Artificial Intelligence

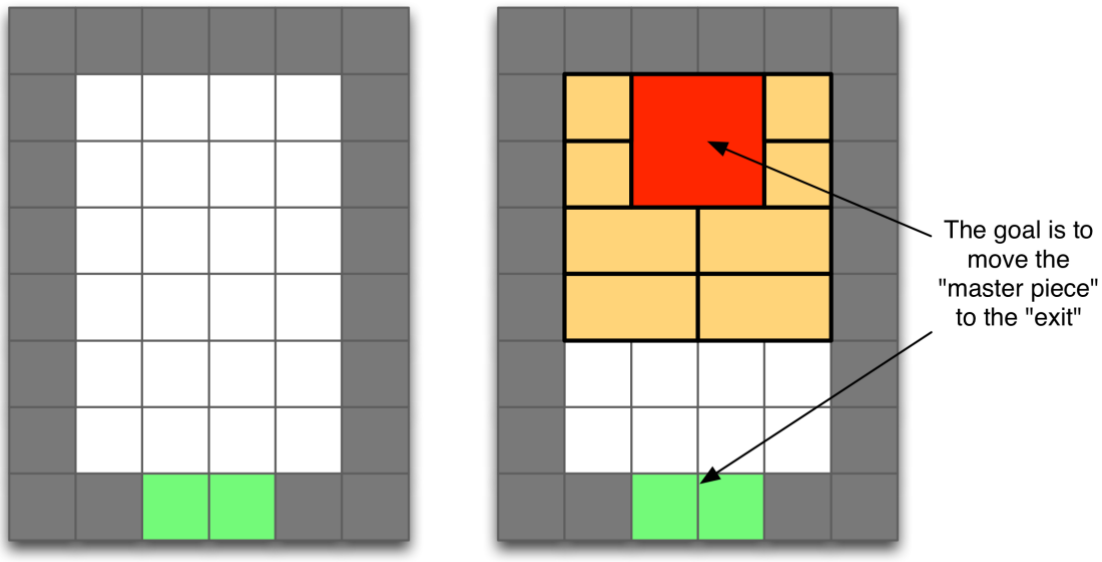
Sliding Brick Puzzle (12 points)

Part I: State Representation and Move Generation

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 two assignments, you will create a program that can solve the SBP.

- 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 must 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 an 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

In this part, 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, executing the moves, or determining whether we have solved the puzzle. The code for these assignments should be written in Python to run on tux.cs.drexel.edu.

Implementation Setup

The various parts of this assignment will require a shell script that can pass an argument to your code allowing us to be able to run your code with a consistent interface. However, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

To this end, please create a shell script **run.sh** that calls your code and includes 2 command-line arguments that are passed to your code. For example, a shell script for **python3** might look as follows:

```
#!/bin/sh
if [ "$#" -eq 3 ]; then
      python3 sbp.py "$1" "$2" "$3"
elif [ "$#" -eq 2 ]; then
      python3 sbp.py "$1" "$2"
else
      python3 sbp.py "$1"
fi
```

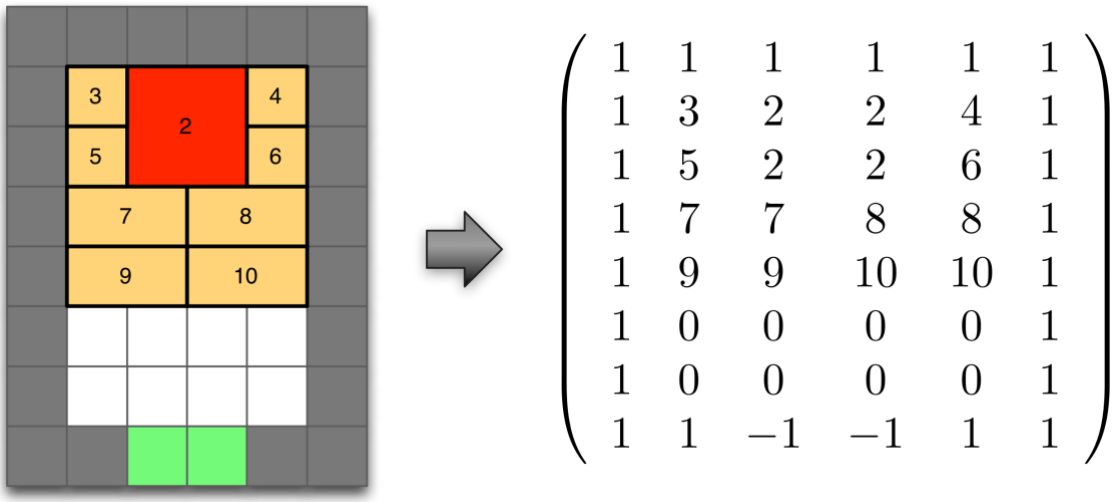
Your code will need to accept these two arguments and use them properly for each particular command. As you'll see in the sections below, running the code will have the general format:

```
sh run.sh <command> [<optional-argument>]
```

Again, this scheme will allow you to test your code thoroughly, and also allow us to test your code using a variety of arguments.

State Representation (1 pts)

In this task, you will write code to represent the game state, load a game state from disk, and display a game state on the screen. You will have to create a *class* (recommended) or just a set of functions to complete these tasks.
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 greater than 2: represents each of the other bricks

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.

We will assume the input state will have the following format: (make sure to export errors in case the input string is wrong)

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

Four different example levels are included with the zip file: *SBP-level0.txt*, *SBP-level1.txt*, *SBP-level2.txt*, and *SBP-level3.txt*.

Your first task is to implement a function that allows you to load a game state from a disk. The input to the function should be just the name of the file.

After that, you should implement a function that can print the board (game state) in ASCII characters. For example, if you load the file *SBP-level0.txt*, the display state function should output 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,
```

For our testing purposes, you should make this print function runnable from the command line with the "**print**" command as the first argument and the filename as the second argument where the file represents a board configuration or state. Here are two examples of running from the command line:

```
> sh run.sh print "SBP-level1.txt"

5,5,
 1, 1, 1, 1, 1,
 1, 3, 2, 2, 1,
 1, 0, 4, 5, 1,
-1, 0, 6, 7, 1,
 1, 1, 1, 1, 1,

> sh run.sh print "SBP-level2.txt"

6,5,
 1, 1, 1, 1, 1, 1,
 1, 0, 3, 2, 2, 1,
 1, 0, 3, 4, 5, 1,
-1, 6, 6, 7, 8, 1,
 1, 1, 1, 1, 1, 1,
```

Cloning

Implement a function that *clones* a state. The function should return a separate board that is identical to the original one. This function will become important in later parts.

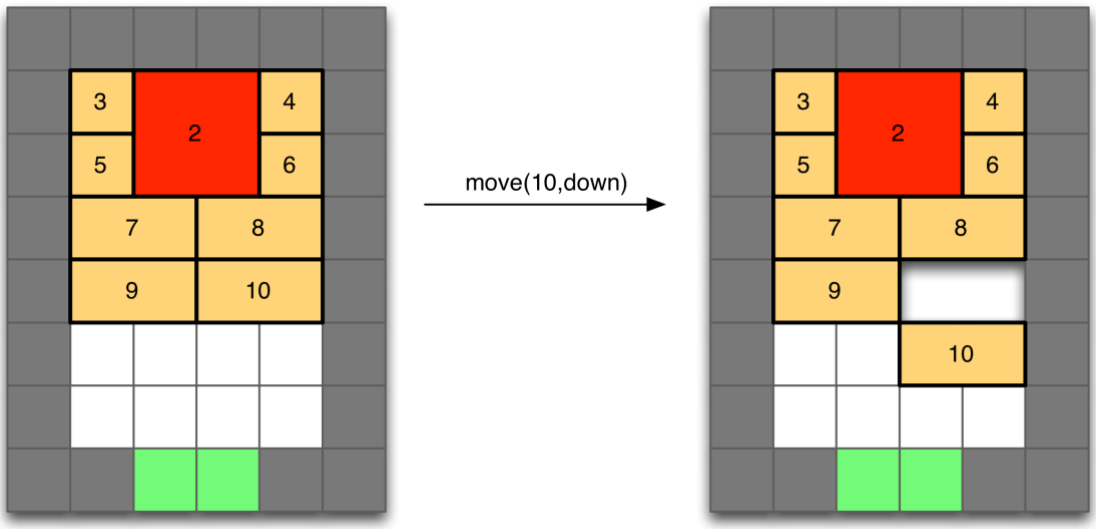
Identifying Solutions (1 pts)

Write a function that returns *true* if a given state represents a solved puzzle (i.e., if the master brick is on top of the goal). Notice that checking this is very easy, you only need 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). Then, augment the possible commands to accept a "**done**" command that prints *true* or *False* depending on whether the given board, read from a file, is at the solution state or not. For example:

```
> sh run.sh done "SBP-level0.txt"
False
> sh run.sh done "SBP-level0-solved.txt"
True
```

Available Moves (2 pts)

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):



For this part, implement a function that returns all the moves that can be done in a given board.

To make this easier, first, we can implement a function that, given a state and a piece, it returns a list of all the moves the piece can perform (notice that the list can be empty). The moves are *up*, *down*, *left*, and *right*. After that, implement a function that, given a board, 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).

Augment the possible commands to accept a "**availableMoves**" command that prints all the moves that can be done in a given board, read from a file. For example:

```
> sh run.sh availableMoves "SBP-level1.txt"
(3, down)
(4, left)
(6, left)
```

Apply Move (2 pts)

Implement a function that, given a state and a move, it performs the move in the state.

For our testing, we should be able to run this command, from the command line with the "**applyMove**" command as the first argument, a file, representing the board, as the second argument, and the move as the third. Running this command from the command line should output the new board after applying the move. For example:

```
> sh run.sh applyMove "SBP-level1.txt" "(3,down)"

5,5,
 1, 1, 1, 1, 1,
 1, 0, 2, 2, 1,
 1, 3, 4, 5, 1,
-1, 0, 6, 7, 1,
 1, 1, 1, 1, 1,
```

Also, for this part, implement a function 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). This will be helpful in future parts.

State Comparison (1 pts)

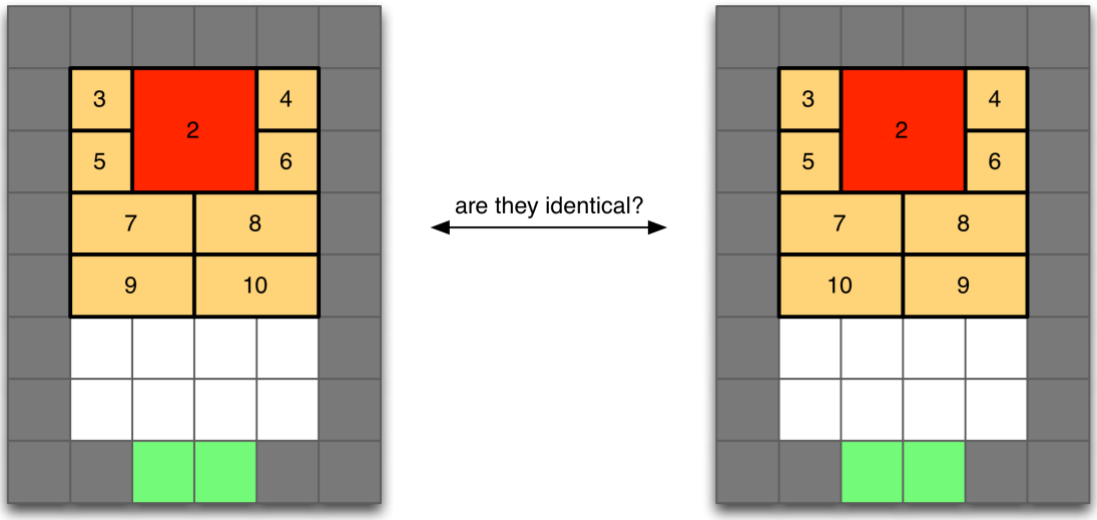
Write a function that compares two states, and returns *true* if they are identical, and *false* if they are not. Do so using the simplest possible approach: just iterate over each position in the matrix that represents the state, and compare the integers one by one. If they are all identical, the states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

Same as before, for this part we will test your code from the command line with the "**compare**" command as the first argument, and two file as second and third arguments:

```
> sh run.sh compare "SBP-level0.txt" "SBP-level0-test.txt"
False
> sh run.sh compare "SBP-level0.txt" "SBP-level0.txt"
True
```

Normalization (2 pts)

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

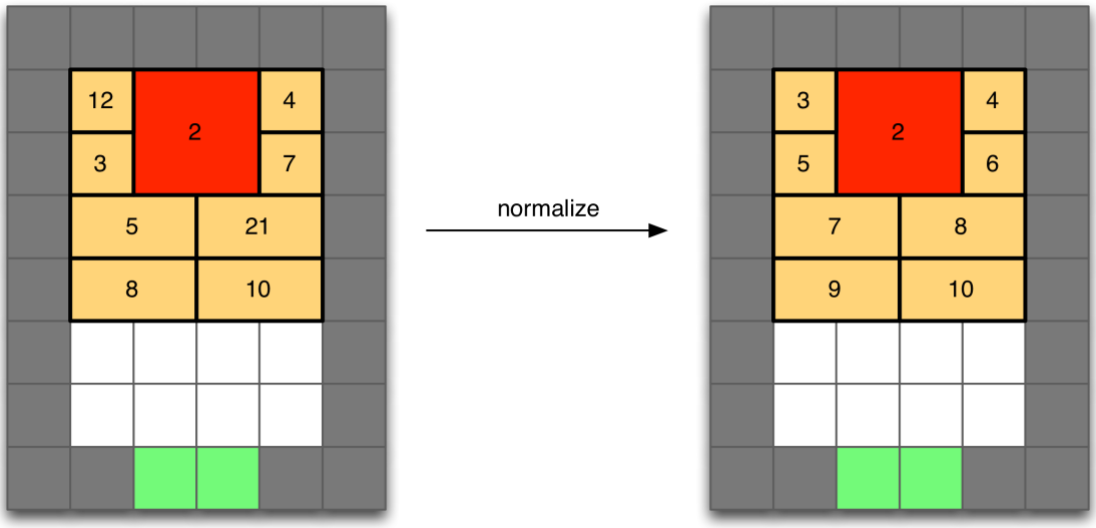


The previous function will consider these two states as different. However, it's obvious that the states are equivalent. In order to solve this problem, we are going to define a *normal form* for a state:

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 (the 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 that, given a board, it returns its 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 simply 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 boards and see if they are equivalent or not. You can test if your version works with this board *SBP-test-not-normalized.txt*. Make sure you obtain the same result as in the figure above. Also, we should be able to run it from the command line with the **norm**" command as the first argument and the file as the second:

```
> sh run.sh norm " SBP-test-not-normalized.txt"

6,8,
 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,
```

Random Walks (3 pts)

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.

For this part, we should be able to run a random walk from the command line with the "**random**" command as the first argument, the file as the second, and N as the third. This command should print both the move and the game state on the screen after each iteration of the method. For example, by loading the file *SBP-level0.txt*, and executing a random walk with *N = 3*, a possible output can be this:

```
> sh run.sh random " SBP-level0.txt" 3

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,
```

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit:

- Your Python code for this assignment and the board files.
- Your **run.sh** shell script that can be run as noted in the examples.
- We should be able to run all the functions mentioned above (**print**, **done**, **availableMoves**, **applyMoves**, **compare**, **norm**, and **random**) using your **run.sh** shell script.
- A PDF document with written documentation containing a few paragraphs explaining your program and results showing testing of your routines. If you did anything extra, discuss it here.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—do not email 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 or someone from IRT before the assignment is due. If you have any doubts, complete your work early so that someone can help you.