README.md

Sokoban Environment - NYU CS-GY6613

Prerequisites

Requires Python 3.8 to run

Install libraries

```
$ pip install -r requirements.txt
```

Run the Game

Solve as a human

```
$ python3 game.py --play  $ python3 game.py --agent Human
```

Solve with an agent

```
$ python3 game.py --agent [AGENT-NAME-HERE]
```

```
$ python3 game.py --agent BFS #run game with BFS agent
```

```
$ python3 game.py --agent AStar --no_render #run game with AStar agent without rendering
```

Solve with fast_game.py

```
$ python3 game.py --agent [AGENT-NAME-HERE]
```

```
$ python3 fast_game.py —-agent BFS #run all 100 levels with BFS agent
```

```
$ python3 fast_game.py —-agent HillClimber -trials 3 #run all 100 levels 3 times with HillClimber agent
```

Parameters

`--play` - run the game as a human player

`--no_render` - run the AI solver without showing the game screen

`--agent [NAME]` - the type of agent to use [Human, DoNothing, Random, BFS, DFS, AStar, HillClimber, Genetic, MCTS]

`--level [#]` - which level to test (0-99) or 'random' for a randomly selected level that an agent can solve in at most 2000 iterations. These levels can be found in the 'assets/gen_levels/' folder (default=0)

`--iterations [#]` - how many iterations to allow the agent to search for (default=3000)

`--solve_speed [#]` - how fast (in ms) to show each step of the solution being executed on the game screen

`--trials [#]` - number of repeated trials to run the levels for *(used only in fast_game.py)* (default=1)

Agent Types

Agent_py

- **Agent()** - base class for the Agents
- **RandomAgent()** - agent that returns list of 20 random directions
- **DoNothingAgent()** - agent that makes no movement for 20 steps
- **BFSAgent()** - agent that solves the level using Breadth First Search
- **DFSAgent()** - agent that solves the level using Depth First Search
- **AStarAgent()** - agent that solves the level using A* Search
- **HillClimberAgent()** - agent that solves the level using HillClimber Search algorithm
- **GeneticAgent()** - agent that solves the level using Genetic Search algorithm
- **MCTSAgent()** - agent that solves the level using Monte Carlo Tree Search algorithm

Code Functions

These are the only functions you need to concern yourselves with to complete the assignments. **WARNING: DO NOT MODIFY THESE FUNCTIONS!**

Sokoban_py

- **state.clone()** - creates a full copy of the current state (for use in initializing Nodes or for feedforward simulation of states without modifying the original) **Use with HillClimber to test sequences**
- **state.checkWin()** - checks if the game has been won in this state *(return type: bool)*
- **state.update(x,y)** - updates the state with the given direction in the form *x,y* where *x* is the change in x axis position and *y* is the change in y axis position. Used to feed-forward a state. **Use with HillClimber Agent to test sequences.**

Helper_py

- **Other functions**
  - **getHeuristic(state)** - returns the remaining heuristic cost for the current state - a.k.a. distance to win condition (return type: int). **Use with HillClimber Agent to compare states at the end of sequence simulations**
  - **directions** - list of all possible directions (x,y) the agent/player can take **Use with HillClimber Agent to mutate sequences**
- **Node Class**

    - **__init__(state, parent, action)** - where *state* is the current layout of the game map, *parent* is the Node object preceding the state, and *action* is the dictionary XY direction used to reach the state *(return type: Node object)*
    - **checkWin()** - returns if the game is in a win state where all of the goals are covered by crates *(return type: bool)*
    - **getActions()** - returns the sequence of actions taken from the initial node to the current node *(return type: str list)*
    - **getHeuristic()** - returns the remaining heuristic cost for the current state - a.k.a. distance to win condition *(return type: int)*
    - **getHash()** - returns a unique hash for the current game state consisting of the positions of the player, goals, and crates made of a string of integers - for use of keeping track of visited states and comparing Nodes *(return type: str)*
    - **getChildren()** - retrieves the next consecutive Nodes of the current state by expanding all possible directional actions *(return type: Node list)*
    - **getCost()** - returns the depth of the node in the search tree *(return type: int)*

- **MCTSNode Class (extension of Node() for use with the MCTSAgent only)**

    - **__init__(state, parent, action, maxDist)** - modified to include variable to keep track of number of times visited *(self.n)*, variable to keep track of score *(self.q)*, and variable to keep make score value larger as solution gets nearer *(self.maxDist)*
    - **getChildren(visited)** - returns the node's children if already made - otherwise creates new children based on whether states have been visited yet and saves them for use later *(self.children)*
    - **calcEvalScore(state)** - calculates the evaluation score for a state compared to the node by examining the heurstic value compared to the starting heuristic value (larger = better = higher score) - for use with the rollout and general MCTS algorithm functions

FAQs

- *I'm getting errors when installing the requirements.txt file*
    - Make sure you use Python version 3.8
- *What is `bestNode` ?*
    - `bestNode` keeps track of the node in the search tree that either contains the winning sequence or has the closest winning sequence (i.e. the best heuristic value.) The agent must return a sequence, so in the case that the solution is not found in the maximum number of iterations, the agent will return the `bestNode` 's action sequence. As such, you should update `bestNode` every iteration and evaluate it against the current node's heuristic. `getCost()` can be used as a tie-breaker between nodes.
- *What is `iterations` and `max_iterations` ?*
    - `iterations` keeps track of how many nodes you can expand upon in the tree, and `max_iterations` is the number of nodes you are allowed to search. In the case of evolutionary search, this is the number of times the solution is allowed to evolve. This variable is implemented to prevent infinite loops from occuring in case your algorithm has an error. Each level in the Sokoban framework should be able to be solved in 3000 iterations (the default number) or less for every AI solver.
- *Can I use other libraries?*
    - No. All the functions you need are already included in the code template. Do not import any internal or external libraries for any reason.
- *I get infinite loops / My agent is running in circles. / My agent keeps going back to the same place.*
    - Make sure you are using **`node_a.getHash() == node_b.getHash()`** to check the equivalency of 2 nodes and NOT `node_a == node_b`
- *Is it ok if my agent's solution is less than 50 steps but returned a solution length of 50 anyways?*
    - Yes, as long as it reaches the win state of the level.
- *My agent doesn't win all of the levels, is this ok?*
    - That's fine, as long as the final solution gets somewhat close to the win state.
    - We will grade on algorithm implementation rather than level completion.
    - However, for sanity checks, the agents should have around the following accuracies when tested on all 100 levels:

Agent Type	Accuracy (%)
BFS	~89%
DFS	~60%
A*	~99%
Hillclimber	~75%
Genetic	~90%
MCTS	~67%

- *How do I update and evaluate a state without using the Node class?*

    -
```
sequence = [directions[0],directions[3]]  #list of actions in the form {x: #, y: #}
mod_state = state.clone       #make a clone of the initial state to modify

#update the state with the sequence of actions
for s in sequence:
    mod_state.update(s[x],s[y])

h = getHeuristic(mod_state)   #save the resulting heuristic value from the updated state
w = mod_state.checkWin()      #check if sequence created a win state
```

- *What is `coinFlip` ?*
    - Use the value of `coinFlip` to determine whether a genome should mutate. For example:

```
if random.random() < coinFlip:
    #mutate
else:
    #do nothing
```