Backtracking and Branch-and-Bound

Practical Exercises

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

Spring 2024

**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

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.

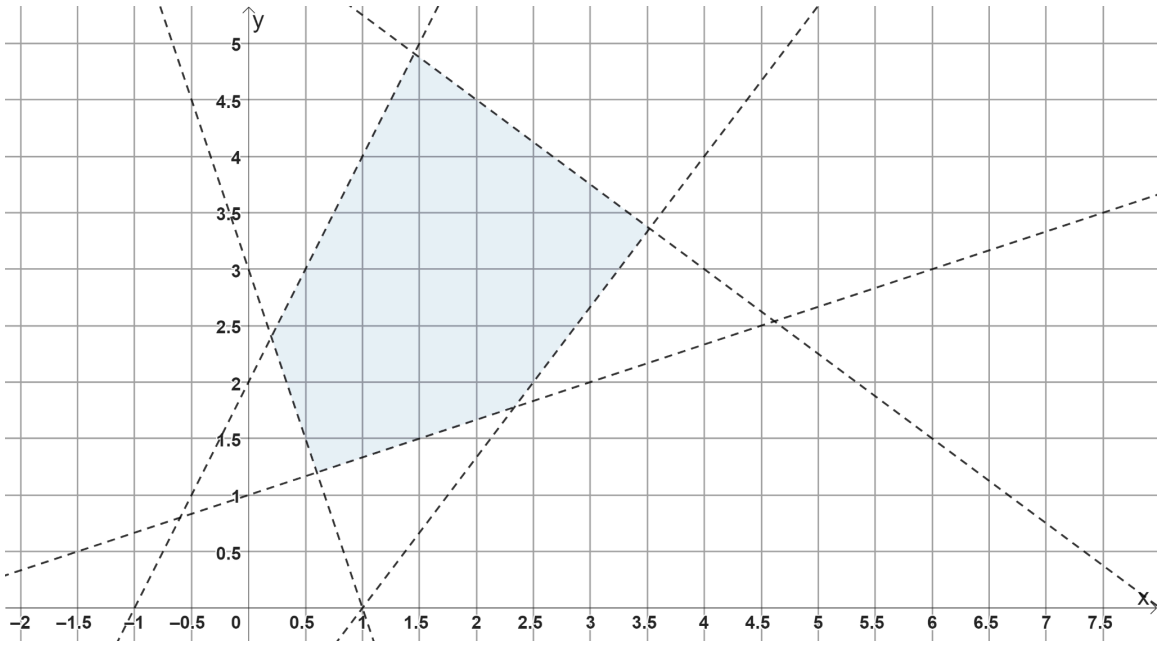
B - Branch-and-Bound

Exercise 2

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 3

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.

C – Extra exercises

Exercise 4

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 file exercises.h), 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. 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 5

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.

Exercise 6

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 7

The **activity selection problem** is concerned with the selection of non-conflicting activities to perform within a given time frame, given a set **A** of activities (a_i), each marked by a start time (s_i) and finish time (f_i). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming a given priority and that a person can only work on a single activity at a time. Consider the function *activitySelection* below (which uses the **Activity** class):

Input example: **A** = { $a_1(10, 20)$, $a_2(30, 35)$, $a_3(5, 15)$, $a_4(10, 40)$, $a_5(40, 50)$ }
Expected result: {a_3, a_2, a_5}

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

Implement *activitySelectionBT* using a strategy based on backtracking in the Activity class (see *exercises.h* file).
Suggestion: Implement the < (less than) operator in the **Activity** class.

Exercise 8

The **Traveling Salesman Problem (TSP)** consists in finding the path in a weighted graph that traverses each vertex once and exactly once and then returns to the initial node such that the sum of the weights of the edges used is minimized.

To simplify, consider that the initial city is always node 0 and that the graph is complete (*i.e.*, it has one edge connecting every node to every other node). The distances may not be symmetric, that is, the distance from node A to B may be different from the distance from B to A.

Input example: `dists` = [[0, 10, 15, 20], [5, 0, 9, 10], [6, 13, 0, 1], [8, 8, 9,0]], **n**=4
Expected result: 35, `path` = [0, 1, 3, 2]

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

Implement *tspBT* using a strategy based on backtracking.