# Algorithms & Data Structures II (course 1DL231)
# Uppsala University – Autumn 2020
# Report for Assignment 1 by Team 23

William EULAU Adam ERIKSSON

13th November 2020

## Part 1
# Weightlifting problem

## A Introduction

With a brute-force running time of $\mathcal{O}(2^n)$, by testing all possible combinations of the given set, a lower-complexity algorithm is of course preferable. Applying a dynamic programming approach, an optimal substructure can be proved and recursion derived. Given a set $P$ and a target weight $w$, there could exist a subset $P'$, of length $n$, which adds up to the weight. If an item, $k$, is removed from the subset which adds up to $w$, there will exist a subset of length $n-1$ which will add up to $w - w_k$, where $w_k$ is the weight of the item $k$. This optimal substructure shows that the optimal solution can be constructed efficiently from the optimal solution of it's sub-problems, and proves that the bottom-up method is suitable for a dynamic programming implementation.

Starting from zero plates in $P$ considered, a boolean value represents if there is a subset of $P$ which adds up to each weight from 0 to $w$. Sequentially, considering one plate at the time, a matrix of size $(n+1) \times (w+1)$ will be filled, where $n$ is the length of the set $P$ and $w$ is the desired sum, or, in this case, weight. This matrix, enabling a dynamic programming implementation, is presented in the following section.

## B Dynamic Programming Implementation

To initialize the $(n+1) \times (w+1)$ matrix, the base cases need first to be considered. In our example, we have two, which are introduced in the following sentences. Given from 0 to $n$ number of plates, there will be a subset which adds up to zero, when all considered plates are excluded. Therefore, the first column $[i][0]$ of the matrix is filled with the boolean value $True$. The other base case is given no plates, the plates will not add up to all sums larger than zero. This results in the first row $[0][j]$ is filled with the value $False$, except for the $j = 0$ as zero plates can add up to sum of zero. The initialized matrix is represented in Figure 1.

| | | Sum | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | w-3 | w-2 | w-1 | w |
| | 0 | T | F | F | F | F | F | F | F | F | F | F | F |
| | 1 | T | None | None | None | None | None | None | None | None | None | None | None |
| **no. of elements** | ... | T | None | None | None | None | None | None | None | None | None | None | None |
| | n-1 | T | None | None | None | None | None | None | None | None | None | None | None |
| | n | T | None | None | None | None | None | None | None | None | None | None | None |

Figure 1: The initialized dynamic programming matrix of size $w+1 \times n+1$, with base cases implemented, where T symbolises boolean value $True$ and F $False$.

Then, the matrix will can be filled with boolean values according to a recursion as follows. Starting from the upper-left most undefined value, at index [1][1], boolean values $b$ are assigned, filling up every row before moving to the next. $i$ represents $k_i$, the i:th item of $P$, and $j$ represents a sum $j \leq w$.

$$b[i][j] = \begin{cases} b[i-1][j] & (\text{if } j < k_i) \\ b[i-1][j] \;||\; b[i-1][j-k_{i-1}] & (\text{if } j \geq k_i) \end{cases}$$

In layman terms, the boolean value $b$ answers the question: "with the first ($i$-1) plates to choose from, can one obtain the weight sum of $j$?". $j \geq k_i$, $b$ is decided by an OR-operator, $||$. If either there is a sum $j$ achievable or there is a sum $j - k_{i-1}$ not considering $k_i$, the sum $j$ is achievable considering the plate $k_i$. Per defintion of the OR-operator, if either one of the sums are $True$, then $b$ will also be $True$. When all values of the $n+1 \times w+1$ matrix are filled, the solution is found in the bottom-right corner, at index $[i][j] = [n][w]$. The recursion code implementation is presented below.

```
#initialize initial values of matrix
for i in range(len(plate_list) + 1):
    dp_matrix[i][0] = True

for i in range(1, weight + 1):
    dp_matrix[0][i] = False

#fill matrix with remaining values
for i in range(1, len(plate_list) + 1):
    for j in range(1, weight + 1):
        if j < plate_list[i - 1]: #according to assignment report
            dp_matrix[i][j] = dp_matrix[i - 1][j]
        if j >= plate_list[i - 1]:
            dp_matrix[i][j] = (dp_matrix[i - 1][j] or
                               dp_matrix[i - 1][j - plate_list[i - 1]])

return dp_matrix[len(plate_list)][weight]
```

Listing 1: Code implementation of base case initialization and recursion to fill the dynamic programming matrix.

## C Returning Subset

With the boolean matrix constructed, a reverse walk can be implemented to identify and return the optimal subset and solution of the problem. Starting at the bottom-right corner, the matrix

is traversed in backwards fashion, observing if the i:th element of $P$, $k_i$, was used to sum up the final weight $w$. If there is a solution, the last item of $P$, $k_n$ for the final weight $w$ has value $b[n][w] = True$. If the boolean value for the final weight of the second to last item, $k_{n-1}$, is also True, then the final item $k_i$ was not used to sum the final weight. Otherwise, $k_n$ was not excluded in the sum, and $k_n$ would be appended to solution subset, and the weight updated as $w = w - k_n$. This is done sequentially for all items $k$ of the set $P$, until the updated weight is $w = 0$. If the last value of the matrix is not $True$, there is no solution to the problem, and an empty set is returned. This implementation is presented as code below.

```
1    dynamicWeight = weight #initialize a dynamic weight, which will decrease as /
         the reverse walk proceeds
2    subset = []
3    for i in range((len(plate_list)),0,-1):
4        #start from last index
5        if dp_matrix[i][dynamicWeight] == True and dp_matrix[i][dynamicWeight] != /
             dp_matrix[i-1][dynamicWeight]:
6            #if the boolean values are different, the i:th item of the set was used
7            weightUsed = plate_list[i-1]
8            subset.append(weightUsed)
9            dynamicWeight -= weightUsed
10   return set(subset)
```

Listing 2: Code implementation of reverse walk of dynamic programming matrix to obtain which subset $P'$ sums up to the weight $w$.

## D Time Complexity Analysis

In contrast of the brute-force $\mathcal{O}(2^n)$ algorithm, we argue that the dynamic programming significantly improves the running time, to $\mathcal{O}(n \cdot w)$. Instead of computing the sum of every possible subset, a matrix of size $(n+1) \times (w+1)$ is constructed. To fill the matrix $(n+1) \cdot (w+1)$ iterations are required. As the recursive call, filling the matrix is obtainable within *scalar* time for every iteration, the matrix construction is done within the time complexity $\mathcal{O}(n \cdot w)$. Initializing the base cases and returning the subset solution are all done at a lower time complexity than filling the matrix, and therefore do not affect the upper-bound time complexity.

# Part 2
# Ring problem

## A Introduction

In order to solve the task of finding ring structures in undirected graphs, a modified Depth-First Search (DFS) algorithm is implemented. A full DFS, according to the course literature, will keep track of timestamps and visit all available nodes in sequence and mark them as finished when all of the neighboring nodes has been visited and in turn finished. However, the full DFS implementation in this case provides redundant information under a too large time complexity. In the this and the following sections, we will present a modified DFS-algorithm which finds ring structures in undirected graphs and argue that it takes time $\mathcal{O}(|V|)$, where $V$ is the number of vertices, or nodes, in the graph. A cycle is detected if an edge is pointing to an already visited non-parent node (from now on referred to as the target node). Two dictionaries are created; one storing boolean values of if nodes of the graph have been visited and one storing the parent

node of each node. The algorithm starts by initiating the two dictionaries described above and mark each node as not visited ($False$) and without parent ($None$). As shown in the code below, these iterations in for loops take time $\mathcal{O}(n)$ where $n$ represents the number of nodes in the graph. A stack is created, initialized as an empty list, to ensure the depth-first property of the modified DFS algorithm. New found nodes are appended to the stack, and pursued before other nodes according to the First In Last Out nature of stacks.

```
visited = dict()
parent = dict()
stack = []
for node in nodes:
    visited[node] = False
    parent[node] = None
```

Listing 3: Dictionary and stack initialization of modified DFS-algorithm.

A loop over all nodes is run to perform the DFS. If the stack is empty, a not visited node will be selected. Otherwise, the last node of the stack will be examined. The selected node is then set as visited, the non-parent neighboring nodes are marked as the children of the selected node and appended to the stack. If an already visited, non-parent, node is detected, a cycle is detected and the loop is broken. If the loop iterates through all nodes without detecting a cycle, there is no cycle and the function will return $False$.

```
iterations = len(nodes)
#backward track from the graph nodes, taking the last index of nodes. All /
    nodes which are visited,
#are updated in visitedd dictionary, but also removed from nodes list.
for i in range(iterations-1,-1,-1):
    if(len(stack) != 0):
        #take from stack
        node = stack.pop()
    else:
        #take node instead
        node = nodes[i]
    visited[node] = True
    neighbors = G.neighbors(node)
    for neighbor in neighbors:
        if(parent[node] != neighbor): #if neighbor isn't parent
            if(visited[neighbor]): #if neighbor is already visited
                #Cycle detected!
                return True
            else:
                parent[neighbor] = node
                stack.append(neighbor)
                #explore neighbor first, by appendding in the stack (stack is /
                    prioritized)
    nodes.remove(node) #removal when node is visited
return False
```

Listing 4: Modified depth-first search algorithm detecting cycles in time $\mathcal{O}(|V|)$.

## B Extended Algorithm Returning Ring Edges

As stated above, the ring algorithm returned a boolean value specifying if there is a cycle in the undirected graph. The algorithm is extended to return the edges creating the found ring structure, if there is one. A modified ring function - in the code named "ring_mod" - is created to return the necessary information to extract the edges of the ring structure; specifically, the

current node, the already visited non-parent detected node (or, the target node), and the parent dictionary. The return statements of the modified function is presented below.

```
17                return True, node, neighbor, parent

23    return False, None, None, None #hasCycle, node, neighbor, parent
```

Listing 5: Return statements in the modified ring algorithm. The first row is implemented if there is a cycle in the graph and the second row if there is None.

The returned information is used, in the function "ring_extended", to track the cycle backwards, until it has through a parent node track-back reached the target node. Rebuilding the ring structure takes $\mathcal{O}(m)$ time, where $m$ are the nodes spanning the ring, and are less or equal to the total amount of nodes in the graph.

```
1    hasCycle, node, target, parent = ring_mod(G) #get if there was a cycle, or ring,
2    if(hasCycle): #the node where the cycle was found,
3       #trackback #the cycle connecter, and the parent of the node
4       completeLoop = False
5       loop = [(target,node)] #initialize loop
6       while not completeLoop: #until loop is complete, iterate
7          loop.append((node,parent[node]))
8          node = parent[node] #update node
9          if node == target:
10            completeLoop = True #if cycle is complete
11      return (True,loop)
12   else:
13      return (False,[]) #return empty list is there is no ring
```

Listing 6: Code implementation of of backtracking in order to return full cycle.

## C Time Complexity Analysis

Using a modified DFS algorithm, a ring structure is detected by visiting every node of the input graph. Two dictionary data-structures are initialized to save information on every node. Iterating over every node of the graph, the data-structure are filled and neighbors identified in *scalar* time. Therefore, the time complexity of the ring functions is $\mathcal{O}(|V|)$ where $V$ are the vertices, or nodes, of the graph. In contrast of the full DFS algorithm, running on $\mathcal{O}(|V| + |E|)$ time, considering all nodes and edges, the ring function only visits all nodes once. There could be at most $n - 1$ tree edges in a cyclic graph with $n$ nodes. This is because a ring structure in the graph is at the latest detected after visiting the last node. In sum, all recursion and data-structure initialization are completed under $\mathcal{O}(|V|)$ time, creating a less time consuming algorithm than the full DFS, finding ring structures only considering the vertices of the graph.

# Part 3
# Code Appendix

## A Weightlifting Problem

```
from src.weightlifting_data import data # noqa
from typing import List, Set # noqa
import unittest # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger("put name here")
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa

__all__ = ['weightlifting', 'weightlifting_subset']


# In the function head below, "P" is the set of weights and "weight" is the / weight we are
# trying to achieve.
def weightlifting(P: Set[int], weight: int) -> bool:
    '''
    Sig: Set[int], int -> bool
    Pre:
    Post:
    Ex: P = {2, 32, 234, 35, 12332, 1, 7, 56}
         weightlifting(P, 299) = True
         weightlifting(P, 11) = False
    '''
    plate_list = list(P)

    #initialize matrix
    dp_matrix = [
        [None for i in range(weight + 1)] for j in range(len(plate_list) + 1)
    ]

    #initialize initial values of matrix
    for i in range(len(plate_list) + 1):
        dp_matrix[i][0] = True

    for i in range(1, weight + 1):
        dp_matrix[0][i] = False

    #fill matrix with remaining values
    for i in range(1, len(plate_list) + 1):
        for j in range(1, weight + 1):
            if j < plate_list[i - 1]: #according to assignment report
                dp_matrix[i][j] = dp_matrix[i - 1][j]
            if j >= plate_list[i - 1]:
                dp_matrix[i][j] = (dp_matrix[i - 1][j] or
                                   dp_matrix[i - 1][j - plate_list[i - 1]])

    return dp_matrix[len(plate_list)][weight]

#test with reverse walk
def weightlifting_subset(P: Set[int], weight: int) -> bool:
```

```
    '''
    Sig: Set[int], int -> bool
    Pre:
    Post:
    Ex: P = {2, 32, 234, 35, 12332, 1, 7, 56}
         weightlifting(P, 299) = True
         weightlifting(P, 11) = False
    '''
    plate_list = list(P)
    #first part copied from weightlifting fucntion code.
    #First the matrix is initialized. Then a reverse walk of the matrix is performed.
    dp_matrix = [
        [None for i in range(weight + 1)] for j in range(len(plate_list) + 1)
    ]

    for i in range(len(plate_list) + 1):
        dp_matrix[i][0] = True

    for i in range(1, weight + 1):
        dp_matrix[0][i] = False

    for i in range(1, len(plate_list) + 1):
        for j in range(1, weight + 1):
            if j < plate_list[i - 1]:
                dp_matrix[i][j] = dp_matrix[i - 1][j]
            if j >= plate_list[i - 1]:
                dp_matrix[i][j] = (dp_matrix[i - 1][j] or
                                   dp_matrix[i - 1][j - plate_list[i - 1]])

    #matrix constructed, reverse walk if there is a subset.
    dynamicWeight = weight #initialize a dynamic weight, which will decrease as the reverse walk proceeds
    subset = []
    for i in range((len(plate_list)),0,-1):
        #start from last index
        if dp_matrix[i][dynamicWeight] == True and dp_matrix[i][dynamicWeight] != dp_matrix[i-1][dynamicWeight]:
            #if the boolean values are different, the i:th item of the set was used
            weightUsed = plate_list[i-1]
            subset.append(weightUsed)
            dynamicWeight -= weightUsed
    return set(subset)
```

Listing 7: Full code supplied in weightlifting.py.

# B Ring Problem

```
from typing import Set, Tuple # noqa
import unittest # noqa
from src.graph import Graph # noqa
from src.ring_data import data # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger("put name here")
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa

```

```
13 __all__ = ['ring', 'ring_extended']
14
15
16 def ring(G: Graph) -> bool:
17     """
18     Sig: Graph G(V, E) -> bool
19     Pre:
20     Post:
21     Ex: Sanity tests below
22          ring(g1) = False
23          ring(g2) = True
24     """
25     #modified ring(G) function, returning a bool, but also the current node,
26                                        #the node parent and cycle connector node
27     nodes = G.nodes.copy()
28     #initialize dictionaries O(n)
29     visited = dict()
30     parent = dict()
31     stack = []
32     for node in nodes:
33        visited[node] = False
34        parent[node] = None
35
36     #check all nodes
37     iterations = len(nodes)
38     #backward track from the graph nodes, taking the last index of nodes. All /
           nodes which are visited,
39     #are updated in visitedd dictionary, but also removed from nodes list.
40     for i in range(iterations-1,-1,-1):
41        if(len(stack) != 0):
42           #take from stack
43           node = stack.pop()
44        else:
45           #take node instead
46           node = nodes[i]
47        visited[node] = True
48        neighbors = G.neighbors(node)
49        for neighbor in neighbors:
50           if(parent[node] != neighbor): #if neighbor isn't parent
51              if(visited[neighbor]): #if neighbor is already visited
52                 #Cycle detected!
53                 return True
54              else:
55                 parent[neighbor] = node
56                 stack.append(neighbor)
57                 #explore neighbor first, by appendding in the stack (stack is /
                       prioritized)
58        nodes.remove(node) #removal when node is visited
59     return False
60
61
62 def ring_extended(G: Graph) -> Tuple[bool, Set[Tuple[str, str]]]:
63     """
64     Sig: Graph G(V,E) -> Tuple[bool, List[Tuple[str, str]]]
65     Pre:
66     Post:
67     Ex: Sanity tests below
68          ring_extended(g1) = False, []
69          ring_extended(g2) = True, [('a','c'),('c','f'),
70                                     ('f','h'),('h','g'),('g','d'),('d','f'),
```

```
71                              ('f','a')]
72      """
73      hasCycle, node, target, parent = ring_mod(G) #get if there was a cycle, or ring,
74      if(hasCycle): #the node where the cycle was found,
75          #trackback #the cycle connecter, and the parent of the node
76          completeLoop = False
77          loop = [(target,node)] #initialize loop
78          while not completeLoop: #until loop is complete, iterate
79              loop.append((node,parent[node]))
80              node = parent[node] #update node
81              if node == target:
82                  completeLoop = True #if cycle is complete
83          return (True,loop)
84      else:
85          return (False,[]) #return empty list is there is no ring
86
87  def ring_mod(G: Graph) -> Tuple[bool, str, str, dict]:
88      #modified ring(G) function, returning a bool, but also the current node,
89                                          #the node parent and cycle connector node
90      nodes = G.nodes.copy()
91      #initialize dictionaries O(n)
92      visited = dict()
93      parent = dict()
94      stack = []
95      for node in nodes:
96          visited[node] = False
97          parent[node] = None
98
99      #check all nodes
100     iterations = len(nodes)
101     #backward track from the graph nodes, taking the last index of nodes. All /
            nodes which are visited,
102     #are updated in visitedd dictionary, but also removed from nodes list.
103     for i in range(iterations-1,-1,-1):
104         if(len(stack) != 0):
105             #if there is a node in the stack, take it
106             node = stack.pop()
107         else:
108             #take a node instead
109             node = nodes[i]
110
111         visited[node] = True
112         neighbors = G.neighbors(node)
113         for neighbor in neighbors:
114             if(parent[node] != neighbor): #if neighbor isn't parent
115                 if(visited[neighbor]): #if neighbor is already visited
116                     #Cycle detected!
117                     return True, node, neighbor, parent
118                 else:
119                     parent[neighbor] = node
120                     stack.append(neighbor)
121                     #explore neighbor first, by appendding in the stack (stack is /
                            prioritized)
122         nodes.remove(node) #removal when node is visited
123     return False, None, None, None #hasCycle, node, neighbor, parent
```

Listing 8: Full code supplied in ring.py.