Computer Science 245
Spring 2008

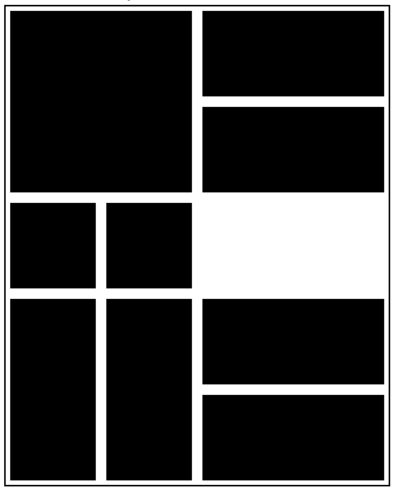
Programming Assignment 4
Sliding Tile Puzzle

Due Wednesday, May 7 2008

This assignment is adapted from Mike Clancy's Sliding Blocks assignment
`http://nifty.stanford.edu/2007/clancy-slidingblocks/`

1 Sliding Tile Puzzles

A "Sliding Tile" puzzle conists of a number of rectangular blocks in a tray. The objective is to slide the pieces without lifting any of them to achieve a certain configuation. One of the classic sliding tile puzzles is the Pennant Puzzle by L.W. Hardy:



The object is to move the 2x2 piece to the lower left corner of the puzzle, which can be accomplised in 59 moves.

2 Assignment

For your next assignment, you will write a program named Solver.java that produces a solution to a sliding-block puzzle (if a solution exists). Your program need not produce the solution with the fewest number of moves, but it must take as little execution time as possible.

2.1 Program Input

Your program will take two command-line parameters:

- A file that represents the initial tray configuration. The first line of this file is the length and width of the tray. Each subsequent line of the file will contain four numbers describing a block in the tray: The length of the piece, the width of the piece, the row of the upper-left corner of the piece (starting from 0), and the column of the upper-left corner of the piece (starting from 0). Thus, the tray for the Pennant puzzle above would be:

```
5 4
2 2 0 0
```

```
1 2 0 2
1 2 1 2
1 1 2 0
1 1 2 1
2 1 3 0
2 1 3 1
1 2 3 2
1 2 4 2
```

Blocks may appear in any order in the initial tray configuration file. You can assume that the tray will have a width no larger than 255, and a length no larger than 255.

- A file that specifies a goal configuration. Each line of this file contains four numbers (length, width, row, column) which describe the final location of a block in the goal. The goal file will not necessarily specify the desired position for all of the blocks in the puzzle. The goal file for the Pennant puzzle would be:

  ```
  2 2 3 0
  ```

  If there is more than one 2x2 block in the puzzle, the goal is to have *any* of the 2x2 blocks at the (row,column) position (3, 0)

  The goal configuration file can specify any number of blocks (up to the total number of blocks in the puzzle.) For example, if the goal was to move the two 1x1 blocks over two spaces to the right, then the goal configuration file would be:

  ```
  1 1 2 2
  1 1 2 3
  ```

  Blocks may appear in any order in the goal configuration file.

2.2 Program Output

A solution to the puzzle is a sequence of block moves. Each block can be moved either horizontally or vertically into an adjacent space. Blocks cannot be rotated. Your program should produce a list of block moves to standard out that lead to a solution. Each block move in your output should consist of the row and column of the upper-left corner of the block to move, followed by the row and column of the upper-left corner's updated position. For example, if the initial configuration was the pennant puzzle:

Initial Configuration file:

```
5 4
2 2 0 0
1 2 0 2
1 2 1 2
1 1 2 0
1 1 2 1
2 1 3 0
2 1 3 1
1 2 3 2
1 2 4 2
```

and the goal was to move the two 1x1 blocks over to the right:

Goal Configuration file:

```
1 1 2 2
1 1 2 3
```

Then a valid output of your program would be:

```
2 1 2 2
2 2 2 3
2 0 2 1
2 1 2 2
```

Not all problems have solutions. If the goal cannot be reached from the initial configuration, your program should output the string

```
No solution
```

3 Time / Space Tradeoffs

Your program will search the graph of possible move sequences to find a solution to the puzzle. You will need to make several decisions in the design of your program:

- The program will generate moves possible from a given configuration. This will involve examination either of the blocks in the tray or of the empty space in the tray. Should the tray be stored as a list of blocks/empty spaces to optimize move generation, or should the locations in the tray be represented explicitly? If the former, should blocks/spaces in the list be sorted?
- Prior to each move, the program must check whether the desired configuration has been achieved. What tray representation optimizes this operation? If this representation is incompatible with implementations that optimize move generation, how should the conflict be resolved?
- Once it has a collection of possible next moves, the program will choose one to examine next. Should the tree of possible move sequences be processed depth first, breadth first, or some other way?
- Should block moves of more than one space be considered? Why or why not?
- The program needs to make and unmake moves. Again, a representation that optimizes these operations may not be so good for others. Determine how to evaluate tradeoffs among representations.
- The program must detect configurations that have previously been seen in order to avoid infinite cycling. Hashing is a good technique to apply here. What's a good hash function for configurations? The default limits for Java memory allocation may limit the maximum number of configurations that the table can contain. How can this constraint be accommodated, and what effect does it have on other operations?

4 Writeup

In addition to your code, you will need to submit a writeup of your project that answers all of the following questions:

- A description of the overall organization of your programalgorithms and data structuresthat lists operations on blocks, trays, and the collection of trays seen earlier in the solution search. Diagrams will be useful here to show the correspondence between an abstract tray and your tray implementation. This description should contain enough detail for one of your classmates to understand clearly how the corresponding code would work.
- A description of any other files you're submitting, other than Solver.java
- An explanation of how you addressed efficiency concerns in your program:

- What alternatives did you consider?
- Why did you accept or reject them? (In particular, you should explain each choice of an array or vector over a linked list and vice-versa, and each choice of a sorted array or vector over an unsorted array vector and vice-versa.)
- How much memory do your data structures consume when processing typical examples? The jvm parameter `-verbose:gc` will help you here
- How fast do your algorithms work on those examples? The unix time command will help you here.
- In retrospect, what bad choices did you make and how would you have made them differently?
- If you were to make one more improvement to speed up the program, what would it be, and what is your evidence for expecting a significant speedup?

  Again, your descriptions of data structures and algorithms, both those that you have implemented and those you have rejected, should contain enough detail for one of your classmates to understand how the corresponding code would work.

5 Grading

Your project will be graded on a scale of 100, with points allocated as follows:

- 30 points for the writeup
- 45 points for correctly working on the easy problems, found on the course website. There are 28 easy problems on the course website. If your program correctly solves at least 25 of them within two minutes per problem, you will recieve the full 45 points. For each easy problem less than 25 that you fail to solve, you will be deducted 2 points, down to a minimum of 0. So, if you complete 25-28 problems, you will receive 45 points. If you complete 24 problems, you will receive 43 points, if you complete 23 problems you will receive 41 points, and so on.
- 25 points for correctly working on the hard problems. There are 13 hard problems on the website – you will receive 3 points per hard problem solved, up to a maximum of 25. As with the easy problems, each puzzle must be solved within 2 minutes to count.

6 Provided Files

In addition to the easy and hard problems, the website includes the file Test.jar, which can be used to test your solutions. The main executable in Test.jar takes two command line parameters, the initial configuration of the problem and the goal configuration. The list of moves is read from standard input. Thus, to test your Solver, you would type:

```
java Solver initialConfig goalConfig | java -jar Test.jar initialConfig goalConfig
```

I have also included a GUI for the tester, so you can see how your solution is working. You can use the gui with the command:

```
java Solver initialConfig goalConfig | java -jar Test.jar -gui initialConfig goalConfig
```

7 Hints

- Unlike the previous 3 assignments, you are given lots of freedom about how to complete the project. While this makes the problem more fun, it also will require a bit more time.
- You will need to do a bit of tuning to solve the hard problems. For instance, you will need to use a hash table to store the previously seen configurations – you will likely not want to use the hash table from the previous assignment without modification, since you will not need to store key/value pairs, you will instead just need to know if a key is in the table or not.

- You may also need to run some experiments to see when you run out of memory, to get the hard problems to work.
- Spend some time thinking about how you should represent the board – both the working board that you are using to make moves, and the representation you are storing in the hash table to check for duplicates. You may want to use different representations for different tasks.
- Once your program works on the easy problems, you may still have a substatial amount of work ahead of you to work on the hard problems. To get full credit for this assignemnt, you will need to spend time tuning and optimizing once it works for the easy problems.