Project: BLOCKS

Project 3
Computer Science 61BL, Summer 2009
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Colleen Lewis, Kaushik Iyer, David Zeng, Jonathan Kotker and George Wang

Contents

1 Deadlines

This project is time-consuming, please start early. You are required to work with 1 or 2 partners on this project. You cannot work alone on this project. Your final solution to this project is due **Tuesday, August 11** at **10PM**. Note that the final will take place on **August 13** from **5-8PM**, so finishing early will give you more time to prepare for the final.

There will be one checkoff, discussed in section for this project, that is due at the beginning of lab on Monday, 3rd August. The checkoff is described in detail in section 7.

Submit one solution per partnership and include your names *and* instructional logins in all files that you submit. Your solution directory, should include **all `.java` files** that relate to your solution and a file called `README` whose contents we discuss later in this specification at 6. Detailed design, coding and testing of this project should be your own work or that of your partners.

2 Overview

The past few weeks have introduced you to the foundations of data structures through data structure choice, design and time-space efficiency. This project aims to allow you to demonstrate your understanding of these areas while writing a solver for Klotski puzzles. Enter Project 3! `BLOCKS`: **B**rilliant **L**ogical **O**perations on **C**ode for **K**lotski **S**olvers.

Those of you who spend much time in toy stores may be familiar with "sliding-block" puzzles, also known as Klotski puzzles. They consist of a number of rectangular blocks in a tray; the problem is to slide the pieces without lifting any out of the tray, and achieve a certain goal configuration. An example[1] is shown in Figure 1.

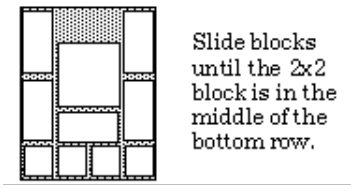

Figure 1: Example Klotski puzzle

"Virtual" versions of these puzzles are available on the Internet, for example, http://www.puzzleworld.org/SlidingBlockPuzzles/pennant.htm.

3 Your Task

3.1 Design and Overview

Write a program named `Solver.java` that produces a solution to a sliding-block puzzle (if a solution exists) in as little execution time as possible. Your program need not produce the shortest possible sequence of moves. Input to your program will come from the command line and from files named there. You should not only use the `Solver` class for all your code. Create and design classes as you need. We discuss how to test your code in section 8.

3.2 Arguments

- An optional first argument will be a string whose first two characters are "`-o`" and whose remaining characters specify information about what debugging output the program should produce. (You may choose the format of this information.) The string "`-ooptions`" should cause the program to print

[1] from Winning Ways, E.R. Berlekamp et al., Academic Press, 1982

all the debugging options and the effect of each option. If the "−o" argument is not provided, your program should produce no output beyond that required to display a solution to the puzzle.

- The next argument will name a file that specifies an initial tray configuration. Line 1 of this file will contain two positive integers, the length (number of rows) and width (number of columns) of the tray. Each subsequent line of this file will contain four nonnegative integers describing a block in the tray: the length and width of the block (both greater than 0), and the row and column of the upper left corner of the block. (The upper left corner of the tray is row 0, column 0.) Blocks are indistinguishable except for their size, and may appear in any order in this file. Thus the tray depicted in Figure 1 might be represented in the file as follows:

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

Figure 2: Sample input file

- The last argument will be the name of a file that specifies a desired final or goal configuration. This file is similar in format to the initial configuration file. Each line of this file contains four nonnegative integers: the length and width of the block (both greater than 0), and the desired position of the upper left corner of the block. This file will not necessarily contain entries for all blocks in the tray. Blocks may appear in any order in this file. The goal configuration mentioned in Figure 1 is represented by the single line

```
2 2 3 1
```

Figure 3: goal file

If there were more than one 2-by-2 block in the initial configuration, the one-line goal would specify the position of any of the 2-by-2 blocks. Figure 4 shows a goal configuration in which three of the 1-by-1 blocks have a specified arrangement, along with the corresponding goal file. Again, if there are more than three 1-by-1 blocks in the initial configuration, it doesn't matter which three of them end up in the specified goal positions.

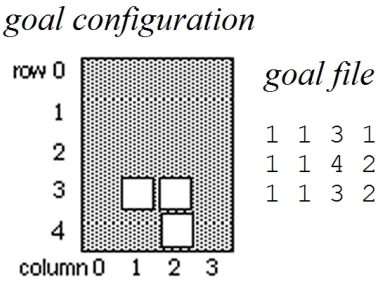

Figure 4: goal file

- You should check that command-line arguments are correctly specified, but you may assume that the initial and goal configuration files are free of errors. You may also assume that the length and width of a tray are no larger than 256.

3.3 How to Run Your Code

3.3.1 From the Command Line

Your program should be able to be run with the UNIX command

`java Solver [-oinfo] initialConfigFile goalConfigFile`

where the `-o` argument is optional, `info` is the debugging information you supply, and `initialConfigFile` and `goalConfigFile` name the files containing the initial block configuration and the goal configuration respectively.

3.3.2 From Eclipse

To enter command line arguments in `Eclipse` go to the `Run` menu, and then to the `Open Run Dialog` option. Enter arguments in the `Arguments` tab just as you would from the command line, then ensure that your `Working Directory` is the directory where your input and output files are stored (use the `File System` option to indicate this).

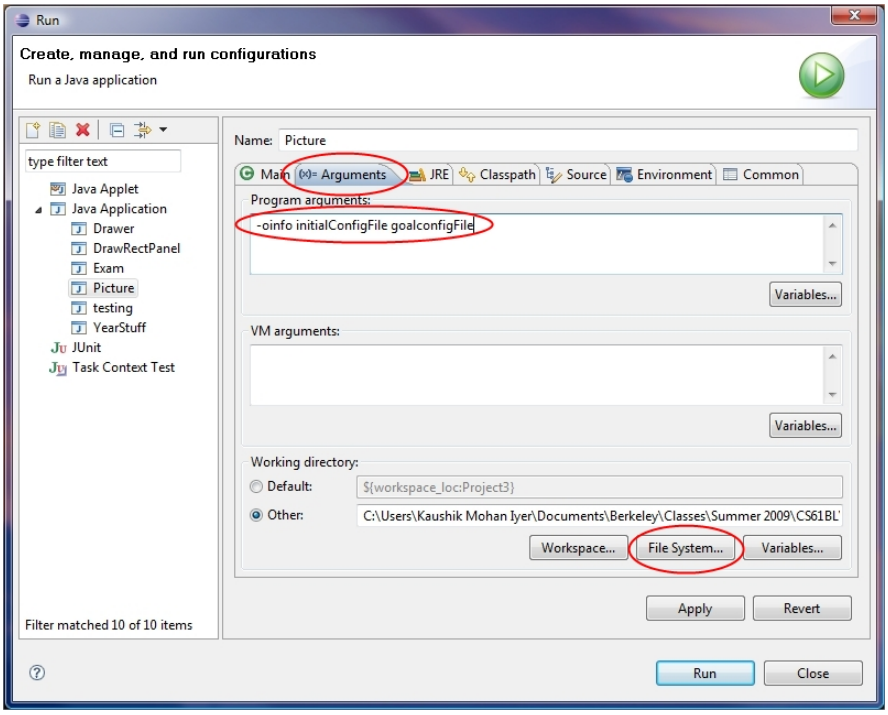

Figure 5: How to enter command line arguments in Eclipse

3.4 Output

A solution to the puzzle will represent a sequence of position changes of blocks that, when starting with the specified initial configuration, ends up with blocks in the positions specified in the goal. The only legal moves are those that slide a block horizontally or vertically—not both—into adjacent empty space. Blocks may only be moved an integer number of spaces, and either the row or the column will be the same in the start position as in the end position for each move.

Your program should produce a line of **printed** output for each block move that leads to a solution. Each such line will contain four integers: the starting row and column of the upper left corner of the moving block, followed by the upper left corner's updated coordinates. Example output appears below; the indicated moves, applied to the starting configuration of Figure 1, achieve the goal in Figure 2. (The annotations would not appear in the solution output).

1	1	0	1	move the 2×2 block up
3	1	2	1	move the 1×2 block up
4	1	3	1	move the 1×1 block up
4	2	3	2	move another 1×1 block up
4	0	4	2	move the leftmost 1×1 block two spaces over

Figure 6: Sample Output

If your program, run with debugging output disabled, finds a solution to the puzzle, it should exit normally after producing only output as just described, that is, the lines representing block moves that solve the puzzle. In particular, if the initial configuration satisfies the goal, your program should exit normally after producing no output.

If your program cannot find a solution to the puzzle, it should exit with the call

System.exit(1);

again after producing no output.

4 Time/space tradeoffs

Basically, your program will search the tree of possible move sequences to find a solution to the puzzle. It will implement several operations; questions you are to consider in your program design include those outlined below.

- 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?

Some of these questions can be answered with careful analysis. Others require empirical evidence. Incorporate in your program sufficient output information (governed by debugging options—see below) to provide this evidence. Discuss your answers to all these questions in your `README` file (see below).

5 Miscellaneous Requirements

5.1 Space Constraints

The amount of space your program needs is not an important consideration, except that your program has to fit in the default allocation of memory provided on the workstations in 275 Soda. An experiment we recommend is to determine how many configurations you can add to a hash table before you run out of memory. (The blocks in the puzzle described in Figure 1 may be moved into 65880 different configurations).

5.2 Debugging Requirements

You should associate debugging output with program events appropriately, and choose an appropriate debugging level for each set of output. Your debugging output facility should allow the user to select both the classes for which output is produced and the level of detail of output. Any interesting event that happens in your program—e.g. making/unmaking a move, encountering a previously seen configuration, determining the set of possible moves, or comparing a configuration with the goal—should be displayed by debugging output at some level. You should also incorporate output that will help you make implementation decisions about time/memory tradeoffs. Describe your debugging output facility in your `README` file (see section 6).

5.3 Style Guide

- Your code should display good documentation and style.
  - Provide an overview comment with each class that describes the abstract object and any invariants on the abstract object state (e.g. 'A Counter represents a mutable, non-negative, non-decreasing integer counter').
  - Accompany each method with descriptions of its preconditions and effects or return value.
  - When throwing exceptions, supply informative messages.
  - Give your variables and methods informative names that conform to conventions used earlier this semester (class names capitalized, names of constants in all upper case, and names of data members uncapitalized).
  - Indent your code appropriately. Highlight your code and press `Ctrl` + `I` or `Ctrl` + `Shift` + `F`.
  - All `if`, `else`, `while`, `do`, and `for` statements should use braces, even if they are not syntactically required.
  - Numerical constants with special meaning should always be represented by all-caps `final static` constants.
  - All class, method, field, and variable names should be meaningful to a human reader.
  - Methods should not exceed about 50 lines. Any method that long can probably be broken up into logical pieces. The same is probably true for any method that needs more than 7 levels of indentation.
  - Avoid unnecessary duplicated code; if you use the same (or very similar) fifteen lines of code in two different places, those lines should probably be a separate method call.
  - Programs should be easy to read.
  - Keep lines of reasonable length (say, 72 characters, with a maximum of 80 characters) for readability.
- All data fields and methods within each class must have the proper `public/private/protected/package` qualifier. In particular, you are not allowed to make things public that would let a user corrupt your data structures (even if none of your own code would do this).
- You are to include a `isOK` method with each nontrivial class. When a class's debugging option is enabled, a call to `isOK` should accompany each change to objects in the class. The `isOK` method should throw an `IllegalStateException` with an informative message if it finds a problem.

- Organize your program into classes that allow easy substitution of efficient code for inefficient code or of one algorithm for another (e.g. depth-first move processing for breadth-first) in each area. Use straightforward algorithms where possible. Your methods should not be overly long, complex, or repetitive.

6 README AND DOCUMENTATION

A substantial part of the credit for this project comes from the README submitted in your proj3 directory. This file should include the information listed below, answering all the questions in each category.

- An explanation of how you split the work for this assignment between members of your partnership and why you split it this way. Also indicate the number of points that should be awarded to each partner if you were to distribute 10 extra bonus points for your project between members of your partnership. We hope the answer is that you did pair programming for every piece of the project. This project has an extremely large design component, so working together on one computer through pair programming is likely to lead to un-buggy code, be a good experience and prevent "my partner isn't pulling their weight" situations.
- A description of the overall organization of your submitted program—algorithms and data structures—that 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 another CS 61BL student to understand clearly how the corresponding code would work.
- A description of your `isOK` method and whether it was useful in revealing program bugs.
- A description of any other files you're submitting.
- An explanation of how you addressed efficiency concerns in your program:
  - What data structures choices did you consider for the tray? What operations did you optimize: fast generation of possible moves, fast comparison of the current confiugration with the goal, or making a move? How did the considerations conflict?
  - How did you choose a hash function for trays? How did your choice optimize the need for fast computation, minimal collisions and economical use of memory?
  - How did you choose between moving blocks one square at a time and making longer block moves?
  - 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 implemented and those you rejected, should contain enough detail for another CS 61B student to understand how the corresponding code would work. You should defend every instance where you perform an operation more slowly than the theoretical optimum.
- A description of your debugging output facility and how to enable it.
- An explanation of the process by which you constructed a working program:
  - What did you code and test first, and what did you postpone?
  - Why did you build the program in this sequence?
  - What test cases did you use for each of your classes, and how did you choose them?
- Describe the bugs you encountered and fixed, and indicate what if anything you should have done differently to construct your program. Also describe and explain any bugs that remain; a bug you explain will cost you fewer points than a bug you do not mention.

In general, comments in your code will describe information specific to the corresponding class, while the README file contains information that relates classes and describes and provides rationale for design and implementation decisions. However, your README file should be written to be read on its own without a copy of the program code at hand, so there may be some information duplicated in writeup and code.

We **encourage** you to build the README file as you design, code, and test rather than putting it off until the end.

7 Checkoff

Due: By the beginning of lab on **Monday, August 3rd.**
Please bring a write-up / schematic of our design as discussed below. This is an opportunity for you to get feedback on your design from your TA. If you wish to get feedback on your design before the checkoff, just speak to your TA.

1. A list of the classes that you are going to use in your method.
   (a) For each class, list the private / public methods that you plan on creating.
   (b) For each class, list the instance variables you will add and what information you will use them to store.
2. A description of the data structures that you're going to use to represent the problem, and why you're using them.
3. What algorithms you're thinking of using.

8 How to Test your Code

The `~cs61bl/labcode/proj3/testing` directory will contain three folders, `easy`, `medium` and `hard` which contain easy, medium and hard tests! To test your code with a test suite.

1. Make a new folder for your tests.
2. Copy your `Solver` class to that folder.
3. Copy `Checker.class` and `TrayVerifier.class` to that folder from `~cs61bl/labcode/proj3/testing`.
   (a) Copy the following files to that folder: `~cs61bl/labcode/proj3/testing/easy/run.easy`, `~cs61bl/labcode/proj3/testing/medium/run.medium` and `~cs61bl/labcode/proj3/testing/hard/run.hard` to the testing folder.
4. Change permissions to allow you to run each of the files. Do so by entering the following commands in `xterm`:

   `chmod u + x run.easy`

   `chmod u + x run.medium`

   `chmod u + x run.hard`

5. Then go to that folder, and type `run.easy` , `run.medium` and `run.hard` to run the corresponding tests and see if your program behaves as it should.
6. Not all the test cases can be solved. If a test can not be solved, then follow the instructions above in section 3.4.

You can also test your code with a particular test by copying `Checker.class` and `TrayVerifier.class` to the folder where you have your test case and running the following command in `xterm`:

`java Solver init goal | java Checker init goal`

Here `init` and `goal` contain the initial and goal configurations respectively.

9 How to finish this project and SURVIVE

1. Your final takes place 2 days after the project is due. Do **not** place this project above studying for the final!
2. We do **not** require that you pass **all** the tests. Your program points are capped as discussed below in section 10.
3. Expect to have to rearrange your code dramatically after you pass all the `easy` tests. The `hard` tests will require careful design and data structure choice.
4. Do **not** optimize your code at the start. Concentrate on getting code that works. Pass all the `easy` tests. Then optimize.

10 Grading and Submission Details

This project will earn up to 140 points, allocated 80 for the program, 20 for testing and 40 for the writeup. These points will be scaled to 9% of your total grade. Grading will proceed as follows.

1. Your writeup will be examined for information about your tray data structure, and your `isOK` methods for the tray and blocks will be evaluated.
2. Your program will be compiled using the command

   `javac -O Solver.java`

   "You don't have to implement the `-O` command; it's already built into the inline Java compiler."If it fails to compile, you get no more program points. The `-O` (minus-Oh) option turns on optimization, and should be used for production runs. For debugging, use Eclipse.
3. Your program will be run, using the command

   `java Solver initialConfigFile goalConfigFile`

   on the `easy` directory described above. You must correctly solve almost all of these puzzles, using under two minutes of execution time for each puzzle, to earn more than 40 program points. (You are of course not allowed to "hard-code" solutions to these puzzles into your program.)
4. Your program will then be run, again using the command

   `java Solver initialConfigFile goalConfigFile`

   on a selection of more difficult puzzles `medium`, using a lightly loaded EECS workstation configured the same as the Sun Ultra-20 workstations in 275 Soda. (The "clients" command will tell you which workstations these are.) You can gain a maximum of 20 points for this section.
5. Your program will then be run, again using the command

   `java Solver initialConfigFile goalConfigFile`

   on a selection of more difficult puzzles `difficult`, using a lightly loaded EECS workstation configured the same as the Sun Ultra-20 workstations in 275 Soda. You can gain a maximum of 20 points for this section. Note that we will not be supplying arguments to the Java interpreter that modify the default memory allocation or the default maximum size of the system stack.
6. Stylistic and organizational attributes of your program will then be evaluated to complete your program score. These include information supplied in comments and variable names, formatting and use of white space, organization, a correct `isOK` method for each class, and appropriateness of debugging output. We will assign a percentage score to these aspects of your code; 100% means no flaws, 90% means minor flaws, and so on. Your program score is the product of your correctness score (from steps 1 through 5) and the style percentage.

7. Your test cases for helper methods and overall testing approach will be graded for a maximum of 20 points.
8. Your writeup will be evaluated separately to produce the remaining points of your project score.

You will submit all your `.java` files and a `README` file by using the command `submit proj3`. Congratulations!

11 Frequently Asked Questions

Please see the stickied FAQ thread on the google group available at: http://groups.google.com/group/cs61b_su09/browse_thread/thread/85b2bdc78060dede.

12 Acknowledgments

This project is based upon code written by Michael Clancy.

13 Changelog

- v 1.0 released on Monday, July 27th.
- v 1.1 updated to make it clear that test cases can fail. Released on Saturday, August 1st.
- v 1.2 updated to add changing permissions for `run.easy` et al. Released on Saturday, August 1st.
- v 1.3 updated to clarify output, *print* output, not write output to a file.