Code You Have Been Provided in the search engine for sliders

The file search.py, provides a generic search engine framework and code to perform several different search routines. This code will serve as a base for your solver. A brief description of the functionality of search.py follows. The code itself is documented and worth reading.

An object of class StateSpace represents a node in the state space of a generic search problem. The base class defines a fixed interface that is used by the SearchEngine class to perform search in that state space.

For the Sliders problem, we will define a concrete sub-class that inherits from StateSpace. This concrete sub-class will inherit some of the "utility" methods that are implemented in the base class. Each StateSpace object s has the following key attributes:

– s:gval: the g value of that node, i.e., the cost of getting to that state.

– s:parent: the parent StateSpace object of s, i.e., the StateSpace object that has s as a suc-cessor. This will be None if s is the initial state.

– s:action: a string that contains that name of the action that was applied to s:parent to generate s. Will be "START" if s is the initial state.

An object of class SearchEngine se runs the search procedure. A SearchEngine object is initialized with a search strategy ('depth first', 'breadth first', 'best first', 'a star', or 'custom') and a cycle checking level ('none', 'path', or 'full').

Note that SearchEngine depends on two auxiliary classes:

– An object of class sNode sn which represents a node in the search space. Each object sn contains a StateSpace object and additional details: hval, i.e., the heuristic function value of that state and gval, i.e. the cost to arrive at that node from the initial state. An f val f n and weight are tied to search nodes during the execution of a search, where applicable.

– An object of class Open is used to represent the search frontier. The search frontier will be organized in the way that is appropriate for a given search strategy.

When a SearchEngine's search strategy is set to 'custom', you will have to specify the way that f values of nodes are calculated; these values will structure the order of the nodes that are expanded during your search.

Once a SearchEngine object has been instantiated, you can set up a specific search with:

```
init_search(initial_state, goal_fn, [heuristic_function],
[fval_function])
```

and execute that search with:

```
search([timebound], [costbound])
```

The arguments are as follows:

– *initial state* will be an object of type StateSpace; it is your start state.

– *goal_fn(s)* is a function which returns True if a given state s is a goal state and False otherwise.

– *heuristic_fn(s)* is a function that returns a heuristic value for state s. This function will only be used if your search engine has been instantiated to be a heuristic search (e.g. best first).

– *f_val_fn(sNode; weight)* defines f values for states. This function will only be used by your search engine if it has been instantiated to execute a 'custom' search. Note that this function takes in an sNode and that an sNode contains not only a state but additional measures of the state (e.g. a gval). The function also takes in a float weight. It will use the variables that are provided to arrive at an f value calculation for the state contained in the sNode.

– *timebound* is a bound on the amount of time your code will be allowed to execute the search. Once the run time exceeds the time bound, the search must stop; if no solution has been found, the search will return False.

– *costbound* is an optional parameter that is used to set boundaries on the cost of nodes that are explored. This costbound is defined as a list of three values. costbound[0] is used to prune states based on their g-values; any state with a g-value higher than costbound[0] will not be expanded. costbound[1] is used to prune states based on their h-values; any state with an h-value higher than costbound[1] will not be expanded. Finally, costbound[2] is used to prune states based on their f -values; any state with an f-value higher than costbound[2] will not be expanded.

For the sliders problem, we have also provided sliders.py, which specializes StateSpace for this problem. You will therefore not need to encode representations of sliders states or the successor function for sliders! These have been provided to you so that you can focus on implementing good search heuristics and anytime algorithms.

The file sliders.py contains:

An object of class slidersState, which is a StateSpace with these additional key attributes:

– s:width: the width of the sliders grid

– s:height: the height of the sliders grid

– s:tiles: the distribution of the numbers in the grid

slidersState also contains the following key functions:

– successors(): This function generates a list of slidersState that are successors to a given slidersState. Each state will be annotated by the action that was used to arrive at the slidersState. These actions are the string direction-row_or_column, where direction can be “LEFT”, “RIGHT”, “UP”, “DOWN” and row_or_column can be the number of row of column where the action was applied.

– hashable state(): This is a function that calculates a unique index to represents a particular slidersState. It is used to facilitate path and cycle checking.

– print state(): This function prints a slidersState to stdout.

Also note that slidersState.py contains a set of 20 initial states for sliders problems, which are stored in the tuple PROBLEMS. You can use these states to test your implementations.

The file solution.py contains the methods that need to be implemented.