Computer Science 384
St. George Campus

Version W24.1
University of Toronto

Assignment 3: Games

Silent Policy: A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.

Submission Instructions: You must submit your assignment electronically through MarkUs. You will submit the following files:

- `agent.py`

Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time. Only your latest submission will be considered. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission. Ensure that:

- your code runs on teach.cs using `python3` (version 3.10) using only standard imports. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- you do not add any non-standard imports from within the `Python` file you submit (the imports that are already in the template files must remain). Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- you do not change the supplied starter code. Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

Clarifications: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on Quercus. You are responsible for monitoring the clarification page.

Questions: Questions about the assignment should be posted to Piazza.

Introduction

During one of Amazon's company-wide meetings, you learn that their current warehouse is insufficient for its operations. Thus, Amazon executives are looking to build a new one. They found a plot of land that is N hectometers by N hectometers[1] divided into N^2 sub-plots, each being 1 hectare[2]. The only problem is that their main competitor, Nozama, also wants to use that land. To build the new warehouse, Amazon needs to own more land area than its competitor. The landlord gives both companies 2 hectares of land as shown in Figure 1.

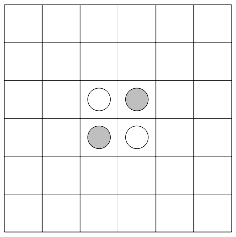
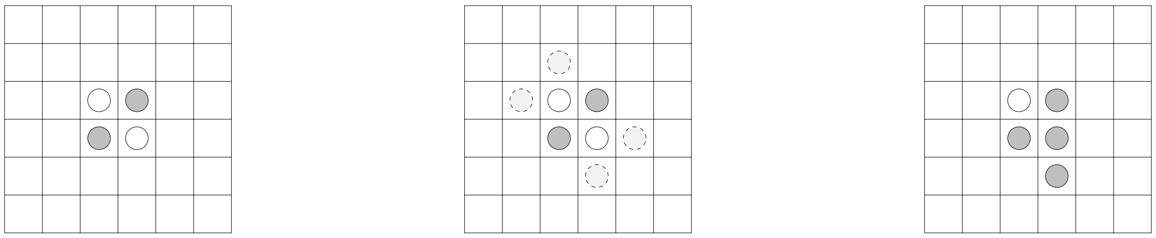


Figure 1: Example of a 36 hectare (6 hectometers by 6 hectometers) plot of land; ● represent one company's land, and ○ represent their competitor's land.

Zoning laws dictate that if a company owns two sub-plots that are in a straight line along the horizontal, vertical or diagonal axes, the company must also own the sub-plots in between. Formally, if ij indexes the sub-plot in the i^{th} row and j^{th} column, and $o[ij]$ is the company that owns that sub-plot, we require the following conditions to be satisfied:

- $\forall ij: \quad o[ij] = o[ik] = O,\ k > j \quad \text{and} \quad o[il] = O',\ j < l < k \quad \Rightarrow \quad O' = O$
- $\forall ij: \quad o[ij] = o[kj] = O,\ k > i \quad \text{and} \quad o[lj] = O',\ i < l < k \quad \Rightarrow \quad O' = O$
- $\forall ij: \quad o[ij] = o[(i+D)(j+D)] = O,\ D > 0 \quad \text{and} \quad o[(i+d)(j+d)],\ 0 < d < D \quad \Rightarrow \quad O' = O$
- $\forall ij: \quad o[ij] = o[(i+D)(j-D)] = O,\ D > 0 \quad \text{and} \quad o[(i+d)(j-d)],\ 0 < d < D \quad \Rightarrow \quad O' = O$

A company can only claim a sub-plot if doing so would violate one of the above conditions. We say that the claim must *bracket* the opponent's the sub-plots along one of the axes. The landlord will then adjust their competitor's ownership to ensure the laws are not violated (see Figure 2).



(a) initial land ownership of the companies

(b) ◌ represent where the first company may claim land

(c) land ownership after the first company claims the (4,2) sub-plot.

Figure 2: Example of how the land ownership changes as each company claims sub-plots of land.

[1] A hectometer is a unit of length equal to 100 meters.

[2] A hectare is a unit of area equal to 1 square-hectometer

The landlord enacts a bidding process. The companies take turns claiming the sub-plots. Based on the rules mentioned, at each company's turn, the company may claim an unoccupied sub-plot if it brackets one or more competitor subplots in along at least one axis (vertical, horizontal, or diagonal). Once the sub-plot is claimed, the ownership of all competitor sub-plots that are bracketed, along any axis, are flipped . If a company cannot legally claim any sub-plot on its turn, the land-lord will stop the process and split the land based on what has already been claimed (i.e., no additional land can be claimed by either company). Randy, the CEO of Nozama, will be claiming the sub-plots for Nozama. Your manager says that Amazon cannot take any chances, and thus, asks you to develop an AI agent that can compete against Randy. For fairness, the land-lord will randomly select which company gets first pick. We will refer to this company as the "first company", and their competitor as the "second company".

The landlord is very busy. If you do not provide them with a decision within 10 seconds, they will automatically disqualify you from the bidding process.

The Starter Code

The code for this assignment consists of several `Python` files, some of which you will need to read and understand in order to complete the assignment. You have been provided:

1. `othello_gui.py`; provides a simple graphical user interface to visualize the bidding process
2. `othello_game.py`; represents the behaviour of the landlord who manages the entire process
3. `othello_shared.py`; contains useful functions that can be used by the AI agents and/or the manager
4. `randy_ai.py`; specifies the behaviour of Randy, the CEO of Nozama
5. `agent.py`; should specify the behaviour of your AI agent

The only file you will submit is `agent.py`. We consider the other files to be starter code, and we will test your code using the original versions of those files. Therefore, when testing your code, you should not modify the starter code.

Each state consists of:

1. the current land ownership of the companies, `board`
2. whose turn is it to claim land, `color` (or `current_player`)

The former is represented as a tuple of tuples of integers, where `0` denotes an empty sub-plot, `1` denotes that the sub-plot is owned by the first company (visualized with dark markers on the bidding GUI), and `2` denotes that the sub-plot is owned by the second company (visualized with the light markers on the bidding GUI). For example, the land ownership in Figure 2 (c) is represented as

```
((0,0,0,0,0,0),
 (0,0,0,0,0,0),
 (0,0,2,1,0,0),
 (0,0,1,1,0,0),          (1)
 (0,0,0,1,0,0),
 (0,0,0,0,0,0))
```

The latter is an integer where `1` denotes the first company's turn, and `2` denotes their competitor's turn. For example, if `current_player=1` and Amazon is the first company, then it is Amazon's turn to claim land. In our implementation, we consider the bidding process as a game, where the companies are the *player*s, the land is the playing *board*, and the number of sub-plots claimed by each company is their current *score*.

To run the code, you can either run two separate agents against each other using:

```
python3 othello_gui.py -d N -a <agent1> -b <agent2>
```

or you can run one agent against an interactive user:

```
python3 othello_gui.py -d N -a <agent1>
```

In both cases, `N` denotes the board size. Optionally, you can also specify three flags:

- `-l <limit>`, where `<limit>` is an integer representing the depth limit (see §Your Tasks - Depth Limiting)
- `-c`, which enables caching (see §Your Tasks - Caching)
- `-o` which enables optimal node ordering for α/β-pruning (see §Your Tasks - Optimizing the Node Order for α/β-Pruning)

Important Technical Disclaimer

The landlord is represented by the `OthelloGameManager` class found within the `othello_game.py` file. This class communicates with the agents via `stdout` and `stdin`. As a result, you cannot print to `stdout` for debugging purposes. Doing so may cause unpredictable behaviour. You may print to `stderr` instead using the `eprint` function within `agent.py`.

The agents and the manger will all run in different processes. In particular, `othello_game.py` will spawn a child process for each agent. The `OthelloGameManager` reads from each agent's `stdout` pipe and writes to the their respective `stdin` pipes. The following protocol is used:

1. each agent sends a string to identify itself (e.g., `randy.py` sends the string `'Randy'`)
2. the `OthelloGameManager` will reply with either `1` or `2` telling each agent whether it is the first or second company
3. when it is an agent's turn to claim land, the `Manager` will send its `stdin` the current total land area claimed by the companies (e.g., `'SCORE 2 7'`, which means that the first company owns 2 hectares, and their competitor owns 7) as well as the current actual land allocation represented as a string
4. the agent must use this information to provide the manager with its intent to claim another sub-plot within 10 seconds by writing a string `'r c'` to its `stdout` representing the row and column of the sub-plot it intends to claim
5. when the landlord wishes to end the process, or when the agent has no legal claims, the `OthelloGameManager` will write the final land area claimed by the agents (e.g., `'FINAL 12 14'`)

Your Tasks

Naive Minimax Implementation

You will do this in two parts:

1. Implement the function, `compute_utility(board, color)`, that computes the utility of a *terminal state* for the specified company (identified with its *color*), which we define to be the number of hectares they own minus the number of hectares their competitor owns. You may find the following function useful:

    - `get_score(board)`, which returns a tuple, `(n1, n2)`, representing the number of hectares owned by the first and second companies, respectively.

2. Implement the function, `select_move_minimax(board, color, limit, caching=0)` that returns a tuple, `(col, row)` which represents the sub-plot that your agent should claim under the minimax strategy. Implement minimax recursively by writing two helper functions:

    (a) `minimax_max_node(board, color, limit, caching=0)`, which computes the utility assuming it is your turn to claim land,

    (b) `mm_min_node(board, color, limit, caching=0)`, which computes the utility assuming it is your opponent's turn to claim land.

    Ignore the `limit`, and `caching` parameters for now. You may find the following functions useful:

    - `get_possible_moves(board, color)` which returns a list of legal claims for the specified company; each claim is a tuple, `(r,c)`, denoting the row and column of the sub-plot.

Once you are done, you can run your MiniMax algorithm via the command line using the flag `-m`. If you issue the command

```
$python3 othello_gui.py -d 4 -a agent.py -m,
```

you can play against your agent on a 4×4 board using MinMax algorithm.

α/β-Pruning

You will now add α/β-pruning to speed up your agent's decision making. Implement the function, `select_move_alphabeta(board, color, limit, caching=0, ordering=0)` that returns a tuple, `(col, row)` which represents the sub-plot that your agent should claim using α/β pruning. Implement Minimax(w/ α/β-pruning) recursively by writing two helper functions:

1. `alphabeta_max_node(board, color, alpha, beta, limit, caching=0, ordering=0)`, which computes the utility assuming it is your turn to claim land,

2. `alphabeta_min_node(board, color, alpha, beta, limit, caching=0, ordering=0)`, which computes the utility assuming it is your opponent's turn to claim land.

Ignore the `limit`, `caching`, and `ordering` parameters for now. Once you are done, you can run your α/β-pruning algorithm via the command line. If you issue the command

```
$python3 othello_gui.py -d 4 -a agent.py,
```

you can play against your agent on a 4×4 board using the α/β-Pruning algorithm.

Optimizing the Node Order for α/β-Pruning

Recall that α/β-pruning is most useful if the nodes are explored in a particular order. Update your code so that it explores node in this optimal order when the `-o` flag is set on the command line. The starter code will call `select_move_alphabeta(board, color, limit, caching=0, ordering=0)` with `ordering=1` when the flag is set.

Depth-Limiting

To speed up your agent further, you can introduce a depth limit. The starter code will call `select_move_minimax` and `select_move_alphabeta` with `limit` set to the appropriate value when the `-l` flag is set. Recursively pass and decrease `limit` into the appropriate helper functions. When `limit=0`, use a heuristic function to estimate the utility of the resulting state. For now, you can simply use `compute_utility` as the heuristic function.

Caching States

We can try to speed up the agent even more by caching states that we have seen before. The starter code will call `select_move_minimax` or `select_move_alphabeta` with `caching=1` when the `-c` flag is set. Update your code as follows:

1. Create a dictionary, `cached_states` as a global variable within `agent.py`.
2. Modify the `select_move_minimax` function to check whether the specified state, `(board, color)` has already been visited and branch as follows:
    - if `(board, color)` has already been visited, get its utility from `cached_states`
    - otherwise, add `(board, color)` to `cached_states`
3. Modify the `select_move_alphabeta` function to check whether the specified state, `(board, color, alpha, beta)` has already been visited and branch as follows:
    - if `(board, color, alpha, beta)` has already been visited, get its utility from `cached_states`
    - otherwise, add `(board, color, alpha, beta)` to `cached_states`

Adding a Custom Heuristic

Implement a custom heuristic `compute_heuristic(board, color)` for the purposes of depth limiting. You can replace the appropriate calls to `compute_utility` for testing purposes, but do NOT use it in the code you submit.