CS510 HW Assignment 1

Due: Sunday Jan. 17th 11:59pm

Part 1: Written Problems (30 points)

1.A: Problem Definition (20 points, 10 each)
Write a formal specification of the following problems as we did in class (PEAS: performance measure, environment, actions, and sensors). Be as explicit as possible, providing
1) (5pts per problem) a specific data representation for the states and actions and
2) (5pts per problem) showing how the results of applying an action to a state are determined.
You do not need to specify an algorithm for solving the problem; possible algorithms for choosing between applicable actions will be covered later:

- **Small Towers of Hanoi**: there are three vertical rods (R1, R2, R3), and three disks (D1, D2, D3). Each disk has a hole in the center, so that it can slide into any of the three rods. Disk D1 is small, disk D2 is medium-sized and disk D3 is large. Initially all three disks are stacked on rod R1, with the largest at the bottom, and the smallest at the top. The goal is to move all the disks to another rod, with the following rules: 1) only one disk can be moved at a time, 2) each move consists of taking the upper disk at one rod, and sliding it onto another rod, placing it on top of any disk already present in that rod, 3) No disk may be placed on top of a smaller disk. (see wikipedia for more info on the Towers of Hanoi, if necessary)
- **Pac Man**: we all know the game of Pac Man. For this exercise, consider a simplified version of Pac Man, where the map is a rectangular grid. At each position of the map there might be: a wall, a pellet (that Pac Man has to eat), Pac Man, or nothing. There are no ghosts, fruits, or any other power-ups. At each time step, Pac Man moves one cell up, down, left or right, the game ends when Pac Man eats all the pellets.

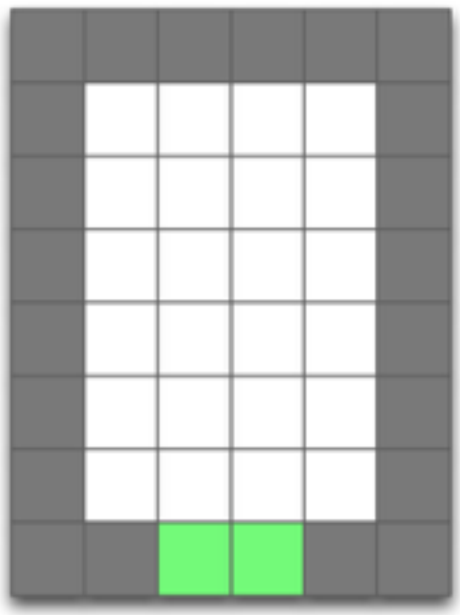
1.B: Forward-Backward Search (10 points)
Eloise claims to be a descendant of Benjamin Franklin. Which would be the easier way to verify Eloise's claim: By showing that Franklin is one of Eloise's ancestors or by showing that Eloise is one of Franklin's descendants? Why?

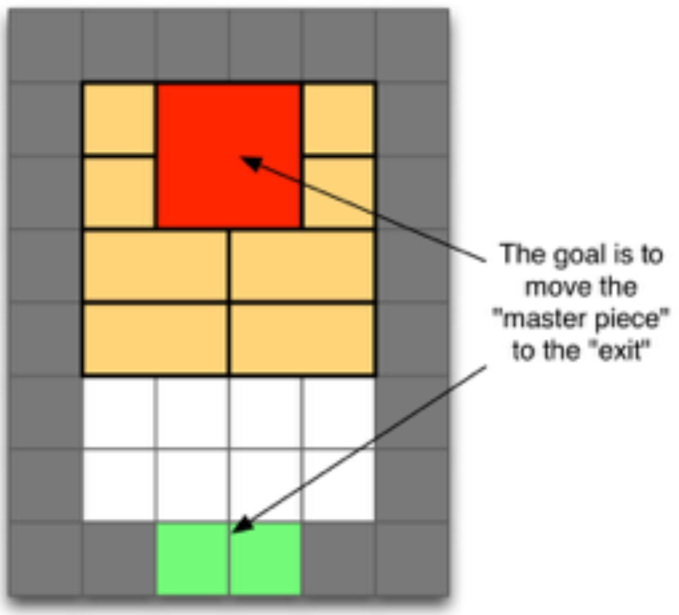
Part 2: Programming Assignment (70 points)

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. And if you like it, you can play a rather challenging one here (although this last one, unfortunately, only runs on Windows machines). For the next several assignments, you will create a program that can solve the SBP. In this assignment you will have to create data structures and functions to represent the game state, perform the various needed operations such as: determining the set of valid moves, execute the moves and determining whether a we have solved the puzzle, and implement some simple searching algorithms.

- 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. No other pieces are allowed to be placed on top of the goal

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 in one of the links above)


a 6 x 8 board

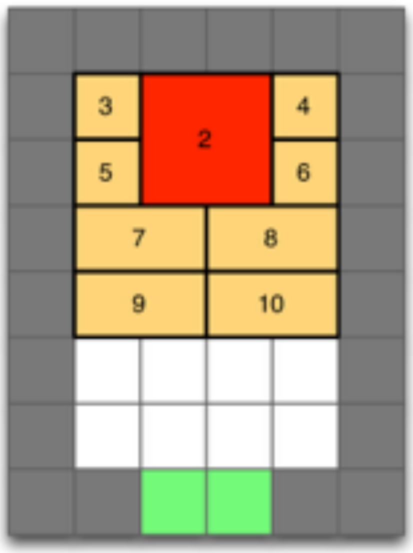


a 6 x 8 with 9 pieces

You can complete this assignment in C, C++, Java, Python, or Lisp:

2.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* (Java, C++) or just a set of functions (C, Lisp) 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 named **loadGameState** 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.

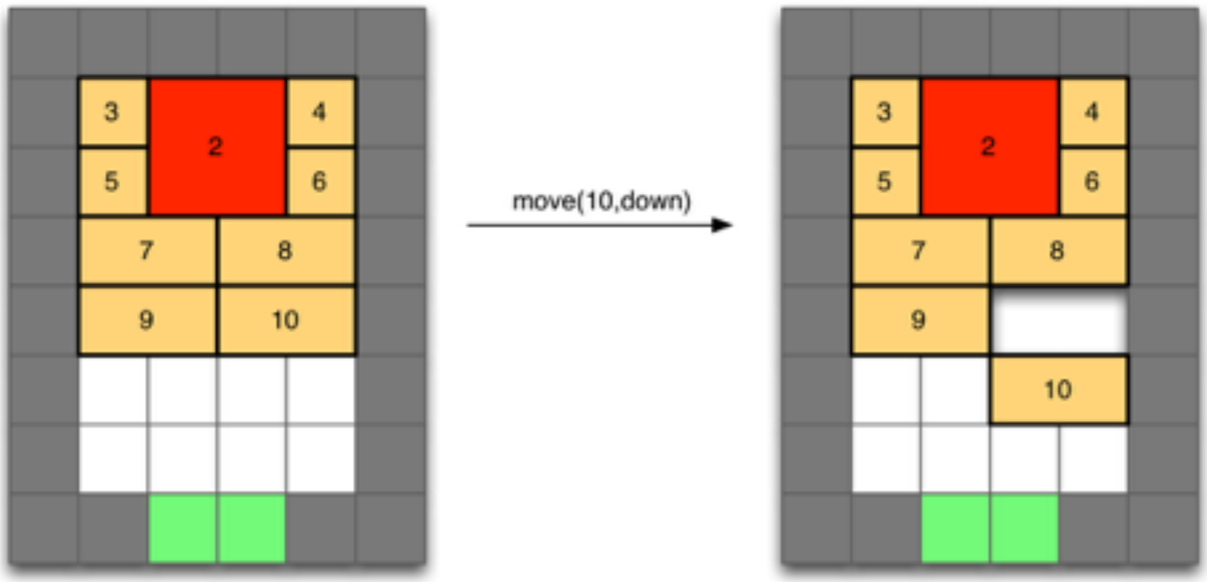
- Write a function named **outputGameState** that outputs the game state in the screen. For example, if you load the file SBP-level0.txt, the display state function should output EXACTLY the following to the screen:
  ```
  5,4,
  1,-1,-1,1,1,
  1,0,3,4,1,
  ```

```
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.

2.B: Puzzle Complete Check
Write a function named **gameStateSolved** 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

2.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 named **allMovesHelp** that given a state and a piece, returns a list of all the moves the piece can perform (notice that the list can be empty). If you are using Java or C++, define a class or a struct to represent a "move". If you are using Lisp, etc. you can just return a list of two elements, representing the move. Feel free to represent the direction (up, down, left, right) however you want. This function should return the moves sorted first by the piece number and then by move in the order up, down, left, right.
- Write a function named **allMoves** that given a state, it returns all the moves that can be done in a board (that is the union of the moves that each individual piece can perform).

This function should return the moves sorted first by the piece number and then by move in the order up, down, left, right.

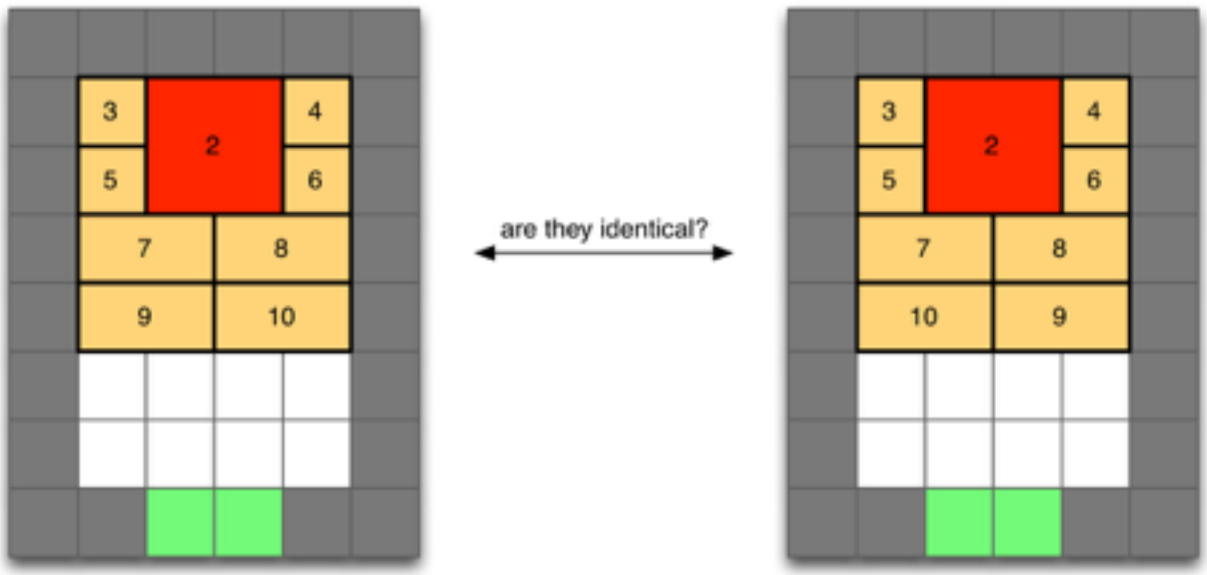
- Implement a function named **applyMove** that given a state and a move, performs the move in the state.
- Implement a function named **applyMoveCloning** that given a state and a move, returns a new state, resulting from applying the move (i.e. first clones the state, and then applies the move).

2.D: State Comparison

Write a function named **stateEqual** 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.

2.E: Normalization

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

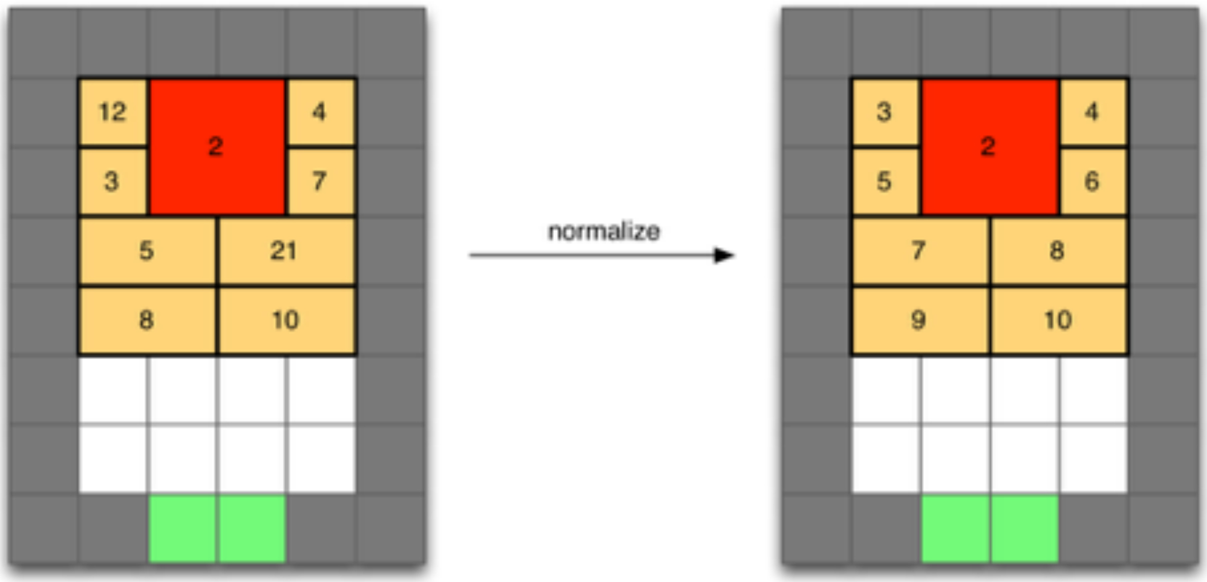


The previous function will consider these two states as different. However, it's quite obvious that the states are equivalent. In order to solve this problem, we are going to define a *normal form* for a state:

If we give an index to each cell in 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 covered by the brick. This is illustrated in the following figure:

Now, a state is in normal form if, given two bricks (that are not the master brick) with numbers n and m. If $n < m$ then we know that $P(n) < P(m)$.

Write a function named **normalizeState** that given a state, it 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 does this:

```
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.

2.F: Random Walks
Write a function that given a state and a positive integer *N*, does the following: 1) generate all the moves that can be generated in the board, 2) select one at random, 3) execute it, 4) normalize the resulting game state, 5) if we have reached the goal, or if we have executed N moves, stop; otherwise, go back to 1.

Please print both the move and the game state on screen after each iteration of the method. For example, loading the file SBP-level0.txt, and executing a random walk with N = 3, a possible output can be this:

```
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.G: Uninformed Search
Using the code you wrote in the preceding sections 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.

Notice that the search space is a graph, so as we will discuss in class 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 screen. Print the list of moves required to solve the state, and the final state of the puzzle, for example:

```
(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,
```

Together with the source code, turn in (in a plain text file called 'output-part2.txt') the output that your program generates for the following four levels: SBP-level0.txt, SBP-level1.txt, SBP-level2.txt, SBP-level3.txt. Also, report how many nodes are explored, how much time does the search take, and the length of the solution found

Using these search strategies, it is unlikely that your program handles puzzles much larger than the ones linked above (in assignment 2 you will implement much better strategies!). But 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.

What to Submit

All homework for this course must be submitted using Blackboard. Do not e-mail your assignment to a TA or Instructor. If you are having difficulty with your Blackboard 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. A PDF document with your answers to the "Written problems to be turned in.” Documents in ANY other formats will receive a zero for that part of the assignment.
2. Your C/C++/Java/Lisp/PROLOG/Python… source code.
   - Remember that your code MUST run on TUX. Development on other platforms is acceptable, but for grading it must run on TUX. Code that does not will receive a zero.
   - Note ALL functions that have names specified in the above text MUST have those given names. Failure to do so will result in a zero for your code.
3. Written documentation for your program including documentation about compiling and running your code for each of random search, breadth-first, and depth-first search.
4. A text file with the output specified for part 2.G.
5. A Make file for your code at the top level of the archive. Regardless of the language you use, you MUST include a makefile that compiles your code. The TA should be able to download, unzip, and run “make” to build your code on TUX. Failure to include this will result in a zero for your code.

Use a compression utility to compress your files into a single file (with a .zip extension) and upload it to the assignment page.

Important: Several future 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, including answers to book questions. All material taken from outside sources must be appropriately cited. If you need assistance with this aspect of the assignment, see a consultant during consulting hours