ECS7002P - AI in Games

TAG: Tabletop Games Framework

Lab 3 - MCTS and Stochastic Games

Before starting this lab, **pull** from the TableTop repository to update your code with the latest version of the framework. You can also use the **Pull** option in GitHub Desktop to update your code.

1 Monte Carlo Tree Search - Implementation

Monte Carlo Tree Search (MCTS) is a highly selective best-first tree search method. This iterative algorithm balances between *exploitation* of the best moves found so far and *exploration* of those that require further investigation, respectively. On each iteration of the algorithm, the standard MCTS performs 4 steps: selection, expansion, simulation and back propagation.

A simple implementation of the MCTS agent can be found at `players.basicMCTS.BasicMCTSPlayer.java`.[1]. The main classes here are:

- `BasicMCTSPlayer`: it's the class that extends from Player.java and implements the agent API.
- `BasicTreeNode`: implements an MCTS node and all the logic required for the MCTS algorithm.
- `BasicMCTSParams`: a class for defining the parameters of MCTS.

1.1 BasicMCTSPlayer

`BasicMCTSPlayer` implements the entry point of the agent. The *_getAction()* method of this agent looks as follows:

```
1  public AbstractAction _getAction(AbstractGameState gameState, List<AbstractAction>
       allActions) {
2    // Search for best action from the root
3    BasicTreeNode root = new BasicTreeNode(this, null, gameState, rnd);
4
5    // mctsSearch does all of the hard work
6    root.mctsSearch();
7
8    // Return best action
9    return root.bestAction();
10 }
```

The call to *mctsSearch()* runs the MCTS algorithm, while the line *root.bestAction()* retrieves the action from the root node that should be executed in the real game.

1.2 BasicMCTSParams

BasicMCTSParams allows you to specify the configuration of MCTS. This includes the main MCTS parameters, such as the exploration constant, the maximum tree depth, the rollout depth and the state evaluation heuristic used to obtain rewards from states found at the end of each iteration. A *players.basicMCTS.BasicMCTSParams* object can be passed to the MCTS constructor upon creation:

```
1 BasicMCTSParams params = new BasicMCTSParams();
2 AbstractPlayer player = new BasicMCTSPlayer(params);
```

The following code shows an example of how to build a BasicMCTSPlayer with an BasicMCTSParams object that includes values set for different algorithmic parameters:

[1] A more complex MCTS player, which includes opponent policies, advanced selection policies and rollout types, among other enhancements, can be found in `players.mcts.MCTSPlayer.java`. You are **not** allowed to use this agent for the assignment.

```
BasicMCTSParams params = new BasicMCTSParams();
params.K = Math.sqrt(2);      //UCB1 Exploration constant
params.rolloutLength = 10;    //Maximum length for the rollouts
params.maxTreeDepth = 5;      //Maximum length the tree can grow.

// Create the player with the given parameters.
BasicMCTSPlayer player = new BasicMCTSPlayer(params);

// The heuristic to evaluate states:
player.setStateHeuristic(new MyConnect4Heuristic());
```

The code above could be written in the main functions of the classes that start the execution, such as `core.Game` or `evaluation.RunGames`, which we have seen previously. Similarly, they can be setup in JSON configuration files for agents.

Finally, an important set of parameters that you can set is the control for the allocated budget for each action decision. The following snippet shows an example of setting a budget of type FM_CALLS and value 1000 (this agent will then use 1000 times the forward model before returning an action to the game):

```
BasicMCTSParams params = new BasicMCTSParams();
params.budgetType = PlayerConstants.BUDGET_FM_CALLS;
params.budget = 1000; //1000 FM calls
```

Other available budgets that you can specify are defined in the enumerator `players.PlayerConstants`. The most relevant are:

- BUDGET_FM_CALLS: Sets the limit on the number of times the model can be rolled forward.
- BUDGET_ITERATIONS: Sets the limit on the number of iterations of the algorithm.
- BUDGET_TIME: Sets the limit on the number of milliseconds the agent can use to make a decision.

Note that these variables are used to indicate an agent when to stop searching. It's responsibility of the agent code to actually respect these limits within its implementation (see below for MCTS).

The easiest way to specify these parameters is, however, via JSON. The file *json/players/basicmcts.json* can be used to run BasicMCTS with custom parameters. Its contents are as follows:

```
{
  "class" : "players.basicMCTS.BasicMCTSParams",
  "K" : 1.0,
  "rolloutLength" : 0,
  "maxTreeDepth" : 30,
  "heuristic" : {
    "class" : "players.heuristics.ScoreHeuristic"
  },
  "budgetType" : "BUDGET_TIME",
  "budget" : 40
}
```

1.3 BasicTreeNode

Tree nodes are implemented in *players.mcts.BasicTreeNode*, and they implement the different steps of MCTS: tree policy, expansion, rollout and back propagation. One of the main methods of this class is *mctsSearch()*, which runs the main loop of the algorithm:

```
while (!stop) {
  // Selection + expansion: navigate tree until a node not fully
  // expanded is found, add a new node to the tree
  BasicTreeNode selected = treePolicy();

  // Monte carlo rollout: return value of MC rollout
  // from the newly added node
  double delta = selected.rollOut();

  // Back up the value of the rollout through the tree
  selected.backUp(delta);

  // Finished iteration
```

```
    numIters++;

    /* Check stopping condition code {...} */
    // ...
}
```

As you can see above, three methods take care of the 4 steps of the algorithm:

- `treePolicy()`: performs selection (using UCB1) and expansion.
- `rollOut()`: performs the Monte Carlo simulation step.
- `backUp()`: performs the Backpropagation step.

Exercise 1 Explore the 3 methods mentioned above and see if you can understand what is the code doing at each step. Do not hesitate to use the debugger to inspect variables and run it step by step if needed.

See if you can answer the following questions:

1. Can you explain the different lines of code included in the UCB1 calculation (method BasicTreeNode.ucb())?
2. How does the selection step stop so that the algorithm moves to the expansion step?
3. In the expansion, how do you select which action must be chosen to add a new node?
4. In the simulation step, how are the actions chosen for the rollout? When does a rollout terminate?
5. How and where (in the code) are the statistics of each node being updated after the state at the end of the rollout has been evaluated?

mctsSearch() also includes the code for controlling when to stop the algorithm when the budget runs out, according to how is set up in the MCTSParams object supplied to the MCTS agent. This code looks as follows:

```
while (!stop) {
    ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();

    /* Main MCTS Steps {...} */

    // Check stopping condition
    PlayerConstants budgetType = player.params.budgetType;
    if (budgetType == BUDGET_TIME) {
        // Time budget: computes avg time per iteration to avoid overspending
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis());
        avgTimeTaken = acumTimeTaken / numIters;
        remaining = elapsedTimer.remainingTimeMillis();
        stop = remaining <= 2 * avgTimeTaken || remaining <= remainingLimit;
    } else if (budgetType == BUDGET_ITERATIONS) {
        // Iteration budget
        stop = numIters >= player.params.budget;
    } else if (budgetType == BUDGET_FM_CALLS) {
        // FM calls budget
        stop = fmCallsCount > player.params.budget;
    }
}
```

Exercise 2 For the case of using BUDGET_FM_CALLS, the variable `fmCallsCount` counts how many time the forward model is used during the execution of the algorithm. Identify where in the code this variable is updated.

2 MCTS to the test

In this section, we are going to try running MCTS against other agents in several games, using different settings. We will be running this from the evaluation.RunGames class using JSON configuration files.

Exercise 3 First of all, create a JSON tournament configuration file with the following parameters:

- Game: TicTacToe
- Number of players: 2
- Mode: Exhaustive.
- Number of matchups: 100.
- Listener: "json/listeners/basiclistener.json"
- DestDir: An output directory of your choice.
- output: An output file for the tournament log of your choice.
- playerDirectory: A directory of your choice to place the ageng JSON files.

Then, create or copy the Agent JSON files for the players Random (*json/players/random.json*) and BasicMCTS (*json/players/basicmcts.json*) to the playerDirectory folder you specified above.

The Random Player has no configuration parameters, but BasicMCTS does. For the first run, configure the MCTS agent with the following values:

- class: players.basicMCTS.BasicMCTSParams
- K: 1.4 (this is approximate $\sqrt{2}$, for the UCB constant a.k.a. 'C').
- rolloutLength: 2
- maxTreeDepth: 3
- budgetType: BUDGET_FM_CALLS
- budget: 1000

Now, run RunGames and observe the results you get.

What happens if you execute the same tournament again but varying the rollout length of MCTS? Use values 3, 5 and 10. What do you observe in the results? Does the win rate of each agent change significantly when using different values? Why do you think this is?

Exercise 4 Now, let's repeat the previous experiment with different agents and games. Set up the parameters to a fixed set of values (for instance, rolloutLength = 10, maxTreeDepth = 5, and 1000 forward model calls as a budget). Complete the table below with the win rates of MCTS and the opponent agent on each pairing. What do you think of the results?

Opponent:	RandomPlayer	FirstActionPlayer	OSLAPlayer
Tic Tac Toe			
Connect 4			
Dots and Boxes			

Exercise 5 Finally, play the three games indicated above between yourself and MCTS. For this, go to core.Game and add the code to run these games with BasicMCTSPlayer (using BasicMCTSParams, as shown above) and the HumanGUIPlayer. Play a handful of games against MCTS on each game to get a feel of the AI strength. Can you beat it often?

3 Stochastic Games

The games we have seen so far (Tic Tac Toe, Connect 4 and Dots and Boxes) are generally considered simple games: they are 2-player, not very long, fully observable and completely deterministic.

A stochastic game is a game that has one or more random elements that affects the rules or interaction between the different components[2]. An example of this is a game where different events happen at the roll of a dice, or when using a shuffled deck of cards.

In this section, we are going to work with **Can't Stop**, a stochastic dice rolling game.

Exercise 6 Learn about how the game Can't Stop works. Some resources are:

- Wiki entry: `https://en.wikipedia.org/wiki/Can%27t_Stop_(board_game)`
- Short video description (3 min): `https://www.youtube.com/watch?v=VUGvOQatVDc`
- Another video (7 min): `https://www.youtube.com/watch?v=npYf28jVAuA`

The interesting aspect of Can't Stop is its stochasticity: the actions available at each step depend on the roll of the dice, and all numbers are not equiprobable (for instance, it's more likely to roll for a 7 than a 2 or a 12). Therefore, the decisions on every turn between stopping and keep rolling dice must be done based on simulations from that level of the tree that are not only stochastic because of the nature of random rollouts, but also due to the variance of actions available on each iteration.

We are going to analyze how MCTS deals with this empirically, with the following exercises:

Exercise 7 First, start playing yourself against BasicMCTS as you did previously, this time playing Can't Stop ("CantStop" in TAG). Keep the parameters as rolloutLength = 10, maxTreeDepth = 5, and 1000 forward model calls as a budget. Play a few games. What do you think of the strength of the algorithm in this game?

Unless you've been unlucky with your rolls, chances are that you've been able to beat MCTS more than once.

Exercise 8 Let's go back to tournaments:

1. Set up a tournament so BasicMCTS plays Can't Stop ("CantStop" in TAG) against the three typical agents: RandomPlayer, FirstActionPlayer and OSLAPlayer. Configure the tournaments to play 100 games each and observe the results. Is BasicMCTS still better than the other agents?
2. Set up now a tournament between two *different* BasicMCTS agents. You'll need to create two different JSON Configuration files in your players directories. Then, modify their parameter values: keep the same forward model budget for both agents to the same value (i.e. 1000), but try different values for the rolloutLength and the maxTreeDepth parameters. What can you conclude about the effect that these parameters have in the play-strength of the agents?

3.1 Writing a heuristic for Can't Stop

In this last section, you are going to implement a Heuristic (concretely, a state evaluation function) for the game Can't Stop. We already saw how to write heuristics in a previous exercise, so please take a look at that if you don't remember.

Let's start with creating the class.

Exercise 9 Create a file called CantStopHeuristic.java in the package `players.heuristics`. The class CantStopHeuristic must implement the interface `IStateHeuristic` and define the following method (leave if empty for now, returning any double value):

```
1 double evaluateState(AbstractGameState gs, int playerId);
```

The next step is to make our BasicMCTSPlayer use this heuristic. For this, you only need to assign it

[2]More formally, a stochastic game is a Markov game with probabilistic transitions between states. We'll see that later in the course.

to the player object using the `setStateheuristic()` function (in code) or as a JSON parameter (if using JSON configuration files). For the latter, your BasicMCTS agent JSON must include a parameter as follows:

```
"heuristic" : {
    "class" : "players.heuristics.CantStopHeuristic"
},
```

When configuring the agent to use this heuristic, you can execute one game to verify you are calling the right heuristic (for example, run in Debug and set a breakpoint in the only line of code of your new `CantStopHeuristic.evaluateState()` method.

Once you've verified that the BasicMCTS agent is using your heuristic, it's time to write some code that actually evaluates a state to drive the search done by MCTS. Several methods exist in CantStopGameState that allow you to query the state of the game.[3] Here's a list:

- `boolean trackComplete(int n)`: returns true if column 'n' is completed.
- `int[] getDice()`: returns the last dice values rolled.
- `int getMarkerPosition(int number, int player)`: returns the position of a marker by player ID 'player' in column 'number'.
- `int getTemporaryMarkerPosition(int number)`: returns the position of a temporary marker (this is, before passing in a turn) in column 'number'.
- `List<Integer> getMarkersMoved()`: returns the list of temporary markers that have been moved so far (before passing the turn).
- `double getGameScore(int playerId)`: returns the current game score for player 'playerId'.
- `void printToConsole()`: prints the current game state to console.

As `CantStopGameState` inherits from `AbstractGameState`, you will also have access to more general functions, such as:

- `int getCurrentPlayer()`: returns the ID of the player to play in the current state.
- `Utils.GameResult getGameStatus()`: indicates if the game has been won, lost, or is still running.
- `Utils.GameResult[] getPlayerResults()`: returns the player results, in an array for all players, indexed by the player ID.
- `boolean isNotTerminal()`: returns true if the game is not over.

Exercise 10 Implement a heuristic function for evaluating game states in CantStopGameState.evaluateState() with the aid of the functions above. Make sure that you cast `AbstractGameState` to `CantStopGameState` (see the footnote about this) in order to have access to the methods from `CantStopGameState`.

Try the effectiveness of your heuristic by running tournaments between two BasicMCTS agents, one that uses your new heuristic and the other one using the default game heuristic (don't set any heuristic for this second agent). Make sure both agents have the same set of parameters values for the exploration constant, rollout length and maximum tree depth, so they play on an equal field.

After the lab: 1 Can you build a heuristic that makes the MCTS agent that uses it defeat the other one at least 75% of the time?

[3] Note that, in order for you to be able to call these functions, you need to cast the AbstractGameState received by parameter in evaluateState() to an object of type CantStopGameState. You can do that with the line: `CantStopGameState cs = (CantStopGameState) gs;`