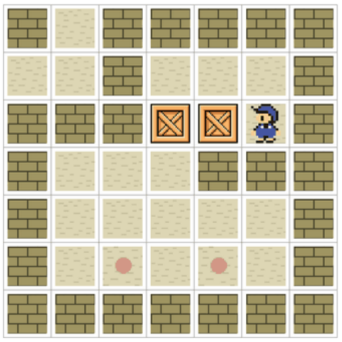
Introduction to Systems Programming (IPS)



Lab 1 - Sokoban game

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)	...
...	...	...	...	...

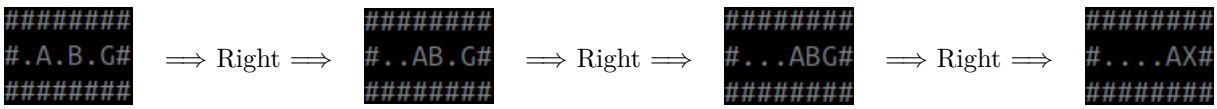
Table 1: Grid coordinates

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:



2.1.2 Show best move

Nothing to do for the first lab. We will add more information about this functionality in Lab 3.

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

Nothing to do for the first lab. We will add more information about this functionality in Lab 2.

2.3 Load Game

Nothing to do for the first lab. We will add more information about this functionality in Lab 2.

2.4 Resume Game

Nothing to do for the first lab. We will add more information about this functionality in Lab 2.

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

1st Submission (33.33%)

In the first submission, the game must be functional, that is, it must be playable from `New game` in the main menu, show the available new game options, update the grid accordingly when moves are selected, and terminate when all boxes are at goal locations. Use the following guidelines to accomplish the tasks:

game.c

Implement the following functionalities:

1. `void print_state(State s);` prints on the screen the grid of the state (see the grid in Fig. 3).
2. `void print_game(Game game);` prints on the screen the score for the current level and the game state.
3. `bool is_terminal(State s);` it returns `true` if all goal locations contain boxes, otherwise it returns `false`.

4. `State move(State s, Option o);` it returns the next state after applying a move option (see Sec. 2.1.1).

session.c

Implement the following functionalities:

1. `void restart_session_game(Session *session);` initializes the game of the session.
2. `void init_session(Session *session);` initializes the `best_score` of every level to 0, and restarts the session of the game.
3. `void print_session(Session *session);` prints on the screen the best score of the current level and the game data (see Fig. 3 for the format).
4. `void new_game_score(Session *session);` assigns the `current_score` to the `best_score` of the current level of the session, if the best score is either 0 (i.e., the level has not been solved yet) or higher than the current score (i.e., the level has been solved with less moves in total).

Note: briefly describe your code with comments below each function name that you implemented.

The **deadline** is before the next lab class. Submit only a compressed file `.zip` which contains all `.c` and `.h` files of the project.