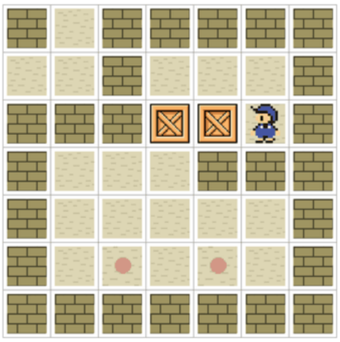
# Introduction to Systems Programming (IPS)



## Lab 3 - Show best move in Sokoban game with recursion

**TL;DR** In the 3rd lab, you will implement a new functionality to show the best move for a given game state. Check all the deatils in Section 2.1.2, as well as in the submission instructions.

## 1 Introduction

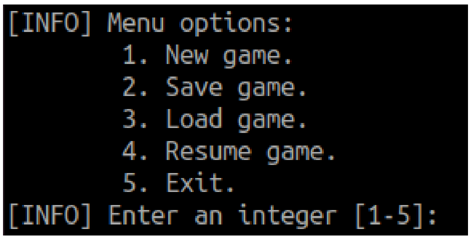
In the beginning, before graphical interfaces, video games were text-based, that is, the game was run entirely on a terminal. The first games to be adapted were board and card games, since their format made it relatively easy to adapt to the terminal. In this project, we will program the **Sokoban** game.

In this game, an agent must minimize the number of steps to push all boxes to goal locations. The agent can freely move between adjacent empty locations in one of the four cardinal directions (i.e., up, right, down and left), or push an adjacent box if the next location in the same direction is a goal or empty location. The walls cannot be traversed nor pushed, so if the agent or a box hits a wall, they will remain in the same location.

## 2 Description

The player will interact with the **Sokoban** game through the terminal, introducing all options by the standard input (keyboard). The application will always run the next flow:

1. Show the main menu with all menu options (see Figure 1).



```
[INFO] Menu options:
        1. New game.
        2. Save game.
        3. Load game.
        4. Resume game.
        5. Exit.
[INFO] Enter an integer [1-5]:
```

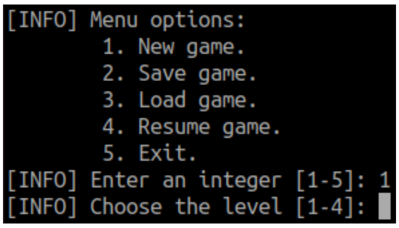
Figure 1: Main menu

2. Get the player option.
3. If applicable, run the functionality chosen in step 2. In case the chosen option is `Exit`, the player will leave the application.
4. Otherwise, go back to step 1.

### 2.1 New Game

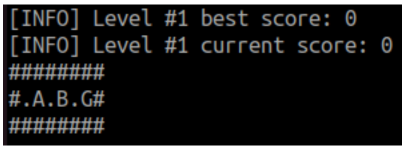
This is the main functionality to implement in the application. First, it will ask which of the four available *levels* will be played (see Fig. 2). Second, it will load and print the chosen *level* grid, the *best score* achieved so far in that level, and the *current score* which is initially 0 (see Fig. 3). Last, a new game starts, showing up the move options (see Fig. 4) for each game state and properly finishing when that state is terminal (i.e., end of the game). The list of new game options consists of:

- [1 – 4], move the agent in one of the four cardinal directions (to be developed in Lab 1).
- [5], show the best next move (to be developed in Lab 3).
- [6], quit the current game (already programmed).



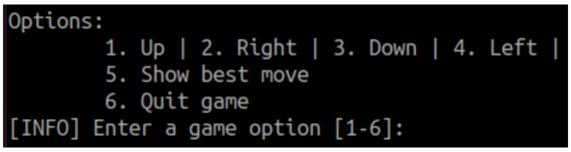
```
[INFO] Menu options:
        1. New game.
        2. Save game.
        3. Load game.
        4. Resume game.
        5. Exit.
[INFO] Enter an integer [1-5]: 1
[INFO] Choose the level [1-4]: 
```

Figure 2: Choose level



```
[INFO] Level #1 best score: 0
[INFO] Level #1 current score: 0
########
#.A.B.G#
########
```

Figure 3: Print Session



```
Options:
        1. Up | 2. Right | 3. Down | 4. Left |
        5. Show best move
        6. Quit game
[INFO] Enter a game option [1-6]:
```

Figure 4: New game options

The *grid coordinates* are interpreted from left to right, and top to bottom as follows:

| | | | | |
|---|---|---|---|---|
| (0,0) | (0,1) | (0,2) | (0,3) | ... |
| (1,0) | (1,1) | (1,2) | (1,3) | ... |
| (2,0) | (2,1) | (2,2) | (2,3) | ... |
| (3,0) | (3,1) | (3,2) | (3,3) | ... |
| ... | ... | ... | ... | ... |

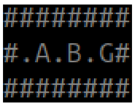
Table 1: Grid coordinates

The *grid display* (see grid in Fig. 3) uses the following legend:

- `'A'`: agent
- `'B'`: box
- `'#'`: wall
- `'.'`: empty location
- `'G'`: empty goal location
- `'X'`: box at goal location
- `'Y'`: agent at goal location

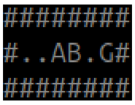
### 2.1.1 Move

There is a `move` option for each of the four cardinal directions (i.e., up, right, down and left). Any of these options updates the agent's location one step in the chosen direction if the next location is empty (*note: this also includes empty goal locations*). If the next location contains a box, both the agent and the box will move in the same direction (i.e., the agent will push the box) if the location after the box is empty (*note: this also includes empty goal locations*). In all other cases, the locations of the agent and the boxes will not be updated. The following images show a sequence of `right` moves after loading `Level #1`, with the leftmost being the initial state and the rightmost being a terminal state with the box already at the goal location:



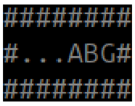
```
########
#.A.B.G#
########
```

⟹ Right ⟹



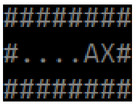
```
########
#..AB.G#
########
```

⟹ Right ⟹



```
########
#...ABG#
########
```

⟹ Right ⟹



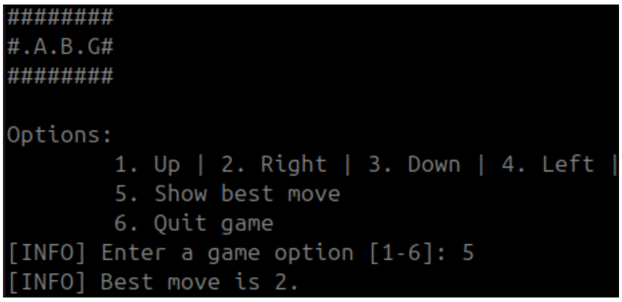
```
########
#....AX#
########
```

### 2.1.2 Show best move

This is the fifth option in the game menu, which runs a recursive function to inform the player what could be the best next move. For that, we need to implement three new functions in `game.c`:

- `Game copy(Game *g);` it does a deep copy of the game `*g` and returns it. Note that the grid must be dynamically allocated on the copy.
- `int recursive_best_score(Game *g, int depth);` returns the minimum achievable score from a given game state. It must do recursive calls for all available options, and return the minimum achieved score among game states that are terminal; otherwise it will return 0 (i.e., the game is not solvable within `MAX_DEPTH` steps). *Hint: do a copy of each game before the recursive call, and free it right after the recursion with* `void free_game(Game *g);`.
- `int show_best_move(Game *g);` it makes a game copy for each available option, applies the move option to the copy, and gets the best score by calling the `recursive_best_score` functionality. At the end, it returns the option with the minimum positive score, otherwise it returns `INVALID_MOVE`.

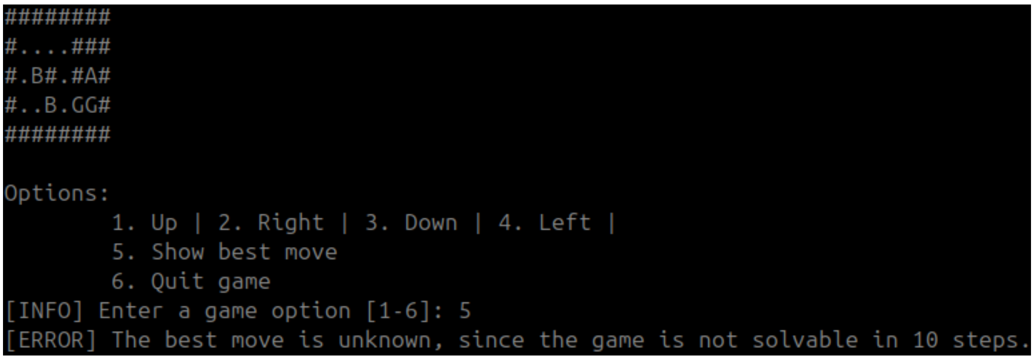
For the recursive function, we need to create a macro `MAX_DEPTH` in `game.h` and set it to some small value (e.g., 10), to handle the depth of the recursive calls. The next images show an example of the best next moves in the given game states:



```
########
#.A.B.G#
########

Options:
        1. Up | 2. Right | 3. Down | 4. Left |
        5. Show best move
        6. Quit game
[INFO] Enter a game option [1-6]: 5
[INFO] Best move is 2.
```

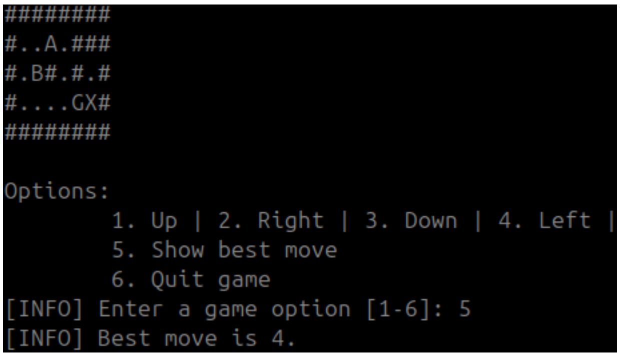
Figure 5: Level 1 - initial state (best move is right)



```
########
#....###
#.B#.#A#
#..B.GG#
########

Options:
        1. Up | 2. Right | 3. Down | 4. Left |
        5. Show best move
        6. Quit game
[INFO] Enter a game option [1-6]: 5
[ERROR] The best move is unknown, since the game is not solvable in 10 steps.
```

Figure 6: Level 2 - initial state (best move is unknown)



```
########
#..A.###
#.B#.#.#
#....GX#
########

Options:
        1. Up | 2. Right | 3. Down | 4. Left |
        5. Show best move
        6. Quit game
[INFO] Enter a game option [1-6]: 5
[INFO] Best move is 4.
```

Figure 7: Level 2 - after some steps (best move is left)

### 2.1.3 Quit Game

This option has been already implemented, so selecting it must quit the current game, going back to the *main menu* again.

## 2.2 Save Game

This is the second option Figure 1. The objective is to save the current game into a file. Recall that `Game` is defined by the integers `score` and `level`, and a `state`, which is a 2D `grid` of characters that represents the current game display of `rows`×`columns` size. The image on the right side shows an example of the format in which a game should be saved. Follow the next steps to properly save a `Game`:

1. Print the game `score` (e.g., `"Score: 2"`).
2. Print the game `level` (e.g., `"Level: 1"`).
3. Print the game `state` to the file as:
   (a) `rows` and `columns` of the `grid` (i.e., the grid dimensions)
   (b) the `grid` display.

```
Score: 2
Level: 1
State:
rows: 3
columns: 8
########
#.A.B.G#
########
```

## 2.3 Load Game

This is the third option in Figure 1. The aim of this function is the opposite of Section 2.2, where a game that was previously saved in a file, must be loaded into the memory of the current session on-the-fly.

That requires to open a file in reading-mode, read the corresponding data, and assign it to the `Game` of the current session. The structure is exactly the same as in the image of the previous Section, and the opened file must be closed once the reading is finished.

*Note that the `grid` must be implemented as a dynamic array, so you should allocate the memory before loading the grid data from the file (i.e., read the board rows and columns, allocate space, and continue reading the board data).*

*Hint: given a* `file` *variable (i.e., an opened `FILE*`), and an integer* `score`*, you can read the first row of the opened file as follows:* `fscanf(file, "Score: %d\n", &score);`

## 2.4 Resume Game

This is the fourth option in Figure 1. Let's assume the current game state is the one in Figure 3, so if we select `"Quit game"` we should go back to the main menu. Once in the main menu, if we select the `"Resume game"` option, the game should proceed from the current state (i.e., continue from the one in Fig. 3).

The same applies if the game is loaded from a file, instead of created from the new game option. *Note that games in a terminal state must not be playable even if resumed.*

## 2.5 Exit

This option is already implemented, and it makes the player to leave the application.

# 3 Provided Resources

To help with implementation, you will receive a set of files, both headers and sources, covering all the necessary modules. These are the files provided for the lab project:

- `common.h` includes all useful standard libraries and defined macros.
- `utils.c/.h`: implement some utility functions to be used in other modules.
- `game.c/.h`: contain the main functionalities to play the Sokoban game. The next are already implemented:
  - `void print_options();` used to print the available options in a game state (i.e., Fig. 4).
  - `bool is_valid_option(Option o);` returns `true` iff the given option is in the valid set (i.e., up, right, down and left moves; show best next move; and quit game), otherwise returns `false`.

- `bool set_level(State *s, unsigned level);` sets the level data to a state (i.e., a specific grid with its corresponding number of rows and columns).
- `void choose_level(Game *g);` chooses a valid level and assigns the game data with the previous functionality.
- `void init_game(Game *g);` sets the game data to default values.

- `session.c/.h`: have the relevant information about the current game state and the best score achieved.
- `main.c/.h`: contains the starting point of any C program, the main function. In addition, it programs all options from the starting menu, but only `New game` is relevant for the first submission.

# 4 Submission instructions

## 3rd Submission (33.33%)

The last submission assumes that all functionalities from Lab 1 and 2 are implemented, so the game is already functional and playable from the `New game` option, games can be saved and loaded from files, and resumed. Now, the next functionalities (see Section 2.1.2) must be implemented to show what is the best available move in the current game state. Also, take the opportunity to create any possible extension of the project (up to +10% for labs, optional). Use the following guidelines to accomplish the tasks:

### game.h/.c

A few functionalities have been defined in **game.h**, as well as, the macros `MAX_DEPTH` and `INVALID_MOVE`. Implement the new functionalities in **game.c**:

1. `Game copy(Game *g);` to make a deep copy of the current game. The copy requires to allocate new memory for the copied grid.
2. `int recursive_best_score(Game *g, int depth);` to perform recursive calls and return the minimum score found for a given game state and depth.
3. `int show_best_move(GameState *game_state);` that calls the `recursive_best_score` for every starting valid move and returns the move (i.e., the option) with lowest score[1].

The **deadline** is right after the 5th lab class. Submit only a compressed file `.zip` which contains all `.c` and `.h` files of the project, and a short abstract about the extension of the project if any.

[1] *Note: if the returned score is* 0*, it means that the task is not solvable within* `MAX_DEPTH` *recursive calls.*