Designing heuristics

Artificial Intelligence

For your second homework, please experiment with heuristics to optimize the A* search for solutions to configurations of my son's game Rushhour. Rushhour is a children's puzzle based on sliding blocks back-and-forth on a grid. One of the blocks represents the family car, which is stuck in traffic: one or more blocks sit in between the family car and an exit the end of the family car's row of the grid. My son solves the puzzle by moving cars within their row or column so that the path from the family car to the exit becomes clear; you and I will solve the puzzle by tuning a heuristic search algorithm to find a series of moves which are as optimal as we can manage.

Stuck in traffic.

Driving free!

Only the family car is allowed to exit; the other vehicles must remain in the frame.

You will receive a substantial amount of working code for your experiments: the Java code you will produce for this homework will include little more than implementating your ideas for heuristic functions[1]. There are links to the code and its Javadoc on Canvas; some starting points for exploring the code are:

- Package rushhour.model is an implementation of the game mechanics — the Rushhour board, cars, possible moves, etc.
  - Class rushhour.model.Boards has some sample initial board configurations.
- Package search.graph is a generic implementation of several of the graph search algorithms we have discussed.
  - The heart of this implementation is class GraphSearcher, which implements the Graph-Search algorithm of Russell and Norvig in its general form. The specific behaviors of the frontier, the explored set, checking for goal nodes, etc. are provided through the generic type arguments, and through the behaviors passed as constructor arguments.
  - You will make particular (if indirect) use of class AStarSearcher, which specializes Graph-Searcher with the priority queue details of A* search.

[1]But there is also written work; keep reading.

- Package rushhour links the generic search implementations with the Rushhour model.
  - Class rushhour.BreadthFirstFinder solves Rushhour puzzles using breadth-first search. This class is your frenemy: On the one hand, this class gives you a working example of how we specialize the general search algorithms to a particular problem. But on the other hand this class is your rival, since the entire point of designing good heuristics with A* is to *beat blind search algorithms like BFS*.
  - For each heuristic function you implement, you will write one class extending rushhour.MovesFinder. Note that the constructor for MovesFinder takes only one argument — the heuristic function. Your subclasses should provide that constructor argument, and nothing more: do not otherwise override any methods inherited from MovesFinder.
  - Finally, you will extend class rushhour.AbstractSolution to wrap up all of your work on one bundle (see the *Deliverables* section below).

    The `run` method of this class is suitable for calling from the `main` method of your concrete `Solution` class, such as with

    ```
    new Solution().run();
    ```

    The given version will apply all of your solvers, plus BFS, to all of the sample boards, and print the results as a table. Of course you are free to override or edit this method locally to print additional calculations useful for your analysis of the effective branching factor.

Deliverables and submitting them

There are three deliverables for this homework:

1. **Contributions to rushhour.model.Boards.** As I write this document, there are a small number of example boards in that class, but we will all find it useful if there are more. Therefore this weekend or early next week I will scan some Rushhour cards, and will ask each of you to encode two for inclusion in that class, and then email me the revised Boards.java. Later in the week, I will update that class with all of your contributions.

   Follow the naming convention of the board which are already in that file, make sure it compiles and runs under BFS, and email me your updated Boards.java file. This portion of the homework is due by **Thursday, October 3**.[2]

2. **Heuristic function implementations.** Using the approaches we studied for designing heuristics, I expect you to try *at least **three** distinct ideas* for heuristics for Rushhour, implementing each one as a separate class extending MovesFinder. Your heuristics should be independent of board size: although the physical toy does use a 6×6 board, the BoardState class which represents one configuration of the board can have a different size set at its creation.

   In addition, you should provide one *additional* MovesFinder extension which combines your individual ideas using pointwise maximization.[3]

[2] If you signed out your board sheet the week after, then this portion is due Thursday, October 10.

[3] See the discussion in the text (Sec. 3.6.2 in Russell and Norvig 2010), for a way to react to the situation where we create different heuristics which all seem good, but without a "single 'clearly best' heuristic."

When you are debugging your code, it may be helpful to use the setDebug method to generate debugging information from running your code. However, the versions of your MovesFinder extensions which you submit should *not* set this flag, or otherwise print output messages.

Finally you should write one additional class Solution in the rushhour package extending rushhour.AbstractSolution. This class simply allows me to run all of your code at once. Your Solution class should look something like:

```
package rushhour;
public class Solution extends AbstractSolution {
  public Solution() {
    super(new MyFinder1(), new MyFinder2(),
          // ...
          new MyFinderNminusOne(), new MyFinderN(),
          new MyComboFinder());
  }

  // Use *exactly* this main method
  public static void main(String[] args) { new Solution().run(); }
}
```

where the MyFinderN's implement your individual heuristic ideas, and MyComboFinder performs the pointwise maximization across them.[4]

Submit your code to Canvas as a zip or tgz file which expands to a src directory containing your Java source at the top-level. I will expand your archive and then run commands like

```
  javac -cp codeIGaveYou.jar src/*.java
```

This portion of the homework is due on **Monday, October 21**.

3. **Written report.** Analyze each of your heuristics, discussing
   - How you derived them.
   - Their properties, especially admissibility, consistency, and complexity. Do not spend time on heuristics which are not admissibile and consistent, and do not duplicate solving the problem in the heurisitic.
   - Their performance, including consideration for each heuristic of
     - Its effective branching factor on each board relative to both the theoretical branching factor for that board, and the effective branching factor by the provided BFS implementation for that board.
     - How stable its effective branching factor is across different boards.

     Gather many numbers!

   Discuss the factors behind each heuristic's advantages, and draw conclusions as to which of your heuristics are better or worse. Submit this report to Canvas as a PDF as discussed in the syllabus.

   This portion of the homework is due on **Monday, October 21**.

[4]Note that except for Solution, I do not care what you actually name these classes — so long as I can compile them and run Solution.

Bug bounty

I wrote the code distributed with this assignment with haste. Although I have run it successfully, I have not thoroughly tested it, and you are as likely as not to find bugs in it. I will award a small amount of extra credit for the first accurate emailed report of each bug in the code. A report must contain a minimal example which triggers the bug; reports which also identify suspected details of the bug will be assessed as more valuable.