

Bloxorz

Due: 11:59 PM 09/29/09

Introduction

In the recent years, as a part of their devious plan to subjugate the human race, the Cylon empire has devised a surefire strategy. To redirect the resources of Earth's greatest minds (you) they have unleashed a flood of psychologically addictive programs, known as Flash games.

Fortunately you have seen through this attack and wish to help your fellow scientists break free of the trap. With the knowledge gained from your AI course, you know that all of these puzzles can be solved using a simple bidirectional A^* search. To avoid exposure to the dangerous Flash environment, your TAs have provided a Java based simulation of one of the most evil and time wasteful games known, Bloxorz.

Now it is up to you to save us all.

- Roll Opening Credits -

The Game

The game this project is based on is available here: <http://www.miniclip.com/games/bloxorz/>.

The objective of the game is to drop the 1x2x1 block through the hole in the middle of the stage without falling off of the sides. Obstacles, such as bridges triggered by switches, may also lie between you and your goal. The block is represented by a single red square if it is in the vertical orientation, and by two red squares if it is in the horizontal orientation.

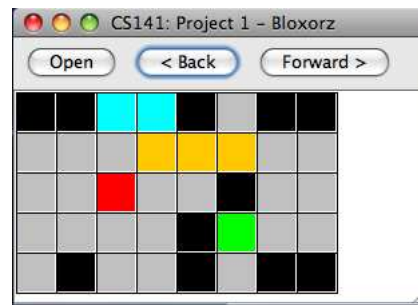
Here is a sample stage:

```

5
8
X X A2 A1 C2 0 X X
0 0 0 Y Y Y 0 0
0 0 S 0 0 C1 0 0
0 0 0 0 X G 0 0
0 X 0 0 X 0 X X

```

Our simulator renders it like so:



The Game Board

The board consists of a grid of $M \times N$ tiles. Each tile has a type:

Safe tiles are safe. The block can stand on these anytime. They are represented by **0**.

The **Goal** tile is the tile the block needs to be on in order to finish the game. The block's orientation must be vertical in order to fall into the goal. It is represented by **G**.

The block may never touch an **empty** tile, even if half of the block is on a safe tile. These tiles are represented by **X**.

Orange tiles are weaker than safe tiles. The block may only touch an orange tile if it is in the horizontal orientation. These tiles are represented by **Y**.

Soft locks activate and deactivate the bridges they are paired up with. Soft locks toggle the bridge they are associated with when the block touches them, regardless of the block's current orientation. After activating a bridge via the lock, touching the lock again will deactivate the bridge. These tiles are represented by **A** and a lock number (for lock-bridge pairs), for example, **A1**.

Hard locks toggle the bridges they are paired up with, but only trigger when the block touches them while in the vertical orientation. These tiles are represented by **B** and a lock number (for lock-bridge pairs), for example, **B1**.

Bridges are initially empty, but become safe when the locks they are paired with are activated. They are represented by **C** and a lock number, for example, **C1**.

The support code includes a parser for board files that will convert the file into an instance of the **Board** class. Your solver will take an instance of this class as input and output the optimal list of legal states that navigate the block into the goal.

You will also be expected to come up with some boards of your own. Boards are of the format:

Number of Rows

Number of Columns

Map of the board, where each row is on a new line and each tile on a row is separated by a space.

What to Do

There are three parts to this assignment.

1. Your first step is to come up with an admissible heuristic. As part of your writeup, you should explain what heuristic you used and prove that it is admissible.
2. Now that you have a heuristic, you'll need to implement bidirectional A^* search in the `Solver` class in the support code.
3. Once your solver works on our test boards, you should come up with three to five additional boards. These boards will not only help you test your solver, but they will also be used to test everyone else's. If you come up with a board that catches issues in many other students' solvers, you will be rewarded with up to 5 points extra credit.

Support Code

You can get the source code you need for this project by running `cs141-install bloxorz`, which will copy all of the files in `/course/cs141/src/bloxorz` to your home directory in `course/cs141/bloxorz`.

The support code is configured as an Eclipse workspace. Simply import the `bloxorz` directory as an existing Eclipse project to get started. If you do not wish to use Eclipse, you can run the support JAR directly so long as `Solver.class` is on your classpath.

Javadocs for all of the support code are available at <http://cs.brown.edu/courses/cs141/bloxorz/index.html>. Here's a summary of the important classes available to you.

Tile: Represents a tile on the game board. It contains the row, column, type of tile, and lock number.

Position: Represents the state of the game. It contains the row, column, and orientation of the block, and a set of activated locks. Our code assumes that if the block is in the horizontal orientation, then the row and column refer to the rightmost, bottommost occupied tile. It provides methods that return a string representation of the state and set the current instance to an input string representation of the state.

Board: Represents the game field. Stores a matrix of `Tiles`, representing the squares of the board, the starting `Position`, and the current `Position`. Provides methods that can check if a given `Position` is legal, can return all of the legal `Positions` available from a given `Position`, and can return all possible goal `Positions`. It also provides a method `moveBlock` that moves the block on the board and automatically toggles a lock if necessary.

Solver: This is the only class you need to modify.

`getSolution()` should take a `Board` as input and return the optimal solution. For our grading purposes, it should also set three public fields.

`solution` should contain the solution you returned when `getSolution` was called.

`num_explored` should contain the number of nodes you explored to find the solution.

`num_moves` should contain the number of moves your solution takes.

What to Hand In

We need:

1. A README containing a high level description of your solution, your heuristic and proof from part one, and any unresolved bugs.
2. All code and jars necessary to run your `Solver` class.
3. The three to five test boards you created for part three.

Our handin script turns in everything in the current directory and all subdirectories. To hand in this project, navigate into the directory containing the files and directories you wish to hand in and run `cs141-handin bloxorz`.