Backtracking and Branch-and-Bound

Solution to the Practical Exercises

Gonçalo Leão

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

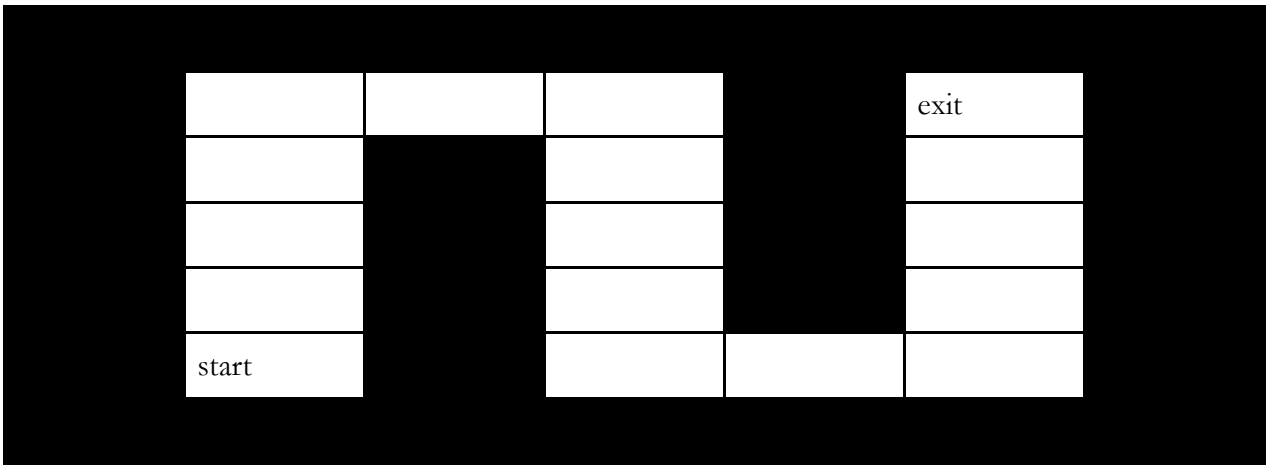
Spring 2023

A - Backtracking

Exercise 1

a) See source code.

b) The temporal complexity of the algorithm in the worst case corresponds to a maze where the maximum number of cells are visited before reaching the exit. This occurs with a maze with no branches and a single zig-zag, winding path as exemplified below for a 7 x 7 maze:



The number of cells visited in this sort of maze is between $n^2/2$ and n^2, and is thus $O(n^2)$. Therefore, since processing a cell can be done in O(1) time, the algorithm runs on $O(n^2)$ time.

Exercise 2

a) The idea of the backtracking solution is to consider including each element of the array as a choice point: either the element is included or it is not.

Pruning occurs when inserting a given element exceeds T. In this case, the whole search tree under the current node can be pruned since it is impossible to reach a valid solution, given that all elements of the array are non-negative.

Pseudo-code for finding a subset with the correct sum:

Input and auxiliary functions:

- A: array of non-negative numbers
- T: desired sum
- subsetSumRec(A, T, i, subset): recursive function for the actual backtracking, which considers the choice point of the i-th value of A and stores the current subset

Output: boolean indicating if the subset exists, subset of A whose sum is T

```
subsetSum(A, T)
    subsetSumRec(A, T, 0, [])

subsetSumRec(A, T, i, subset)
    if(i = A.size())
        return (T = 0), subset
    if(T ≥ A[i])
        // Try including A[curIndex] in the subset
        subset.push(A[i])
        if(subsetSumRec(A, T - A[i], i+1, subset))
            return true, subset
        subset.pop()
    // Try not including A[i]
    return subsetSumRec(A, T, i+1, subset)
```

b) The temporal complexity of the algorithm in the worst case is $O(2^n)$, where n is array A's length. This is due to the search tree having a maximum depth of n (since there are n choice points), with each node having 2 children (since there at most two options in each choice point), and processing each tree node taking O(1) since only a fixed amount of constant-time operations are performed. The complexity is better than the one for brute-force since the current sum is updated throughout the recursion process rather than being computed from scratch at each choice point (to be more precise, instead of the running sum, the algorithm decreases the value of T as more elements are added).

The spatial complexity of the algorithm is Θ(n), not only for the worst-case, but for any other case. This is due to space always being allocated to the current candidate, whose size scales with n. If there is a valid solution, then space is also allocated for the solution, which also scales with n. The recursion stack also scales with n assuming that the array A is shared amongst all recursive calls and that there is a single instance of the subset being passed around in the recursive calls. All the other variables have a fixed size with respect to n.

c) Source code.

d) Not yet in the code, but it is rather trivial to implement given the definition of the two bounds as described in the solution of Problem 12.

Exercise 3

a) See source code.

b) The algorithm has a temporal complexity of $O(n*S^n)$ since:

- there are at most S^n candidate solutions (if all n coin denominations have an equal stock of S), which means that the base case is reached at most S^n times;
- in the recursive steps, the algorithm requires O(1) time to perform all the required recursive calls (i.e. the driving function runs in O(1) time);
- in the base step (once a candidate is fully assembled to be tested), the algorithm runs in O(n) time, since, in the worst case, it has to update the best solution found so far (vector-to-vector copy).

Exercise 4

See source code.

Exercise 5

See source code.

Exercise 6

See source code.

Exercise 7

a) See source code.

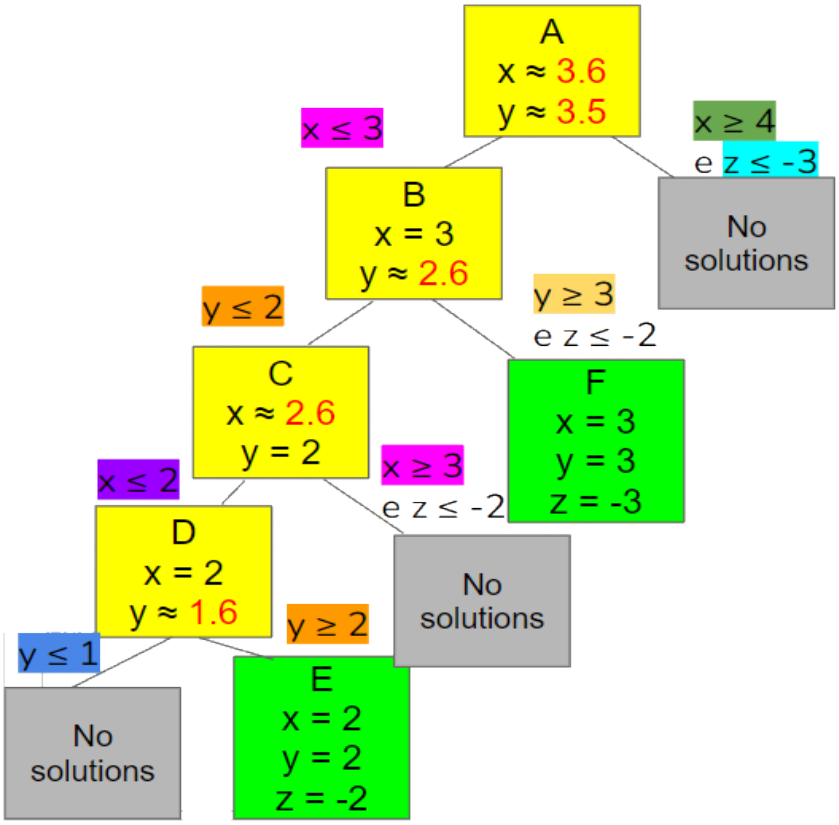
b) See source code.

c) See source code.

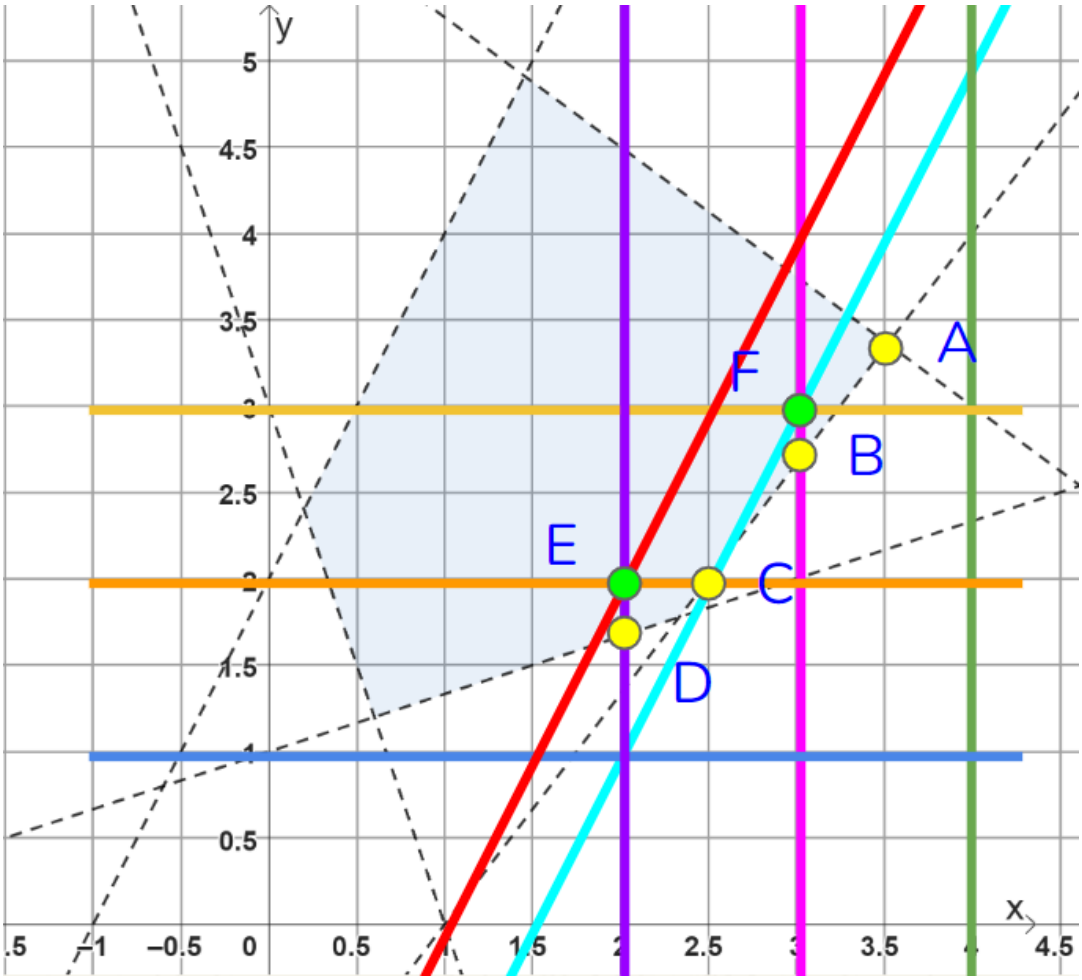
B - Branch-and-Bound

Exercise 8

The image below presents the search tree generated via backtracking during the Branch-and-Bound algorithm.



The plot below shows how the 2D space is partitioned during the search.



The optimal solution is F = (3,3), where z = 3 - 2*3 = -3.

During the search process, another feasible solution is found, E. When E is found, it is not immediately declared to be optimal since there are still nodes left to explore in the tree. Instead, a new constraint is added (represented by a red line), which helps to reduce the search space.

Exercise 9

a) This is a minimization problem since z increases as more constraints are added, i.e. as the node depth increases in the search tree.

b) When node K is reached, the best solution found so far is on node H with z = 265.4. Therefore, the solution in node K must have a z that is lower or equal than 265.4 (265.4 is thus the upper bound)

The father of node K is I, with z = 260. The relaxed LP problems in K and I differ by K having one additional constraint, thus z's value for K cannot be better (i.e. smaller) than the one in I, since the space of admissible solutions for K is a subset of the one for I. Therefore, the solution in node K must have a z that is higher or equal than 260 (260 is thus the lower bound).

To conclude, in node K, $z \in [260, 265.4]$.

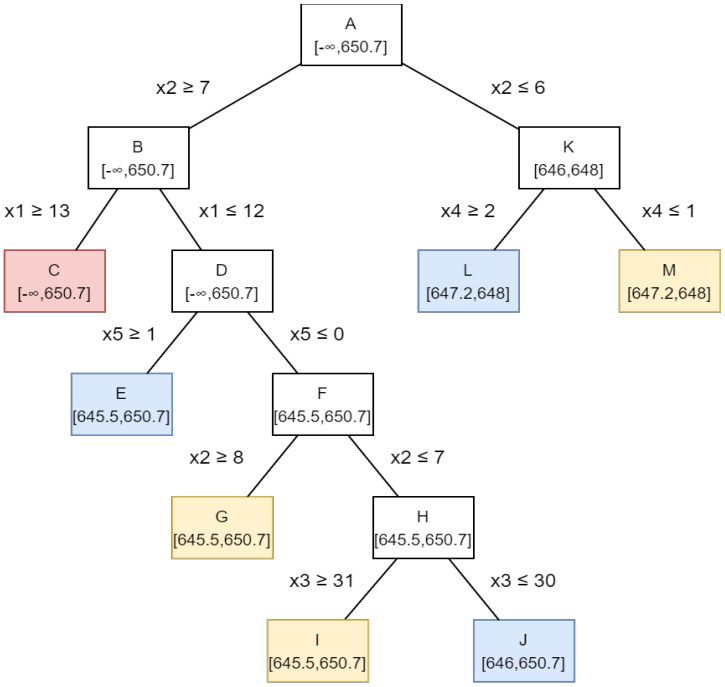
c) The search was conducted using Breadth-First Search (BFS). This can be seen for instance in the root, where each child is explored before reaching the root's grandchildren.

Using Depth-First Search (DFS) is be more suitable for this problem as avoids having to solve the relaxed LP problems of nodes F and G. During the DFS, the solution of node H is found before reaching C, so C's children would be pruned since node C's z value is already higher than the one for H, which implies that F and G would have sub-optimal (i.e. higher) z values.

Exercise 10

a) This is a maximization problem since F decreases as more constraints are added, i.e. as the node depth increases in the search tree.

b) The search tree is depicted below, with the bounds for F at every step.



c) The sole inadmissible solution to the relaxed LP problem is marked in red. The all-solutions that are considered the best found so far are in blue, while those that are immediately pruned are in yellow. For instance, solution G is pruned since it has an F value of 642.7, which is lower than the F value of the current best solution, E (and thus to the lower bound for F).

Since this is a maximization problem:

- the lower bound for F is updated (increased) whenever an all-integer solution with an F value higher than the previous lower bound is found.
- the upper bound for F is updated (decreased) whenever any solution with an F value lower than the previous upper bound is found AND there are no other nodes to explore.

d) The optimal solution corresponds to node L. It is the all-integer solution with the highest F value.

e) The search was conducted using Depth-First Search (DFS). This can be seen for instance in the root, where the children of node B are explored before reaching the second child of the root: node K.

Exercise 11

We now consider the sum-of-subsets problem as described in class for the instance S = (5, 2, 9, 3, 5, 8) and *M* = 20. Does this problem have a solution for M = 20? And what about for M = 4? Provide a solution example and argue that for infeasibility.

Answer: For M = 20 a simple solution is x = (1,1,0,0,1,1) yielding a summation of M=20. For M=4, and since there is no element in W whose value is exactly 4, we are left with the two values 2 and 3 whose addition is larger than 4. No other possible combination of values even comes close to 4, so the problem is infeasible.

Exercise 12

Solve the following sum-of-subset problem W = {1, 3, 4, 5}, M=8 using backtracking showing the complete tree of states and use the bounding functions described in class and determine which states can be pruned.

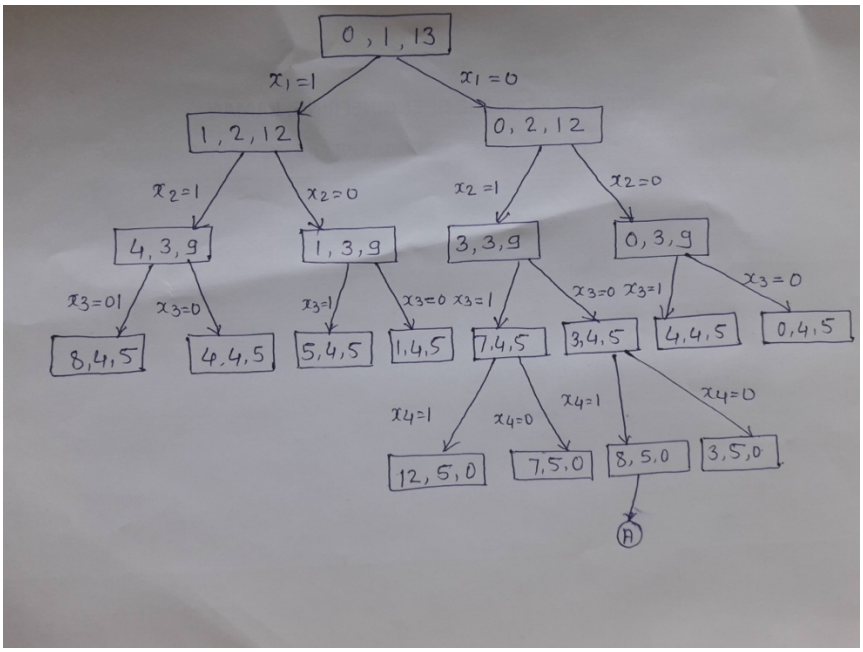
Answer: The tree of exploration states is as shown below. For convenience, the values were already sorted so that the two bounding functions described in class below can be easily applied. The first bounding function check if at a given state the summation is so close to the desired value so that adding an additional value will exceed the value. In other words, the search can be abandoned if:

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > M$$

The second bounding function determines if adding all the remainder elements of the set will yield a summation that is less than the desired value M. In other words, the search can be abandoned if:

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i < M$$

Combining these functions that a simple backtracking search we get the state space shown below where each node is a 3-tuple (`sum`, `lev`, `rem`) where `sum` is the current summation (thus corresponding to the choices of x_i values, `lev` is the tree level and thus which element is being chosen, and `rem` the remainder elements that can still be included in the summation, i.e., the value of $\sum_{i=k+1}^{n} w_i$.



Notice that is a single solution were to be desired, the search would terminate at the forth node (far left node in the tree) to be explored, the node (8,4,5) corresponding to the choice vector x = (1,1,1,0).