CSCI1410 Fall 2018
Assignment 2: Adversarial Search

Code Due Monday, September 24
Writeup Due Thursday, September 27

1 Introduction

In this assignment, you will implement adversarial search algorithms under different sets of assumptions. Then, you will answer two written questions.

2 Your Task

2.1 Coding

In `adversarialsearch.py`, you will implement the following four functions:

1. `minimax`. Implement the minimax algorithm for two-player, constant-sum games. Recall from the game theory lecture that **constant-sum** describes games in which the sum of the rewards earned by the players is the same across all possible outcomes. Your implementation will search all the way down to terminal states.

2. `alpha_beta`. Same as part 1, but use alpha-beta pruning.

3. `alpha_beta_cutoff`. Same as part 2, but this time cutting off search after some given number of turns and applying a given evaluation function. See the function's docstring for more details.

4. `general_minimax`. A generalization of `minimax` (*no* pruning) that supports games with two *or more* players and arbitrary (potentially variable-sum) reward structures and chooses a maximin action. Read the subsection 2.1.1 for details.

For the first three coding tasks (but *not* the fourth), you may assume that players' turns are **alternating**. This means that the second player always takes his turn after the first player, and the first player always takes her turn after the second player. **You may never assume that you are the first player**, but

you should always maximize your score.

Each function that you will implement takes in an `AdversarialSearchProblem` as an input—much like how your search algorithms took in a `SearchProblem` in the previous assignment—and outputs an action to take. We strongly recommend reading and understanding `adversarialsearchproblem.py` before writing any code.

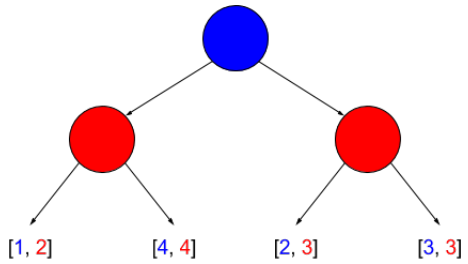
2.1.1 Minimax for Variable-Sum Games

The Goal: The goal of each agent is still to maximize his own points, *irrespective* of the points that the other agents earn. That is, as Player 1, you should prefer [3, 100] to [2, 0].

The Algorithm: It is not clear how to generalize the minimax algorithm from the constant-sum setting to the variable-sum setting. There at least two reasonable approaches: the faithful approach and the conservative approach.

1. **Faithful Approach:** Prescribe the rational action assuming Common Knowledge of Rationality (CKR) between the players. Recall from the game theory lecture that CKR means that every player is rational, and every player knows that every player is rational, and every player knows that every player knows that every player is rational, and so on, ad infinitum. Here, "rational" describes a player who chooses the action to maximize their points given their beliefs about what the other players will do.
2. **Conservative Approach:** Prescribe the maximin action. Recall from the game theory lecture that the maximin action is the one that ensures the best worst-case scenario for the actor.

Here is an example that illustrates the difference between the faithful and conservative approaches in this setting.



In the diagram above, there are just two players: Blue and Red. The lists at the bottom represent terminal states of the game, in which the first number is Blue's points, and the second number is Red's points.

The Blue player can move Left or Right.
If Blue takes the faithful approach, Blue thinks about what Red would do in each subgame, assuming that Red is rational. If Blue were to move Left, then a rational Red would move Right, earning payoff [4, 4] for the players, and if Blue were to move Right, then a rational Red would assess Left and Right as equally favorable options, earning payoff [2, 3] or [3, 3] for the players. After completing this analysis, Blue would decide to move Left, since doing so would lead to more points under the assumption of CKR.

In contrast, if Blue takes the conservative approach, what would Blue do? If Blue moves Left, then the worst-case scenario earns 1 point, and if Blue moves Right, then the worst-case scenario earns 2 points. Trying to make the best worst-case scenario, the conservative approach prescribes Right in this case.

The generalization of minimax that you will implement (for `general_minimax` in the stencil) will take the conservative approach and choose a maximin action.

2.2 Written Questions

1. The two approaches for generalizing minimax in variable-sum games in section 2.1.1 are both reasonable because they are both true of minimax in two-player, constant-sum games.
   Argue *informally* that in two-player, constant-sum games, an action is a maximin actions if and only if it is the rational action to play under CKR.

2. A friend says, "Use the minimax algorithm if you can! It produces the best decisions possible in games." Do you agree with their assertion? If not, give at least two reasons (not examples, but *reasons*) why you think the assertion is wrong. Limit your response to 3 sentences.

3 The Code Files

3.1 Files to Modify

- `adversarialsearch.py` is where you will complete the coding task.

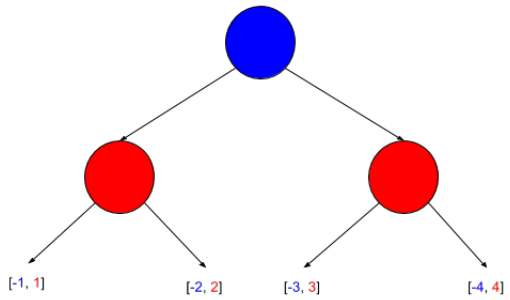
3.2 Necessary Source Code

- `adversarialsearchproblem.py` contains two abstract classes: `AdversarialSearchProblem` and `GameState`.
  `AdversarialSearchProblem` is analogous to `SearchProblem` from the previous assignment; it contains methods that all adversarial search problems have in common. All of the algorithms that you will implement in this assignment take in an `AdversarialSearchProblem`. The `GameState` class is also abstract. It represents a game state. The only requirement of a game state that is imposed by the `GameState` class is that a game state

have some method of getting the player who will move next, and implement that method as the `player_to_move` method. The subclasses of `AdversarialSearchProblem` are in `gamedag.py` and `tttproblem.py`. See them for specific examples of how this works.

3.3 Optional Code for Testing

- `gamedag.py` implements a generalization of a game tree—a game DAG (directed, acyclic graph)—as an `AdversarialSearchProblem`. This helps you create simple, abstract adversarial search problems to make testing easier. Below is a visualization of the DAG created in the `exampleDAG` function in `adversarialsearch`. Running each of your functions on a DAG of this size should take no more than 1 second.



3.4 Optional Code for Fun

We have implemented TicTacToe as an implementation of `AdversarialSearchProblem` for you. It may be useful for sanity-checking your algorithms and understand ASPs, but it is mainly meant for fun, so that you can see the algorithms come to life! See the code file for details.

- `tttproblem.py` includes the Tic-Tac-Toe game.
- `gamerunner.py` allows you to run and visualize games using your algorithms. It's general enough that if you want to create a new game as an ASP, you can use `gamerunner.py` to run it.

4 Help

You can find pseudocode for the minimax algorithm on page 166 of the textbook. You can find pseudocode for the alpha-beta pruning algorithm on page 170 of the textbook. Note that you will need to modify the pseudocode significantly to work with `AdversarialSearchProblem`s, but this should give you an idea about how the algorithms work.

Additionally, we have found this resource helpful for understanding alpha-beta pruning.

5 Grading

We will give you your score based on the rubric in `rubric.txt`.
For both alpha_beta and alpha_beta_cutoff, you will earn points for "expands game states in a correct way." To evaluate this, the autograder will look only at the sequence of (GameState, action) pairs on which `transition(.)` is called. What we're looking for here is that, in whatever order you visit the game states, you prune all the branches that you should when visiting in that order.
Avoid superfluous calls to `transition(.)` they will throw off the auto-grader.

6 Install and Handin Instructions

To install, run `cs1410_install Adversarial_Search` from within your cs1410 directory.

To hand in, run `cs1410_handin Adversarial_Search` from within your cs1410 directory.

In addition, please turn in the written portion of the assignment via Gradescope.