FINAL PROJECT

COEN 352 - Fall 2016

Due Date: 05.12.2016 with by 11:59PM (we will discuss when & where exactly to Demo and when & how to submit the final Report+Code in class).

Please manage your time so that you can finish all parts of this assignment on time.

All homework assignments as well as final project will be posted on the course's Moodle website. You must submit your answers to them using Moodle.

The objective of this project is to get you to practice problem solving and advanced data structures and algorithms.

The game of 'Walls' is comprised of an NxN board of squares (like chess, for example) and two players (say A and B), where N is odd and greater than 1. Each game is played twice, one time with player A starting the game and another with player B making the first move. The game has a *start state*, *play rules* that determine legal moves and an *end state*. Once a game ends, there is a certain accounting that decides the *game result*. These concepts are explained below. Design the game so you are able to test its actual run-time on the same computational infrastructure (CPU, RAM etc.) and under the same conditions (running the same OS, with no other application running) for values of N spanning the discrete range: {3,5,7}.

Your objective, as the programmer is to design, in detail, build and test an *artificial agent* (AA for short) capable of playing the game, as optimally as possible, against a human player or another AA. To do so you need to build a ***search tree*** (often called game tree) that includes every possible move starting from the initial state of the game, and all possible future moves, all the way down to the leaf nodes (dead-ends with no possibility of future brick-laying moves). If it is not practically feasible to build the whole tree, then it is acceptable to build a game tree down to less than the maximum possible number of levels, but in that case non-terminal nodes will also have to be assigned a value to guide decision making. One way of assigning values to nodes is by attaching to each node a pair representing the number of bricks belonging to both sides at that point in the game (e.g., <4,2>). In fact, it is also possible to build the tree incrementally, adding one/several levels at a time and only when necessary (e.g., expanding it whenever there is no single best recommendation available to the search algorithm). It is worth noting that once a player secures more than half the nodes in a game (of any size) then that player has won the game (e.g., <5,4> in a walls game with 3x3 squares).

An alternative to using search trees is the use of ***rule-based*** machine learning. A collection of rules controlling a game player is called a *strategy*. A single rule for playing

walls can take the form: IF <neighborhood_mask> is true THEN take <action>. The action set is {*try-forward*, *turn-left*, *turn-right*}. The neighborhood_mask depends on the size of the neighbourhood. By definition, the neighborhood_mask is an MxM matrix of individual conditions centered on the player itself, with M being an odd number and 1 < M <= N. The possible conditions are: *edge* (E), *empty* (0), *brick* of the opposing player (B), the opposing player's *orientation* (N, E, W or S), your own *brick* (A), your own player's *orientation* at the center square (N, E, W or S) and *don't-care* (-). If there is both a player and a brick occupying the *same* location then your player will only *sense* the location and orientation of the player at that location. An example of a rule for M=3 is: < - - N B **E** 0 - A - : *attempt-forward* >. [Hint: such a rule can be applied strictly as is or taking symmetry into consideration, which will make its application more complicated, but will cut down on the number of rules necessary to effectively play a game].

A neighborhood_mask is true if and only if every condition in it is true, otherwise it is false. And, if a neighbourhood_mask is true then its rule is valid and can be activated. There can be any number of rules in a rule list but only the first valid one is activated. If there are no valid rules then the player automatically executes a default move, which is *try-forward*.

Start state. The game starts with a *clear* board with the two players placed at horizontally symmetric positions, which are at equal distances from and as close as possible to the *center* of the board (without overlapping). A player has a 'face' which indicates the *direction* of its forward movement. At the start of the game each player is facing *outward* towards the closest edge to it.

Play rules. The players take turns making moves. A player can: try and move forward (*try-forward*), turn 90 degrees left (*turn-left*) or turn 90 degrees right (*turn-right*). A player *can* only move into an empty square or into a square with a 'brick' bearing its name. A player *cannot* move into a square that has a 'brick' of the opposite type, or off the edge of the board, or into a square currently occupied by the opposite player. Anytime player *X* moves forward from an unmarked square, that square is filled with a brick marked *X* (signifying its origin).

End state. The game ends when either every square on the board contains a brick or *both* players are *stalled*. A player is stalled, *by definition*, if it makes 9 consecutive moves without laying a new brick. A stalled player is not allowed to make any more moves.

Game result. Once the end state of the game is reached, the player with the greater number of bricks bearing its name wins 2 points (the other receiving 0). If both players lay the same number of bricks then the result is a draw, with each player receiving 1 point.

Machine Learning. The machine learning algorithm takes the form of an Evolutionary Algorithm (or EA). A candidate solution is a particular strategy for playing the game. In an EA, there is a population of strategies, which are evaluated and hence, used as a basis

for the generation of the next population of strategies. This cycle of evaluation and generation repeats, until the termination criteria are satisfied. The evolutionary algorithm is described below.

Only for the very first cycle, a population of (say 1000) randomly initialized strategies is created: call that generation Y. Each strategy should have between 5 and 50 rules.

The evolutionary cycle. A random sample of (say 5) strategies from are taken at random from the current population and is involved in a tournament, which will give 1 winning strategy. In a tournament, every strategy plays a double-game against every other strategy (for a total of 10 double-games). A double-game are two walls games, but with each player starting one of the two games. At the conclusion of the whole tournament, each strategy will accumulate 0 or more points. The strategy with the greatest number of points is the winner; if 2 or more have equal points, the strategy with the fewest number of rules is preferred; if that does not resolve the situation, one of the strategies with equal top-most points and equal sizes are selected at random. The strategies that participate in a tournament are not allowed to participate in another tournament. The tournaments repeat until all members of the population have played. Since there are (1000) strategies, this round of tournaments will result in the selection of (200) *parents.* If none of the parents satisfies the *termination criteria* (described below) then these parents will be used to generate, via *recombination* and four types of *mutation,* 800 *children*. The children (800) will then be added to their parents (200) to constitute the next population of 1000 game-playing strategies. [To further combat premature convergence, I suggest that every 5 or so evolutionary cycles, 700 children - rather than 800 - are produced via recombination and mutation, plus 100 children completely randomly generated, as had been done for the very first population]. This new population forms generation Y+1, which will go through the same tournaments described at the start of this paragraph, and so on. Only the satisfaction of the termination criteria will halt the evolutionary algorithm.

The termination criteria is satisfied if (a) one (or more) parent(s) can beat a fixed strategy, in a double-game, a fixed strategy that you devise and believe is optimal or at least very good. The termination criteria must also include a (b) exhaustion condition that halts the evolutionary cycle after a pre-set fixed number of generations (say 1000), in case condition (a) is not met.

The generation of children. As stated above, if the termination criteria are not met by a fit parent or due to exhaustion, a pool of children is generated via recombination and mutation. On an alternating basis, 2 parents will be recombined to generate 2 children, and then 2 other parents which will each be mutated to generate 2 children. To recombine two parents, some rules are copied from one parent while the rest of the rules are copied from the other parent. To mutate a child, the program will first pick from: *add* or *delete, re-order* or *modify* mutations, with (say) equal probability [modify and re-order will be the least disruptive mutations]. An add mutation inserts a randomly generated rule at a randomly chosen location within the rule list. A delete mutation deletes a randomly chosen rule from the rule list. The re-order mutation picks a rule at random and shifts it

up/down by a random number of positions [treating the rule list as if it were circular]. The modify mutation picks a rule at random and hence either: (a) changes a randomly chosen part of the neighborhood_mask, randomly (e.g., changing a “B” at some position in the mask to a “-” at the same position), or (b) changes the action part (e.g., replacing “try-forward” with a “turn-left”).

Please note that random in the context of this problem means *uniformly random*. Random actions (such as random selection of plyers for a tournament) must be *truly* uniformly random, or else the performance of the algorithm will degenerate, due to implicit bias.

You are allowed to either use tree search or evolutionary algorithms or even your own approach to carrying out this project - as long as you produce your own AA (artificial agent). The solution must work for at least 3 different game dimensions (e.g., 3, 5 and 7) in order for the solution to be (theoretically) somewhat general, and to be able to gather experimental evidence about the scalability of your solution. The ultimate aim comprises empirically derived formulae expressing (a) the actual time (in real seconds or CPU cycles or counter values) and (b) order (O) of its time complexity of the AA as a function of the dimension of the game. The time here refers to average total time spent by an AA in playing a game (making all the moves from start to finish). The AA is the program used to make moves in a game of a given dimension rather than any program used to develop or evolve the AA.

Programming Language and IDE:

You can use Java or C++.
Use Eclipse for Java and Visual Studio for C++.
If you write you program on Mac Machine, make sure it runs on a Windows machine.
If the marker cannot run your program, you will lose at least half of the assignment’s mark.

Continue

Documentation:

Metadata

- The programming assignment **title**—e.g., "FinalProject"
- The name and netid of the **student** submitting the assignment.

Design and Implementation

- A high-level **flow chart** describing the main *algorithms* you employed augmented, where necessary, with
- Simply drawn **figures** outlining the major *data structures* that are built and manipulated by the main algorithms.

Testing

(a) **Screenshots** of results

(b) **Brief report** on the performance of the program, especially with respect to its scalability.

How to Submit the Final Project:

At the top of each assignment, include comments that consist of your name, your student ID and the assignment name or number.

When you complete your FinalProject, you will have Source code(s), Documentation and executable file:

Fristname_Lastname_FinalProject.cpp (or .java)
Firstname_Lastname_FinalProject.doc (or docx or pdf)
Firstname_Lastname_FinalProject.exe (or jar)

(First name and Last name are your actual name, i.e. Nancy Nelson)

Here is the instruction on how to submit your assignments, in this case, FinalProject**:**

- Create a folder on the desktop, name it: Your Name_FinalProject (i.e. FinalProject Nancy)
- Locate your assignment FinalProject **(source code, documentation, executable file)**, Copy and Paste them inside **Your name** FinalProject folder
- Close the folder (if it is open)
- Right click on the folder, select "Send To" and then, click on the "Compressed (zipped) Folder"

Now, you should see another folder on the desktop with the "zip" extension: "Your Name_ FinalProject.zip"

Now you should be ready to upload your folder and submit it using the Moodle.

Continue

Honesty Policy: It is expected that all students will conduct themselves in an honest manner, and never claim work which is not their own. Violating this policy will result in a substantial grade penalty, and could result in expulsion from the University. However, students are allowed to discuss programming assignments with others and receive debugging help from others.

****

IMPORTANT:
You must give a proper name to your files prior to submitting them. Each file must start with your first name, and then, last name and Assignment Number. Here is an example:
david_jones**_FinalProject**.zip

IMPORTANT:
You will be responsible to see that your homework is submitted correctly.

IMPORTANT:
If it has been submitted incorrectly, you can resubmit it. The system keeps track of the last file submitted for each program. If it is resubmitted after the due date, it will be considered late. The program automatically dates and times your program and identifies if it is late or not.

IMPORTANT:
Files submitted for homework may be electronically compared to detect cheating.

VERY IMPORTANT:

Each assignment is due **PRIOR** to 11:59PM. If an assignment is submitted at 12:00AM or after, even it may be 1 or 2 minutes late, it is considered a late assignment and will get zero! No Exception Can Be Made!
Always double-check before submitting your assignments.