CS 380: Artificial Intelligence

Assignment A1: State and Action Representation

The Rotator Puzzle (20 pts)

A "Rotator Puzzle" is actually a class of puzzles involving what we'll call a *rotator*: a cylindrical puzzle with sliding tiles like those shown below. Each row of the puzzle can be rotated to spin the tiles around the central axis of the cylinder. There is one tile missing—an empty space—which allows tiles to slide up and down along the length of the cylinder. Solving the puzzle involves sliding and rotating the tiles to a state where each vertical column contains tiles of the same color (like some of the puzzles in the figure below).

See also: https://www.youtube.com/watch?v=wcKIX1CIDBw

For this assignment, we will write the code needed to represent a single state of the rotator puzzle, to compute possible next states after executing a single action, and to execute a simple walk through the state space.

Implementation Setup

This assignment, and all assignments for this course, will use **python3** running on **tux.cs.drexel.edu**. Please note that **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

Your code will need to accept up to two command-line arguments and use them properly for each particular command. As you'll see in the sections below, running the code will have the general format:

```
> python3 rotator.py <command> [<optional-argument>]
```

This scheme will allow you to test your code thoroughly, and also allow us to test your code using a variety of arguments. You can use **sys.argv[index]** to get the argument at a given index, where the command is at index 1 and the optional argument is at index 2.

State Representation

For this assignment, you first need to create a representation for the current problem state. The rotator puzzle can be characterized as a 2-dimensional grid of tiles where the rows wrap around the cylinder and the columns run down the length of the cylinder. For example, in the figure below, the table of numbers on the right shows a simplified

representation of the puzzle on the left. The middle (colored) part of the table represents the parts of the puzzle visible to us, but note that the tiles wrap around the cylinder, so there are tiles we cannot see in the picture. We assign a number to each color; for example, '4' is green, '5' is blue, '6' is red, and '8' is gray. Also note that, because the cylinder wraps around, the '1' tiles in the table are actually next to the '8' tiles. There is also an empty space in the table that represents the missing tile (not visible in the puzzle picture).

1 2 3 **4** **8** **6** 7 8
1 2 3 **4** **5** **6** 7 8
1 2 3 **4** **5** **6** 7

Putting this all together, let's assume we can represent the state of the puzzle with a string that looks like the following:

```
Default            "12345|1234 |12354"
```

The vertical line '|' represents the divider between the rows, with the top row first. The space character ' ' represents the empty tile space. We will consider the state above to be the *default state* (to be used in the sections below).

Write a class **State** that takes a string representation like the one above and stores the information relevant for that state. Your code will need to extract the information from the state string and then store this information in whatever variables you think are appropriate. Your code should work for a puzzle of any size with at least 2 rows and at least 2 columns. You can assume that the input string is well-formed—for instance, you can assume that the string has one character per tile, has the same number of characters per row, and has exactly one empty space.

As a first step, you should implement a **print** function that can print the current state in as an ASCII state string (like those above). It should be runnable from the command line with the "**print**" command as the first argument and the state as the second argument. If the state argument is not provided, please use the default state noted above. Here are a few examples of running from the command line:

```
> python3 rotator.py print
12345|1234 |12354
> python3 rotator.py print "123|123| 23"
123|123| 23
> python3 rotator.py print "fedcba|acdfeb|bcaed "
fedcba|acdfeb|bcaed
```

Identifying Solutions

Write a method that determines whether the given state is at the solution state. As mentioned, a solution state is any state where all the tiles in a given column (besides the empty space, of course) have the same color.

Please add a method **is_goal()** to your State class to check whether the state is a solution state. Then, add a "**goal**" command-line argument to your code command that prints "**True**" or "**False**" depending on whether the given state is at the solution state or not. For example:

```
> python3 rotator.py goal
False
> python3 rotator.py goal "1234|1234|123 "
True
> python3 rotator.py goal "fedcba|acdfeb|bcaed "
False
> python3 rotator.py goal "ac b|acdb|acdb"
True
```

Computing Possible Actions

Your next task is compute possible actions from a given state. The rotator puzzle has two categories of actions from any given state, and we can characterize actions with a string representation just as we did for states:

- **slide(x,y,x2,y2)** – Slide the tile at (x, y) to the empty space at (x2, y2), where *y* = 0 represents the top row of the rotator.
- **rotate(y,dx)** – Rotate a row left or right by 1 tile, where *dx* can be either 1 (where each tile rotates to the right) or –1 (where each tile rotates to the left).

Note that slides or rotations of more than 1 tile can be accomplished by multiple 1-tile moves, so we don't need to account for those separately. Also note that not all actions are legal from a given state (e.g., we can't slide a tile down into the empty space if the empty space is at the top).

Write a class **Action** that embodies this information, including a **__str__()** method to build a string representation like that above. You may use subclasses as well if desired.

Then, implement a method **actions()** in your State class that returns a list of possible legal actions from the current state. In addition, the list of actions returned by this method *must be sorted according to the action's string representation*.

Finally, add an "**actions**" command-line argument to your code, such that the program prints possible actions for either the given state (if the optional argument is provided) or for the default state (if the optional argument is not provided). Below is the desired output for the default state and a different sample state:

```
> python3 rotator.py actions
rotate(0,-1)
rotate(0,1)
rotate(1,-1)
rotate(1,1)
rotate(2,-1)
rotate(2,1)
slide(4,0,4,1)
slide(4,2,4,1)
> python3 rotator.py actions "fedcba|acdfeb|bcaed "
rotate(0,-1)
rotate(0,1)
rotate(1,-1)
rotate(1,1)
```

```
rotate(2,-1)
rotate(2,1)
slide(5,1,5,2)
```

Executing Actions

Your final task is to execute actions on a state and then implement a "walk" through states.

First, implement a method for your State class, **execute(action)**, that executes the action in the current state (thus changing the state to a new state). It's also a good idea to include three other State methods for convenience: **__str__()** to build a string representation of a state (useful for printing and debugging), **__eq__()** to check whether the state is equal to another state (useful for checking for redundant states), and **clone()** which returns a cloned instance of the current state (useful for generating next states, since you can clone the current state and execute an action without affecting the original state).

Then, implement a walk through states as follows. Add a command-line command "**walk***<i>*" which starts with the current state (i.e., the given state or the default state), prints the state, gets its possible actions, executes actions[*i*] (so *i*=0 means executing the first action, 1 the second, etc.), and iterates. The iteration should continue until it arrives at a state that has already been seen, and then stops. Here are four examples of this walk:

```
> python3 rotator.py walk1
12345|1234 |12354
51234|1234 |12354
45123|1234 |12354
34512|1234 |12354
23451|1234 |12354
> python3 rotator.py walk2 "acdb|ac b|acdb"
acdb|ac b|acdb
acdb|c ba|acdb
acdb| bac|acdb
acdb|bac |acdb
> python3 rotator.py walk1 "21|1 "
21|1
12|1
> python3 rotator.py walk6 "fedcba|acdfeb|bcaed "
fedcba|acdfeb|bcaed
fedcba|acdfe |bcaedb
fedcb |acdfea|bcaedb
```

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run on tux will receive a grade of zero.

For this assignment, you must submit only one file:

- **rotator.py** – your Python code for this assignment

The code file must be submitted electronically using Blackboard—please do not email your assignment to the TA or instructor. If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment it due. If you have any doubts, complete your work early so that someone can help you.

Academic Honesty

Please remember that all material submitted for this assignment (and all assignments for this course) must be created by you, on your own, without help from anyone except the course TA(s) or instructor(s). Any material taken from outside sources must be appropriately cited.