CAL POLY

101 202 480

Instructor Info

Projects

Tile Driver I

Tile Driver II

Biogimmickry

Mine Shafted

Board Stupid

Course Details

Description

Assessment

Resources

Calendar

Policies

Enrollment

Code Style

Student Conduct

Accommodation

Guides

Casey Submission System

Getting Started

Project V: Board Stupid

Due Mon, Aug 9 by 9PM Pacific Time

To begin, use the following file: http://users.csc.calpoly.edu/~dkauffma/480/boardstupid.py

- To download locally, right-click link and choose "Save link as...".
- To download and unzip on a CSL server run:
  `wget http://users.csc.calpoly.edu/~dkauffma/480/boardstupid.py`
  `unzip boardstupid.py`

Description

In some search problems, including many games, one or more agents may be attempting to thwart your ambitions. Adversarial search algorithms provide a way to increase the likelihood of reaching a desired state (a winning condition) while other agents try to prevent you from reaching this goal. In the simplest case, this competition is between two players, traditionally labeled `MAX` and `MIN`, who select moves that respectively maximize or minimize an estimated utility value. These utility values guide each player through the game tree toward a favorable goal.

When experimenting with adversarial search methods, Tic-Tac-Toe is a popular choice due to the simplicity of its rules. The state space can also easily be expanded by increasing the size and number of the boards. Even with four 4x4 boards - known as 3D Tic-Tac-Toe (or Qubic) - the game tree becomes far too large for an exhaustive search. See this document for a visual representation of all the 3D Tic-Tac-Toe winning conditions. Whereas classic Tic-Tac-Toe has 8 winning conditions, the 3D version has 76 winning conditions (verify this number for yourself to ensure you understand the rules).

The Multi-Armed Bandit Problem

Suppose you are at a casino (i.e. you don't value your free time) and you decide to play the slot machines (i.e. you *really* don't value your free time). These slot machines (or "one-armed bandits") may have different payout rates, which can only be determined by observation. As you begin experimenting by playing different machines, you find

Secure Shell (SSH)

Vim Editor

that some machines initially appear to pay out more often than others, but the number of experiments has been too small to be conclusive. After `N` plays, which machine should you play next?

This problem demonstrates the **exploration** versus **exploitation** tradeoff. By continuing to play machines that have thus far paid well, you are exploiting them; however, another machine could have a higher payout rate if your observations are insufficient, requiring you to occasionally explore other machines to see if those options become better.

When selecting the next machine to play, the exploration versus exploitation tradeoff can be balanced using the calculation for an Upper Confidence Bound (UCB).

$$\frac{W}{N} + C * \sqrt{\frac{log\ T}{N}}$$

In this equation, `W` refers to the number of wins from the machine under consideration, `N` is the number of times that machine was played, and `T` represents the total number of plays on all machines. The variable `C` is used as an **exploration bias** parameter to help control the frequency of exploration (`√2` serves as a typical baseline). The UCB is then calculated for each machine, with the highest-valued machine selected for the next experiment.

Monte Carlo Tree Search

In almost all games, it is impractical to do an exhaustive search of the game tree. Some approaches to finding good moves in large trees involve engineering game features for use in a state evaluation function. While effective if the features closely reflect likely outcomes, evaluation functions are specific to the game for which they are developed and ill-suited for many complex games such as Go.

Monte Carlo Tree Search is a method that uses sampling to estimate the quality of a move by exploring many paths through the move and tallying the number that result in victory. The algorithm works by performing a large number of tree traversals, each known as a playout. During these playouts, a frontier is maintained (see below).

On each iteration of the search, the following actions are performed.

- **Selection:** Starting from the root of the game tree (which represents the current state of the game), traverse down the tree

by selecting the child state at each level with the highest UCB. (States yet to be traversed may be assigned the value `C` or something similar.) Note that, at each level, `T` is equivalent to the `N` of the parent; that is, the total number of attempts of a level is equal to the number of attempts through the parent of the level. Continue until a frontier or terminal state is reached (whichever comes first).

- If a state is selected that is non-terminal and on the frontier:
  - **Expansion:** Add one of its children at random to the frontier and traverse to it. Remove its parent from the frontier. It is recommended to maintain an explored set for former-frontier states so that their win ratios can still be accessed and updated. Only one state should be added to the frontier on each iteration of the search.
  - **Simulation:** Perform a random walk toward a terminal state.
- **Backpropagation:** Once at a terminal state, determine the outcome of the game with respect to the current player: win, loss, or tie. Each outcome should have a different value, as determined through experimentation. Return this value back up through the tree to the root, updating each state's win ratio appropriately along the way.

Implementation

Allowed Modules: math, random

For this assignment, a 3D board will be represented as a 4-tuple of 2D boards, with each 2D board represented as a tuple of integers, using `0` for an empty space and `1` and `-1` for `MAX` and `MIN`, respectively. Each 2D board has `16` spaces, for a total of `64` spaces per game state. The initial state is shown below.

```
((0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
```

In addition, some spaces may be set to `None` before the game begins. These spaces should be considered barriers that no player may select for a move. This feature allows for the reduction of the search space, which can simplify the creation of tests.

See the starter file for the `GameState` class, which provides a representation of game states, including the current player, the actions available at that state, and the utility of the state, as well as a means to traverse the game tree.

```
find_best_move(state: GameState) -> None
```

Search the game tree for the optimal move for the current player, storing the index of the move in the given GameState object's `selected` attribute. The move must be an integer indicating an index in the 3D board - ranging from `0` to `63` - with `0` as the index of the top-left space of the top board and `63` as the index of the bottom-right space of the bottom board.

This function must perform a Monte Carlo Tree Search to select a move, calling additional functions as necessary. During the search, whenever a better move is found, the `selected` attribute should be immediately updated for retrieval by the instructor's game driver. Be sure to update `selected` with the move that has the highest **win ratio**, not highest UCB.

Each call to this function will be given a set number of seconds to run; when the time limit is reached, the index stored in `selected` will be used. Note that this function will only be evaluated on finding the best move for isolated boards; in other words, it will **not** be run multiple times in succession on the same board until a player wins.

Submission

On a CSL server with `boardstupid.py` in your current directory:

Instructor	Command
Daniel Kauffman	`/home/dkauffma/casey 480 boardstupid`