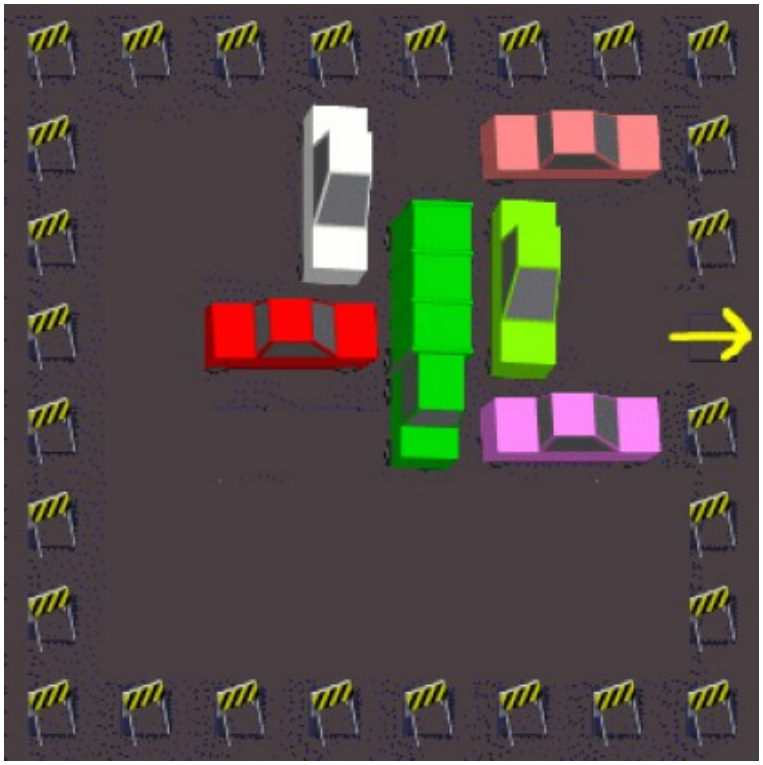
AI Project: Rush Hour[1]

In this programming assignment, you will use the A* algorithm to solve instances of the Rush Hour puzzle. This will involve implementing a graph-search version of A*, along with three heuristics, and testing your implementation on several Rush Hour puzzles. Code is being provided for handling input/output, for representing states, search nodes and puzzles, etc. Your job will be simply to fill in code for A* and the heuristics.

1 Rush Hour puzzles

This assignment focuses on Rush Hour puzzles such as the following:



In Rush Hour puzzles such as this one, the red car is stuck in traffic and is trying to escape. Cars can be moved up and down or left and right. 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, e.g. at one of the sites listed at `http://britton.disted.camosun.bc.ca/rushhour/links.html`.

[1]This text was taken verbatim from an artificial intelligence course website at Princeton University (no author given). The web page has since been removed. We merely changed the layout, but kept the original wording with minute modifications.

2 A* and heuristics for Rush Hour

One of the main parts of this project is implementing A* and three heuristics for solving Rush Hour puzzles. Your implementation of A* should use a graph-search, not a tree-search. In other words, it should check to be sure that it is not re-exploring parts of the search space that have already been explored. The goal of the search is to solve the puzzle in the fewest moves possible, so the cost of a search path should simply be the number of legal moves made.

Here are the three heuristics that you should implement and test A* on:

1. The trivial zero heuristic whose value is equal to zero in all states. Note that using A* with this heuristic is equivalent to breadth-first search. (Actually, an implementation of this trivial heuristic is provided.)
2. The blocking heuristic which is equal to zero at any goal state, and is equal to one plus the number of cars blocking the path to the exit in all other states. For instance, in the state above, there are two cars (namely, the two green ones) on the path between the red car and the exit. Therefore, in this state, the blocking heuristic would be equal to three.
3. A third, advanced heuristic of your own choosing and invention. Your heuristic should be consistent (as defined in class), and you should aim for a heuristic that will be at least as effective as the blocking heuristic. A trivial heuristic, comparable to the zero heuristic in triviality, would not be appropriate.

In your presentation, you should include a clear and precise demonstration of the advanced heuristic that you chose to implement. You also should include a brief but convincing argument of why both the blocking heuristic and your advanced heuristic are consistent, and therefore appropriate for use with A* graph search.

3 The code we are providing

We are providing code for handling input/output, representing states, search nodes, etc. Your job is simply to fill in A* and the two non-trivial heuristics.

The class `Puzzle` represents an instance of a Rush Hour 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. In addition, this class maintains a counter of the number of states that have been expanded for this puzzle. Methods for accessing, incrementing or resetting this counter are also provided.

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 here as part of the puzzle, while the variable dimension is stored as part of the state (class `State`).

The goal car (the one we are trying to move to the exit) is always assigned index 0.

The class `State` 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 are `hashCode` and `equals` methods which override the same methods provided by the `Object` class. These are provided so that `State` objects representing the identical configuration will be considered equal.

The class `Node` represents a search node of the puzzle, including an actual state, the depth of the node, and a link to the parent node that makes it possible to trace back a path to the root node. Methods are provided for accessing information about the search node, and for expanding it (i.e., obtaining a list of all nodes immediately reachable from it, corresponding to the immediately reachable states). Note the distinction between states and search nodes as discussed in class.

Your A* implementation will take as input a heuristic. Heuristics are classes that implement the `Heuristic` interface that we are providing. Such classes must include a method for computing the value of the heuristic at a given state. The `ZeroHeuristic` implementation has been provided.

We also are providing a class called `BranchingFactor` that includes a static method for computing an average branching factor.

Finally, the class `RushHour` consists of a simple `main` for running A* on all three heuristics and for all of the puzzles included in a file named in `argv[0]`. For instance, on unix systems, the command `java RushHour jams.txt` will run the program on all puzzles in the file `jams.txt`. The program prints out the solution for each puzzle and each heuristic, and prints a table summarizing the results. You may wish to modify this `main` to obtain printed results in some other form, or to test your program in some other way.

In addition to the code we are providing, you are welcome to use anything provided in the Standard Java Platform. You may find some of these classes, such as `TreeSet` or `PriorityQueue`, especially helpful.

Locations on the grid of a Rush Hour 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 6×6 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 may be 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 `State` class. For instance, the red car would have a fixed position of 2, and an initial variable 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 6×6 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 rudimentary ASCII form using the `State.print` method. For instance, the puzzle above would be printed as follows:

```
+-------------+
| . . ^ . < > |
| . . v ^ ^ . |
| . < > | v . |
| . . . v < > |
| . . . . . . |
| . . . . . . |
+-------------+
```

We are providing a file called `jams.txt` of the forty puzzles appearing on the `puzzles.com` site. However, you will surely want to test and debug your code on smaller puzzles of your own design.

Below is a summary of how my own implementation performed on the first five puzzles with the zero and blocking heuristics. (If my results seem fishy to you, please let me know right away.)

	ZeroHeuristic			BlockingHeuristic		
name	nodes	dpth	br.fac	nodes	dpth	br.fac
Jam-1	11587	8	3.066	8678	8	2.950
Jam-2	24178	8	3.380	6201	8	2.820
Jam-3	7814	14	1.789	5007	14	1.728
Jam-4	3491	9	2.326	1303	9	2.061
Jam-5	24040	9	2.928	8353	9	2.583

You may notice that the results you achieve for the number of nodes searched are different, and frequently better, than the sample ones above. The number of nodes searched can vary pretty widely, even among "correct" implementations, due to slight variations involving the handling of nodes on the OPEN list that have the same cost. The results above were achieved for an implementation in which such ties are broken in a FIFO order, i.e., the node that was placed on the OPEN list earliest is removed first, an approach that actually appears to be suboptimal. In any case, any way you choose to handle ties is acceptable and the potential variation in results will be taken into account when grading your assignments. (However, see the note below under "debugging tips".)

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 (especially for AI programs) 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. Also, sometimes, the answer can be computed in two different ways; for instance, on this problem, we know that the search depth of the optimal solution should be the same regardless of which heuristic is used.

4 The code that you need to write

Your job is to fill in the A* algorithm which belongs in the constructor of the `AStar` class. A template has been provided. This constructor must take as input both a puzzle and a heuristic. The solution, represented as an array of states, must be returned in the `path` field of the constructed object (or `path` should be set to null if no solution is found). You may also wish to add other fields returning other information.

You also need to provide implementations of the `BlockingHeuristic` and your own `Advanced-`

`Heuristic`, both being implementations of the `Heuristic` interface. Again, templates for these have been provided.

The final code that you turn in for A* and heuristics should not write anything to standard output or standard error. You should not modify any of the "finished" code that we are providing, with the exception of `RushHour.main` which you are welcome to modify.

All code and data files can be obtained from our e-learning platform.

5 Exploring the results

After you have your code working, you should run it on the forty puzzles that we are providing. Then, examine the results, and prepare some slides in your presentation for discussion of the results. Some questions to think about include the following: How did the different heuristics compare in performance on this problem? What about the results was surprising and what was just as you expected? Why is less searching (as measured by the number of search states generated) sometimes smaller for puzzles whose solution is actually deeper? The puzzles are listed roughly in order of supposed difficulty. Is it possible to discern what makes a problem hard or easy for humans in terms of its search characteristics? How does the search approach compare to how a human might solve these problems? What thoughts do you have about how this approach might be improved?

In preparing this discussion, you are welcome and encouraged to explore the results both anecdotally and numerically. For instance, you might want to figure out what made particular puzzles hard or easy. Or, you might want to look at the results ordered by the depth of the solution, or you might want to look at average performance within each of the categories listed on the `puzzles.com` site (beginner, intermediate, etc.). You might even want to make graphical plots of the results.

6 Debugging tips

If you are having trouble debugging your code, here are some things to think about:

When doing a graph-search, we need to check whether we have visited a node, so that we do not explore the same part of the search space twice. How do we do this? Should we check a node to see if it has already been visited when we pull it off the OPEN list, i.e., before we expand it? Or should we check a node before we add it to the OPEN list, in other words, right after it has been generated? For BFS, it doesn't matter. But for A*, it matters a lot. Make sure that your implementation of A* is doing the right thing.

Also, if you are using the `Comparator` interface, keep in mind that an object that implements

`Comparator` needs to impose a total order over the objects that it is likely to see. This means (for instance) that if `compare(o1, o2)` is positive (so that `o1` is considered greater than `o2`), then `compare(o2, o1)` must be negative, for all objects `o1` and `o2`. So, simply setting `compare(o1,o2) > 0` every time there is a tie will not work. (If you think about how priority queues and binary decision trees work, you can see why the ordering over objects has to be a total order, and why inconsistent responses from `compare()` will mess things up.) See the documentation provided with the Standard Java Platform for more details on the requirements for `Comparator` objects.

7 What to turn in

You should turn in the following using our e-learning plattform:

- The classes `AStar.java`, `BlockingHeuristic.java` and `AdvancedHeuristic.java` with the missing code filled in. Preferrable a complete working project in e.g. Eclipse or using ant scripts.
- Any other Java files that you wrote and used, or that are needed by your code.
- If appropriate, a `readme.txt` file explaining briefly how your code is organized, what data structures you are using, or anything else that will help us understand how your code works.

In addition, you should turn in a PDF with a presentation of your results:

- A clear and precise description of the advanced heuristic that you implemented.
- A convincing argument as to why your advanced heuristic and the blocking heuristic are both consistent.
- A brief discussion of your results. You will probably want to attach the results themselves (such as the summary table produced by running the RushHour `main`) to this discussion.

8 What you will be graded on

Correctness of the implementation and creativity in the design of the advanced heuristic will be major factors in your overall grade. Your presentation should be clear, concise, thoughtful and perceptive.

9 Acknowledgments

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