CSCI 561 – Foundations of Artificial Intelligence

Instructor: Dr. K. Narayanaswamy

Assignment 3 – Game Playing

Due: 11:59:59pm, April 5th, 2013

In this assignment, you will write a program that assumes the role of a Go player playing against an opponent. Given a current board configuration, you will use minimax with alpha-beta pruning to determine the next move that will maximize your chance of winning.

Concept and rules for Go

Players and equipment

- **Players:** Go is a game between two players, called Black and White.
- **Board:** Go is played on a plain grid of 19 horizontal and 19 vertical lines, called a *board*.
  - Definition 1 **("Intersection", "Adjacent"):** A point on the board where a horizontal line meets a vertical line is called an *intersection*. Two intersections are said to be *adjacent* if they are connected by a horizontal or vertical line with no other intersections between them.
- **Stones:** Go is played with playing tokens known as *stones*. One player uses black stones and the opponent uses white stones.

Positions

- **Positions:** At any time in the game, each intersection on the board is in one and only one of the following three states: 1) empty; 2) occupied by a black stone; or 3) occupied by a white stone. A *position* consists of an indication of the state of each intersection.
  - Definition 2 **("Connected"):** In a given position, two stones of the same color (or two empty intersections) are said to be *connected* if it is possible to pass from one to the other by a succession of stones of that color (or empty intersections, respectively) in which any two consecutive ones are adjacent.
  - Definition 3 **("Liberty"):** In a given position, a *liberty* of a stone is an empty intersection adjacent to that stone or adjacent to a stone which is connected to that stone.

The following figures further illustrate concepts of **"Connected"** and **"Liberty"**.

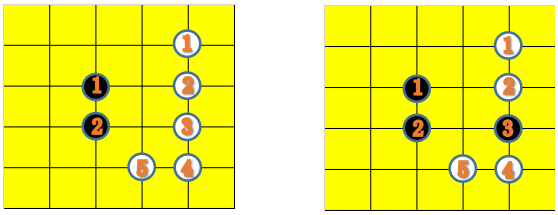

Figure 1. **Connected**. *Left*: black stones 1 and 2 are connected. White stone 1 is connected with white stone 5, since there is a succession of white stones from 1 to 5. *Right*: black stones 1 and 2 are connected, while they are disconnected to black stone 3. White stones 1 and 2 are connected, and white stones 4 and 5 are connected. Nevertheless, white stone 2 is disconnected from white stone 4.

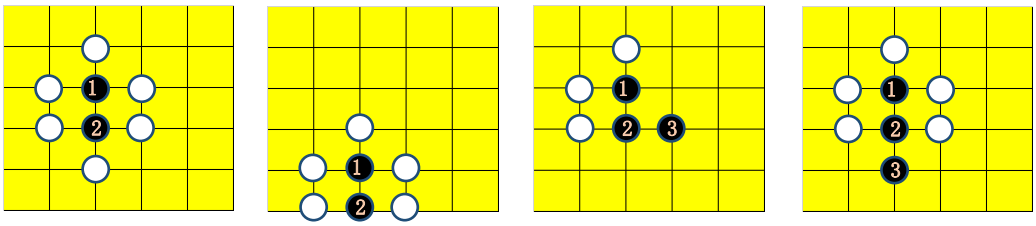

Figure 2 **Liberty**. The above four configurations are results after white placed a stone, and let's give an analysis of liberty of black stone 1 in four cases. The liberty of a stone is defined as: the number of its adjacent empty intersections plus the number of adjacent empty intersections of each of its connected stones (duplicate empty intersections are counted only once). **To compute liberty L_b of a stone b, assume the set of b's connected stones to be S, and let C_i be the set of adjacent empty intersections of stone i, then $L_b = |(\cup_{i \in S} C_i) \cup C_b|$, where $\cup$ is the union of two sets, and $|\ \ |$ is cardinality of a set (i.e. the number of elements a set contains). Note: we don't allow duplicate elements to exist in the union set, therefore, for duplicate elements, we keep only one of them and remove the other.** According to this definition, it's easy to verify that black stone 1 has 0 liberty in case 1, 0 liberty in case 2, 4 liberties in case 3 and 3 liberties in case 4. In case 3, black stone 1 has 1 adjacent empty intersection, and this empty intersection is also the adjacent empty intersection of black stone 3 (duplicate elements). Therefore, this intersection will only be counted once in the computation of black stone 1's liberty.

In our homework, we **modify/simplify the rules**. When programming, you should strictly follow rules defined below.

Playing Rules

- **Initial position:** At the beginning of the game, the board is empty.
- **Turns:** Black places a stone first. The players alternate thereafter.
- **Playing actions:** When it is their turn, a player does the following actions (performed in the prescribed order):
  - Action 1. (Playing a stone) Placing a stone of their color on an empty intersection. It can **never** be moved to another intersection after being played.

- Action 2. (Capture) Removing from the board any stones of their opponent's color that have no liberties, and Labelling these intersection spaces as invalid (i.e. in subsequent steps, a stone is not allowed to be placed on those intersections.)

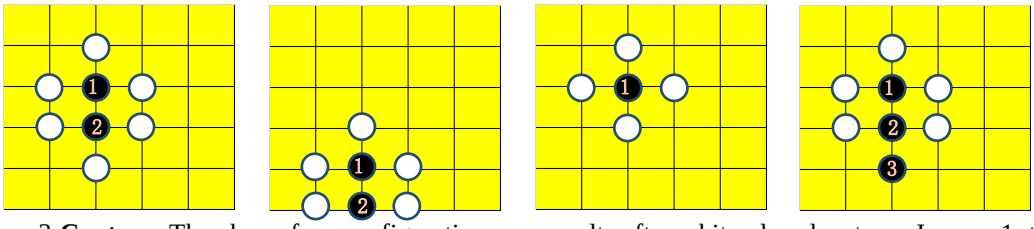

Figure 3 **Capture**. The above four configurations are results after white placed a stone. In case 1, two black stones both have 0 liberty, so they are captured by white and should be removed from the board both. Also these two places are marked as invalid. Similarly, in case 2 and 3, black stones have no liberties and they are captured by white as well. However, in case 4, three black stones all have liberties, and they are NOT captured by white.

Task

Two players, B(black) and White(W), are playing a game of Go. You will write a program that assumes the role of player B, playing against the opponent, player W. Your program will take as input the current board configuration (assuming player B's turn is up next) and will output the next move for player B that yields the best utility for him.

Your program will use minimax algorithm with alpha-beta pruning to determine B's next palcement. However, since it is not feasible to examine all game states of Go, we will limit the search depth to **4**. This means your program will expand nodes up to depth *4* (given the root node is at depth 0). Nodes at depth *d* will not be expanded and are treated as leaf nodes. In counting depth, we add up both players' moves up to that point, so when both players have made a move, the depth is increased by 2.

We will use the following criterion to estimate utility value of each leaf node: **the number of your stones left on the board minus that of your opponent's stones left on the board**. You will want to choose a placement strategy that maximizes this utility value, given player W is also a rational Go player.

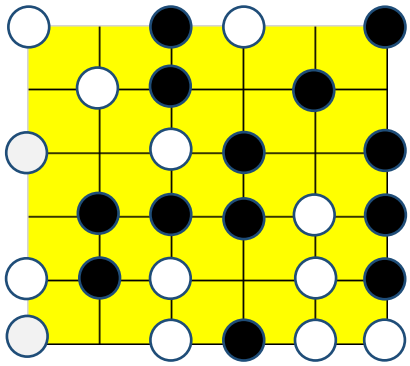

Figure 4 **Initial configuration**. Originally, Go is played on a plain grid of 19 horizontal and 19 vertical lines. In our simplified case, it's played on a grid of 6 horizontal and 6 vertical lines. The initial configuration is shown and **the next step is black's turn**. We use coordinate to define the position of

each intersection: the up-left intersection has coordinate (1,1) and the bottom-right intersection has coordinate (6,6), with x in the horizontal direction and y in the vertical direction.

Program Specifications

Input

There is no input for this homework. You should hard-code all information into your program.

Output

You program should output the optimal strategy for Player B that is determined by minimax with alpha-beta pruning. Apart from that, you have to compare **minimax with alpha-beta pruning** with **minmax algorithm without pruning**, and printout the number of pruned nodes in the former algorithm and running time for two separate algorithms. Although running time is partly determined by your computer hardware, you needn't mention the hardware configuration in your output.

A sample output is as follows:

Sample Output: (Note: these are not necessarily correct outputs. They are only shown to give you a sense of what the output format should look like.)

Best strategy:

Depth 0: Player B places stone at (2,2).
Depth 1: Player W places stone at (3,3).
Depth 2: Player B places stone at (2,3).
Depth 3: Player W places stone at (3,4).
Depth 4: utility value of current board configuration 3.

Comparison:

Minmax with pruning: running time 40.5s;
Minmax with pruning: pruned 500 nodes;
Minmax without pruning: running time 60s.

If there are several placing strategies that result in the same utility, output one of them. The answer is in the format of "Player A places stone at (c,r)" where (c,r) is the position of the stone, with c indicating column index and r indicating row index.

Coding Requirements

 Your program must be written in either Java or C++.

 You may only use the standard libraries (e.g. STL for C++ or Java's standard libraries), and not codes from outside sources. Doing so is considered cheating.

 Your program MUST compile and run on aludra.usc.edu

 Write your own code. Files will be compared and any cheating will be reported.

 The answers output by your program should be calculated from your implementation of the search algorithms. Outputting hard-coded answers is considered cheating and your grade on this assignment will be 0.

Grading Policy

(Out of 100 points)

Programming (90 pts)

60 points for the correct optimal strategy and 30 points for comparison.

Readme.txt (10 pts)

 A brief description of the program structure and any other issues you think will be helpful for the grader to understand your program. (5 pts)

 Instructions on how to compile and execute your code. (5 pts)

 Please include your name, student ID and email address on the top.

 You must submit a program in order to get any credit for the Readme.txt. In short, if you submit ONLY a Readme.txt file you will get 0.

Submission Guidelines

Your program files will all be submitted via blackboard. You MUST follow the guidelines below. Failure to do so will incur a -25 point penalty.

 Compress and zip ALL your homework files (this includes the Readme.txt and all source files) into **one** .zip file. Note that only .zip file extensions are allowed. Other compression extensions such as .tar, rar, or 7z will **NOT** be accepted.

 Name your zip file as follows: firstname_lastname.zip. For example, John_Smith.zip would be a correct file name.

 To submit your assignment, simply select the appropriate assignment link from the Assignments subsection of the course blackboard website. Upload your zip file and click **submit** (clicking send is not enough).

Please make sure ALL source files are included in your zip file when submitted. Errors in submission will be assessed –25 points. A program that does not compile as submitted will be given 0 points.

Only your FINAL submission will be graded.

For policies on late submissions, please see the Syllabus from the course home page. These policies will be enforced with **no exceptions**.

Plagiarism on Programming Assignments: Discussion of concepts, design issues, and programming ideas with colleagues and fellow students is proper and an integral part of learning. However, everyone should do their own coding. Code will be automatically compared and any suspected cases of plagiarism will be investigated and pursued thoroughly. If you happen to reuse snippets of code from the Internet or other sources, please document the sources and nature of what was used in detail to avoid potential problems. If you have any questions or doubts regarding the proper modes of collaboration or using external reference sources, please check with the Professor or TA before you submit your assignment.