MCPlan: A Java-Based Monte-Carlo Planning Library

June 28, 2013

The planning library uses a Java code base that allows new agents and domains to be easily added and to interact with the already present agents and domains. The code base is open source under the GPL 2.0 license. There are currently eight domains and 6 planners included in the library.

The planning algorithms include:

- Random – a baseline agent that selects random actions.
- Uniform Policy Rollout – a policy rollout agent that samples root actions uniformly.
- ϵ-Greedy Policy Rollout – a policy rollout agent that uses ϵ-Greedy exploration at the root.
- UCT – the popular Monte-Carlo Tree Search Algorithm based on use the UCB rule to guide exploration/exploitation.
- Sparse ExpectiMax – an implementation of the sparse sampling algorithm.

Powerpoint slides that overview these agents are available from the main MCPlan web page.

The base class structure consists of the following classes: State, Simulator, Agent and Arbiter. The State classes abstractly represent a state in a given domain. This class just holds information about the state with some accessor methods. It is typical for this class to be immutable. The Simulator class is tied to a specific state and action and computes legal actions and rewards at a given state and the state transition given an action. The simulator is used by the agents to explore the domain and the arbiter to regulate a game. The main method of the Agent class is selectAction that takes a state and simulator as parameters and returns an action. The Arbiter class calls the selectAction method on each Agent when it is time to make a move. The Arbiter class is what governs the legal play of a game. It takes in the agents and simulators needed to play a game. It also records some statistics such as time taken to move. The Arbiter class can run many games in the same domain against the same set of agents and collect statistics on average reward. In order to keep things fair it rotates the move order of agents each game.

The documentation for these agents and the library in general is not yet complete. We expect to produce a significantly updated library with more extensive documentation in the Fall of 2013. Until then feel free to contact Alan Fern (afern@eecs.oregonstate.edu) with questions.

1 Running Tests

The McPlan class is used to run tests. This class takes zero, one or two arguments. If it is run with no arguments then it runs in interactive mode. In this mode the user selects a simulator and agents. This is the only mode that the human agent can be used. If the human agent is run the the user inputs actions and gets feedback about the the current state and legal actions. If there are no human agents then the single game is simulated and the history and results are shown at the end.

If a single argument is passed then it is the file path of a test file. An example test file is given 1. All white space and comments (any line starting with a #) are ignored. The first line read in (that isn't a comment or whitespace) indicates the number of complete games to be simulated. The second line indicates the simulator that will be passed to the arbiter, which is the exact simulator of the game. The third line indicates the internal simulator that each agent will use to make decisions. The reason this distinction is made is that an agent may not have access to the complete real world simulator but only a close approximation. The next two lines are used to specify the agents to run in the domain. Each line is given to an agent and the parameters for that agent are separated by white space. Note that if a domain such as backgammon requires two agents it will throw an exception if not enough agents are included in the test file and if more than two agents are specified it will just use the first two agents. The test file in figure 1 will play 2000 games in the domain of Backgammon between a Random agent and a UCT agent with the number of trajectories set to 1024, C = 1, sparse sample size set to infinity.

The output of the test from 1 "backgammon_random_uct" will be appended to the file "backgammon_random_uct_results" 2. All output will be appended to a file with the name of the input file plus "_results" attached to the end of the name. If that file does not already exist it will be created. Figure 2 shows what the output of figure 1 might look like. The fist line is just the first 3 lines of the test file put onto one line. Each line that follows is given to an agent in the tests. The first value is the average reward while the second parameter is the standard deviation of the average reward. The next two values are the average time per move and the standard deviation in time per move respectively. Then at the end is appended the agent name and the parameters it used. If a second argument is passed to UCTProject then that second argument is the name of the output file instead of the default name.

Another useful feature of the test files is shown in figure 3. This time the domain is Yahtzee and since it is a single agent domain only one agent is given. However, the number of trajectories that UCT takes in as a parameter is given

Figure 1: Test File "backgammon_random_uct"

```
#Number of Simulated Games
2000

#World
Backgammon

#Simulated World
Backgammon

#Agents
Random
UCT 1024 1 -1
```

Figure 2: Output File "backgammon_random_uct_results"

```
#2000 - BackgammonSimulator - BackgammonSimulator
 -0.994, 0.109,   0.014,  0.023, Random
  0.994, 0.109, 437.494, 91.069,    UCT, 1024, 1, -1
```

by an array of values, [128,256,512,1024], instead of a single value. What this does is run 4 separate tests and append the results of each test to the same output file. Each test will use a different value for the number of trajectories. Multiple parameters can be specified in this manner as shown in figure 4. In this case all possible combinations of the parameters are run. Thus there will be output for the following pairs of parameters in the order given: (128,2), (256,2), (512,2), (1024,2), (128,4), (256,4), (512,4), (1024,4). Instead of writing [2,4,6,8,10] you may input [2:4:10]. The first and last values are the first and last values to test respectively. The difference between the first and last values divided by the middle value is the step value. Thus for [2:4:10] it would be $(10 - 2)/4) = 2$ and the results values would be [2,4,6,8,10].

2 Adding New Domains

Any new domain that is added to this library will need to meet a few requirements. All domains consist of at least three parts: a simulator, state and action class. There must be one top level state class that implements the State interface. If the domain has multiple types of states or actions these can in turn inherit from the top level action and state class from that domain. For instance in the domain of Yahtzee there is a single state class, YahtzeeState, that implements State and an action class, YahtzeeAction (There is no Action interface to

Figure 3: Test File 2

```
#Number of Simulated Games
2000

#World
Yahtzee

#Simulated World
Yahtzee

#Agents
UCT [128,256,512,1024] 64 -1
```

Figure 4: Test File 3

```
#Number of Simulated Games
2000

#World
Yahtzee

#Simulated World
Yahtzee

#Agents
UCT [128,256,512,1024] 64 -1
```

implement). Since in Yahtzee there are two different actions that can be performed, namely selecting dice to re-roll or selecting a category to score Yahtzee has two other action classes, YahtzeeRollAction and YahtzeeSelectAction that inherit from YahtzeeAction. YahtzeeAction is never used as an object. The only reason to have it is so that the Yahtzee simulator, YahtzeeSimulator, can use generics to specify that it only uses actions of type YahtzeeAction and states of type YahtzeeState. This protects the simulator from being misused by passing in a state or action that is not a Yahtzee state or action.

This library was designed to use immutable state and action objects (objects that hold data that can be accessed by other objects but cannot be modified). Instead of changing a state a new state object is created to replace the old one. This has advantages and disadvantages. If a mutable action or state object is created then some of the basic simulator methods also need to be overridden since they assume that the objects are immutable.

The simulator class itself is a mutable object. It contains the methods for taking an action and computing the rewards and legal actions at a given state. A simulator has a method takeAction that replaces its current state with a new state object based on the passed in action. It is important for a simulator to check that the passed in action is indeed a legal action from that state or there could be problems. Also, the simulator keeps a record of the legal actions and rewards array for the current state. These should be updated whenever the state changes. The getLegalActions and getRewards methods don't need to be overridden as they just return these objects. This means that it is desirable to make some kind of method that computes the legal action and rewards that can be called whenever the state changes. For all of the current simulators computeLegalActions and computeRewards are used as private methods although the names don't matter.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant Numbers 0958482 and IIS-0905678. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.