Modified SOKOBAN Game Problem

1 Objective

Develop an efficient and heuristic application to solve the SOKOBAN game problem, based on the A* algorithm. It will be implemented in the C++ programming language (without using STL containers or algorithms), using makefile, and structuring good practices of object-oriented code.

2 A* Algorithm

This type of procedure is used when the answer to a specific problem corresponds to a sequence of steps to perform (`https://en.wikipedia.org/wiki/A*_search_algorithm`). The algorithm is based on the notion of state; this contains the value of each variable in the problem. Then, the steps correspond to changes in the values of the variables of said state. This forms a graph, where each node is a state with particular values of said variables and the steps are directed connections between nodes. Additionally, there is a system start state and a condition for the final state. The solution corresponds to the description of a path between the initial state and some final solution.

The A* algorithm (alg 2) builds the graph as it executes and ensures not to repeat already visited nodes. To do this, it maintains a data structure for nodes to visit and those already visited.

Algorithm 1 A* Search

```
Require: (start, goal)
OpenSet = {start}
ClosedSet = ∅
while OpenSet is not empty do
    current = OpenSet→ pop()
    ClosedSet→ push(current)
    if current = goal then
        return reconstruct path from goal to start
    end if
    for each neighbor of current do
        if neighbor ∈ ClosedSet OR neighbor is not valid then
            continue
        end if
        OpenSet→ push(neighbor)
    end for
end while
return not Found
```

The A* algorithm allows solving combinations of operations to reach a certain result. In practice, said algorithm is used when a set of operations that change the system state is available.

In engineering there is an infinity of applications in which it is used. For example, the case of an autonomous robot that needs to reach a certain destination given a map where the operations are steps in certain directions. In computer security, it must be tested if security barriers can be bypassed; a sequence of operations that manages to breach the system is sought. In operations management, the goal is to find the sequence of buy/sell operations that allows achieving a certain objective. Another classic application is solving the Rubik's cube or the 8-puzzle.

3 Problem to solve: Generalized SOKOBAN Game

This task consists of solving the generalized SOKOBAN game using the A* algorithm together with heuristics to accelerate the search for a solution to the problem.

The SOKOBAN Game (`https://en.wikipedia.org/wiki/Sokoban`) consists of a board with a player (Fig. 1), with fixed walls and movable boxes that can only be pushed. For example, if a box is adjacent to a wall, it can no longer separate from it, or if it is in a corner of two walls, it can no longer move. The idea is that the player present on the board pushes the boxes until moving them all to locations marked on the board.

Boxes can only be pushed by the character present in the game and cannot be pulled. This means that certain locations exist that leave a box immobilized, as is the case of a corner. In other cases, boxes can block accesses or corridors that can leave the game without a possible solution from that moment.

This game can be generalized, considering the following complexities:

- **Player energy level**: The player has an energy level that is spent as operations are executed. The operations performed are moving, pushing blocked or unlocked boxes. If they run out of energy they will not be able to move.
- **Boxes with locks**: Boxes can be locked by a padlock to the floor. Keys are scattered on the board. To move a locked box, one must be positioned on top of the key associated with the box which is automatically picked up by the player. A player can only carry a single key. When trying to push the locked box, it will automatically unlock and the key will disappear.
- **Automatic doors**: Certain cells can have a door that opens and closes according to a certain predefined frequency. When closed, the door acts as a rigid wall, when open it behaves as a space through which one can circulate. Doors have a time for which they remain open and closed for another configurable time. If a box or player is located just before closing, it remains open so it moves if pushed. If the door is still in the closing period, only the block moves but the player does not enter the door. Time runs in discrete steps of successful player actions.

4 Implementation

An efficient implementation written in a clean and clear manner is required, in the C++ programming language. For this first task, STL library containers and algorithms cannot be used. It is expected that all data structures be implemented in one class for each of them. Object orientation must be maintained: There should be no loose functions, only classes and main functions for testing and the main program. It is important that the final program efficiently solves the problem. Each class must have an associated program for testing purposes.

The main program (main.cpp) should display a menu, where one of its options is "Exit". This menu also includes an option to load a configuration file that contains the necessary information describing the generalized Sokoban board in its initial layout and necessary values for this game. The output that should be shown after selecting the menu option that solves the problem is

a list of operations U:Up D:Down R:Right L:Left that indicate the player's movements along with the time in milliseconds used to solve, this is in case the board has a solution. In case it has no solution, a message "Game without solution" and the associated time are shown. The menu must also have options to: show the board according to the steps in which the game is solved, show the steps in which the board is modified if a sequence of U,D,R,L movements is entered.

The configuration file has the following syntax at the beginning:

```
[META]
NAME = <string>
WIDTH = <int>
HEIGHT = <int>
ENERGY_LIMIT = <int>
MOVE_COST = <int>
PUSH_COST = <int>

[DOORS]
<ID> OPEN=<k> CLOSE=<l> PHASE=<p> INITIAL=<0|1>
```

What appears between brackets [] is the section name. The [META] section indicates the metadata necessary for this game. Among these is the name (NAME) which is a string, the board width (WIDTH) which is an integer, the board height (HEIGHT) which is an integer, the player's total energy at the start (ENERGY_LIMIT) which is an integer, the energy cost the player has when moving freely but without pushing (MOVE_COST), the cost consumed when pushing (PUSH_COST). There is also a [DOORS] section that configures doors and is optional (can be empty). Each line contains the following information: ID which is an integer from 1 to 9 representing the door, the OPEN value which is an integer indicating the time it remains open, the CLOSE value which is an integer indicating the time it remains closed, PHASE is a number indicating how much time passes from the start for it to change state, INITIAL indicates what the door's initial state is (1: open, 0: closed).

Subsequently, there is a [BOARD] section that shows the HEIGHT×WIDTH board where the following characters have the following meaning:

- **Static cells**
  - `#` Solid wall
  - `.` cell with target point
- **Dynamic cells**
  - `@` Player
  - `$` unlocked box on the floor
  - `*` box located at a target point
  - `A, ..., Z` locked box
  - `a, ..., z` key associated with the box if changed to uppercase
  - `1, ..., 9` door, the number is its ID and its behavior is defined in the [DOORS] section

Let's see an example file with a simple problem:

```
[META]
NAME = Easy-2
WIDTH = 8
HEIGHT = 7
ENERGY_LIMIT = 200
MOVE_COST = 1
PUSH_COST = 1

[DOORS]

[BOARD]
########
#      #
# . $  #
# @    #
#      #
#      #
########
```

In this case, the program shows:

```
Resolution time: 1[msec]
Solution found.
Steps:
RUL
```

Additionally, the deliverable must contain:

- A makefile that compiles the main program and test programs with separate compilation.
- Each class consists of 2 files (.cpp) and (.h), which must be compiled separately.
- A main.cpp with a menu to select input file. The problem is solved, the minimum cost found and the time it took to solve are delivered. It repeats in a loop that includes an exit.
- By efficient it refers to search, insertion, deletion, pop, push operations being at least O(log(N)).