**CS2420 Program 1 Linked List Review**

This assignment focuses on Parking Lot puzzles such as the following:



In Parking Lot puzzles such as this one, the red car is stuck in a parking lot and is trying to escape.  Cars can be moved up and down or left and right (depending on their orientation).  They are allowed to move more than one square in a single move, but cannot move over or through other cars.  The goal is to clear a path so that the red car can escape past the yellow arrow.  You are encouraged to try it out yourself by downloading the game from the app store (Unblock My Car – Park Move Out).

**The code we are providing**

We are providing code for handling input/output and representing states. Feel free to change the code as you see fit.

The class Puzzle represents an instance of a Parking Lot puzzle.  This class includes methods for accessing information about the puzzle that is being represented, as well as for reading in a list of puzzles from a data file.

Note that every car is constrained to only move horizontally or vertically.  Therefore, each car has one dimension along which it is fixed, and another dimension along which it can be moved. The fixed dimension is stored in Puzzle as part of the puzzle, while the variable dimension is stored as part of the current state (class Node).

The car we are trying to move to the exit is always assigned index 0.

The class Node represents a state or configuration of the puzzle. Methods are provided for constructing a state, for accessing information about a state, for printing a state, and for expanding a state (i.e., obtaining a list of all states immediately reachable from it).  Also provided is the equals method for seeing if two Nodes are identical. It overrides the same method provided by the Object class.  The function of this method can be adjusted to count two states as “equal” if they have the same car position (regardless of the way the state was achieved).

Finally, the class TheMain consists of a simple main for running the tests.

Locations on the grid of a ParkingLot puzzle are identified by their (*x*, *y*) coordinates, where the upper left corner is square (0,0).  For instance, in the puzzle above, the red car occupies squares (1,2) and (2,2).  The goal is to move the red car so that it occupies squares (5,2) and (6,2).

Puzzles can be read from a file into memory using the static method  Puzzle.readPuzzlesFromFile.  Such puzzles should be encoded as in the following example representing the puzzle above:

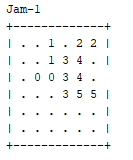
example  
6  
1 2 h 2  
2 0 v 2  
4 0 h 2  
3 1 v 3  
4 1 v 2  
4 3 h 2  
.

The first line assigns a name to the puzzle, in this case "example".  The next line, "6", gives the size of the grid, i.e., this puzzle is defined on a 6x6 grid.  The next line, "1 2 h 2", gives a description of the red car.  Note that the goal car must always be given first.  The first two numbers (1,2) give the (*x*, *y*) coordinates of the upper left corner of the car.  The "h" indicates that the car is horizontally oriented ("v" would have indicated vertical orientation).  The last number "2" indicates that the car has size (i.e., length) 2.  The next line, "2 0 v 2" describes the pink car, and so on.  The last line must consist of a single period, indicating the end of the description of the puzzle.  Multiple puzzles are described consecutively in a single file.

Internally, within the Puzzle class, each car is represented by its orientation (vertical or horizontal), its size and its fixed position.  In addition, the variable position of its upper left corner is stored in the Node class.  For instance, the red car would have a fixed position of 2 (can only move horizontally at y location 2) , and an initial variable position of 1 (x position of 1).  During the course of the search, this variable position might vary between 0 and 5 (inclusive), with 5 indicating that the goal has been attained.  Similarly, the pink car would have a fixed position of 2 and a variable position initially of 0, but varying between 0 and 4.

For this assignment, you can assume that the puzzle is a 6x6 grid, that the goal car always has size 2 and is horizontally oriented, and that all cars have size either 2 or 3.  However, you are free to experiment with puzzles not satisfying these constraints.

Puzzles can be printed in a simple form using the Node.toString()  method.  For instance, the puzzle above would be printed as follows:



We are providing a file called jamsAll.txt of puzzles.  However, you will surely want to test and debug your code on smaller puzzles of your own design. Some of these puzzle are too difficult for our crude methods, but we will return to this problem when we have more skills.

Note that in general we will *not* provide full-scale "reference solutions" for the programming assignments.  Being able to thoroughly test and then debug computer programs is an important and often challenging task requiring skill that is well worth developing.  A good way to test your program is to run it on a variety of small problem instances where you know what the correct behavior should be.

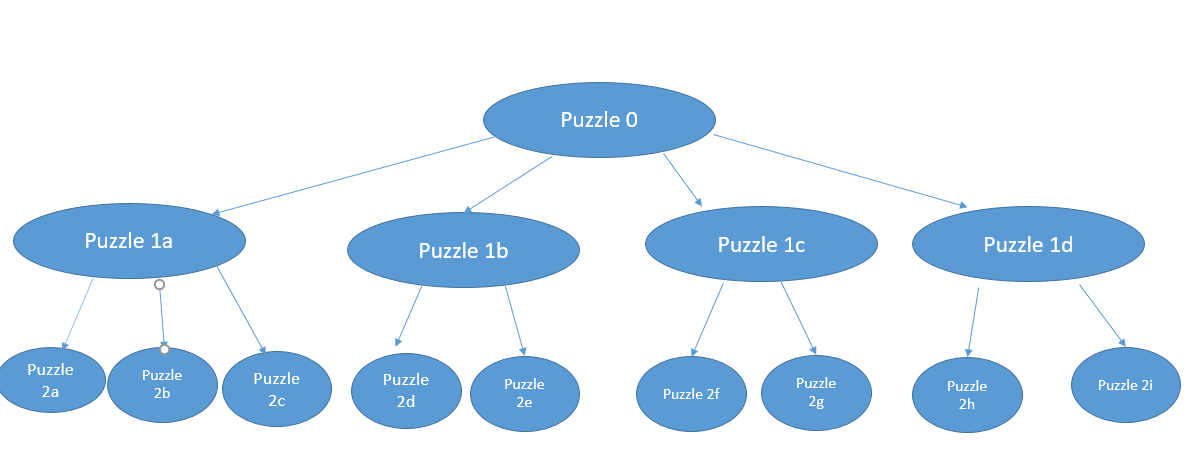
**Solving the problem**

**One way:** One way to solve the problem would be to identify a “best move” and make it. A series of “best moves” would hopefully result in a near-optimal solution. This technique is called a “greedy solution” as you take the move that seems best at the time. This is actually the way humans use to solve the problem. The tricky part is identifying a best move. This method does not guarantee that a solution will be the best possible solution (unless you are really good at predicting the best move).

**Brute force:** We are going to be more methodical.Instead of picking a particular move, we say, “I have ten (or whatever) possible moves. I don’t know which one to try. I want to try ALL of them. Let me try each of them and then (later) pick the one that was the best.” We try all choices from each puzzle state we reach. We use a queue to remember all of the partial puzzles we are considering. If we consider all “one move” states before all “two move” states (and so on), when we find a solution, it will be the shortest possible solution.

A logical tree helps us consider the possibilities. This is a LOGICAL tree, not a coded tree. Starting at “Puzzle 0” you consider each possible moves. [If there were only four choices at a time, we would say this is a tree with a four way branching factor.] We can keep expanding nodes until a “leaf” is the desired (goal) state. That represents an exhaustive search. However, we which order we expand the nodes. Are we going to keep going deeper in the tree [a depth first search]? This has the advantage of being easy to do (as recursion takes care of the returning to a previous node), but has some disadvantages.

We are going to approach it slightly differently. Instead of doing one move after another – we are going to consider all puzzles reachable with one move, then all puzzles reachable with two moves, etc. So in the picture below, we consider Puzzle 0, then Puzzle 1a, Puzzle 1b, Puzzle 1c, Puzzle 1d, and then all the puzzles at the next level. This is like traversing the logical game tree “by level”. This is a called a breadth-first traversal. We examine all one move states, then all two move states, etc.



The bad news is recursion can’t help us with this kind of traversal.You will use a **queue** to store all the possibilities you want to explore. **In order to refresh reference skills, you must use a linked list representation of a queue. This needs to be code that you have written.**

Because we try lots of possibilities, we need keep all the partial solutions we are working on. We define a *Node* to be the current state of the puzzle and the history of how we got to the puzzle. First, insert the initial state into a queue. Then, delete the state from the queue, and insert onto the queue all neighboring states (those that can be reached in one move) of the removed state. Repeat this procedure until you reach the goal state. Because the number of nodes at each level is growing exponentially, it will be more efficient to check if a state is a goal state before putting it on the queue (rather than when the state is removed.

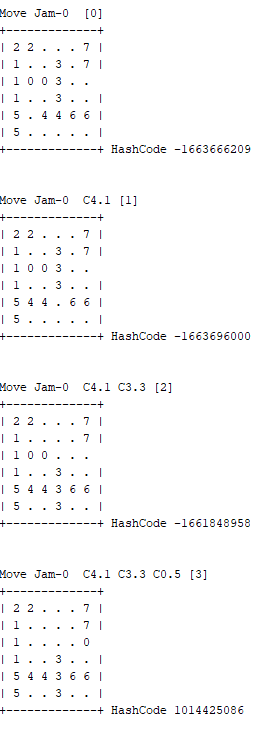
**Optional:** the brute force technique generates thousands of potential boards. You can see why. If the solution takes 10 moves, and each puzzle state has six possible next moves (for example), you would need to try 610 moves. One thing that really helps is to eliminate putting duplicate states on the queue. If you understand hash tables from CS1, you may want to use a hashtable to eliminate duplicate states. Java has a built-in hash table.

Feel free to experiment with techniques to improve the search.

This code creates a “hashcode” which is a number associated with the state. If two hashCodes are different, we know that the states are different. This is helpful in efficiently determining if you are reconsidering a state that has already been considered.

**Output**

**Print out your solution from the initial grid arrangement, to the final arrangement identifying each move. State the depth of the solution and the total number of nodes that were put on the queue. Here is the solution for the first puzzle of the set.**



If you attempt to solve more involved problems, I found it helpful to periodically print the depth of the search and the number of nodes I had created, so that I could verify progress was being made.

**Acknowledgments**

Thanks to Michael Littman and Rob Schapire from whom this assignment was liberally borrowed.