CS 4300: Artificial Intelligence

Assignment: Rubik’s Cube Search

This assignment requires writing the code necessary to use the existing cube model and search tool to search for solutions to solve scrambled cubes.

The AI search library framework must be used, and the required classes must be created according to the search library. Using the rectangle problem’s solution as a reference is encouraged.

Required Classes

- Create header and implementation files in `cs4300-code-ai-agents/prog/RubiksCube`.
- Create an `Action` class that inherits from `ai::Search::Action`. This class is usually simple. Instances need to be able to uniquely identify any of the legal actions.
- Create a `State` class that inherits from `ai::Search::State`. This class will probably have an instance of your cube class as a data member. There are a few required methods, for example `isEqual`.
- Create a `Problem` class that inherits from `ai::Search::Problem`. This class requires at least 4 methods, in addition to the constructor. Look at the `Actions`, `GoalTest`, `Result` and `StepCost` methods.
- Complete the code in `CubeSearch.cpp` to provide the `search_aux` function solution. It is assumed that you will copy the code from the rectangle problem solution as a start, then modify it to work with the cube problem.

Required Functionality

- The code must compile and run.
- The code must give optimal solutions to cubes that are 6 or fewer moves from being solved, in less than 5 minutes on a computer system similar to those available in the lab, assuming that the depth limit is configured such that `l = d`.
- The script `verifier/verify_many.bash` can be run from the verifier directory, if `RubiksCubeSolver` has been built. It will attempt to solve 10 cubes of each difficulty from 1 to 6 steps from the solved.

Output From `verify_many.bash`

On a system from the lab:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0m0.011s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0m0.023s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0m0.177s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0m1.096s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0m12.273s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2m1.432s</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

On a laptop system:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0m0.023s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0m0.054s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0m0.273s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0m1.320s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0m36.575s</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3m51.573s</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Each line in the output has the following information:

- The problem difficulty
- The real clock time to try solve all 10 problems
- The number of problems solved
- The number of problems not solved
- The average number of tree nodes generated
- The average of maximum number of tree nodes stored
- The average solution length

Notes on the above results:

- The laptop apparently takes a longer time to do approximately the same amount of work. No surprise, the CPU is faster on the lab machine.
All of the problems are solvable with the configuration given. Be patient. It took almost 5 minutes to solve the 10 level 6 problems.

Each level requires approximately 10 times as many nodes as the previous level. No surprise with a branching factor of 12.

For my search, I received difference results on different machines. You probably won’t. I actually shuffle the vector before returning it from `Problem::Actions()`. Minor point, not necessary for you.

The laptop averaged 2.8 moves to solve the level 3 problems. This is because the 3 random moves for one of the problems had 2 that counteracted each other, making it have a level 1 solution. Because of my shuffling of `Actions()` that run lucked out and stumbled onto the level 1 solution before finding a level 3 solution. \[(9 \cdot 3 + 1) / 10 \rightarrow 2.8\] average.

**Additional Information**

- Look at [README_CubeSolver.md](#) in the source directory for more suggestions on the implementation of the required code.
- [verifier/verify_many.bash](#) Runs depth limited searches for cubes from 1 to 6 steps from solved. Each search uses a depth limit appropriate for the optimal path length.
- Solve the Cube
- Speed Solving

**Passoff**

Submit your source code by committing and pushing the repository. Your code will be checked by an automated tester to verify that it solves all of the above cubes, and several others with similar difficulty.