CS 591 Programming Project 2

Breakthrough Game

Grade: 100 marks in total (Program: 70%, Team Review: 30%)

Type: Group work (Team size cannot exceed 6)

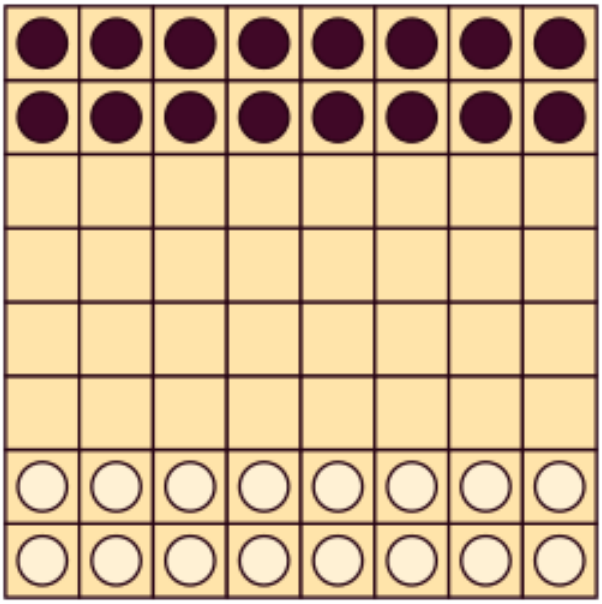
Programming language: Python (preferred), Java, C/C++, or any other advanced programming languages

Hints: Textbook online repository files (games4e.py, games4e.ipynb**)** in aima-python, or similar files in other folders, such as aima-java. A Reference project code package (BreakthroughGame-master.zip) is provided for you to partially utilize to integrate into your own project. But a **less than 60% overlap** should be followed. You may also choose not looking at this reference if you can manage to do the project by your own.

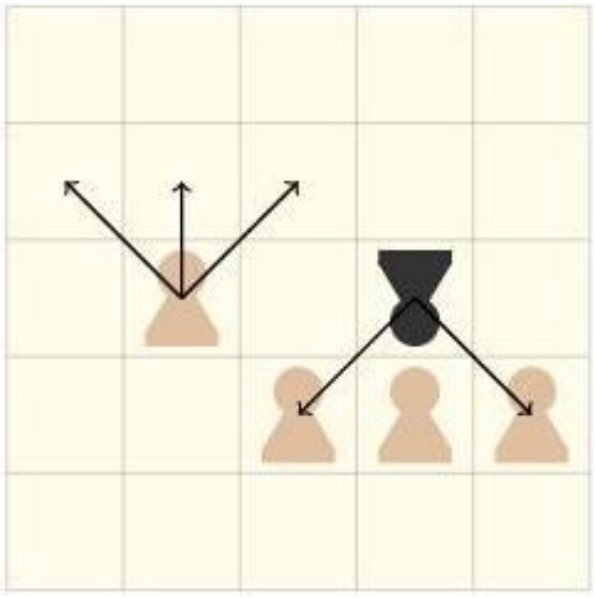
Project Overview: The goal of this assignment is to implement an agent to play a simple 2-player zero-sum game called Breakthrough (please click it to have an overview of the game).

Rules of the game

- The 8x8 board is initially set up as shown below. Each player has 16 workers in their team.



- Players play in alternating turns and can only move one piece (of their own workers) at a time.
- In each turn, a worker can only move *one square* in the forward or diagonally-forward directions, as shown below. Moreover, a worker can 'capture' workers of the enemy team if they are placed diagonally forward from it, as shown in the illustration below. Note that if enemy workers are directly in front of the player, then a capture isn't possible.



- The game finishes when (a) a worker reaches the enemy team's home base (the last row); or (b) when all workers of the enemy team are captured.

For more information, check out Breakthrough's Wikipedia page.

1.1 Minimax and alpha-beta agents

Your task is to implement agents to play the above game, one using **minimax search** and one using **alpha-beta search** as well as two evaluation functions - one which is more **offensive** (i.e., more focused on moving forward and capturing enemy pieces), while the other which is more **defensive** (i.e., more focused on preventing the enemy from moving into your territory or capturing your pieces). The evaluation functions are used to return a value for a position when the depth limit of the search is reached. Try to determine the maximum depth to which it is feasible for you to do the search (for alpha-beta pruning, this depth should be larger than for minimax). The worst-case number of leaf nodes for a tree with a depth of three in this game is roughly 110,592, but in practice is usually between 25,000 - 35,000. Thus, you should at least be able to do minimax search to a depth of three.

We provide the following two dummy heuristics:

- **Defensive Heuristic 1**: The more pieces you have remaining, the higher your value is. The value will be computed according to the formula `2*(number_of_own_pieces_remaining) + random()`.

- **Offensive Heuristic 1**: The more pieces your opponent has remaining, the lower your value is. The value will be computed according to the formula `2*(30 - number_of_opponent_pieces_remaining) + random()`.

  NOTE: `random()` generates a random float uniformly in the semi-open range [0.0, 1.0) as per Python's random(). This noise is added to provide an easy way to break ties.

Your task for this part is:

- Implement minimax search for a search tree depth of 3.
- Implement alpha-beta search for a search tree of depth more than that of minimax.
- Implement **Defensive Heuristic 1** and **Offensive Heuristic 1**.
- Design and implement an **Offensive Heuristic 2** with the idea of beating **Defensive Heuristic 1**.

- Design and implement a **Defensive Heuristic 2** with the idea of beating **Offensive Heuristic 1**.

Then play the following matchups:

1. Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)
2. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
3. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
4. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
5. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
6. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

For each of the above matchups, report the following:

A. The final state of the board (who owns each square) and the winning player.
B. The total number of game tree nodes expanded by each player in the course of the game.
C. The average number of nodes expanded per move and the average amount of time to make a move.
D. The number of opponent workers captured by each player, as well as the total number of moves required till the win.

If you have time, try to run each matchup multiple times to determine whether the noise in the evaluation function has any effect on the final outcome.

Finally, you should summarize any general trends or conclusions that you have observed. How does the type of evaluation function (offensive vs. defensive) affect the outcome? How do different combinations of evaluation functions do against each other?

Tips

- Pseudocode for alpha-beta pruning is given in Figure 5.7, p. 154, in the 4th edition as well as in the Appendix 1.
- For alpha-beta pruning, try to come up with a move ordering to increase the amount of pruning. Discuss any interesting choices in your report.
- For offensive vs. defensive evaluation functions, you may want to define the evaluation function as having two components: your score and your opponent's score. You would then weight these two components differently or combine them in different ways to get a more offensive or more defensive

strategy. You can also read the Breakthrough Wikipedia page for possible features of a position to look for and prioritize for offensive vs. defensive play. Better yet, you should try playing the game yourself to get a better sense of what features or formations can occur.

1.2 Extended rules (for Bonus Points only)

Modify your implementation and evaluation functions to support the rule changes below and run several matchups for alpha-beta agents only with the maximum depth you can manage. You can choose any combination of offensive or defensive agents. Report outcomes of 2-4 representative matchups (or averages over several matchups of the same type), and summarize any interesting trends and differences from Part 1.1.

1. **3 Workers to Base**: To win, 3 workers of a player's team must reach the opponent's base (as opposed to 1 in the original game). A player automatically loses when less than three of their pieces are left on the board. Therefore, the two ways to win the game would be to (a) move 3 pieces to the enemy's home base; (b) capture n-2 of the enemy's pieces, where n is the total number of players the enemy has at the beginning of the game.
2. **Long Rectangular Board**: For this variation, stick to the original, 1-worker-to-enemy-base win criterion. Instead, change the board shape to an oblong rectangle of dimensions 5x10.

1.3 Other Bonus Points

- Design an interface for the game that would allow you to play against the computer. How well do you do if compared to the AI? Does it depend on the depth of search, evaluation function, etc.?
- Implement any advanced techniques from class lectures or your own reading to try to improve efficiency or quality of gameplay.
- Implement a 1-depth greedy heuristic bot, and describe the differences between the gameplays of the greedy bot and the minimax bot.

Report Checklist

Your report should briefly describe your implemented solution and fully answer the questions for every part of the assignment. Your description should focus on the most "interesting" aspects of your solution, i.e., any non-obvious implementation

choices and parameter settings, and what you have found to be especially important for getting good performance. Feel free to include pseudocode or figures if they are needed to clarify your approach. Your report should be self-contained and it should (ideally) make it possible for us to understand your solution without having to run your source code. For full credit, your report should include the following.

1. For everybody: Describe your implementation, especially your choice of Offensive and Defensive Heuristic 2. For matchups 1-6, report items A-D. Identify and discuss any general trends.
2. For Extended rules (For bonus points): For two types of extended rules in 1.2, describe your modified implementation (and especially evaluation functions) and report items A-D for matchups 2 and 3. Discuss any interesting trends that you observe.

Extra credit:

- We reserve the right to give **bonus points** for any advanced exploration or especially challenging or creative solutions that you implement. **If you submit any work for bonus points, be sure it is clearly indicated in your report.**

WARNING: You will not get credit for any solutions that you have obtained, but not included in your report! For example, if your code prints out path cost and number of nodes expanded on each input, but you do not put down the actual numbers in your report, or if you include pictures/files of your output solutions in the zip file but not in your PDF. The only exception is animated paths (videos or animated gifs).

Submission Instructions

You need to submit on **Canvas** by the deadline. Each submission must consist of the following two attachments:

1. A **report** in **PDF format**. As for Assignment 2, the report should briefly describe your implemented solution and fully answer all the questions posed above. **Remember: you will not get credit for any solutions you have obtained, but not included in the report.**

   Project report template (for reference only): a project report may include at least the following items: **abstract**, **problem description**, **algorithm**

(pseudocode), **implementation details**, **running results and analysis**, **conclusions on what you have learned in the project**, and **references** (if applicable, also explicitly cite them in your project report).

2. Your **source code** should be well commented, and it should be easy to see the correspondence between what's in the code and what's in the report. Please include **executables** or various supporting files (e.g., utility libraries) whose content is relevant to the assignment. Please provide a **Readme.txt** file to list the steps to compile and run your program, including the software environment. If we find it necessary to run your code in order to evaluate your solution, we will get in touch with you.

Please **zip** both the report and source code into one package and name it based on the following rule:

"**Last_Name_First_Name_PA2.zip**"

Multiple attempts will be allowed but only your last submission will be graded. No email submission is accepted.
I reserve the right to take off points for not following directions.

Grading Policy:

1. **Functionality (70 points):**

1.1 Defensive Heuristic 1: 10 points
1.2 Offensive Heuristic 1: 10 points
1.3 Defensive Heuristic 2: 10 points
1.4 Offensive Heuristic 2: 10 points
1.5 Six matchups: (5 points for each)
- **1.5.1** Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)
- **1.5.2** Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
- **1.5.3** Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
- **1.5.4** Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
- **1.5.5** Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
- **1.5.6** Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

2. **GUI (extra credit):**
2.1 10 extra credit for those who exceed expectation

3. **Report & Analysis (30 points):**
3.1 Report: 20 points

3.2 Analysis: 10 points

In total: 100 points.

4. Additional Bonus (35 points):
4.1 3 Workers to Base: 10 points
4.2 Long Rectangular Board: 10 points
4.3 Other Bonus Points: 15 points

- Design an interface for the game that would allow you to play against the computer. How well do you do compared to the AI? Does it depend on the depth of search, evaluation function, etc.? (5 points)
- Implement any advanced techniques from class lectures or your own reading to try to improve efficiency or quality of gameplay. (5 points)
- Implement a 1-depth greedy heuristic bot, and describe the differences between the gameplays of the greedy bot and the minimax bot. (5 points)

Late policy:

For every day that your assignment is late, your score gets multiplied by 0.75. The penalty gets saturated after four days, that is, you can still get up to about 32% of the original points by turning in the assignment at all. If you have a compelling reason for not being able to submit the assignment on time and would like to make a special arrangement, you must send me email **at least four days before the due date** (any genuine emergency situations will be handled on an individual basis).

Be sure to also refer to course policies on academic integrity, etc in our syllabus.

Appendix 1: the alpha-beta search algorithm

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Figure 5.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 5.3, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.

Fig 5.3 (see next page):

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 5.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.