Assignment3.md

Assignment 1 - NYU CS-GY6613 - Fall 2020

Tasks:

Assignment in agent.py file:

1. Genetic Algorithm in the class GeneticAgent
2. MCTIS in the class MCTSAgent

After installing the requirements (see below) can test your code with `python3 game.py --agent Genetic` `python3 game.py --agent MCTS`

Deadline:

Tuesday, October 6, 11:55pm

General Instructions:

- Python 3 is required to run the Framework.
- All your code must be inside the agent.py file. **This is the only file you should submit**
- RandomAgent and DoNothingAgent are implemented as example agents.
- External libraries are not allowed (as you won't submit them).
- More instructions to run the environment and auxiliary functions you will need can be seen in the readme file included with the environment, and also listed below for convenience.
- Your code will be graded based on correctness, not on the agent's win rate or score. Do **not** attempt custom modifications (for example, modifying the heuristic) trying to "improve" the algorithm.
- With that said, on the basic map (level 0), both BFS and DFS, implemented correctly, are able to beat the level. Failure to do that probably indicates a mistake on your code.

You will fail the assignment if:

- You try to change any of the system params.
- Your code doesn't run (has errors).
- You don't write the code yourself.
- You submit anything beside agent.py file.
- You change the name of the agent classes.
- You implement your own heuristic.

More instructions to run the environment and auxiliary functions you will need can be seen in the readme file included with the environment, and also listed below for convenience.

GENETIC ALGORITHM INSTRUCTIONS

Each chromosome is an action sequence of length 50.

Use a population of size 10. Implement your algorithm with a mu + lambda strategy with mu = 5 and lambda = 5 :

1. Initialize a new population
2. Keep the best (minumum fitness) 5 individuals from the current population for the next population.
3. Generate the remaining 5 individuals by crossover followed by mutation:
   - Crossover
     - Choose 2 parents by rank selection (see below)
     - Generate a child by picking each gene (action in the action sequence) at random (50% chance) from each parent.
   - Mutation:
     - For each gene of the new child's action sequence (after crossover) do a random test.
     - If the test is < 0.3 (mutation rate), change it to a randomly chosen direction
     - Otherwise, do nothing

Use rank selection to select chromosomes:

- Sort all chromosomes based on fitness.
- Give each chromosome a rank (from 1(worst) to size of population (best)).
- Select chromosomes proportionally to their ranking, for example:
  - Assume we have 5 chromosomes (they have ranks 5,4,3,2,1 where 5 is better than 1)
  - The total of the rankings is 15=5+4+3+2+1.
  - Based on that, the probability to pick each chromosome is 5/15, 4/15, 3/15, 2/15, 1/15.
- Extra reading:
  - You can have a description of the Rank Selection process here (http://www.obitko.com/tutorials/ genetic-algorithms/selection.php).
  - For more details check the roulette wheel selection and use the rank instead of fitness (https://en.wikipedia.org/wiki/Fitness_proportionate_selection).

MCTS INSTRUCTIONS

The skeleton of the code is already implemented in getSolution and a helper Node class in MCTSNode. You just need to implement the four methods below:

1. treePolicty()
   - Use getChildren from the MCTSNode class to get a list of the node's children. This method already generates and adds the children to the current node for you if it has no children.
   - If the node has any children with no visits (n = 0), pick one of the unvisited children at random.
   - Otherwise, use bestChildUCT() to get the node with the best UCT score.
2. bestChildUCT()
   - Use the UCT formula with c=1 to find the best node to vist.
3. rollout()
   - Play the game with random actions (using state.update()) up to a depth of 7 or until you find a winning node.
   - Return the heuristic using node.calcEvalScore().
4. backpropagation()
   - Go up the tree adding 1 to the number of visits and the score to the total score.

Prerequisites

Requires python3 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 MCTS #run game with MCTS agent
```

Parameters

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

`--agent [NAME]` - the type of agent to use

`--level [#]` - which level to test (0-488) or 'random' for a randomly selected level that an agent can solve in at most 2000 iterations (default=0)

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

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 GeneticAAgent 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 GeneticAAgent to test sequences.**

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

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 GeneticAgent to compare states at the end of sequence simulations**
  - **directions** - list of all possible directions (x,y) the agent/player can take **Use with GeneticAAgent 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())**
  - **__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