CSC148, Assignment #3
due April 3rd, 2015, 10 p.m.

deadline to declare your assignment team: March 23rd, 10 p.m.

overview

In this assignment you will choose one of two options to continue studying computer game strategies.

option A: You will build on your solution to minimax in order to implement three optimizations:

> **memoization:** Avoid redundancy by storing the value returned by recursive calls.
>
> **pruning:** Avoid investigating moves that you can tell will not change the result.
>
> **myopia:** Limit how far the algorithm looks ahead.

option B: You will investigate the space of game sequences in order to compute metrics such as:

> **size:** How many game states, and final game states, are there?
>
> **redundancy:** What portion of game states are repetitive?

Important!

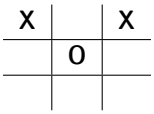
You must choose to implement either **option A**, or **option B**, but not both. Only one of these options will be graded (see "submitting your work" below).

option A

Minimax performs its job perfectly, but often slowly. When you consider that this algorithm must follow every possible sequence of moves to some end state (win, lose, or tie), you'll agree that this can make the computation exponentially large. This explains why, for example, it performs acceptably on small problems, for example subtract square with small starting values, but takes unacceptably long for larger values such as a 4×4 game of tippy or subtract square with a starting value in the 1000s. Some games, such as chess or go, have such an enormous number of possible sequences of moves that minimax on its own is completely impractical.

memoization

You will have probably noticed that any trace of the possible positions (game states) that can be generated from a given position has a lot of redundancy. For example, this fairly early position in tippy can be generated in two ways:



In one instance, an **X** could have been placed in the top left corner, then an **O** in the centre, and then an **X** in the top right. In another instance, **X** could have been placed in the top right corner, then an **O** in the centre, and then an **X** in the top left corner.

The redundancy is a problem for the minimax algorithm. In our simple implementation to date, we encounter these two equivalent positions, and then calculate the value of each possible move from these positions, and so on, **twice**! This extra computation becomes quite noticeable when you consider that some positions may be encountered by minimax much more than twice.

There is a simple solution, though. The first time you encounter a game position and calculate its value, you can store that result in a dictionary. If you subsequently encounter the same position, rather than re-calculating the value, you simply return the value stored in the dictionary. This approach has a big impact on performance.

There are a few technical issues to be solved. Where should the dictionary that is storing the returned values (and perhaps the corresponding moves) be stored? One choice is to store it in the **StrategyMinimaxMemoize** instance that also contains the **suggest_move** method. Another choice is to define it as a local variable in **suggest_move**, and then define a recursive helper that takes the dictionary as a parameter. Either of these choices is preferable to a global dictionary, which is vulnerable to being accessed by other code.

Another technical issue is what key to use for the dictionary. Since **GameState**s are mutable, they aren't allowed as dictionary keys. Think about some attributes of a **GameState** that characterize it very well and are immutable — perhaps the value returned by **__repr__** or **__str__**.

You will know you've succeeded in memoization if certain executions of your minimax algorithm that weren't feasible become feasible. For example, it may have been the case that the minimax version of **suggest_move(99)** exceeded your patience in providing a move, but after memoization it promptly suggests subtracting 4.

You will declare class **StrategyMinimaxMemoize** in file **strategy_minimax_memoize.py**. You should **StrategyMinimaxMemoize** a direct subclass of **Strategy**. You will need to implement method **suggest_move** to incorporate the memoization technique.

pruning

Minimax is presented with a huge tree of game state (position) sequences. However, some careful consideration shows that, in many situations, minimax may ignore huge portions of the tree, since the position sequences in those portions won't change the outcome of the game. Here's why.

Suppose our hero, minimax, is examining a position where p2 has the next move with 31 moves to choose from. After examining 5 of those moves, minimax sees one that leads to a guaranteed tie (a score of 0.0 in our scheme). So, from the remaining 26 moves, minimax will never choose one that guarantees only a loss (a score of -1.0), since minimax always takes the max score. It sure would be nice to cut short long computations of moves that lead to something less than a tie for p2. Here's how to do that.

Minimax evaluates each of the remaining 26 moves available to p2 by considering how good p1's responses are. And it rates p1's responses by considering p2's reponses to them, and then p1's responses to those, and so on. Somewhere in the tree of moves and counter-moves, suppose minimax sees a move for p1 that guarantees a tie, or 0.0. Now minimax knows that the maximum score it will deliver for p1 in that recursive call will be **at least** 0.0. On the other hand, minimax knows (from the previous paragraph) that it can achieve a score of **at least** 0.0 for p2. Now any score higher than 0.0 for p1 represents a score of less than -1×0.0 for p2, so minimax knows that there's no need to continue to examine the remaining moves for p1. Remember, the result of minimax's deliberation will be a score for one of the 26 moves p2 is considering, and it will ignore any result lower than 0.0. This realization allows p1 to abandon further computation, since it wouldn't change the outcome.

This approach can be generalized. When minimax begins its work, it knows it can guarantee both p1 and p2 at least -1.0 — they cannot do worse than lose. As minimax evaluates the moves leading out from any position for a player, it tries to maximize the score for each position those moves lead to. However, minimax may stop this search for the maximum when it meets or exceeds -1 times the score we already know is guaranteed to the opponent. Optimize minimax by keeping track of the score already guaranteed to each opponent, and abandoning further search whenever the score guaranteed to p1 is ≥ -1 times the score guaranteed to p2.

You will declare class **StrategyMinimaxPrune** in file **strategy_minimax_prune.py**. You should make **StrategyMinimaxPrune** a direct subclass of **Strategy**. You will implement method **suggest_move** to incorporate the pruning technique.

myopia

Perhaps a compromise is in order. Have miminax look ahead to game end states, provided these occur in less than some threshold, say n, moves. If it looks ahead n moves and the game has not ended, then it should use its best guess — **rough_outcome** — to provide a score for that game position. This technique will not be as accurate as other optimizations of minimax, but for sufficiently small n, it will perform in a reasonable amount of time.

You will declare **StrategyMinimaxMyopic** in the file **strategy_minimax_myopic.py**. You should make this a subclass either of **Strategy**. You will implement **suggest_move** to incorporate the myopic technique.

what to hand in

Navigate to A3 on MarkUs, and submit the following files:

strategy_minimax_memoize.py This contains your implementation of class **StrategyMinimaxMemoize** along with method **suggest_move**. These are worth 25% of the credit for A3.

strategy_minimax_prune.py This contains your implementation of class **StrategyMinimaxPrune** along with the method **suggest_move** with the pruning optimization. These are worth 30% of the credit for A3.

strategy_minimax_myopic.py This contains your implementation of class **StrategyMinimaxMyopic** along with the method **suggest_move** with the myopic optimization. These are worth 25% of the credit for A3.

game_view.py Modify this file so that, when it is evaluated, the user may choose one of the three new strategies, in addition to the strategies from A2, when playing either **subtract square** or **tippy**. This is worth 10% of the credit for A3.

Use pep8.py on your submissions (worth 5%), and be sure to follow style guidelines (worth 5%).

Although you are welcome to experiment with combining some of the optimization techniques, such combinations are not part of the assignment.

option B

your job

Download module game_state_tree.py from the course website. Read the class **GameStateNode**, and understand how it's **grow** method is responsible for creating a tree of game states. Your first job is to implement

grow. Next, look at the functions for analyzing game state trees, which are defined outside the class **GameStateNode**, but in the module **game_state_tree**. Read and understand the docstring specification for each function, and then implement it. You may implement the functions in any order.

Notice that the functions do not depend on any of the code you have written for assignment 1 or 2. They depend only on classes **GameState** and, in one case, class **SubtractSquareState**, which we have provided. Do not change either of those classes when you test your code.

Many of the doctest examples are based on the example tree shown in the Assignment 2 handout for the game subtract a square. Make sure that you understand that tree fully.

In all of the doctest examples, the **GameStateNode**s used for testing contain specifically **SubtractSquareState**s. However, unless a function specifies that it only works for the game subtract a square, be prepared that we may test it on another game.

sometimes you'll need a helper

All of the functions can be written recursively, and most or all of them would be extremely difficult and awkward to write without recursion.

For some of the functions, the type contract is exactly what you need in order to conduct the recursion. In other cases, the type contract is inadequate: either the recursive call needs additional information (that isn't passed through the existing parameters) in order to do its job, or it needs to tell back additional information (that is not part of the existing return type) so that the caller can do its job. You saw this in Assignment 2, where **suggest_move** only returned a move, but in the recursion it was necessary to report back also a score. In such situations, create a helper function that has the appropriate type contract, and call it to do most of the work.

Of course, you are also welcome to use helper functions just to carve off some of the work.

If you create a helper function, prefix its name with an underscore in order to indicate that it is not intended for use by client code.

using sets of game states

For some functions, you will need to keep track of the game states that you have already seen, and to check whether a particular game state is among them. In such circumstances, you must maintain a set of game states. An alternative would be to maintain a list of game states, but checking whether an item is in a set is faster than checking whether an item is in a list. This difference becomes crucial when there are many game states.

Here is a small example of how sets work in Python:

```
>>> s = set()
>>> s.add(3)
>>> s.add(55)
>>> 3 in s
True
>>> 5 in s
False
```

In order to make checking set membership fast, sets rely on having elements that are immutable. As a result, you cannot directly add a **GameState** object to a set; you must convert it to something immutable first. This can easily be done by calling either **__str__** or **__repr__**.

what to hand in

Navigate to A2 on MarkUs, and submit your completed version of file **game_state_tree.py**.

Use **pep8** on your submissions (worth 5%), and be sure to follow style guidelines (worth 5%).

Declaring your assignment team

You may do this assignment alone or in a team of either 2 or 3 students. Your partner(s) may be from any section of the course on St George campus. You must declare your team (even if you are working solo) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working solo, say so. If you are working with other(s):

First: one of you needs to "invite" the other(s) to be partners, providing MarkUs with their cdf user name(s).

Second: the invited student(s) must accept the invitation.

Important: there must be **only** one inviter, and other group members accept **after** being invited, if you want MarkUs to set up your group properly.

To accept an invitation, find "Group Information" on the appropriate Assignment page, find the invitation listed there, and click on "Join".

Submitting your work

Submit all your code on MarkUs by 10 p.m. on April 3rd. Click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". You can submit a new version of a file later (before the deadline, of course); look in the "Replace" column.

Only one team member should submit the assignment. Because you declared your team, all of you will get credit for the work.

If you submit a non-empty game_state_tree.py file, then you will be graded only on option B. If you submit **game_state_tree.py**, and then change your mind, you should re-submit an empty file with the same name, **game_state_tree.py** up until the submission deadline.

There is a 5% deduction for each hour late any part of your submission is. A good strategy is to begin submitting the parts of your assignment that work **early**, and keep submitting improved versions as the deadline approaches. Only the last version of each file is graded.