Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2021
Report for Assignment 1 by Team 15

Jakob Nordgren Aljaz Kovac

19th November 2021

Part 1

The Weightlifting Problem

We want to define the optimal solution as $OPT : i \times w \longrightarrow \{True, False\}$, where $i \in \{1, ..., n\}$, and w is the sum we are looking for. We first observe that if $w = 0$, then $P' = \emptyset$ is a solution. Further, if $P = \emptyset$ and $w > 0$ then there obviously cannot be a solution set. Now for $P = \{w_1, ..., w_n\}$, we want to know if there exists a set $P' \subseteq P$ such that:

$$w = \sum_{w_j \in P'} w_j.$$

Say that there is a solution. Then for any $w_i \in P$, either w_i is in the solution set or it is not. If $w_i \in P'$, then there must be a solution for a subproblem of size $n-1$ with $w' = w - w_i$, that is $OPT(i-1, w-w_i)$. If $w_i \notin P'$ then the solution must be in $OPT(i-1, w)$. In either case, we have subproblems of strictly smaller size.

A Recursive equation

This problem can be represented with the following recursive equation:

$$OPT(i, w) = \begin{cases} True & \text{if } w = 0 \\ False & \text{if } i = 0, w > 0 \\ OPT(i-1, w) & \text{if } w < w_i \\ OPT(i-1, w-w_i) \text{ or } OPT(i-1, w) & \text{otherwise} \end{cases}$$

The above equation defines the problem in terms of strictly smaller subproblems. Thus this problem does indeed have the optimal substructure property.

For $i > 0, w > 0$, $OPT(i, w)$ will in the worst case expand to 2^i instances of $OPT(i-j, w)$ where $j \in \{0, .., i\}$. For example, a possible expansion of $OPT(3, w)$ could be:

$$\begin{aligned} & OPT(3, w) \\ =& OPT(2, w-w_3) \; or \; OPT(2, w) \\ =& OPT(1, w-w_3-w_2) \; or \; OPT(1, w-w_3) \; or \; OPT(1, w-w_2) \; or \; OPT(1, w) \\ =& OPT(0, w-w_3-w_2-w_1) \; or \; OPT(0, w-w_3-w_2) \; or \\ & OPT(0, w-w_3-w_1) \; or \; OPT(0, w-w_3) \; or \\ & OPT(0, w-w_2-w_1) \; or \; OPT(0, w-w_2) \; or \\ & OPT(0, w-w_1) \; or \; OPT(0, w) \end{aligned}$$

It's clear that we will compute $OPT(i, w')$ with i^2 "different" values of w'. Now let s be the sum of all $a_i \in P$. Then for large enough problem sizes (meaning a large enough number of elements in P), and $s < w < i^2$, since there cannot be more than w distinct sums, we will have overlapping subproblems. Another way to demonstrate instances where subproblems are overlapping would be to imagine that $P = \{1, 2, 3\}$ in the example above. Then we would for example compute $OPT(0, w-3)$ twice. Dynamic programming can thus be used to solve this problem in an efficient way.

B Bottom-Up Iteration and Top-Down Recursion

In part A, we defined an optimal solution $OPT : i \times w \longrightarrow True, False$, as a recursive equation. This could be implemented either with bottom-up iteration or top-down recursion with the same time complexity. We chose to use bottom-up iteration and the optimal solution is computed by populating a table over the possible inputs to $OPT(i, w)$. Given the base cases of the recursive equation we will begin with a table that looks like this:

i	T			
$i-1$	T			
...	T			
0	T	F	F	F
	0	...	$w-1$	w

Our program, shown in Listing 1 iterates over all the entries in this table from left to right, starting at the bottom row. By computing the table values in this way, we make sure to always solve the smallest possible subproblem and store the result so it can be used while computing larger subproblems. This is a typical example of bottom-up iteration using dynamic programming.

C Extending the algorithm

To return the elements of P', we will build a simple algorithm that traverses the populated OPT table. To do this, we first check the value of entry i, w. If $OPT(i, w) = False$, then $P' = \emptyset$, otherwise we find the smallest $j \leq i \ s.t. \ OPT(j, w) = True$, which is anologous to finding the bottom-most T in column w of our table. Since $OPT(j, w) = True$, but $OPT(j-1, w) = False$, we can add w_j to P'. Then, the remaining elements of P' must be the set that is the optimal solution to $OPT(j-1, w-w_j)$, so we find the smallest $j' \leq j \ s.t. \ OPT(j', (w-w_j)) = True$, add w'_j to P' and so on until we have $OPT(j'', 0) = True$, and now P' contains the optimal solution. Listing 2 shows the extension part of the algorithm.

D Time complexity analysis

The algorithm for determining the existence of a subset sum will perform a fixed amount of $O(1)$ operations for each combination of (i, w). So the time complexity is $O(nw)$. Determining the P' will take at most i steps, so this operation is linear, i.e. $O(n)$. Therefore the time complexity for the extended algorithm is $O(2nw) = O(nw)$

Part 2
Augmenting Path Detection in Network Graphs

A Design and implementation of the algorithm

We chose to use a depth first search, with the extra criteria that en edge (u, v) can only be included in the search path if $capacity(u, v) > flow(u, v)$. As shown in Listing 3, we

```

def weightlifting(P: Set[int], weight: int) -> bool:
   '''
   Sig: Set[int], int -> bool
   Pre: Elements of 'P' and 'weight' are non-negative integers
   Post: Return True if there is a subset of 'P' that sums to 'weight',
         False otherwise
   Ex: P = {2, 32, 234, 35, 12332, 1, 7, 56}
        weightlifting(P, 299) = True
        weightlifting(P, 11) = False
   '''
   plate_list = list(P)
   # Initialise the dynamic programming matrix
   dp_matrix = [
      [None for i in range(weight + 1)] for j in range(len(plate_list) + 1)
   ]

   n=len(plate_list)

   # Build matrix dp_matrix[][] in bottom up manner
   for i in range(n + 1):
      # Variant: n - i
      for j in range(weight + 1):
         # Variant: weight - j
         # Invariant: dp_matrix[0..i][0..j] contains already computed solutions
         if j == 0:
            dp_matrix[i][j]=True
         elif (i==0 and j>0):
            dp_matrix[i][j]=False
         elif plate_list[i-1]>j:
            dp_matrix[i][j]=dp_matrix[i-1][j]
         else: dp_matrix[i][j]=dp_matrix[i-1][j-plate_list[i-1]] or /
             dp_matrix[i-1][j]
   return dp_matrix[n][weight]
```

Listing 1: Python implementation of the solution to the weightlifting problem, using bottom-up iteration.

implemented a nested function that uses a top-down approach to recursively visit all neighbors of nodes until either all nodes are visited or until it encounters the sink node. A neighbor node u of a node v is only visited if it has not been visited before and there is an edge (u, v) that has a capacity greater than the flow.

B Extended algorithm

Listing 4 shows the extended algorithm. Here we keep a list of edges that start off empty. If, while visiting node v, the recursive call visiting the neighboring node u returns true we add the edge (v, u) to the beginning of the path list. Thus, when the algorithm is finished, the path is either empty, meaning that no recursive call yielded $True$, or it will contain all edges in the path from source to sink, in order.

```
    # Now backtrack through the matrix and pick out the weights that were included
    # in the solution, adding them to the solution_set
    solution_set = set()
    # The solution is in the top right-corner of the matrix
    res = dp_matrix[n][weight]
    # Initialize the row index to the weight given
    j = weight

    for i in range(n, 0, -1):
       # Variant: i
       if res == False:
          break
       # If true does not come from the row above, we include the item
       if res == True and res != dp_matrix[i-1][j]:
          # We include the item
          solution_set.add(plate_list[i-1])
          # And move the cursor to the row above, and to the index that equals
          # index_in_row = previous_index_in_row - the weight of the item
          j = j - plate_list[i-1]
          res = dp_matrix[n-1][j]
    return solution_set
```

Listing 2: Extension part of the algorithm. After the simple algorithm is finished (not shown here, but identical to the previous algorithm), the solution table is traversed to find all members of the subset.

C Complexity Analysis

The extended algorithm consists of a recursive function which has a loop. All other operations are counted as constant. As seen in listing 4, a recursive call is only made for nodes not already visited. Thus, we never visit a node more than once. Inside the function `dfs(v)` we loop through all neighbors of v. The number of neighbors for a node is equal to the number of outgoing edges from that node. So, in summary, in the worst case, we will do some amount of work in $\mathcal{O}(1)$ time for each outgoing edge from each node. Of course the total number of outgoing edges from all nodes is the same as the total number of edges in G (an incoming edge to a node must be outgoing from another node). So the running time of the extended algorithm depends only on the number of edges, independent of the number of nodes and time complexity is $\mathcal{O}(|E|)$.

```python
def augmenting(G: Graph, s: str, t: str) -> bool:
   """
   Sig: Graph G(V, E), str, str -> bool
   Pre: There are no incoming edges to 's',
         no outgoing edges from 't'.
         If there is an edge from u to v in 'G', then
         there cannot be an edge from v to u in 'G'.
         'G' has no self-loops.
         Capacities and flows of edges in 'G' are non-negative.
   Post: Return true if there is an augmenting path (as defined in the assignment)
         from 's' to 't', False otherwise.
   Ex: Sanity tests below
        augmenting(g1, 'a', 'f') = False
        augmenting(g2, 'a', 'f') = True
   """

   visited = set()

   def dfs(v: str):
      # Recursion variant: (Nubmer of nodes in G) - len(visited)
      if v == t:
         return True
      visited.add(v)
      for neighbor in G.neighbors(v):
         # Loop Variant: len(G.neighbors(v)) - (index of 'neighbor' in /
             G.neighbors(v))
         if neighbor not in visited and G.capacity(v, neighbor) > G.flow(v, /
             neighbor):
            if dfs(neighbor):
               return True
      return False

   return dfs(s)
```
Listing 3: Python implementation of the solution to the augmenting path problem, using top-down recursion.

```
def augmenting_extended(G: Graph, s: str, t: str) \
                  -> Tuple[bool, List[Tuple[str, str]]]:
   """
   Sig: Graph G(V,E), str, str -> Tuple[bool, List[Tuple[str, str]]]
   Pre: There are no incoming edges to 's',
         no outgoing edges from 't'.
         If there is an edge from u to v in 'G', then
         there cannot be an edge from v to u in 'G'.
         'G' has no self-loops.
         Capacities and flows of edges in 'G' are non-negative.
   Post: Returns (True, path) if there is an augmenting path from 's' to 't',
         where path is a list of all edges in the augmenting path,
         (False, []) otherwise.
   Ex: Sanity tests below
        augmenting_extended(g1, 'a', 'f') = False, []
        augmenting_extended(g2, 'a', 'f') = True, [('a', 'c'), ('c', 'b'),
                                          ('b', 'd'), ('d', 'f')]
   """

   visited = set()
   path = []

   def dfs(v):
      # Recursion Variant: (Nubmer of nodes in G) - len(visited)
      # Invariant: 'path' contains an augmented path from 's' to 'v'
      visited.add(v)
      if v == t:
         return True
      for neighbor in G.neighbors(v):
         # Loop Variant: len(G.neighbors(v)) - (index of 'neighbor' in /
             G.neighbors(v))
         if neighbor not in visited and G.capacity(v, neighbor) > G.flow(v, /
             neighbor):
            if dfs(neighbor):
               path.insert(0, (v, neighbor))
               return True
      return False

   return dfs(s), path
```
Listing 4: Extension of Listing 3, where each edge in the augmenting path is stored in a list