CSCI 561: Foundations of Artificial Intelligence

Instructor: Prof. Laurent Itti

Homework #2: Adversarial Search

Due on Oct 20 at 11:59pm, 2014

Introduction

In this project, you will write a program to determine the next move for a player in the Reversi game using the Greedy, Minimax, and Alpha-Beta pruning algorithms with positional weight evaluation functions.

The rules of the Reversi game can be found at http://en.wikipedia.org/wiki/Reversi [1] and interactive examples can be found at http://www.samsoft.org.uk/reversi/ [2]. In the Othello version of this game, the game begins with four disks placed in a square in the middle of the grid, two facing light-up, two pieces with the dark side up, with same-colored disks on a diagonal with each other. However, in this assignment, the starting position will be specified in the input file and may be different.

Tasks

In this assignment, you will write a program to determine the next move by implementing the following algorithms:

2.1 Greedy ;

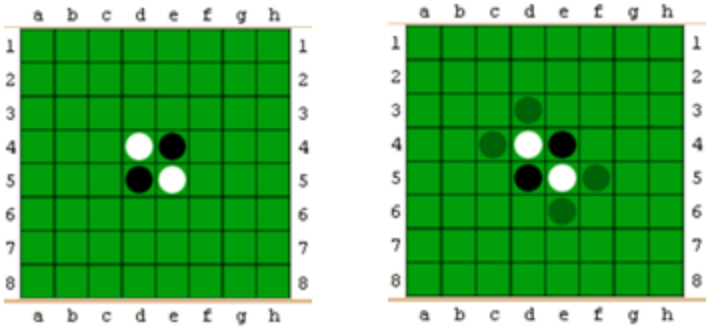
2.2 Minimax ;

2.3 Alpha-Beta ;

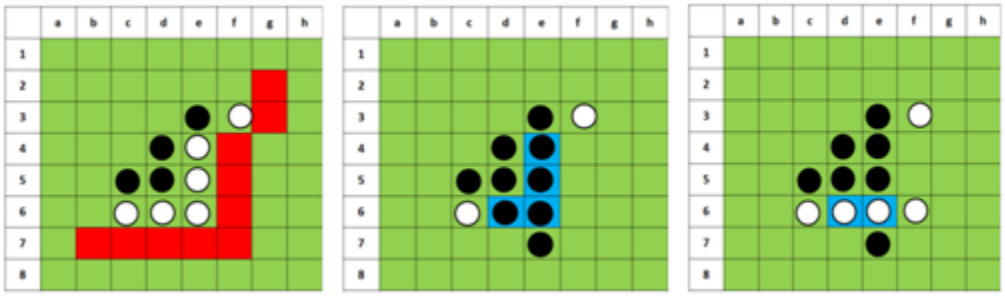
In addition, we will have the competition, which is voluntary and for a bonus prize. You will create an agent that can play this game against another agent. The competition is optional and will not affect your grade in the course.

Legal moves and Flips/Captures

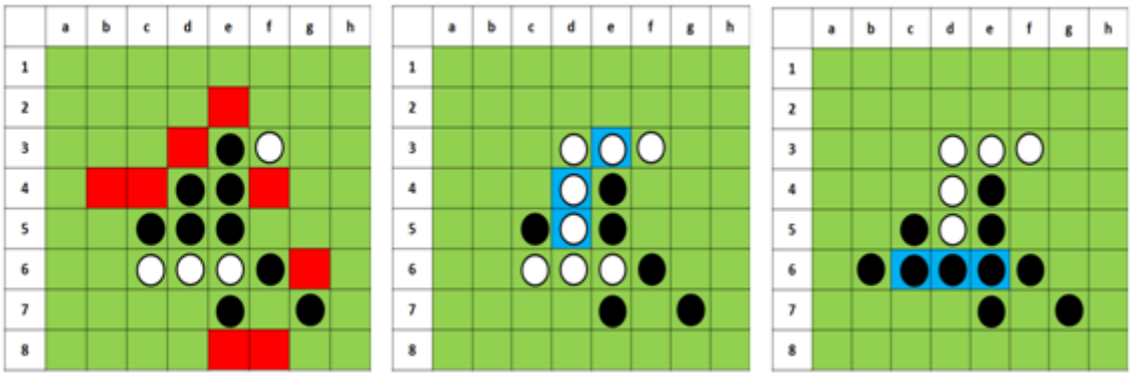
Assume that the current position is as shown in the left image below and that the current turn is Black's. Black must place a piece with the black side up on the board in such a position that there exists at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another black piece, with one or more contiguous light pieces between them. The right image shows all the legal moves available to the Black player (see the translucent circles below that highlight the legal Black moves). If one player cannot make a valid move, play passes back to the other player.



Example of Flips/Captures: In the left image below, the legal Black moves are shown as red cells. After placing the piece at e7, Black turns over (i.e., flips or captures) all White pieces existing on a straight line between the newly added piece and any anchoring Black pieces[1]. All reversed pieces are shown in blue cell in the middle image. In the right image, White can reverse some of those pieces back on his turn if those Black pieces are on a straight line between the new White piece and any anchoring White pieces.



Example of Flips/Captures: In the left image, the available legal White moves are shown as red cells. After placing a new White piece at d3, all captured pieces are shown in blue cells in the center image. In the right image, Black can use his own turn to reverse some pieces back if he chooses so. However, in this example, Black makes a different move, essentially choosing to flip other pieces.

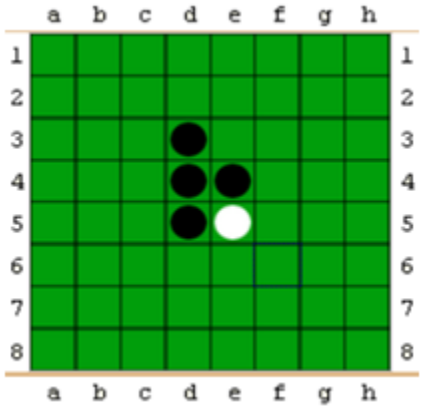


Pass Move and End Game

If one player cannot make a valid move, play **passes** back to the other player. When neither player can move, the game ends. This occurs when the grid has filled up or when neither player can legally place a piece in any of the remaining squares [1].

Evaluation function: positional weights

	a	b	c	d	e	f	g	h
1	99	-8	8	6	6	8	-8	99
2	-8	-24	-4	-3	-3	-4	-24	-8
3	8	-4	7	4	4	7	-4	8
4	6	-3	4	0	0	4	-3	6
5	6	-3	4	0	0	4	-3	6
6	8	-4	7	4	4	7	-4	8
7	-8	-24	-4	-3	-3	-4	-24	-8
8	99	-8	8	6	6	8	-8	99



Each position on the board has different strategic value. For example, the corners have higher strategic values than others. This evaluation function assigns different values for each position. The evaluation function of positional weights can be computed by

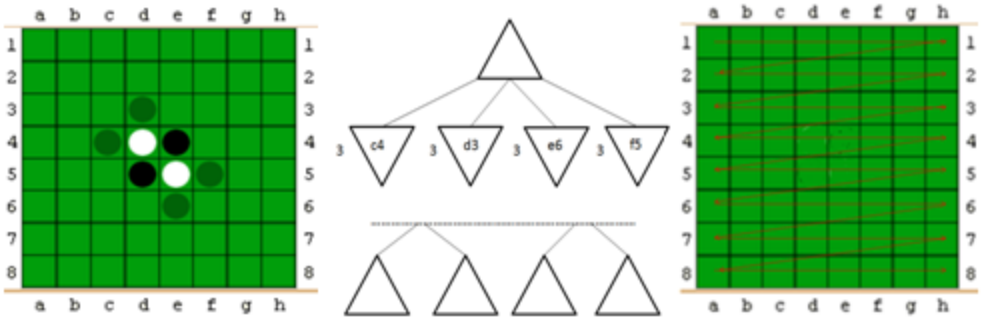
E(s) = Weight_player - Weight_opponent

For example, if the player is black, the board in the picture above has evaluated value of E(s) = (4+0+0+0) - (0) = 4. Note: you can hence assume that your agent is always the "max" player.

Note: The leaf node values are always calculated from this evaluation function. Although this may not be a good estimation for the end game node (and it is a deviation from "official" Reversi rules), you should comply with this rule for simplicity (i.e., you do not need to worry about possible ordering complications between terminal utility values and evaluation values at non-terminal nodes). You can come up with the better evaluation function in the competition task.

Tie Breaking and Expand order

Ties between nodes are broken by selecting the node that is first in positional order on the right figure above. For example, if all legal moves (c4, d3, e6, f5) have the same evaluated values, the program must pick d3 according to the tie breaker rule on the right figure.
Your traverse order must be in the positional order also. For example, your program will traverse on d3, c4, f5, e6 branch in order.



Board size

The board size is fixed. It has 8 rows and 8 columns. The number of cells is 64.

Pseudo Codes

1 Greedy: It is a special case of Minimax. The cut-off depth is always 1. Thus, the algorithm is very simple. You only need to pick the action which has the highest evaluation value.
2 Minimax: AIMA Figure 5.3 (Minimax without cut-off) and section 5.4.2 (Explanation of Cutting off search)
3 Alpha-Beta: AIMA Figure 5.3 (Alpha-Beta without cut-off) and section 5.4.2 (Explanation of Cutting off search)

Compilability

You should take full responsibility to make your code executable. You are required to write Makefile to compile (if necessary) and run your code. In your Makefile, you are required to have 2 targets: **agent** and **run**. The **agent** target is to compile, if needed, otherwise it may do nothing. If "**make agent"** fails, your grade is **0**. The **run** target is to execute your program. If "make run" fails sometimes, you will get a partial score out of 100. We will run N test cases in total. if "**make run**" passes and the output is correct on all N cases, then your score is **100**. Otherwise, if only G tests pass out of N, then the score is 50 - 50*(N-G)/N (i.e., 50*G/N). Please refer to the example Makefile files below:

Example for a C++ agent:

agent: agent.cpp
  g++ agent.cpp -o agent
 	
run: agent
  ./agent

Example for a Java agent:

agent: agent.class
 	
agent.class: agent.java
  javac agent.java
 	
run: agent.class
  java agent

For python or other non-compiled languages:
agent: run: ./agent.py

Input:

You are provided with a file input.txt that describes the current state of the game.

<task#> Greedy = 1, MiniMax = 2, Alpha-beta = 3, Competition =4

<your player: X or O>

<cutting off depth >

<current state as follows:>

***: blank cell**

X: Black player

O: White Player

Example:

```
2
X
2
********
********
********
***OX***
***XO***
********
********
********
```

Output:

The program should output the Greedy algorithm in the format:

<next state>

Example:

```
********
********
***X****
***XX***
***XO***
********
********
********
```

The program should output the Minimax algorithm in the format:

```
<next state>
<traverse log>
```

Example:

```
********
********
***X****
***XX***
***XO***
********
********
********
Node,Depth,Value
root,0,-Infinity
d3,1,Infinity
c3,2,-3
d3,1,-3
e3,2,0
d3,1,-3
c5,2,0
d3,1,-3
root,0,-3
c4,1,Infinity
c3,2,-3
c4,1,-3
e3,2,0
c4,1,-3
c5,2,0
c4,1,-3
root,0,-3
f5,1,Infinity
f4,2,0
f5,1,0
d6,2,0
f5,1,0
f6,2,-3
```

f5,1,-3
root,0,-3
e6,1,Infinity
f4,2,0
e6,1,0
d6,2,0
e6,1,0
f6,2,-3
e6,1,-3
root,0,-3

Note: The Minimax traverse log requires 3 columns. Each column is separated by ",". Three columns are node, depth and value.

"Node": is the node name which refers to the move that is made by the agent. For example, the black player places a piece at the position "d3". The node name is d3 and has depth 1. Then, the white player places a piece at the position "c3" and has depth 2. There are **two special node** names: "root" and "pass". "root" is the name for the root node. "pass" is the name for the special move "pass". Agent can make the pass move only when it cannot make any valid move.

"Depth": is the depth of the node. The root node has depth zero.

"Value": is the value of the node. The value is initialized to **"-Infinity"** for the max node (your agent is always max) and **"Infinity"** for the min (your agent's opponent) node. The value will be updated when its children return the value to the node. The value of leaf nodes is the evaluated value, for example, **c3,2,-3.0**

The algorithm traverses from the root node. The log should show both when:

1. The algorithm traverses down to the node.
2. The value of the node is updated from its children.

For example, the log shows value of the node "d3" when traversing from the root. The log shows the node "d3" again when the node is updated from its children "c3", "e3" and so on. The leaf nodes have no children. Thus, the log shows only once when the algorithm traverses a leaf node and the value is the evaluated value, for example, **c3,2,-3.0**.where -3.0 is the evaluated value.

The program should output the Alpha-Beta algorithm in the format:
<next state>
<traverse log>
Example:
****
****

```
***X****
***XX***
***XO***
********
********
********
Node,Depth,Value,Alpha,Beta
root,0,-Infinity,-Infinity,Infinity
d3,1,Infinity,-Infinity,Infinity
c3,2,-3,-Infinity,Infinity
d3,1,-3,-Infinity,-3
e3,2,0,-Infinity,-3
d3,1,-3,-Infinity,-3
c5,2,0.0,-Infinity,-3
d3,1,-3,-Infinity,-3
root,0,-3,-3,Infinity
c4,1,Infinity,-3,Infinity
c3,2,-3,-3,Infinity
c4,1,-3,-3,-3
root,0,-3,-3,Infinity
f5,1,Infinity,-3,Infinity
f4,2,0,-3,Infinity
f5,1,0,-3,0
d6,2,0,-3,0
f5,1,0,-3,0
f6,2,-3,-3,0
f5,1,-3,-3,-3
root,0,-3,-3,Infinity
e6,1,Infinity,-3,Infinity
f4,2,0,-3,Infinity
e6,1,0,-3,0
d6,2,0,-3,0
e6,1,0,-3,0
f6,2,-3,-3,0
e6,1,-3,-3,-3
root,0,-3,-3,Infinity
```

Note: The Alpha-Beta traverse log requires 5 columns. Each column is separated by ",". Five columns are node, depth, alpha, beta. The description is same with the Minimax log. However, you need to show the alpha and beta values in the Alpha-Beta traverse log.

Note: the examples above are for file format specification only and are not guaranteed to be correct. **Output filename is fixed to output.txt.** The grader will examine the file output.txt for grading. Several correct possible answers may exist.

Grading Notice:

1. If the grader is unable to compile or execute your code successfully on aludra, you will not receive any credits. Your Makefile must contain the "agent" part to compile (or do nothing if no compilation is needed) and the "run" part to execute your program. You are allowed to use standard libraries only, such as C++ STL. You have to implement any other function or method only by yourself.

2. In the Hw2.zip, you can find two examples and the grading script. Please modify your program to follow the file IO as the given two examples and your Makefile to work with the grading script.

3. For Java users, please make sure your JAVA compiler version is 1.6 on Aludra.

4. For C/C++ users, please make sure your C/C++ code is compilable for gcc 4.2.1 on Aludra.

Deliverables:

You are required to hand in all the code that you wrote to complete this assignment. You are free to implement in any language of your choice. However, if you code in C++ or Java, the TA will be better able to assist you. Also, we require that your code be able to run from the command line on the USC aludra.usc.edu server. For information on accessing aludra, please see (https://itservices.usc.edu/web/hosting/students)

The deadline for this assignment is Oct 20, at 11:59 pm Los Angeles time. Please turn in all materials as a .zip file via Blackboard, with the title format [firstname]_[lastname]_HW2.zip (e.g., Tommy_Trojan_HW2.zip).