Backtracking and Branch-and-Bound

Practical Exercises

*Gonçalo Leão**

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

Spring 2023

**Various of these problems were originally developed by the team of instructor of the 2021 CAL (Concepção de Algoritmos") course of the L.EIC at the Faculty of Engineering of the University of Porto (FEUP), University of Porto and adapted and extended here by the author.*

A - Backtracking

Exercise 1

The objective of this exercise is to find the exit of a 10 by 10 labyrinth. The initial position is always at (1, 1). The labyrinth is modeled as a 10x10 matrix of integers, where a 0 represents a wall, a 1 represents free space, and a 2 represents the exit.

a) Implement the *findGoal* function (see *Labirinth.h* and *Labirinth.cpp*), which finds the way to the exit using backtracking algorithms. This function should call itself recursively until it finds the solution. In each decision point in the labyrinth, the only possible actions are to move left, right, up or down (see *Labirinth.h* and *Labirinth.cpp*). Once the exit is found, a message should be printed to the screen. It is suggested that you use a matrix to keep track of the points which have been visited already.
b) Assuming that the maze is located within a square n x n grid, what is the temporal complexity of the algorithm in the worst case, with respect to n?

Exercise 2

Consider the same description for the **subset sum problem** as in the TP2 sheet.

```
bool subsetSumBT(unsigned int A[], unsigned int n, unsigned int T,
        unsigned int subset[], unsigned int &subsetSize)
```

a) Propose in pseudo-code a backtracking algorithm to solve this problem. Your algorithm should return two outputs: a boolean indicating if the subset exists and the subset itself.
b) Indicate and justify the algorithm's temporal and spatial complexity, in the worst case, with respect to n.
c) Implement *subsetSumBT* using the proposed algorithm.

d) Use the bounding function described in class to shorten the exploration of possible solution states (see Problem 12).

Exercise 3

Consider the same description for the **change-making problem** as in the TP2 sheet.

```
bool changeMakingBT(unsigned int C[], unsigned int Stock[],
 unsigned int n, unsigned int T, unsigned int usedCoins[])
```

a) Implement *changeMakingBT* using a strategy based on backtracking.
b) Indicate and justify the algorithm's temporal complexity, in the worst case, with respect to the number of coin denominations, n, and the maximum stock of any of the coins, S.

Exercise 4

Consider the same description for the **activity selection problem** as in the TP3 sheet.

```
vector<Activity> activitySelectionBT(vector<Activity> A)
```

Implement *activitySelectionBT* using a strategy based on backtracking.

Exercise 5

Consider the same description for the **0-1 knapsack problem** as in the TP2 sheet.

```
unsigned int knapsackBT(unsigned int values[], unsigned int weights[],
      unsigned int n, unsigned int maxWeight, bool usedItems[])
```

Implement *knapsackBT* using a strategy based on backtracking.

Exercise 6

Consider the same description for the **traveling salesman problem (TSP)** as in the TP2 sheet.

```
unsigned int tspBT(const unsigned int **dists, unsigned int n, unsigned
                                int path[])
```

Implement *tspBT* using a strategy based on backtracking.

Exercise 7

Sudoku [http://en.wikipedia.org/wiki/Sudoku] is a game in which the objective is to fill a 9x9 matrix with numbers from 1 to 9, without repeating numbers in any row, column or 3x3 blocks.

a) Implement the *solve* function, which (efficiently) solves Sudoku of any degree of difficulty, using a backtracking algorithm. The algorithm should work recursively: it should fill in a cell and call itself to solve the remaining puzzle.

   Suggestion: Use the following greedy algorithm to choose the cell to fill in: select the cell with the minimum number of possible values (ideally 1). Implement this algorithm in the *findBestCell* function.

b) Implement the *countSolutions* function, which determines the multiplicity of solutions (no solution, one solution or more than one solution) of a given Sudoku.

   Suggestion: adapt the implementation of the *solve* function.

c) Implement the *generate* function, which automatically generates Sudoku.

   Suggestion: starting with an empty Sudoku, randomly choose a cell and a number; if the cell is not filled in and the chosen number is valid for that cell, fill it in and analyze the multiplicity of solutions; if the Sudoku becomes impossible, clear the cell; if it has one solution, terminate; otherwise, continue the process.

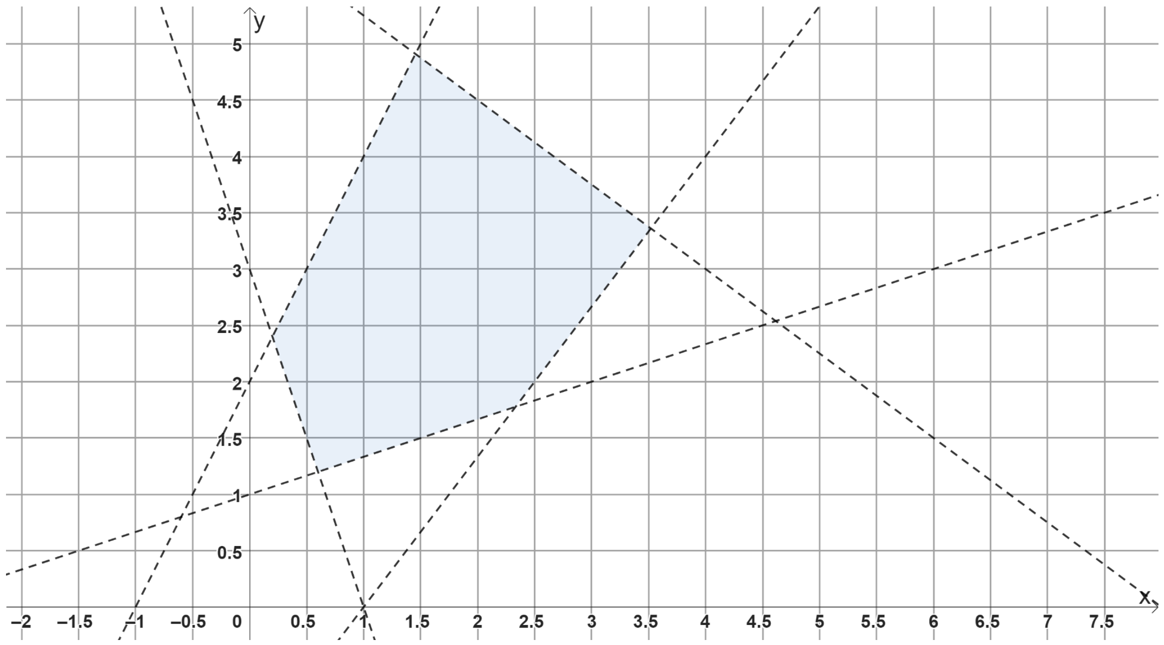
B - Branch-and-Bound

Exercise 8

The figure below depicts the graphical representation of an Integer Programming (ILP) problem with two decision variables, x and y. A set of constraints in the problem limited the feasible region to the polygon highlighted in blue. The objective function, to be minimized, is z = x - 2*y.

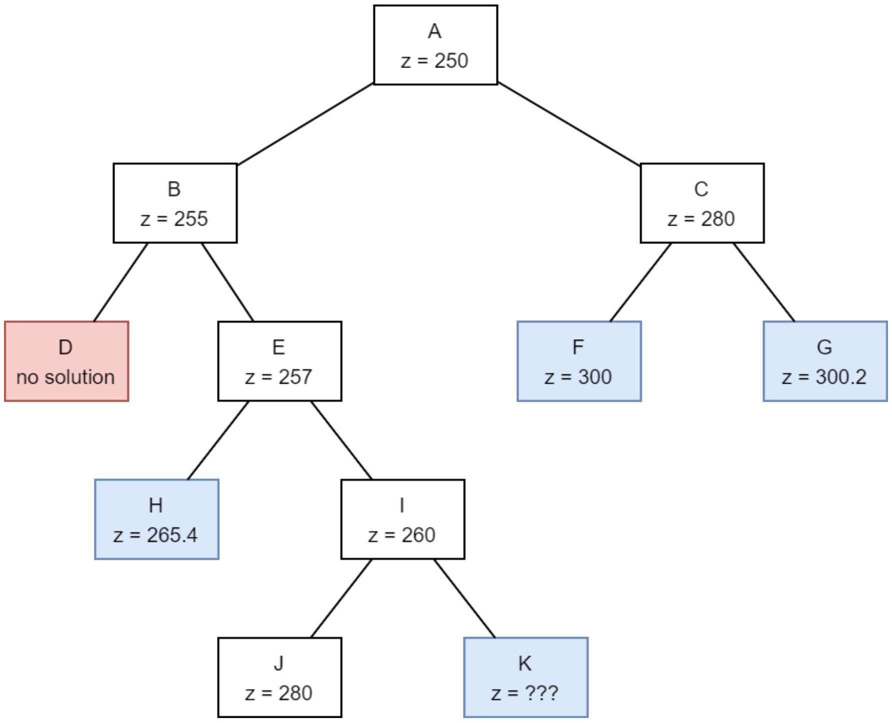
Using the Branch-and-Bound algorithmic approach, find the optimal solution. Explain the main steps of the algorithm by drawing a search tree and represent the optimal solution graphically in the plot.

The search for the optimal solution should be conducted using Depth-First Search (DFS).



Exercise 9

A computer running the Branch-and-Bound algorithm for an Integer Linear Programming problem produced the search tree below. This problem has an objective function known as z.



The nodes of the search tree were explored in alphabetical order. The nodes in blue correspond to all-integer solutions, while the node in red represents an inadmissible solution to the relaxed Linear Programming problem.

a) Determine and justify whether this is a maximization or minimization problem.
b) Assuming node K contains the optimal solution, determine this node's lower and upper bounds for z.
c) Which algorithm was used to explore the solution space (Depth-First Search or Breadth-First Search)? Justify your answer. Is this strategy the best suited one for this problem?

Exercise 10

Another computer running the Branch-and-Bound approach for an Integer Programming problem produced the set of solutions below, in order. This problem has five decision variables (x1, x2, x3, x4, x5) and one objective function known as F.

Solution	x1	x2	x3	x4	x5	F
A	13	6.667	30	0	0	650.7
B	12.4	7	30	0	0	650
C	13	7	30	0	0	no solution
D	12	7	30	0	0.25	647.8
E	11	7	30	0	1	645.5
F	12	7.2	30	0	0	647.7
G	10.5	8	30	0	0	642.7
H	12	7	30.45	0	0	647.5
I	12.33	7	31	0	0	642
J	12	7	30	0	0	646
K	13	6	30	1.5	0	648
L	16	5	30	2	0	647.2
M	12.5	5	30	0	0	647

a) Determine and justify whether this is a maximization or minimization problem.
b) Draw the search tree produced by the computer. In each node, indicate the current lower and upper bound for the objective function F.
c) In the search tree, identify the nodes that correspond to:
   i) Inadmissible solutions to the relaxed Linear Programming problem.
   ii) All-integer solutions. Distinguish those that are considered the best solution found so far from those that are pruned when they are found.
d) Which node of the tree corresponds to the optimal solution?
e) Which algorithm was used to explore the solution space? Justify your answer.

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.

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.