Assignment: Rubik’s Cube Search II

This assignment requires writing additional code in the Rubik’s Cube search program. It assumes that the work for the previous assignment has been successfully completed.

The assignment also requires exploration of the Rubik’s Cube problem through use of the cube solving program built.

Requirements

- Allow the user to specify a set of goal states using the init goal command.
- Allow the user to specify the set of legal moves using the moves command.
- Enable A* and Greedy searches by providing a Problem::Heuristic.
- Research solution plans for assigned cube problems.

Goal States

The user interface allows the init goal command to configure a vector of cubes in AppData.goal_cubes. You can look in CubeSolver.cpp for data.goal_cubes for all of the places it is used in the user interface.

The intention is for the search_init() function to use this vector to set the goal for the Problem class. This can be handled by adding a data member in the Problem class and passing in the vector of goal cubes in the constructor for the class. Finally, Problem::GoalTest() should look for the current cube to == any one of the cubes in the vector.

The cubes are allowed to use the wildcard, *, when specifying a cube description. For example, we can specify that the goal is to place the white/green edge cubie like this:

```
init goal ***ww**** *g**g**** *****r***** *****b***** *****o***** *****y****
```

While the goal to solve the first step of the beginner’s method look’s like this:

```
init goal *w*ww+w* *g**g**** *r**r***** *b**b***** *o**o***** ****y****
```

If we want to have more than one goal cube, we just need to specify multiple cubes on the command line. This initialization will allow either the white/green or the white/red edge cubie placed to be a goal state.

```
init goal 
   ***ww**** *g**g**** *****r***** *****b***** *****o***** ****y**** 
      ***w**w* ***g**g**** *r**r***** *****b***** *****o***** ****y****
```

As a side note, if the last character of a line is \, then the line input is continued on the next line.

An additional command in the user interface allows the definition of short names for cube configurations. For example, this command allows us to use white_green_cubie to describe a cube with the white/green edge cubie placed correctly.

```
define cube white_green_cubie ***ww**** *g**g**** *****r***** *****b***** *****o***** ****y****
define cube white_red_cubie *****w**** *****g**** *r**r***** *****b***** *****o***** ****y****
```

When defining cube names, you can also specify multiple cube descriptions, and the resulting cube will be the logical and of the cubes. The cubes being merged must be the same everywhere that they are not wildcarded.

```
define cube two_edges white_green_cubie white_red_cubie
```

To see the defined cubes use show defined_cubes.

```
cube> config cube_display 2
cube> show defined_cubes
Defined Cubes:
two_edges ***ww**** *g**g**** *****r***** *****b***** *****o***** ****y****
white_green_cubie ***ww**** *g**g**** *****r***** *****b***** *****o***** ****y****
```
After cubes are defined, the name of the cube can be used, anywhere you could use the full cube description. For example:

```plaintext
init goal white_green_cubie
```

Using these configurations of goal states, will allow the user to search for partial completion solutions to the cube with flexibility.

**Moves and the `MoveSet`**

The `moves` command allows the user to change the list of legal moves to use in searching and shuffling. Here’s an example command to restrict the set of allowed moves:

```plaintext
moves B D F L R'
```

To see the current list of allowed moves use `show moves`.

```plaintext
cube> show moves
Moves:
 B : B
 D : D
 F : F
 L : L
 R' : R'
cube>
```

The allowed moves are stored in `AppData.move_set`, which is a `MoveSet` object. To connect this information to the search process, implement `Problem::setAllowedMoves()` as described in the README, and use the moves in the `Problem::Actions()` method for searching. Remember to call `setAllowedMoves()` when you create a `Problem` object.

There are some sequences of moves that are useful when solving the cube by hand. For example Lars Petrus has named the move `Allan` that rotates the positions of three edge cubies on the top face. The `define move` command can be used to give names to sequences of moves.

```plaintext
define move Allan F F U' L' R' F F L' R U' F F
```

When a move sequence is defined, the combined transformations of the moves are calculated once, and the newly defined move is applied using the same memory and CPU as a single simple move. The costs of the combined moves are summed, and assigned as the cost of the combination move.

To view the moves defined, use `show defined_moves`.

Defined moves can also be used in the `moves` command to create the set of allowed moves.

```plaintext
moves D L R' Allan
```

Using these configurations of moves and allowed moves, will allow the user to search for for solutions with flexibility. Either restricting the allowed moves for solving some stages of the cube, or adding desirable combination moves to search options.

**Problem::Heuristic()**

For the A* and greedy search frontiers to work, an estimate of the distance from a state to some goal state must be made. This is implemented in the `Problem::Heuristic()` method. Add the method declaration to your problem class, then implement it.

```plaintext
virtual double Heuristic(const ai::Search::State * const state_in) const;
```

A simple heuristic function for the Rubik’s cube would be to calculate the number of facelets that do not match the goal facelet in the same position, and divide by 20. The number 20 is because a single quarter turn will change the color of up to 20 facelets. For this assignment, this simple heuristic is sufficient. For future assignments, you will be asked to create better heuristics.

Thought: How should you handle wildcards when calculating the heuristic?
Research

Write a command script for that will attempt to solve some 2x2x2 section of a cube configuration. The plan can follow any of several plans. Here are some example plans:

1- Create a set of goal states, one for each of the eight possible 2x2x2 solutions. Leave the non-essential facelets as wildcards. Search using this set of goal states. The solution is a path to a 2x2x2 solved cube. It doesn’t matter which one.

2- Create a single goal state for one of the eight possible 2x2x2 solutions. Leave the non-essential facelets as wild cards. Search using this goal. The solution is a path to the 2x2x2 solve cube.

3- Create an intermediate goal for a solution towards a 2x2x2 solution. For example, instead of the goal being the whole white/red/green 2x2x2 solved, it might be having the white/green and white/red edges placed. Then create a goal for the full 2x2x2 solution. Perform two sequential searches, first for the intermediate goal, then for the full goal. Be sure to have \texttt{config apply solution 1}. Now, you complete two searches. The intention is that the sum of the work of searches will be less than the work for a single search to the full goal. You may even break it up further, into many intermediate steps. For this problem there are only 4 cubies to put into place, so at most there would be 3 intermediate and 1 final goal.

4- If you see a different approach consider using it instead.

There is a set of 100 scrambled cubes linked below. Use your command script to solve as many of them as possible. Recommendations:

\begin{itemize}
  \item \texttt{config solution display 1}, will give you one-line output of the search. Much easier to use if you are trying to process lots of data. Easier to process with automated tools, or to import into a spreadsheet.
  \item \texttt{generation limit le7}, don’t go much beyond 10 million nodes in the tree, unless you need to use your computer as a heater.
  \item \texttt{search graph astar} will use the most intelligent search. I would use either this or \texttt{config ids limit 6} with \texttt{search tree ids}.
  \item Creating a file of cube definitions is very useful. Use descriptive names for the defined cubes. You can \texttt{run cube definitions.cmd} or whatever you file’s name is to include it in your script. This makes the definitions reusable in future work.
  \item Consider using shell scripting or python to create a “master” command script that does

\begin{verbatim}
  echo init cube yyyywrbwbrowwryorwooyoygbbygbogbbooggbwrryrrwww run your_solver.cmd
\end{verbatim}

for each of the cubes in the scrambled cubes file. In short, automate anything you can. Use your time and effort in other ways than copy-paste.
  \item Produce a report, that includes your command script, and statistics on how many of the scrambled cubes the script was able to solve, and how many it was not able to solve. For the solved cubes, what was the average number of nodes generated? (If you take the intermediate goal approach, this would be the sum of all searches.) For the solved cubes, what was the average solution cost? (How many moves in the solution.)
\end{itemize}

Additional Information

\begin{itemize}
  \item \texttt{Solve the Cube}
  \item \texttt{Speed Solving}
  \item \texttt{scrambled_cubes.txt}
  \item \texttt{Deep Heuristic-learning in the Rubik's Cube Domain: an Experimental Evaluation}
\end{itemize}

Passoff

Submit your source code by committing and pushing the repository. Submit your report to Canvas. 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.