IN101: Sokoban

Author : Christophe Garion and Fabrice Frances
Audience : SUPAERO 1A
Date : February 11, 2019

1 Objectives

The objectives of this project are the following:

- write an implementation of the Sokoban classic puzzle game. This implementation will allow to load a map from a file, to replay a game and to interactively play Sokoban
- write a solver for the Sokoban game
- use an efficient data structure to accelerate the solver

Each objective is presented in the following.

2 A Sokoban game in C

Sokoban (sôko-ban, "warehouse keeper" in Japanese) is a classic puzzle game from the eighties. The player pushes boxes around a warehouse trying to get them to a set of storage locations. In figure 1, the player has to place the boxes on the red dots (the darker box is already on a red dot). Notice that:

- the player cannot go outside the warehouse
- the player can only push boxes
- the player can only push one box at a time

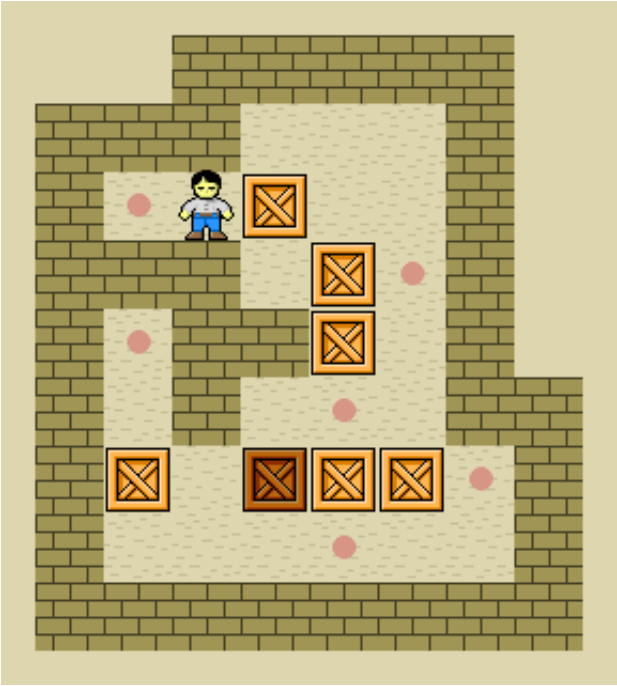


Figure 1: A Sokoban game (credits: Carloseow at English Wikipedia, CC-BY-NC-SA 3.0 license)

The first part of the project is to implement Sokoban in C. Your implementation will be minimalistic and use the console (you will print the map on the console and the user will enter moves with the keyboard). You will also have to implement a map loader and a "replayer" in order to verify your implementation.
See section 5 for more details on how to implement Sokoban game.

3 A Sokoban solver

The next part of the project is to implement a solver for Sokoban in C. The idea is to explore the possible configurations starting from an initial one until a winning configuration has been found or no new configuration can be found. Given a configuration, we will first explore the four configurations generated by the possible moves (going north, south, east or west) before exploring further configurations. This is clearly a breadth-first search algorithm, as seen in session P5.
An example is presented on figure 2 (maps are represented by characters, see 5.1). Starting from the configuration at the root of the tree, we will first generate and check configurations at first level. Configurations generated from the `N` first move are then explored. The next explored configurations would be the ones generated from the `S` configuration, then the ones generated from the `W` configuration etc.

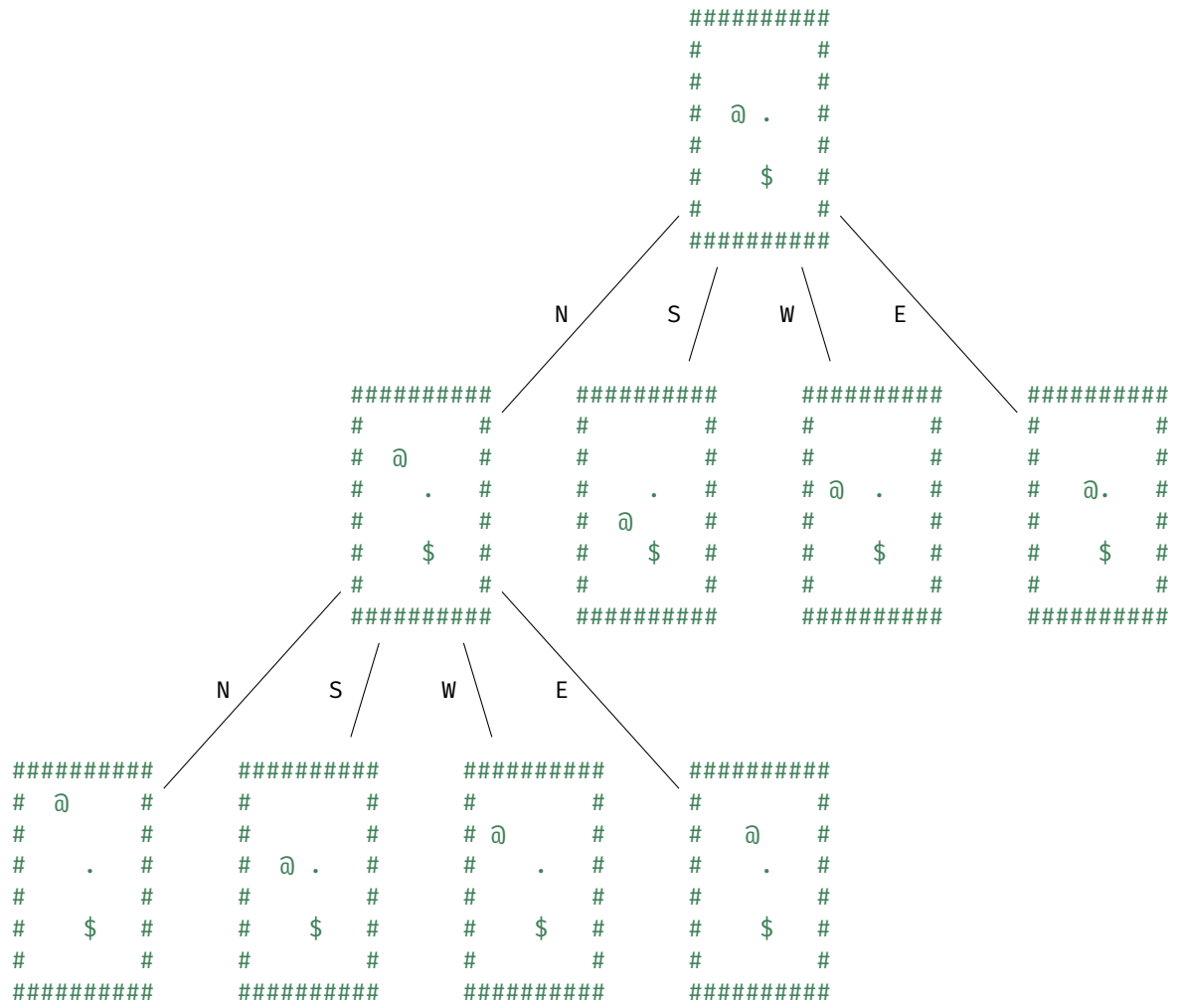


Figure 2: Exploring some configurations

As seen in session P5 on trees, we will not explicitely build the tree presented in figure 2 but we will use a queue data structure (cf. session M6) to store the configurations to explore. At any time, the queue contains part of the configurations that are at a distance of d moves from the initial configuration, followed by part of the configurations that are at a distance $d + 1$ moves. A configuration contains a map, the action that produced it and refers to the previous configuration.

Algorithm 3.1 presents the solver algorithm and some comments on the algorithm are presented in the following:

- the queue `search_queue` contains configurations and its goal is to store the next configurations to explore. As it is a queue, the oldest configurations will be explored first.
- the queue `dequeued_queue` contains the configurations already explored in order to free memory more easily at the end of the execution of the solver.
- the list `explored_list` contains map already explored. It is used to verify if a newly produced map has already been explored. It is also used to free map memory, as it contains all the created maps.
- the main loop works as follows:
  - the queue `search_queue` is dequeued: its oldest element is removed from it. The corresponding map `current_map` is extracted. `new_map`) that can be built from `current_map`
  - if `current_map` is a winning map, exit the **`while`** loop
  - a loop explore the four possible maps (noted
  - if `new_map` has already been explored (i.e. it is in `explored_list`), then continue the for loop for the next map to build else add `new_map` in `explored_list` (it has been explored) and in `search_queue` (it must be later examined)
- when exiting the **`while`** loop
  - either there is a winning map. You have then to build the corresponding plan
  - either there is no plan to win the game starting from the initial map
- finally, free all allocated memory

More details about the implementation of the algorithm and the corresponding data structures are given in section 5.6.

4 Optimizing the solver

The bottleneck in algorithm 3.1 is the search of the current map in the list of already explored maps. It is of course in $O(n)$ with n the size of the list. So, can we do better?
The answer is of course YES! You will see in sessions P5 and M7 a data structure called binary search tree (BST). A BST allows to insert or retrieve an element in logarithmic time, so if we can use a BST instead of the list, it will be a huge improvement of the time complexity of our algorithm.
A BST needs comparable values to work (in session M7, we will implement a BST containing **`int`** values). As we want to manipulate maps in the BST, we need to be able to compare maps (see section 5.7 for more details).
You may wonder if this improvement is real. Table 1 gives execution times of both version of solver compiled with GCC option `-O3` on my machine (which is quite the same as the ones available at SI). BST is clearly a winner...
Conclusion: choosing a good data structure improves a lot your algorithm!

5 Implementation details

5.1 Game map

The game map elements will be represented by **`char`** values:

- `'@'`: the player on an empty cell
- `'+'`: the player on a storage destination
- `'$'`: a box on an empty cell
- `'*'`: a box on a storage destination
- `'.'`: a storage destination without player nor box
- `' '`: empty space

Algorithm 3.1: Sokoban solver algorithm

Input: an initial map initial_map
Output: a plan to solve initial_map if initial_map is solvable

```
1  create new queue of configurations search_queue;
2  create new queue of configurations dequeued_queue;
3  create new list of maps explored_list
4  enqueue(search_queue, initial_map) ;
5  add initial_map in explored_list;
6  while search_queue is not empty do
7  |  current_cell := dequeue(search_queue) ;
8  |  current_map := get map from current_cell;
9  |  enqueue(dequeued_queue, current_cell);
10 |  if current_map is a winning map then
11 |  |  exit while loop ;
12 |  end
13 |  foreach dir ∈ {NORTH, SOUTH, EAST, WEST} do
14 |  |  new_map := compute new map from current_map and dir;
15 |  |  if new_map is in explored_list then
16 |  |  |  free memory allocated for new_map;
17 |  |  |  continue for loop;
18 |  |  end
19 |  |  add new_map in explored_list;
20 |  |  add new cell in search_queue using new_map, current_cell and dir;
21 |  end
22 end
23 if current_map is a winning map then
24 |  build winning plan ;
25 else
26 |  prepare a response saying that there is no plan ;
27 end
28 free memory allocated for the plans stored in explored_list;
29 free memory allocated for explored_list;
30 free memory allocated for search_queue and its cells ;
31 free memory allocated for dequeued_queue and its cells ;
```

file	win	plan length	# of explored nodes	time for list (s)	time for BST (s)
./data/soko_dumber.in	✔	2	5	0.000005	0.000005
./data/soko_dumb.in	✔	7	111	0.000086	0.000062
./data/soko_dumb_imp.in	✘		1,805	0.014606	0.001895
./data/soko81.in	✔	31	31,906	4.647807	0.043387
./data/soko99.in	✔	36	147,508	218.615019	0.291131
./data/soko1.in	✔	184	193,895	318.875331	0.63285

Table 1: Comparison of solver execution times with linked list or BST

- `'#'`: a wall

The game map will be implemented with a structure containing:

- the height and the width of the map
- the player position
- a pointer to a **`char`** representing the elements of the map

The pointer to **`char`** in the structure will point on a *dynamically allocated* memory region that can be assimilated to an array of **`char`**. This region will contain the width × height **`char`** values representing the map[1]. To access the map element at line `i` and column j, you should access the element in the array at index `i * width + j` where `width` is the width of the map. You should write access and modification functions for the map in order to ease the readability of your code.

Do not hesitate to use the **`#define`** directive of the preprocessor to define constants to improve the readibility of your code. For instance

```
#define BOY_ON '+'
```

defines a symbol `BOY_ON` that will be replaced by `'+'` in your program by the preprocessor. Do not add a `;` at the end of the **`#define`** line!

5.2 The map loader

The map loader reads a text file containing the map description and returns a dynamically allocated structure representing the map (cf. section 5.1). You will find in the `data` subdirectory of your repository several file examples. For instance, `soko_dumb.in` (a simple map that can be used for testing purposes) is the following:

```
10 8
##########
#        #
#        #
#  @ .   #
#        #
#    $   #
#        #
##########
```

- the first line contains the width and the height of the map separated by a space
- the other lines contains the map with the convention described in section 5.1

Look at appendix A to understand how to read data from a file (`"%s"` is the format string used in `scanf` to read a string). The `app-ex-loader.c` file contains a example program that reads a file given as input on the command line,

[1] The 2D map will therefore be "flattened" in a 1D array

gets the dimensions of the map and print them and print each line of the map. You can use it to write your `load` function.
You must provide a test for your loader in the `test-loader.c` file. This program should:

- read a file using `main argv` argument
- build the corresponding structure
- print the structure using a printing function defined in `sokoban.c`

To test your implementation, simply try the previous program with different examples and verify that the output on the console is the same as the content of the file.

5.3 Moving the player

When writing the function that moves the player in a given direction, do not forget of course to respect the rules of the game (cannot push several boxes, cannot go through walls etc). Moreover, when called on a pointer to a current map `p_map`, the function should:

- return `p_map` if the movement is not possible (player against a wall etc)
- return a new dynamically allocated map if the movement is possible (i.e. you should not change the original map)

Table 2 gives you test instances for this function: execute your `move` function on the map contained in the original file with the provided movement and compare the result with the map contained in the provided result file. These tests must be implemented in a `test-move.c` file in which all comparisons are checked with the `assert` function (see lab M5).

5.4 Replay

In order to verify your implementation and the plan returned by the solver, you have to implement a `replay` function in `sokoban.c` that takes as arguments a map, the length of the plan and a string representing the movements in the plan. For instance, when called with `7` and `"NEWSNNEE"`, the replayer should execute the following actions: go north, go east, go west, go south, go north two times, go east (notice that the string is partially "executed").
Beware, maps generated during the execution of `replay` should be freed at the end of its execution, except of course the last one that is returned.
Table 3 gives you test instances for this function: execute your `replay` function on the map contained in the original file with the provided plan and compare the result with the map contained in the provided result file. These tests must be implemented in a `test-replay.c` file in which all comparisons are checked with the `assert` function (see lab M5). The big plan for `soko1.in` is given in the `soko1-plan.txt` file.
Finally, write a program in a `replay.c` file that allows to replay a game from the command line. For instance:

```
./replay data/soko1.in 7 "NNNEEESSS"
```

should execute the 7 first movements of the `"NNNEEESSS"` plan starting from the map contained in `data/soko1.in` and print the final map.

The `atoi` function of the `stdlib` C library allows to convert a string to an **`int`**. Therefore, `atoi("76")` returns the **`int`** value `76`. It will be useful in the `replay` program, as all elements in the `argv` parameter of `main` are strings.

5.5 Interactive player

The interactive player program must be contained in the `play.c` file. When launching the program, you must give the file containing the initial map on the command line. The program then wait for the user to give a command by typing a key for the player direction (for instance `n`, `s`, `e` or `w` or the arrow keys). If the user enters `q`, the program exits.

	original file	movement	result file
simple player movements	soko_move_1.in	N	soko_move_1_N.in
	soko_move_1.in	S	soko_move_1_S.in
	soko_move_1.in	E	soko_move_1_E.in
	soko_move_1.in	W	soko_move_1_W.in
player movement on dest.	soko_move_2.in	N	soko_move_2_N.in
	soko_move_2.in	S	soko_move_2_S.in
	soko_move_2.in	E	soko_move_2_E.in
	soko_move_2.in	W	soko_move_2_W.in
move box to east	soko_move_3_1.in	E	soko_move_3_1_E.in
	soko_move_3_2.in	E	soko_move_3_2_E.in
	soko_move_3_3.in	E	soko_move_3_3_E.in
	soko_move_3_4.in	E	soko_move_3_4_E.in
move box to west	soko_move_4_1.in	W	soko_move_4_1_W.in
	soko_move_4_2.in	W	soko_move_4_2_W.in
	soko_move_4_3.in	W	soko_move_4_3_W.in
	soko_move_4_4.in	W	soko_move_4_4_W.in
move box to north	soko_move_5_1.in	N	soko_move_5_1_N.in
	soko_move_5_2.in	N	soko_move_5_2_N.in
	soko_move_5_3.in	N	soko_move_5_3_N.in
	soko_move_5_4.in	N	soko_move_5_4_N.in
move box to south	soko_move_6_1.in	S	soko_move_6_1_S.in
	soko_move_6_2.in	S	soko_move_6_2_S.in
	soko_move_6_3.in	S	soko_move_6_3_S.in
	soko_move_6_4.in	S	soko_move_6_4_S.in
does nothing	soko_move_W_E.in	E	soko_move_W_E.in
	soko_move_W_W.in	W	soko_move_W_W.in
	soko_move_W_N.in	N	soko_move_W_N.in
	soko_move_W_S.in	S	soko_move_W_S.in
	soko_move_B_W_E.in	E	soko_move_B_W_E.in
	soko_move_B_W_W.in	W	soko_move_B_W_W.in
	soko_move_B_W_N.in	N	soko_move_B_W_N.in
	soko_move_B_W_S.in	S	soko_move_B_W_S.in
	soko_move_B_B_E.in	E	soko_move_B_B_E.in
	soko_move_B_B_W.in	W	soko_move_B_B_W.in
	soko_move_B_B_N.in	N	soko_move_B_B_N.in
	soko_move_B_B_S.in	S	soko_move_B_B_S.in

Table 2: Unit tests for move function

The GUI you will have to develop will use the SDL library and functions declared and documented in the `gui.h` file. A complete example of use is given in the `app-ex-gui.c` file, you can build the corresponding executable with `make app-ex-gui` and execute it with `./app-ex-gui`.

When using SDL, you will have memory leaks (please complain to SDL developers). Do not use Valgrind to test memory leaks with your `play` program.

5.6 Sokoban solver

In this section, we will give some implementation details about the Sokoban solver, particularly the needed data structures.

5.6.1 List of already explored maps

A linked list will be used to store the already explored maps during the execution of the solver. You can simply reuse your implementation of a linked list done during sessions M5 and M6, simply replace the `int` value contained in

original file	plan length	plan	result file
`soko_dumb.in`	3	`"SSSEENN"`	`soko_dumb_1.in`
`soko_dumb.in`	5	`"SSSEENN"`	`soko_dumb_2.in`
`soko_dumb.in`	7	`"SSSEENN"`	`soko_dumb_win.in`
`soko_dumb.in`	7	`"ESEESSWNN"`	`soko_dumb_3.in`
`soko_dumb.in`	9	`"ESEESSWNN"`	`soko_dumb_win.in`
`soko1.in`	100	big plan	`soko1_1.in`
`soko1.in`	150	big plan	`soko1_2.in`
`soko1.in`	184	big plan	`soko1_win.in`

Table 3: Unit tests for `replay` function

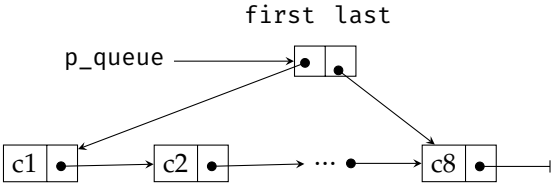
cells by a pointer to a map.
Your specification and implementation of the linked list should be contained in the `linked_list_map.h` and `linked_list_map.c` files. You should provide tests for your list in the `test_linked_list_map.c` file. For instance, add some maps in the list and verify that you can check if the list contains them or not.

5.6.2 Queue

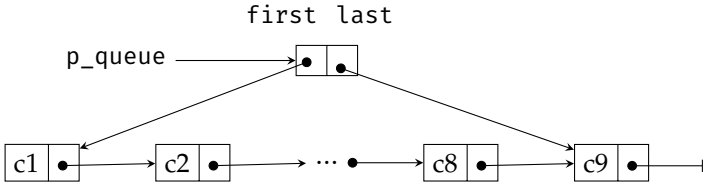
There are two queues used in the solver. Remember that a queue is a linear data structure with two operations: `enqueue`, adding an element at the tail of the queue, and `dequeue`, removing the element at the head of the queue (hence the oldest element of the queue).
First, let us precise what should contain the cells of the queue (you may choose another representation if you prefer):

- a pointer to map
- a pointer to the next cell in the queue
- a pointer to the "mother" cell of the considered cell, i.e. the cell containing the map that generated the map contained in the current cell. This will be useful in order to build the plan
- the action that generated the considered cell (useful to build the plan)
- the depth of the cell in the search tree, which should be $1+$ the depth of the mother of the cell. This will be also useful to build the plan

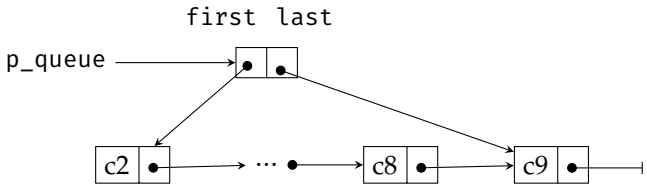
Second, you may have implemented a queue during session M6 using a double-linked list. This is a perfectly correct solution and you may reuse it. You may also use a simple linked list to implement a queue, inserting all elements at the beginning of the list and dequeuing the last element, but this implementation will not be efficient as you have to traverse the whole list to find the last element of the list (hence a linear complexity).
A simple solution to obtain a queue with a complexity in $O(1)$ for both `enqueue` and `dequeue` operations is to use two pointers to cells in the queue: one for the first cell in the queue, and another one for the last cell in the queue.
For instance, let us consider the following queue (only the pointer to the next cell is represented in the cells, in particular pointer to the mother cell is missing):



After "enqueueing" cell `c10` in the queue, the queue will be:



When "dequeueing" the queue, cell `c1` is returned and the queue is:



Of course, you have to pay attention to boundary corners cases: inserting an element in an empty queue, dequeuing a queue with only one element etc.
Your specification and implementation of the queue should be contained in the `queue_map.h` and `queue_map.c` files. You should provide tests for your queue in the `test_queue_map.c` file. For instance, enqueue some maps and verify that they are dequeued in the correct order.

5.6.3 Computing plan

In order to build the plan when finding a winning map and managing the case when there is no winning plan, you should define a structure `stats` containing:

- a boolean stating that there is a winning plan or not
- the corresponding plan (i.e. a string and its length) if it exists
- possibly other values like the number of explored nodes, the time elapsed to solve the game etc.

To build the plan, if you have a cell containing the winning map, you know that the length of the string needed to contain the plan is the depth of the node + 1 (to correctly add the character `'\0'` at the end of the string). You can then use the mother cell relation to "go up" the explored nodes and write the plan.

5.6.4 Freeing memory

When freeing memory, do not free memory allocated for the plan in the cells. All generated plans should be contained in the `linked_list_map` structure, therefore you should first deallocate memory allocated for cells in the queues, then deallocate maps in explored list or BST and then the cells in explored list or BST.

5.6.5 Testing your solver

The `tests-solver.csv` file in the `data` directory contains some plans for different initial maps (all of them are also in your `data` directory):

- there is only one map that has no solutions (`soko_dumb_imp.in`)
- solving time is in seconds and have been mesured on my machine

You should write a program in a `test-solver.c` file in order to verify that you obtain the same results (at least for "easy" games). Beware, some plans are really difficult to find!

When compiling your programs, use the `-O3` option in GCC: it aggressively optimize the compiled code. For this project, I personally have a code that is 4 times faster on my machine with this option!

5.7 Implementing the BST

To implement the BST you may use your own version of BST implemented in lab session M7. The BST will not contain **`int`** values, but maps. To compare maps, you may use the `strncmp` functions of the `string.h` library. `strncmp(s1, s2, size)` compares the `size` first characters of `s1` and `s2` and returns:

- a strictly negative number if `s1` $<$ `s2`
- a strictly positive number if `s1` $>$ `s2`
- `0` otherwise

5.8 Development plan

You may apply the following development plan:

1. `sokoban.h` and `sokoban.c`: define necessary structures and the `print_map` function.

   Notice that you may test your print function by defining a map structure directly in a program (i.e. without loading a file)

2. `loader.h`, `loader.c` and `test-loader.c`: define the loader and test it on several examples.

3. `sokoban.h`, `sokoban.c` and `test-move.c`: define a function to move the player in a given direction.

   You may test this function in a program by loading a map, executing the "move" function and printing the resulting map before writing the complete tests defined in section 5.3.

4. `sokoban.h`, `sokoban.c`, `test-replay.c`: define a function to replay a game and test it with the given examples.

5. `replay.c`: program to replay a game from the command line.

6. `play.c`: define an interactive player for Sokoban (your game implementation should now be correct if you have passed the tests for `move` and `replay`).

7. `linked_list_map.h`, `linked_list_map.c`, `test-linked-list-map.c`: define the linked list needed to store maps and test it.

8. `queue_map.h`, `queue_map.c`, `test-queue-map.c`: define the queue containing maps and test it.

9. `solver.h`, `solver.c`, `test-solver.c`: implement and test the solver.

10. `solve.c`: program to solve a Sokoban game given as an argument of the command line.

11. `bst_map.h`, `bst_map.c`, `test-bst-map.c`: define the BST containing maps and test it. Define also a new solve program using the BST.

Do not try to implement several functionalities at once, it will not work! Implement **and test** your code incrementally.
See section 8 to see how you will be graded.

5.9 Summary of files to produce

	specification	implementation
game	`sokoban.h`	`sokoban.c`
loader	`loader.h`	`loader.c`
list	`linked_list_map.h`	`linked_list_map.c`
queue	`queue_map.h`	`queue_map.c`
solver	`solver.h`	`solver.c`
test of loader		`test-loader.c`
test of move		`test-move.c`
test of replay		`test-replay.c`
replayer		`replay.c`
interactive player		`play.c`
test of list		`test-linked-list-map.c`
test of queue		`test-queue-map.c`
test of solver		`test-solver.c`
solver application		`solve.c`

6 Acknowledgements

This project is inspired from an project done during the "Introduction to Functional Programming in OCaml" online lecture by R. di Cosmo, Y. Regis-Gianas and R. Treinen. The lecture is available on FUN MOOC. I strongly recommend to attend this online lecture if you are interested in algorithms and programming.
The BMP icons for the game have been designed by Borgar Þorsteinsson and are licensed under the Creative Commons Attribution 3.0 license (see `https://github.com/borgar/sokoban-skins`).

7 Work to do and requirements

What you have to do and the project requirements are summarized in the following points. The project is due **03/29/2019 23h59**. **The code retrieved from your repository at this date will be considered as the final version of your project.**
You will have to commit working code for the Sokoban player (not solver) on **03/19/2019 23h59**. If not, penalty will be applied on your final grade.

7.1 Requirements about functionalities

[R1] you must implement the following functionalities: a program to play Sokoban, a solver for Sokoban game, an optimized version of the solver and tests as defined in section 5.

[R2] you may implement one of the following extensions: use a hash table to store the explored configurations, eliminate "wrong" configurations (for instance configurations for which boxes cannot be moved). You may also find yourself an interesting extension. These extensions will be taken into account in your final grade only if the basic functionalities are correctly working.

7.2 Requirements about implementation

[R3] your implementation must be written in C.

[R4] your code must compile with the `-std=c99 -Wall -Werror` options.

[R5] your code must work on the ISAE computers under Linux.

[R6] you must write

- a program in the `play.c` file that allows to play Sokoban
- a program in the `replay.c` file that shows that you can read an initial state and simulate the execution of a plan
- a program in the `solve.c` file that solves a Sokoban problem

If you have not implemented one functionality, do not code the corresponding part of the program.

[R7] you must write the test files as defined in section 5.

[R8] you may add programs to test your implementation. They must be located in files beginning with `app-`.

[R9] your data files (initial state) should be located in the `data` directory and have a `.in` suffix.

[R10] you must complete the given Makefile with at least a `compile-all` target that compiles all your programs. Use the Makefiles provided during the lab sessions.

[R11] your code must not produce errors when using the Valgrind tool.

[R12] you may reuse code **YOU** have implemented during the lab sessions

7.3 Requirements about code conventions

[R13] your header files must be documented, possibly with the Doxygen tool (look at lab sessions and the corresponding refcard on LMS). Do not document preconditions or postconditions, only functions behavior, their returns and their parameters. You should add normal C comments in your code to explain your implementation if they are relevant.

[R14] your C code must follow the code conventions explained during the lecture.

7.4 Requirements about your repository

[R15] you must commit your work at each successful functionality implementation. An intelligible message should be added to the commit. **Use the `add-files-svn` rule in the provided Makefile to add files you create in the repository.**

[R16] you must follow the following directories convention: your `.c` files must be in the `src` directory and `.h` files in the `include` directory. You may put the `.o` and executable files wherever you want.

8 Grading

Your final grade will be calculated as presented on table 4. **Beware, your tests will be used to evaluate your work!**

Part	Details	Grade	Comments
Sokoban game			
	Sokoban structures	1	
	loader	1	
	basic movement	1	
	replayer	2	
	interactive play	1	using GUI
Sokoban solver			
	linked list	1	you may use the solution of M5
	queue	2	
	solver	2	
	BST + solver	2	
Code quality			
	overall quality	4	variables naming, documentation etc.
	tests quality	2	
	your code does not compile	-5	
	your code produces errors with valgrind	-2	maximum malus, it depends on the number of errors
	incorrect use of SVN	-1	not useful commit messages etc.

Table 4: Grade details

A successful extension implementation will give you up to 4 points **if the basic functionalities are correctly implemented**.
Notice that you may have a really good grade even if you do not implement all the functionalities: **the quality of your code is really important!**

Your code will be analyzed by JPlag [2], a code plagiarism detector. DO NOT TAKE CODE FROM ANOTHER STUDENT, EVEN IF YOU TRY TO DISGUISE IT.
If you are convinced of plagiarism, you will obtain the "R" grade for the module.

A File Input/Output in C

If you want to read or write data to a file in C, you should use the structures and functions provided by the `stdio.h` header file. We will present here only simple cases when wanting to read or write formatted data from or to a text file. You may find more information in [3] or manual pages for the functions that are described in the following.

A.1 Opening and closing files

In order to read from files (or write to files), you must first open them. `stdio.h` provides a type, `FILE`, to represent a data stream from or/and to a particular file. Let us suppose that we want to read the content of a file `data.txt`. To open the file, we will use the `fopen` function and get a pointer to a `FILE` object:

```
FILE *p_file = fopen("data.txt", "r+");
```

Some remarks:

- `"data.txt"` represents the path to the file, it is here a relative path (the file `data.txt` must be in the current directory).
- `"r"` is the mode of the stream and specifies what you want to do with the file. You will mainly use the following modes:
  - `"r"` to read a file
  - `"w"` to write to a file
  - `"r+"` to read and write to a file
- if there are some errors when opening the file, the returned pointer is `NULL`. You should therefore test `p_file`.

When you have finished working with a file, **you should close it.** This is particularly true when writing to a file, as the file may have been put in central memory and the modifications you have done to the file might not be commited by the operating system (if you want to know more, you may refer to [1]). To close a file, simply call the `fclose` function on your pointer:

```
fclose(p_file);
```

A.2 Reading from files

If your file contains only **readable characters** and you know that data will follow a given format, then you may use the `fscanf` function. `fscanf` behaves like the `scanf` function we have seen in lab M1, but it works on files instead of standard input.
Let us look at the `fscanf` signature. It needs:

- a `FILE *` pointer to the file you want to read from
- a **format string** in the same format than the `printf` function
- several pointer variables to store what you read

Let us take an example. Let us suppose that we want to read **`int`** values from a file with the following format:

```
2 - 4
4 - 12
5 - 42
13 - 2323
```

Each line of the file has two **`int`** values separated by the string `" - "`. Let us suppose that `first_value` and `second_value` are **`int`** variables in which we want to store the read values, then the call to `fscanf` may be:

```
fscanf(p_file, "%d - %d", &first_value, &second_value);
```

The format string is `"%d - %d"`: we expect an integer, then a space, then `-`, then a space, then another integer. `fscanf` will return the number of input items successfully matched and assigned to the variables. Therefore, if `fscanf` returns 2, then the line has been correctly read and the variables assigned.

You may wonder how you may know when the file has been completely read. The `EOF` special value is returned by `fscanf` when it encounters the end of the file.

You will find in the `src` directory of your repository a simple program in the `read-file.c` file that tries to print all lines of a file respecting the previous syntax (cf. listing 1). You can compile it with the following command (there is also a corresponding target in the provided Makefile):

```
gcc -std=c99 -o read-file read-file.c
```

You can execute it on the file `data.txt` with the following command[2]:

```
./read-file data.txt
```

Try it on different inputs to verify that it works correctly, particularly when there are errors in the data file.

Listing 1: A simple program to read text files

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    int first_int     = 0;
    int second_int    = 0;
    int line_nb       = 1;
    int fscanf_result = 0;

    fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);

    while (fscanf_result != EOF) {
        if (fscanf_result != 2) {
            fprintf(stderr, "Line number %d is not syntactically correct!\n",
                    line_nb);
            exit(EXIT_FAILURE);
        }

        printf("first value at line %d: %d\n", line_nb, first_int);
        printf("second value at line %d: %d\n", line_nb, second_int);

        line_nb = line_nb + 1;
        fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);
    }

    fclose(p_file);
```

[2] `data.txt` must be in the same directory in the command, but you can use relative or absolute paths to call `read-file` on files that are not in the current directory.

```
    p_file = NULL;

    return 0;
}
```

A.3 Writing to files

If you want to write formatted content to a file, you must first open the file with the `"r+"` or `"w"` modes with `fopen`. You then use the `fprintf` function. It behaves like the `printf` function, but it takes as first argument a `FILE *` pointer to the file you want to write to.
For instance, the program presented on listing 2 writes the factorial of the 10 first natural numbers to the file `fact.txt`. The program is available in the `src` directory of your repository and a compilation target is provided in your Makefile.

Listing 2: A simple program to write some computations in a text file

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *p_file = NULL;

    p_file = fopen("fact.txt", "w");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot write to file fact.txt!\n");
        exit(EXIT_FAILURE);
    }

    int fact = 1;

    for (int i = 0; i < 10; i++) {
        fprintf(p_file, "%d! = %d\n", i, fact);

        fact = fact * (i + 1);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

References

[1] Ulrich Drepper. “What Every Programmer Should Know About Memory”. Nov. 21, 2007. URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[2] Karlsruhe Institute of Technology. *JPlag – Detecting Software Plagiarism*. 2016. URL: https://jplag.ipd.kit.edu/.

[3] Wikibooks. *C Programming, File IO — Wikibooks, The Free Textbook Project*. 2015. URL: https://en.wikibooks.org/wiki/C_Programming/File_IO.

License CC BY-NC-SA 3.0