the optimal schedule.

A set A of the tasks is independent if there exists a schedule for these tasks such that no tasks are late. So, the set of early tasks for a schedule forms an independent set of tasks 'I' denote the set of all independent sets of tasks.

For any set of tasks A, A is independent if for t = 0, 1, 2, …, n we have
$N_t(A) \le t$
where $N_t(A)$ denote the number of tasks in A whose dead line is t or earlier , i.e., if the tasks in A are scheduled in order of monotonically increasing dead lines, then no task is late.

Example. Let n = 4(P_1, P_2, P_3, P_4) = (100, 10, 15, 27) and (d_1, d_2, d_3, d_4) = (2, 1, 2, 1)
where P_i are profits on processes or job and d_i are dead line of completion. Find the optimal schedule.

Solution. Max. dead line is 2 so max. number of processes that are scheduled is 2.

Feasible Solution	Processing Sequence	Value
(1, 2)	(2, 1)	10 + 100 = 110
(1, 3)	(1, 3) or (3, 1)	100 + 15 = 115
(1, 4)	(4, 1)	27 + 100 = 127
(2, 3)	(2, 3)	25
(3, 4)	(4, 3)	42
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27

Thus, the optimal schedule is (4, 1) and profit 127.

V. APPLICATIONS

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, Dijkstra's algorithm for finding single-source shortest paths, and the algorithm for finding optimum Huffman trees.

VI. CONCLUSION

Greedy algorithms are usually easy to think of, easy to implement and run fast. Proving their correctness may require rigorous mathematical proofs and is sometimes insidious hard. In addition, greedy algorithms are infamous for being tricky. Missing even a very small detail can be fatal. But when you have nothing else at your disposal, they may be the only salvation. With backtracking or dynamic programming you are on a relatively safe ground. With greedy instead, it is more like walking on a mined field. Everything looks fine on the surface, but the hidden part may backfire on you when you least expect. While there are some standardized problems, most of the problems solvable by this method call for heuristics. There is no general template on how to apply the greedy method to a given problem, however the problem specification might give you a good insight. In some cases there are a lot of greedy assumptions one can make, but only few of them are correct. They can provide excellent challenge opportunities.

REFERENCES

[1] Wikipedia
[2] Google
[3] Algorithms Design and Analysis by Udit Agarwal

AUTHORS

First Author – Annu Malik, M.Tech (Persuing), CSE(Network Security), SES BPSMV Uniersity, Khanpur Kalan, Sonepat, Haryana and annu1990malik@gmailcom.
Second Author – Anju Sharma, M.Tech (Persuing), CSE(Network Security), SES BPSMV Uniersity, Khanpur Kalan, Sonepat, Haryana and anjusharma264@gmailcom.
Third Author – Mr. Vinod Saroha (Guide), M.Tech(Assistant Professor in CSE and IT Department), SES BPSMV Uniersity, Khanpur Kalan, Sonepat, Haryana) and vnd.saroha@gmailcom.

Correspondence Author – Annu Malik, annu1990malik@gmail.com, sbit.cse08408@gmail.com), 08813034437.