CS510 HW Assignment 1

Due date: 10/15/2018

Programming Assignment (100 points)

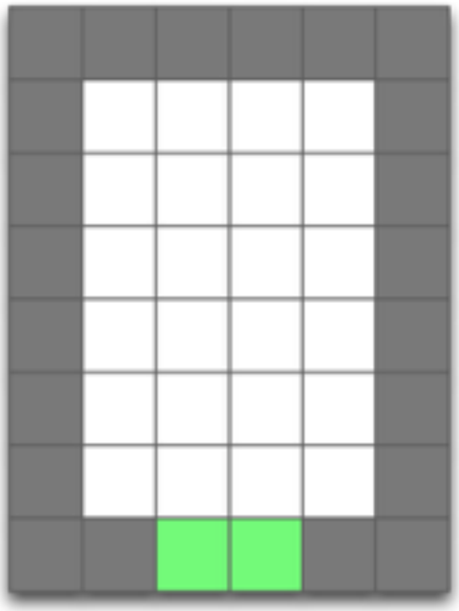
Part 1

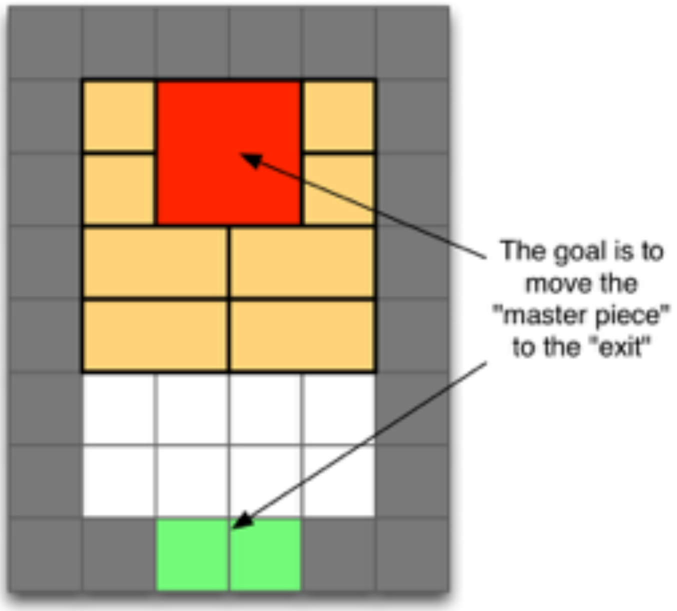
Sliding Brick Puzzle

Many of you might have played one or another version of the "sliding brick puzzle" (SBP). If you have not, you can play one here. You can also find a video here. For the next several assignments, you will create a program that can solve the SBP. In this assignment you will just have to create data structures and functions to represent the game state, and perform the various needed operations such as: determining the set of valid moves, execute the moves or determine whether we have solved the puzzle.

- A sliding brick puzzle is played on a rectangular *w* by *h* board (*w* cells wide and *h* cells tall). Each cell in the board can be either *free*, have a *wall*, or be the *goal*.
- On top of the board (over some of the free cells) there is a set of solid pieces (or *bricks*) that can be moved around the board. One of the bricks is special (the *master brick*).
- A move consists of sliding one of the bricks one cell up, down, left or right. Notice that bricks collide with either walls or other bricks, so we cannot move a brick on top of another. Bricks can only slide, they cannot rotate nor be flipped.
- To solve the puzzle, we have to find a sequence of moves that allows you to move the master brick on top of the goal cell(s). No other pieces are allowed to be placed on top of the goal cell(s).

Here is an illustration of a particular configuration of a SBP (but if you still do not understand how the game works, just see this video, or play the game at one of the links above).

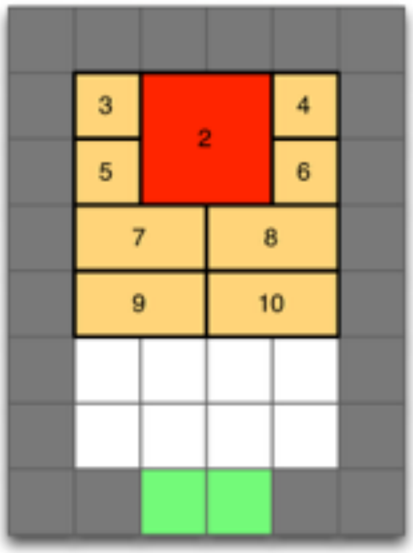

a 6 x 8 board



a 6 x 8 with 9 pieces

Complete this assignment in C/C++ or Python:

1.A: State representation
In this task, you will write code to represent the game state, load a game state from disk, and display a game state in the screen. Depending on the programming language you choose, you will have to create a *class* (C++) or just a set of functions (C) for completing this task. We will represent the game state as an integer matrix, as shown in this example:



$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 2 & 4 & 1 \\ 1 & 5 & 2 & 2 & 6 & 1 \\ 1 & 7 & 7 & 8 & 8 & 1 \\ 1 & 9 & 9 & 10 & 10 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & -1 & -1 & 1 & 1 \end{pmatrix}$$

The matrix will have the same dimensions as the game board, and each position in the matrix has the following meaning:

- -1: represents the goal
- 0: means that the cell is empty
- 1: means that there is a wall
- 2: means that the master brick is on top of this cell
- any number higher or equal than 3: represents each of the other bricks

Thus, as you can see, each piece in the board is assigned an integer: the master brick is assigned number 2, and the other bricks are assigned numbers starting from 3.

- Write a function that allows you to load a game state from disk. The input to the function should be just the name of the file. The file format that you have to use is the following:

```
w,h,
Row 1 of the matrix with values separated by commas,
...
Row h of the matrix with values separated by commas,
```

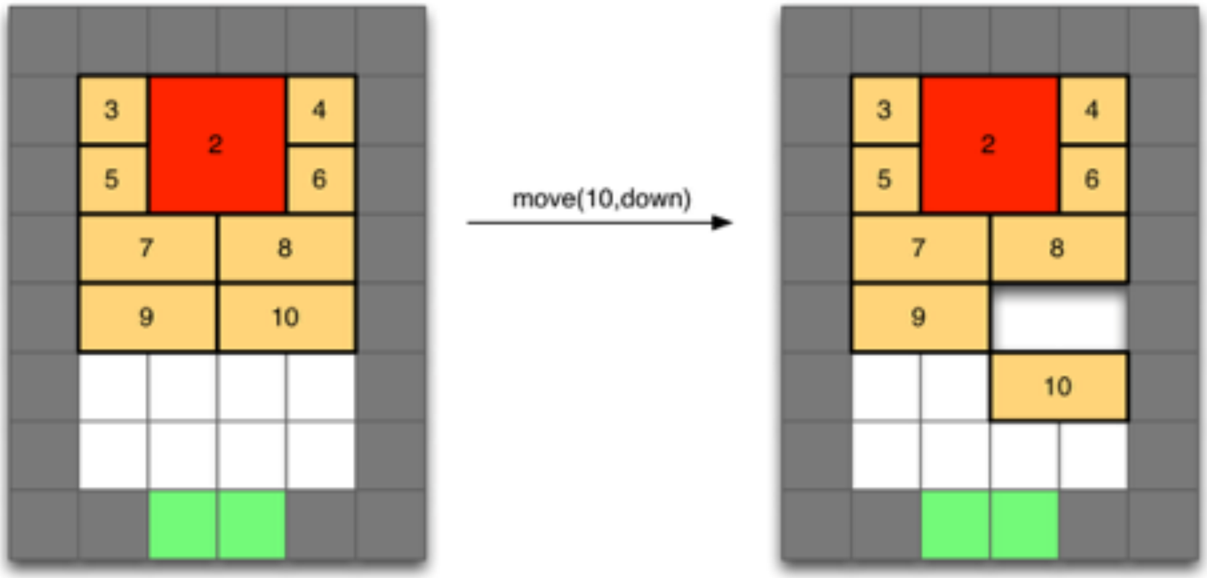
  You can use the four included files as examples: SBP-level0.txt, SBP-level1.txt, SBP-level2.txt, SBP-level3.txt.

- Write a function that outputs the game state on the screen. For example, if you load the file SBP-level0.txt, the display state function must output the following to the screen **(pay attention to spaces and newlines)**
```
5,4,
1,-1,-1,1,1,
1,0,3,4,1,
1,0,2,2,1,
1,1,1,1,1,
```

- Write a function that *clones* a state. That is that returns a separate state that is identical to the original one.

1.B: Puzzle Complete Check
Write a function that returns *true* if a given state represent a solved puzzle (i.e. if the master brick is on top of the goal). Notice that checking this is very easy, since you only have to go over the matrix, and see if there is any cell with the value -1. If there is, that means that the puzzle is not solved, if there is not, then the puzzle is solved (since only the master brick can cover the goal cells). For example, your function should return false with SBP-level0.txt, but true with SBP-level0-solved.txt

1.C: Move Generation
Since each piece has a unique integer identifier, we will represent moves as a pair (piece,direction). Each piece can only move one cell at a time, in any of the four directions. For example a possible move in the following board is (10,down):

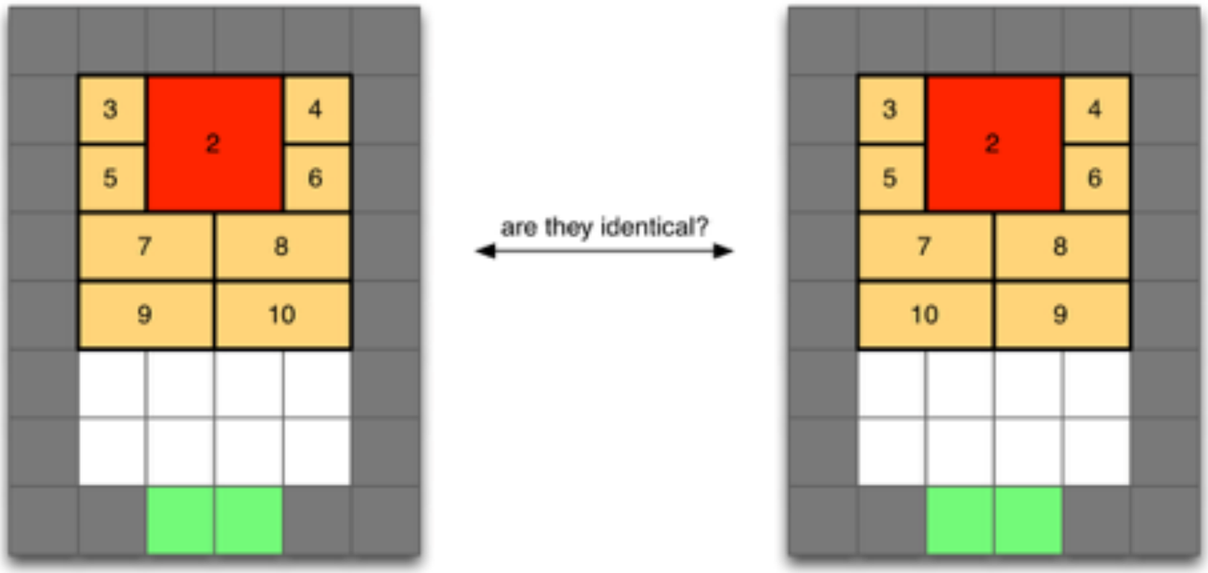


- Write a function that, given a state and a piece, returns a list of all moves the piece can perform (notice that the list can be empty). If you are using C++, define a class and, if using C, a struct to represent a "move". Feel free to encode the direction (up, down, left, right) however you want. For example, you can use a *character* to represent direction ('u','d','l','r'), or an integer. That is up to you.

- Write a function that, given a state, returns all moves that can be performed on a board (that is, the union of the moves that each individual piece can perform).
- Implement a function 'applyMove' that, given a state and a move, performs the move in that state.
- Implement a function 'applyMoveCloning' that, given a state and a move, returns a new state, resulting from applying the move (i.e. first clones the state, and then applies the move).

1.D: State Comparison
Write a function that compares two states, and returns *true* if they are identical, and *false* if they are not. Do so using the simplest possible approach: just iterate over each position in the matrix that represents the state, and compare the integers one by one. If they are all identical, the states are identical, otherwise they are not.

1.E: Normalization
Notice that the previous state comparison function has a problem. Consider the following two states:

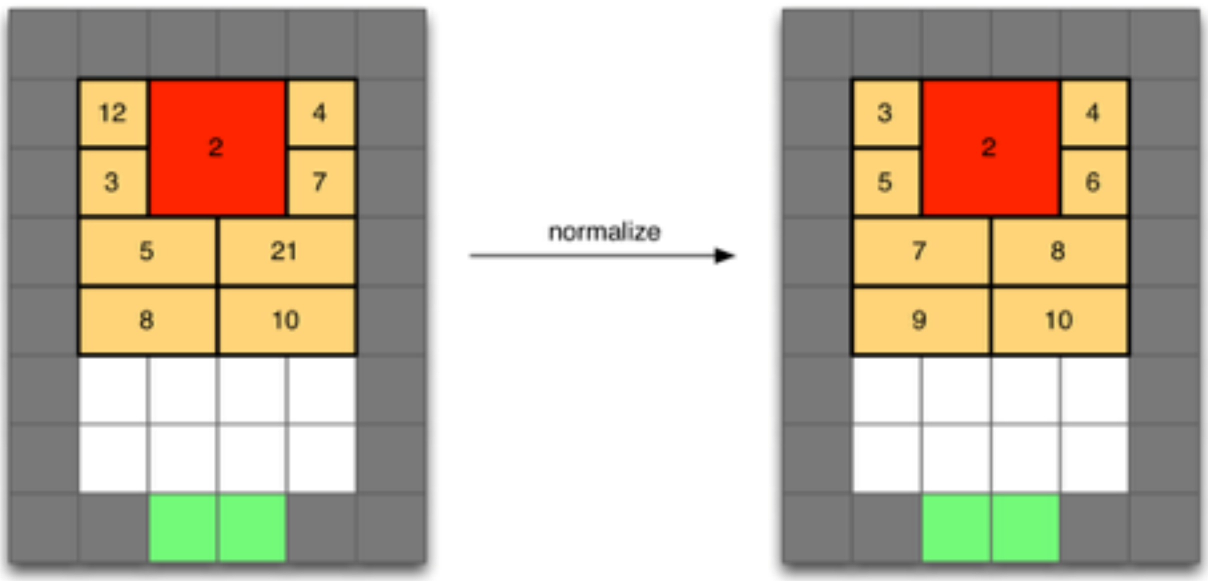


The previous function will consider these two states as different. However, it's quite obvious that the states are equivalent. In order to solve this problem, we are going to define a *normal form* for a state:

If we give an index to each cell on the board starting from the top-left corner and going down row by row from left to right (top-left corner has index 0, the cell right next to it has index 1, etc.), then we can assign an index *I(b)* to a brick *b*, as the smallest index of the cells covered by *b*.

Now, a state is in normal form if, for any two bricks (that are not the master brick) with numbers *n* and *m* such that $n < m$, it holds that $I(n) < I(m)$.

Write a function that, given a state, transforms it into normal form. See the expected effect of this function in this image:



Notice that this is simpler than it seems. It can be done with the following algorithm:

```
int nextIdx = 3;
for(i = 0;i < h;i++) {
  for(j = 0;j < w;j++) {
    if (matrix[j][i]==nextIdx) {
      nextIdx++;
    } else if (matrix[j][i] > nextIdx) {
      swapIdx(nextIdx,matrix[j][i]);
      nextIdx++;
    }
  }
}
```

Where the swapIdx function is:

```
swapIdx(int idx1,int idx2) {
  for(i = 0;i < h;i++) {
    for(j = 0;j < w;j++) {
      if (matrix[j][i]==idx1) {
        matrix[j][i]=idx2;
      } else if (matrix[j][i]==idx2) {
        matrix[j][i]=idx1;
      }
    }
  }
}
```

This normalization function will be very useful in the following assignments to compare game states, and see if they are equivalent or not. You can test if your version works with this state SBP-test-not-normalized.txt. Make sure you obtain the same result as in the figure above.

1.F: Random Walks
Write a function that, given a state and a positive integer *N*, performs the steps following: 1) generates all moves that can be executed on the board in its current state, 2) selects one at random, 3) executes it, 4) normalizes the resulting game state, 5) if we have reached the goal, or if we have executed N moves, stops; otherwise, goes back to step (1).

Please print both the move and the game state on screen after each iteration of the method. The required output format is discussed below.

Part 2

Using the code you wrote for part 1, write:

- A function that solves a given sliding bricks puzzle using a **breadth-first search**.
- A function that solves a given sliding bricks puzzle using a **depth-first search.**
- A function that solves a given sliding bricks puzzle using an **iterative deepening search**

Notice that the search space is a graph, so you will have to keep track of all the states visited so far, and make sure your algorithm does not get stuck in loops.

When the solution is found, it should be printed to the screen. Print the list of moves required to solve the state, and the final state of the puzzle, for example (**pay attention to spaces and newlines**):

```
(2,left)
(4,down)
(3,right)
(2,up)
(2,up)
5,4,
1,2,2,1,1,
1,0,0,3,1,
1,0,0,4,1,
1,1,1,1,1,
```

Main Function
Write a main function that, when the program is run, calls the necessary functions you wrote above in order to accomplish the following:

1. Load the file SBP-level0.txt from the current directory and execute a random walk from 1.F with N=3. The output must be of the form (**pay attention to spaces and newlines**):

```
5,4,
1,-1,-1,1,1,
1,0,3,4,1,
1,0,2,2,1,
1,1,1,1,1,

(2,left)

5,4,
1,-1,-1,1,1,
1,0,3,4,1,
1,2,2,0,1,
1,1,1,1,1,

(2,right)

5,4,
1,-1,-1,1,1,
1,0,3,4,1,
1,0,2,2,1,
1,1,1,1,1,

(3,left)

5,4,
1,-1,-1,1,1,
1,3,0,4,1,
1,0,2,2,1,
1,1,1,1,1,
```

2. Load file SBP-level1.txt from the current directory and display to the screen the solution obtained using breadth-first search in the format specified at the beginning of the “Part 2” section, followed by a line of the form:

#nodes time length

where *#nodes* is the number of nodes explored, *time* is the time the search took in seconds and fractions of seconds (e.g., 2.53 for 2 seconds and 53/100) and *length* is the length of the solution found.

3. Display to the screen the solution for SBP-level1.txt obtained using depth-first search in the format specified at the beginning of the “Part 2” section, followed by the line that shows number of nodes explored, time taken and length of the solution.
4. Display to the screen the solution for SBP-level1.txt obtained using iterative deepening search in the format specified at the beginning of the “Part 2” section, followed by the line that shows number of nodes explored, time taken and length of the solution.

IMPORTANT: write all of the code above to be run from command line and to display its output to the console. Do not create any graphical user interfaces and do not ask the user for any input.

To help you with testing, additional files have been provided. Your functions will likely be able to handle SBP- level2.txt, SBP-level3.txt. Using these search strategies, it is unlikely that your program will handle puzzles much larger than those (in next assignment, you will implement much better strategies). In case you want to test out the limits of your program, you can use these more complex puzzles: SBP- bricks-level1.txt, SBP-bricks-level2.txt, SBP-bricks-level3.txt, SBP-bricks-level4.txt, SBP- bricks-level5.txt, SBP-bricks-level6.txt, SBP-bricks-level7.txt.

Extra Credit

Implement A* search for solving a given sliding bricks puzzle

What to Submit

All homework for this course must be submitted using Bb Learn. Do not e-mail your assignment to a TA or Instructor. If you are having difficulty with your Bb Learn account, you are responsible for resolving these problems with a TA, an Instructor, or someone from IRT, before the assignment is due. If you have any doubts, complete your work early so that a TA or someone from IRT can help you if you have difficulty.

For this assignment, you must submit:

1. Your C/C++ or Python source code (make sure it is well commented and works on tux)
2. A written documentation for your program, including your name, what files are and what they contain, also the compilation and execution instructions.
3. A Makefile or a Bash script named hw1.sh (for use on tux) in a way that
    i. Running the executable file named “hw1” generated by command “make” in the same directory
    ii. Or running ./hw1.sh

   must implement the main function and associated functions/methods described above on tux. **(without this file, your code cannot be compiled nor verified)**
4. A plain text file called “output-hw1.txt” and **generated on tux (very important!)** which shows the output your program generates when run. You can easily generate this file using redirection, e.g.: “./hw1 > output-hw1.txt”.
5. Use a compression utility to compress your files into a single file (with a .zip extension) and upload it to the assignment page.

Grading Rubric:

Item	Points
Random Walk (proper algorithm, well commented)	10
BFS (proper algorithm, optimal solution, well commented)	22
DFS (proper algorithm, optimal solution, well commented)	22
IDS (proper algorithm, optimal solution, well commented)	22
README	9
Output-hw1.txt generated on tux	15
A* (proper algorithm, optimal solution, well commented)	10 (extra)

Important: Almost all of the rest of your assignments will build on top of this assignment. So, make sure that you do a solid job with its design and implementation. Otherwise, you will have problems in the future.

Academic Honesty

You must compose all program and written material yourself. All material taken from outside sources must be appropriately cited. If you need assistance with this aspect of the assignment, see a TA during office hours.