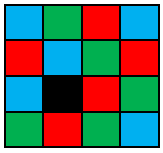
CS 380: Artificial Intelligence

Assignment 2: Uninformed & Informed Search

The RGB Puzzle (20 pts)

The RGB (Red, Green, Blue) Puzzle is a sliding puzzle where 15 colored tiles are placed on a 4x4 grid, like the following:



In the puzzle, there are 5 tiles for each of the three colors (red, green, blue), plus one empty square (shown in black). Like a typical sliding puzzle, a person can slide a tile vertically or horizontally into the empty square, thus moving the tiles around. (The tiles cannot be picked up or moved in any other way.) The goal of the RGB puzzle is to move the tiles to arrive at a state where no two adjacent tiles have the same color; the state shown above is one of many possible solution states.

In the previous assignment, we took a sample problem and implemented a state representation that provides the building blocks for search. In this assignment, we take the next step and implement search algorithms to find a solution.

Implementation Setup

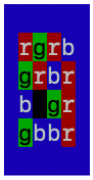
As part of the assignment, you have been given a file **rgb.py** that provides an implementation of the state representation for the RGB puzzle, including computation of possible legal actions. As in Assignment 1, we use a simple string as a base representation for a state:

```
Default     "rgrb|grbr|b gr|gbbr"
```

This string embeds 4 rows of the game state, where each row is a 4-character sequence with the letters **'r'** (red), **'g'** (green), **'b'** (blue), or **' '** (a space for the empty square). The four rows are separated by the vertical line '|'.

The code in **rgb.py** parses this string for you and creates an internal representation with a 2-dimensional array, with helper functions to get and put characters in the cell locations. It also includes methods **is_goal()** to determine whether the current state is a solution state, **actions()** to return the possible legal actions from the current state, and **execute(action)** to execute an action within the current state. In addition, the code allows for command-line arguments just as you implemented in Assignment 1, with some extra utility functions to print color states to the terminal. You can try these commands as examples (all of which use the default state string above):

```
> python3 rgb.py print
```



```
> python3 rgb.py goal
False
> python3 rgb.py actions
slide(0,2,1,2)
slide(1,1,1,2)
slide(1,3,1,2)
slide(2,2,1,2)
```

Implementing an Agent

For this assignment, you will be coding an agent in a file **agent.py** that interacts with and solves a given RGB puzzle using various algorithms. Like the first assignment, we will use **python3** running on **tux.cs.drexel.edu**, and as before, **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 questions about what is allowable, please email the instructor).

Node and Agent Classes

As discussed in lecture, when searching through a state space, we typically use a node structure where each node includes both a state and some extra information used by the search—namely, a pointer to its parent node (to maintain path information for how we arrived at this state), and sometimes a state value (for informed search algorithms like A* where we assign a numeric value to a state). Your first task is to implement a class **Node** that captures this information. This code should be put in **agent.py**.

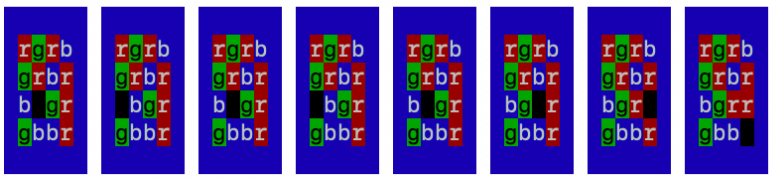
Next, also in **agent.py**, implement a new class **Agent** which will serve as the foundation for building the search algorithms below. Specifically, each of the search algorithms will be a method within the Agent class. Note that this separation between Agent and State classes allows us to use the same agent for multiple problem domains, and also allows us to use different agents on the same problem domain; we will not make much use of this flexibility in this assignment, but future assignments will exploit this architecture to a fuller extent.

Random Walk

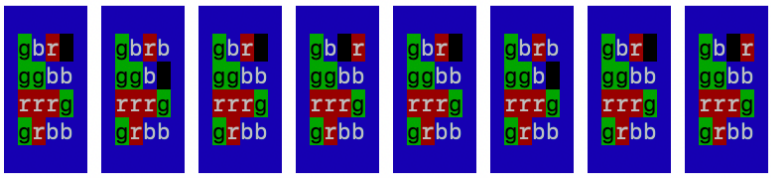
Write a method for your Agent class, **random_walk(state, n)**, that does a random walk through the state space. Specifically, given a positive integer *N*, the random walk should generate all the possible next states from the current state, select one at random, then repeat until *N* states have been visited. Your code should build nodes (using your Node class) and, when the walk is completed, use the final node to generate and return the list of *N* states visited on the random walk.

Create a new file **main.py** and add a "**random**" command-line command to print the resulting sequence of states returned by **agent.random_walk()**. It should work on a state given on the command line, or, if there is no state argument, on the default state (see above). Please assume that *N*=8 here (this can be hard-coded into **main.py**). The code you've been given in **util.py** includes a helper function that will pretty-print a list of states: **util.pprint(states)**. Here are a couple examples:

```
> python3 main.py random
```



```
> python3 main.py random "gbr |ggbb|rrrg|grbb"
```



Note that, because this is a random walk, the code will do different things for each run, so your runs will likely not match those above.

Breadth-First Search, Depth-First Search, and A* Search

Write methods for your Agent class **bfs(state)**, **dfs(state)**, and **a_star(state, heuristic)** that implement the respective search algorithms.

First, note that all of these algorithms have a common base algorithm, described at a high level in the "Graph-Search" algorithm in Lecture 4. In essence, all of these algorithms:

- Maintain a list of open nodes, namely the nodes that have yet to be considered in the search process. The initial list of open nodes contains only the given start state.
- Maintain a list of closed nodes, namely the nodes that have already been considered. The initial list of closed nodes is empty, but as a node is considered, it is added onto this list.
- The algorithms differ with respect to how they select a node from the open-node list to consider: breadth-first search and A* pop off the first node in the list, whereas depth-first search pops off the last node in the list. In addition, A* gives each node a value (i.e., $f(n) = g(n) + h(n)$), and the list of open nodes should be kept sorted by value at each iteration.
- When a new node is considered, if that node is the goal, we are done and can return the solution node. Otherwise, we expand the node by adding its next states to the open-node list, and continue searching.

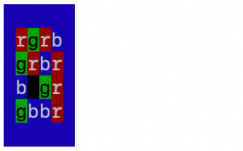
Write a method for your Agent class **_search()** with whatever arguments are needed to implement this base algorithm. Then, your implementations of **bfs()**, **dfs()**, and **astar()** should call this base method with appropriate arguments to differentiate how they work. (In other words, the real work of the search is being done in the base **_search()** method; the other methods are simply telling this method how to search.)

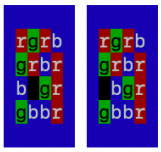
For A* search, note that there is one additional argument, the heuristic function, passed into the agent method **a_star(state, heuristic)**. The heuristic function $h(n)$ provides a value for a given state. For the RGB puzzle, you will need to choose and implement an *admissible* heuristic function $h(n)$, such that A* can reasonably estimate the minimum cost from a given state to the goal state. This heuristic function should be put into **main.py** so that it can be passed to agent when running the A* algorithm.

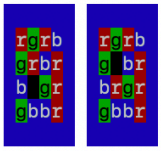
In addition, at each iteration of the search, print the path to the node being considered—the "path" here being the list of states from the initial state to the current state. You should again use the provided utility function **util.pprint(states)** to print the list of states.

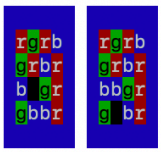
Finally, add command-line commands "**bfs**", "**dfs**", and "**a_star**" to **main.py** such that the commands call their respective search algorithms. Here are some examples:

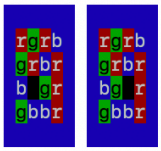
```
> python3 main.py bfs
```

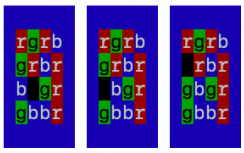




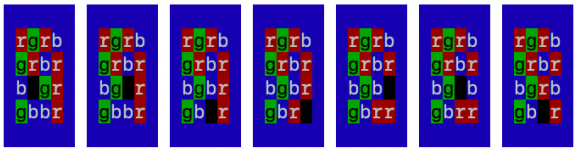








... and the rest of the paths searched,
ending with the solution and number of paths explored ...



```
309
```

In this case, BFS took 309 iterations before finding a solution. Because the given code in **rgb.py** sorts actions into a particular order, your BFS results should be the same. Your DFS may be faster or slower, but also may not find an optimal solution path. Your A* result should be better than BFS and still find the optimal solution path. You may want to design your own examples that are very close to a solution to make it easier to test your code.

Just for Fun: Searching on Assignment 1 [optional!]

If you finished the above without too much trouble, feel free to try integrating your agent search code with your implementation for Assignment 1 to see if it can solve those problems as well. This is an optional task with no extra credit, just for fun. ☺

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 there will receive a grade of zero.

For this assignment, you should submit the following Python files:

- **rgb.py** – the original RGB puzzle code given to you for this assignment* (you should not modify this file in any way!) ***See Assignment Page on Blackboard**
- **util.py** – the original utilities code given to you for this assignment* (you should not modify this file in any way!) ***See Assignment Page on Blackboard**
- **agent.py** – your code for the search agent
- **main.py** – your code for running everything with command-line arguments

Please use a compression utility to compress your files into a single ZIP file (not RAR or any other compression format). The final ZIP file must be submitted electronically using Blackboard—please do not email your assignment to a 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.