**SUBJECT**: DESIGN AND ANALYSIS OF ALGORITHMS

**CODE**: 503040

Duration: 150 minutes

Allowed to use materials.

**LAB 08: Greedy Technique**

# Objectives

Understand the properties of Greedy Technique for algorithm design

Be able to design, implement, and analyze Greedy algorithms solving common problems.

# Idea

Constructs a solution to an *optimization problem* piece by

piece through a sequence of choices that are:

* *feasible, i.e. satisfying the constraints*
* *locally optimal (with respect to some neighborhood definition)*
* *greedy (in terms of some measure), and irrevocable*

For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations. We are mostly interested in the former case in this class.

## An example of a Greedy algorithm

Implement and analyze a Greedy algorithm to give change for a specific amount n with the least number of coins of the denominations d1 > d2 > . . . > dm.

The implementation in Python is presented as follows

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **def** **make\_change**(n, d):  """  gives change for a specific amount n with the least number of coins  of the denominations d1 > d2 > . . . > dm  input:  n(int) - amount of money to be taken  d(array of int) - array of coin denominations  output:  coins(array of int) - array of coin numbers corresponding to d  """  m = len(d)  coins = [**0** **for** \_ **in** range(m)]  **for** i **in** range(m):  coins[i] = n // d[i]  n = n % d[i]  **return** coins  #testcase  d = [**25**, **10**, **5**, **1**]  n = **48**  **print**(make\_change(n, d))  #output: [1, 2, 0, 3] | |

Analysis:

1/ Basic operation: assignment on line 14

2/ Worst case: as average case

3/Counting the number of basic operations in the worst case:

…

**Time efficiency**

***T*(*m*) = *m* ∈ Θ(*m*)**

# Exercises

For each problem in this part, implement (in Python) and analyze a greedy algorithm to solve the problem.

**Warm up**

1. Activity Selection Problem

Given a set of activities, along with the starting and finishing time of each activity, find the maximum number of activities performed by a single person assuming that a person can only work on a single activity at a time.

For example,

Input: Following set of activities  
   
(1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13), (12, 14)  
   
Output:  
   
(1, 4), (5, 7), (8, 11), (12, 14)

Hint: The idea is to initially sort the activities in increasing order of their finish times and create a set S to store the selected activities and initialize it with the first activity.

1. Job Sequencing Problem with Deadlines

Given a list of tasks with deadlines and total profit earned on completing a task, find the maximum profit earned by executing the tasks within the specified deadlines. Assume that each task takes one unit of time (for example: from 0 to 1, from 1 to 2, from 2 to 3,…) to complete, and a task can’t execute beyond its deadline. Also, only a single task will be executed at a time.

For example, consider the following set of tasks with a deadline and the profit associated with it. If we choose tasks 1, 3, 4, 5, 6, 7, 8, and 9, we can achieve a maximum profit of 109. Note that task 2 and task 10 are left out.



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5(25) | 6(20) | 3(18) | 1(15) | 8(10) |  |  |  |  |  |

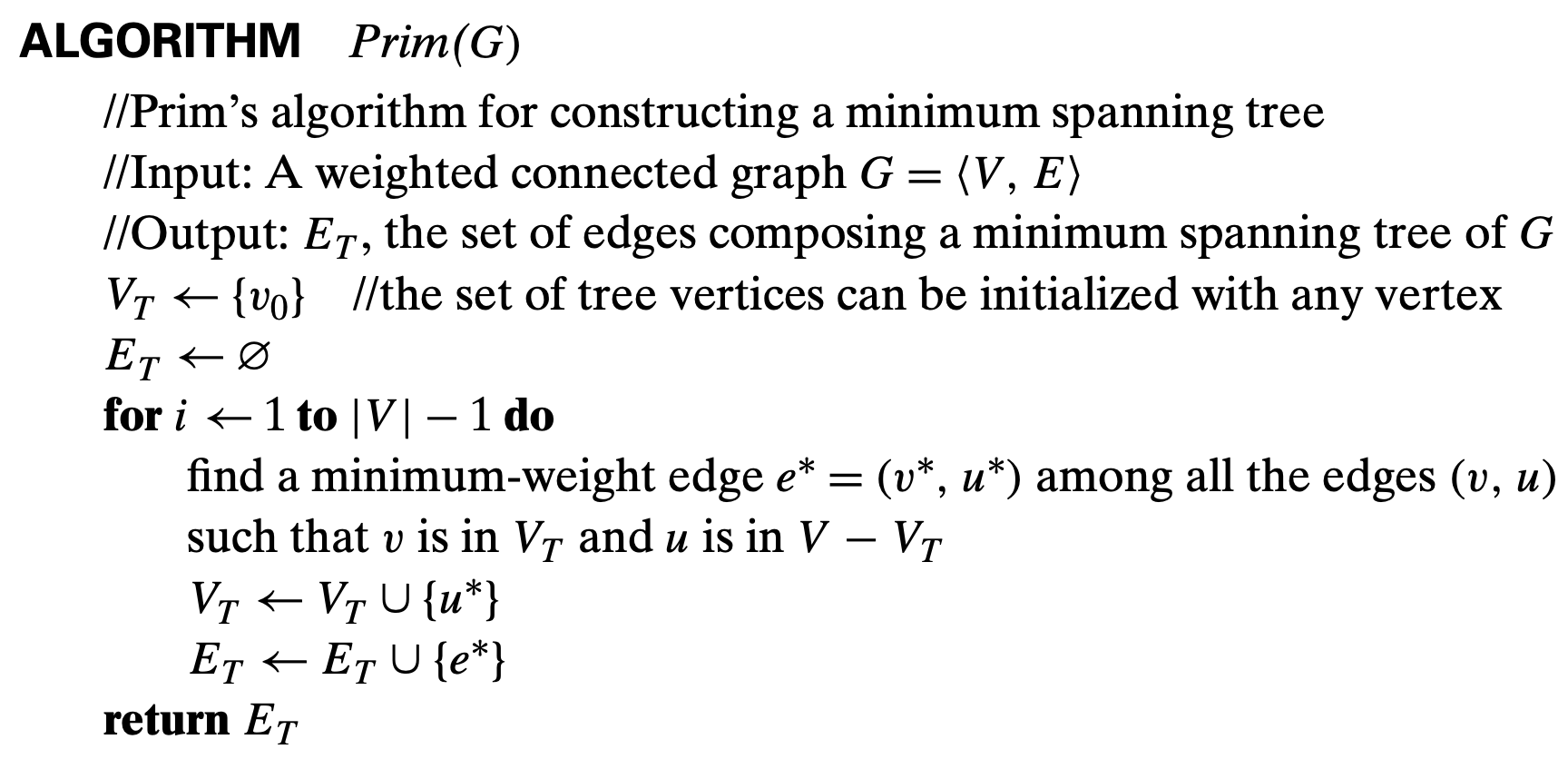
Hint: The idea is simple – consider each task in decreasing order of their profits and schedule it in the latest possible free slot that meets its deadline. If no such slot is there, don’t schedule the task.

**Intermediate exercises**

The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

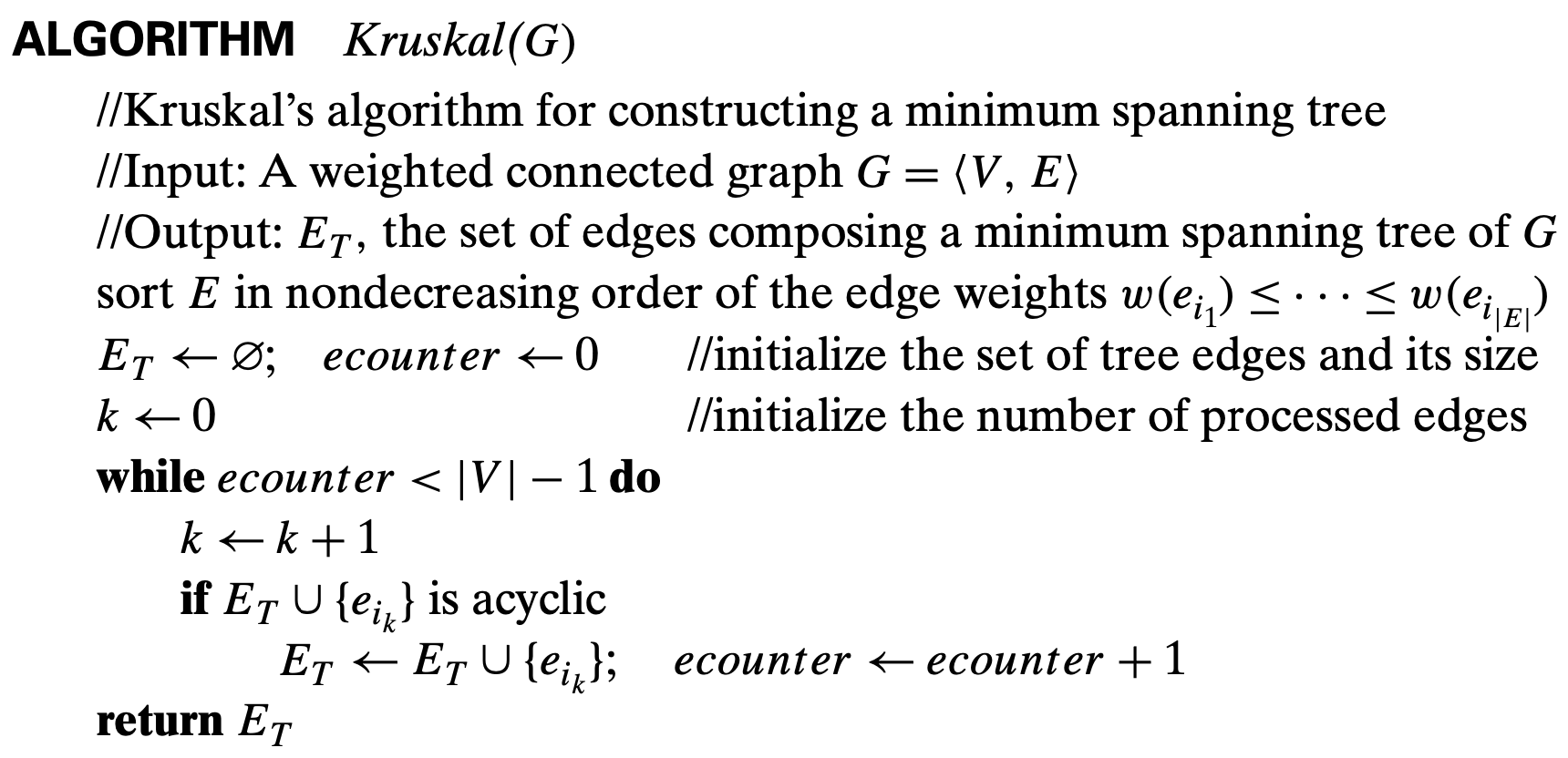
1. Prim’s Algorithm

Prim’s algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph’s vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph’s vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n − 1, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.



1. Kruskal’s Algorithm

The algorithm begins by sorting the graph’s edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.



References:

1. Introduction to The Design & Analysis of Algorithms, 3rd edition, ISBN 978-0-13-231681-1 by Anany Levitin published by Pearson Education © 2012.
2. https://www.techiedelight.com/