UDACITY

‹ Return to Classroom

Build a Game Playing Agent

REVIEW

CODE REVIEW 2

HISTORY

Meets Specifications

Congratulations, you made it! 🎇

You showcased your skills in dealing with a **game-playing agent**, creating **heuristics**, comparing the **results**, and showing the results to the **stakeholders**. You can be proud of your job, and now you can move on toward your next goal!

In this project, you already applied many concepts you went through in the previous lessons, and that's amazing! However, there's something new to learn, especially in the **AI** field. That's why I suggest you check the documents listed below:

- Search Heuristics for Isolation
- A Survey of Monte Carlo Tree Search Methods

Good, that's all for now! I hope you enjoyed working on this project!
See you on the next projects.

Keep up the good work!

Best

Game Agent Implementation

(AUTOGRADED) Game playing agent can return an action.

- `.get_action()` method calls `self.queue.put()` at least once before the time limit expires

Correct! (Note: this rubric item was graded automatically.)

(AUTOGRADED) Game playing agent can play a full game.

- `CustomPlayer` successfully plays as both player 1 and player 2 in a full game to a terminal state (i.e., the agent does not deadlock during search, return an invalid action, or raise an exception during a game)

Correct! (Note: this rubric item was graded automatically.)

Experimental Results & Report

`CustomAgent` class implements at least one of the following:

- Custom heuristic (must not be one of the heuristics from lectures, and cannot *only* be a combination of the number of liberties available to each agent)
- Opening book (must be at least 4 plies deep)
- Implements an advanced technique not covered in lecture (e.g., killer heuristic, principle variation search, Monte Carlo tree search, etc.)

Great job! You have opted for the **Advanced Heuristic** implementation, and your algorithm passed all tests.

If you want to improve your algorithm to get better results, I suggest:

- Reading this document, which covers the **strategy game programming**. Therein you can find explanations of the **variation search** and **killer moves**.
- Read more **killer heuristics (and others)** here
- Check the **Principal Variation Search** algorithm starting from Wikipedia

Submission includes a table or chart with data from an experiment to evaluate the performance of their agent. The experiment should include an appropriate performance baseline. (Suggested baselines shown below.)

Advanced Heuristic

- Baseline: #my_moves - #opponent_moves heuristic from lecture (should use `fair_matches` flag in run_match.py)
  Opening book
- Baseline: randomly choosing an opening move (should *not* use `fair_matches` flag in run_match.py)
  Advanced Search Techniques

- Baseline: student must specify an appropriate baseline for comparison (student must decide whether or not `fair_matches` flag should be used)

Good job here! **Visualization** is a critical skill all **AI Engineers** have to put under their belt because usually, we have to deal with stakeholders that do not have our background. In this case, a simple **table** that shows the **performance** of the algorithms is a great choice.

Agent	Adversary	fair_matches	Num. Rounds	Win Rate
MINIMAX (α, β deepening.)	MINIMAX	used	100	77.6% (BASELINE)
MCTS	MINIMAX	used	100	80.0%
MCTS	GREEDY	used	100	93.8%
MCTS	RANDOM	used	100	98.8%

Table for the following CRITIERA 1.

Submission includes a short answer to the applicable questions below. (A short answer should be at least 1-2 sentences at most a small paragraph.)

NOTE: students only need to answer the questions relevant to the techniques they implemented. They may choose *one* set of questions if their agent incorporates multiple techniques.

Advanced Heuristic

- What features of the game does your heuristic incorporate, and why do you think those features matter in evaluating states during search?
- Analyze the search depth your agent achieves using your custom heuristic. Does search speed matter more or less than accuracy to the performance of your heuristic?

Opening book

- Describe your process for collecting statistics to build your opening book. How did you choose states to sample? And how did you perform rollouts to determine a winner?
- What opening moves does your book suggest are most effective on an empty board for player 1 and what is player 2's best reply?

Advanced Search Techniques

- Choose a baseline search algorithm for comparison (for example, alpha-beta search with iterative deepening, etc.). How much performance difference does your agent show compared to the baseline?
- Why do you think the technique you chose was more (or less) effective than the baseline?

Advanced Search Techniques

- Choose a baseline search algorithm for comparison (for example, alpha-beta search with iterative deepening, etc.). How much performance difference does your agent show compared to the baseline?
  ➔ The baseline search algorithm for comparison in this report is the minimax search algorithm with alpha-beta iterative deepening. Even if the algorithm itself is very slightly different from the original minimax algorithm, the performance of the alpha-beta search is way better than the minimax algorithm. With a simple heuristics, the performance can greatly be improved.
- Why do you think the technique you chose was more (or less) effective than the baseline?
  ➔ As MCTS concentrates on the more promising subtrees, the game tree in MCTS asymmetrically grows, so that it can achieve better results in games than classical algorithms with a high branching factor. However, this does not imply that MCTS is always better than the minimax search(generally better with using the same computational resource). While running MCTS, some branches leading to a loss can be chosen(some difficult solutions not easily searched with random approaches). The AlphaGo's loss in the fourth game against Lee Sedol may be related to this issue. The performance of MCTS can greatly improve, for example, with an opening book.

The answers to the **questions** are exhaustive and show you grokked the concept explained in the course. Let me suggest some **resource** for further learning:

- Check this article to read more about when it's better to aim at **speed** or **accuracy**
- Here is another article that explains how to prioritize **speed** and **accuracy**

DOWNLOAD PROJECT

2 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review

START

UDACITY

‹ Return to Classroom

Build a Game Playing Agent

REVIEW

CODE REVIEW 2

HISTORY

▾ my_custom_player.py 2

```
import random
from math import log, sqrt
from typing import List
from time import time
from isolation import Isolation
from isolation.isolation import Action
from sample_players import DataPlayer

TIME_LIMIT_IN_SECONDS = 0.145
VAL_MUL_UCB1 = 2
TH_NUM_PLAYS = 20


class CustomPlayer(DataPlayer):
    """ Implement your own agent to play knight's Isolation

    The get_action() method is the only required method for this project.
    You can modify the interface for get_action by adding named parameters
    with default values, but the function MUST remain compatible with the
    default interface.

    **********************************************************************
    NOTES:
    - The test cases will NOT be run on a machine with GPU access, nor be
      suitable for using any other machine learning techniques.

    - You can pass state forward to your agent on the next turn by assigning
      any pickleable object to the self.context attribute.
    **********************************************************************
    """

```

```
    def get_action(self, state: Isolation):
        """ Employ an adversarial search technique to choose an action
        available in the current state calls self.queue.put(ACTION) at least

        This method must call self.queue.put(ACTION) at least once, and may
        call it as many times as you want; the caller will be responsible
        for cutting off the function after the search time limit has expired.

        See RandomPlayer and GreedyPlayer in sample_players for more examples.

        **********************************************************************
        NOTE:
        - The caller is responsible for cutting off search, so calling
          get_action() from your own code will create an infinite loop!
          Refer to (and use!) the Isolation.play() function to run games.
        **********************************************************************
        """
        # TODO: Replace the example implementation below with your own search
        #       method by combining techniques from lecture
        #
        # EXAMPLE: choose a random move without any search--this function MUST
        #          call self.queue.put(ACTION) at least once before time expires
        #          (the timer is automatically managed for you)
        # self.queue.put(random.choice(state.actions()))
```

SUGGESTION

To keep the **code readable** I suggest removing **unused code**. Usually, we leave **commented-out code quality standard** in terms of **Code Readability**.

```
        tree = {}
        root_node = self._get_node_mcts(state, tree)
        mc_searcher = MCSearcher(tree, root_node)

        start_time = time()
        while time() - start_time < TIME_LIMIT_IN_SECONDS:
            mc_searcher.iterate()

        self._put_action_in_queue(root_node)

    def _put_action_in_queue(self, root_node: 'NodeMCTS'):
        children = root_node.children

        if children:
            action = max(children, key=lambda x: x.plays).action
        else:
            action = random.choice(root_node.state.actions())
```

AWESOME

The **Game Tree** is quite big when the game starts and selecting a **random move** is a reasonable alterr

```

        self.queue.put(action)

    def _get_node_mcts(self, state: Isolation, tree: dict) -> 'NodeMCTS':
        if state in tree.keys():
            node_mcts = tree[state]
        else:
            node_mcts = self._create_root(state, tree)
```

```python

        return node_mcts

    # noinspection PyMethodMayBeStatic
    def _create_root(self, state: Isolation, tree: dict) -> 'NodeMCTS':
        node_mcts = NodeMCTS(state)
        tree[state] = node_mcts

        return node_mcts


class NodeMCTS:
    __slots__ = ('state', 'action', 'parent', 'children', 'plays', 'wins')

    def __init__(self, state: Isolation, action: Action = None, parent: 'NodeMCTS' = None)
        self.state = state
        self.action = action
        self.parent = parent
        self.children = []
        self.plays = 0
        self.wins = 0.0

    def create_child(self, action: Action, state: Isolation) -> 'NodeMCTS':
        child = NodeMCTS(state, action=action, parent=self)
        self.children.append(child)

        return child


class MCSearcher:
    __slots__ = ('_tree', '_root_node')

    def __init__(self, tree: dict, root_node: NodeMCTS):
        self._tree = tree
        self._root_node = root_node

    def iterate(self):
        leaf_node = self._get_leaf_node(self._root_node)
        expanded_node = self._expand_children(leaf_node)
        self._pass_val_to_parent_nodes(self._get_utility_from_simulation(expanded_node.sta

    def _get_leaf_node(self, node: NodeMCTS) -> NodeMCTS:
        while True:
            children = node.children

            if children:
                for child in children:
                    if child.plays == 0:
                        return child

                if node.plays < TH_NUM_PLAYS:
                    node = random.choice(children)
                else:
                    node = self._ucb1(children)
            else:
                return node

    def _ucb1(self, children: List[NodeMCTS]) -> NodeMCTS:
        list_values = []
        log_children0_parent_plays = log(children[0].parent.plays)

        for child in children:
            value = child.wins / child.plays + VAL_MUL_UCB1 * sqrt(log_children0_parent_pl
            list_values.append((value, child))

```

```
            return max(list_values, key=lambda x: x[0])[1]

    def _expand_children(self, leaf_node: NodeMCTS) -> NodeMCTS:
        return leaf_node if leaf_node.state.terminal_test() else random.choice(self._creat

    def _create_children(self, parent_node: NodeMCTS) -> List[NodeMCTS]:
        for action in parent_node.state.actions():
            state = parent_node.state.result(action)
            self._tree[state] = parent_node.create_child(action, state)

        return parent_node.children

    # noinspection PyMethodMayBeStatic
    def _get_utility_from_simulation(self, state: Isolation, leaf_player_id: int) -> float
        while True:
            if state.terminal_test():
                return state.utility(leaf_player_id)

            state = state.result(random.choice(state.actions()))

    def _pass_val_to_parent_nodes(self, utility: float, node: NodeMCTS):
        leaf_player = node.state.player()
        while node:
            node.plays += 1

            if utility == 0:
                node.wins += 0.5
            else:
                player = node.state.player()
                if (utility < 0 and player == leaf_player) or (utility > 0 and player != l
                    node.wins += 1

            if node == self._root_node:
                return
            else:
                node = node.parent
```

RETURN TO PATH

Rate this review

START