CSCI 4150 Introduction to Artificial Intelligence, Fall 2002
Assignment 7 (150 points): out Tuesday November 19, due Friday December 6

Learning to play Connect 4

In this assignment, you will implement a genetic algorithm to learn an evaluation function to play Connect 4. The evaluation function will be a mathematical function of feature detector outputs; we will represent the evaluation function using an expression tree.

The support code will include procedures to generate random expression trees, to turn an expression tree into an evaluation function, to randomly pick members of a population, and to play Connect 4 games using alpha-beta minimax. You will have to write procedures to perform crossover, mutation, and fitness evaluation, as well as the overall genetic algorithm.

In the first part of the assignment, you will follow the prescribed approach to implementing the genetic algorithm. In the last part, you will be asked to design your own variation and test its performance.

Notes

- This assignment will have a shortened period for late work. There will only be a single late period (7.5% penalty) which ends at midnight the night of Sunday December 8. No late work will be accepted after this time because of the reading days, Monday and Tuesday December 9 and 10.

  Late written work may be turned in by 10am Monday December 9 to Shannon Bornt in Amos Eaton 132, Dave Siebecker's or Kris Beevers's mailbox in the Amos Eaton Lounge, or at the CS main office in Lally 207.

- We have structured this assignment so that it can be run in a reasonable amount (under 1 hour) of computation time. Please plan accordingly, i.e., you won't all be able to run the assignment on `freebsd.remote` at the same time.

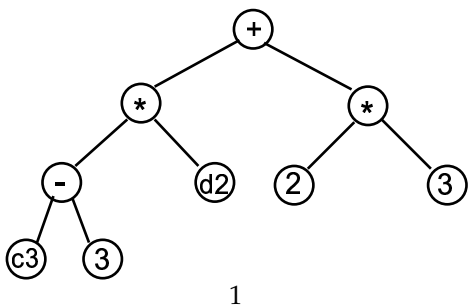
  While this is significantly more computation time than for previous assignments, it is short for an unsupervised learning method. This has been achieved by limiting the population size, using an approximate method of evaluating population fitness, and some other assumptions.

- For Problems 1–3, the web tester will check for answers that match those created by the solutions, so it is important that you implement the algorithms as described. In particular, you should use the support code for picking random nodes and elements. (The pseudorandom number generator will be seeded in the same way that was used to produce the test answers.)

  Problem 4 will be tested based on performance since there are many minor variations that would result in different (specific) answers.

A. Expression trees, crossover, and mutation

In this assignment, the evaluation function will compute a score for a given board using a mathematical function of feature detector outputs. This function will be represented as an expression tree, For example, the expression tree below represents the mathematical expression $(c3 - 3) * d2 + 2 * 3$



We will represent an expression tree in Scheme as a list of the following form:

```
(operator <left-subtree> <right-subtree>)
```

For example, the above expression tree would be represented as:

```
(+ (* (- c3
         3)
      d2)
   (* 2
      3))
```

We will disallow expression trees of depth 0, i.e. where the root node is a leaf node, because such a tree cannot be crossed-over with another expression tree.

The operators for this assignment are limited to the following:

```
operators:  +  -  *
```

Note that the operators used in expression trees are the *symbols* `+`, `-`, and `*`, not the procedures by the same name.

The leaf nodes may be any of the following constants or features.

```
constants:  1 2 3 5
features:   r2 c2 d2 r3 c3 d3 x y z
```

The first six features are based on the provided feature detectors from Assignment 4 that detect open rows, columns, and diagonals. The first character is "`r`" for row, "c" for column, and "d" for diagonal. The second character is "2" for an open sequence of two pieces, "3" for an open sequence of three pieces. The last three features (`x`, `y`, and `z`) are the result of my secret feature detector.

These features are evaluated for a single player, so the actual evaluation function uses the expression tree twice: once to compute a value from the MAX player's features and once to compute a value from the MIN player's features. The value returned by the evaluation function is the difference of the two.

A.1 Support code

- `(random-operator)`
  `(random-leaf)`

  Returns a randomly selected value for a node. A leaf node is either a constant or a feature.

- `(operator etree)`
  `(left-subtree etree)`
  `(right-subtree etree)`

  Accessors for an expression tree.

- `(leaf-node? node)`

  Returns `#t` if `node` is a leaf node.

- `(random-node-directions etree)`

  Returns "directions" to a randomly selected node. These directions consist of a list where each element is either `left` or `right`.

  This procedure has been carefully written to select a node from a tree with equal probability over all nodes (except for the root node which it will not select). To find the selected node, start at the root node, and take the left or right subtree according to the first element of the directions. Repeat this process for the remaining directions. When you run out of directions, you are at the selected node.

For example, using the example expression tree above, the directions `(right)` indicates the right child of the root node which is a * operator. If the directions were `(left left right)`, then the selected node would be the value 3 leaf node at depth 3. Note that there are multiple occurrences of the * operator node and constant 3 leaf node in the tree which is why we need "directions" to find the selected node.

- `(decide-direction etree)`

  Given an expression tree `etree`, this procedure will randomly return one of the symbols `use-this-node`, `use-left-subtree`, or `use-right-subtree`. It is designed so that repeated application will randomly select a node from the tree with uniform probability over all nodes.

- `(print-etree etree)`

  Nicely prints an expression tree.

- `(initial-population size depth)`

  Randomly creates `size` expression trees, each of which may as deep as the given `depth`.

- `(create-eval-fn etree)`

  Given an expression tree, this procedure returns a procedure that can be used as an evaluation function. Just as in Assignment 4, the returned procedure takes three arguments: `board`, `current-player`, and `max-player`.

A.2 Problems

Problem 1 (20 points) Write the procedure:

```
(crossover mother father)
```

which picks a random subtree from each parent and then splices the top part of each parent with the subtree from the other parent. This procedure should return a list containing the two resulting expression trees.

I suggest the following approach to this problem:

- Select random nodes from each parent using the `random-node-directions` procedure.
- Write a procedure `(random-subtree tree directions)` which returns the subtree with the selected node as root node, and get the subtree from each parent.
- Write a procedure `(splice-subtree etree subtree-directions new-subtree)` that takes an expression tree `etree`, directions to the randomly selected node in that subtree, and the subtree that will replace it in the new tree. This procedure should return a newly created tree that splices the `new-subtree` in at the appropriate place. (Actually, only the path down to the affected subtree needs to be newly created; the other subtrees can be the same.)

  essentially make a copy of the top half of the subtree.

Problem 2 (16 points) Write the procedure:

```
(mutate etree)
```

that recursively descends an expression tree (using `decide-direction` to randomly choose the path of descent and where to stop). The thus selected node should then be replaced with a randomly selected operator or leaf node as appropriate (using `random-operator` or `random-leaf`). Note that it is possible for this tree to remain the same after mutation if the same node value is randomly selected.

This procedure should return the mutated expression tree.

B. Fitness evaluation

Ideally, we would run a round-robin tournament in order to evaluate the fitness of the population, but this would take far too much time. Instead, we will randomly pick a number of "reference players" from the population. Each member of the population (including the reference players themselves) will play one game (as `X`) against each of the reference players.

We will compute a score based on the outcome of a game and the number of moves:

$$S = \begin{cases} C_W + (43 - \textit{number-of-moves})W_W & \text{if X wins} \\ C_L - (43 - \textit{number-of-moves})W_L & \text{if O wins} \\ 0 & \text{otherwise} \end{cases}$$

where C_W, C_L, W_W, and W_L are constants.weights for winning and losing. Use the values $C_W = 100$, $C_L = 0$, $W_W = 0.8$ and $W_L = 3$ for Problems 3–5. You should define variables `win-constant`, `lose-constant`, `win-weight` and `lose-weight` to represent these values.

The fitness value for a test player is the sum of its scores with each reference player.

B.1 Support code

- `(create-c4-player eval-fn depth)`

  This procedure uses my alpha-beta minimax solutions to turn an evaluation function into a "player function" as is Assignment 4. Error checking code has been removed in the interest of speed since evaluation functions produced by `create-eval-fn` should not generate any errors.

  You should define a variable `ab-minimax-depth` which controls how deep the search should go. Use a value of 3 for Problems 3–5.

- `(play-c4 x-player o-player)`

  This is the same procedure from Assignment 4 except that it does not print anything to the screen. It returns a list where the first element is the winner (either `X`, `O`, or `draw`) and the second element is the number of moves.

  If you want to see a game, you can load the Assignment 4 support code and run a game.

- `(pick-random-elements lst how-many)`

  This procedure will pick `how-many` elements from the list `lst` and return them in a list. Once an element has been selected, it is removed from consideration so that it cannot be picked twice. (There can be duplicate elements in the return list only if there are duplicate elements in `lst`).

  You should use this procedure to select the reference players from the population.

All of the procedures from the `connect4.scm` file and all the Connect 4 procedures from the `a4code.com` file are included in the support code for this assignment even though they are not listed above.

B.2 Problems

Problem 3 (24 points) Write the procedure:

```
(evaluate-fitness population num-reference-players)
```

which evaluates the population as described above. It should return a list of the form:

```
((fitness-value-1 expression-tree-1)
 (fitness-value-2 expression-tree-2)
 ...)
```

One implementation detail to note is that the `create-eval-fn` is relatively expensive, so you should do this only once per member per generation instead of, say, creating the evaluation every time that member plays a game.

C. Genetic algorithm

Recall that our basic procedure for genetic algorithms is as follows:

- P $\leftarrow$ randomly generated hypotheses
- Compute Fitness(p) for each $p \in P$
- Repeatedly create a new generation P_s and update:
  - *Select*: randomly select $(1-r)|P|$ members of P to add to P_s according to the probability:
    $$Pr(p_i) = \frac{\text{Fitness}(p_i)}{\sum_{p \in P} \text{Fitness}(p)}$$
  - *Crossover*: randomly select $\frac{r|P|}{2}$ pairs of hypotheses from P according to $Pr(\cdot)$. For each pair, use the crossover operator to produce two offspring, and add them to P_s
  - *Mutate*: randomly select $m|P_s|$ members of P_s and mutate the hypothesis
  - $P \leftarrow P_s$
  - Compute Fitness(p) for each $p \in P$

  While $\max_{p \in P} \text{Fitness}(p) < T$
- Return $\text{argmax}_{p \in P} \text{Fitness}(p)$

We will make some minor modifications from this algorithm: the algorithm will be run for only a fixed number of iterations and that the number (instead of fraction) of the population will be specified for selection and mutation.

C.1 Support code

- `(randomly-select-unique-members evaluated-population how-many)`

  This procedure randomly selects members from a population with probability proportional to their fitness. The `evaluated-population` argument should be the output of the `evaluate-fitness` procedure. The list of members returned will contain no duplicates (as long as there are no duplicates in the population.

  There is a variable `fittest-relative-likelihood` which controls the relative likelihood of the fittest member being randomly selected compared to the least-fit member. It is currently set to 5.0. This is a parameter you can change for Problem 6.

  This procedure should be used to select the survivors for the next generation.

- `(randomly-select-members evaluated-population how-many)`

  This procedure randomly selects members from a population with probability proportional to their fitness. Unlike the above procedure, this procedure may pick a single member of the population more than once. The `fittest-relative-likelihood` variable (described above) affects this procedure's operation also.

  This procedure should be used to select the parents for crossover.

- `(separate-random-elements lst how-many)`

  This procedure randomly selects `how-many` elements from a list where each element is equally likely to be chosen. After picking an element, it is removed from the list so that it cannot be picked again. This procedure returns a list where the first element is a list of all the chosen elements and the second element is a list of the remaining elements.

  This procedure should be used to separate the members that will be mutated from the rest of the population.

- `(fittest-member evaluated-population)`

  Given an evaluated population, this procedure returns one of the members with the highest fitness value.

C.2 Problems

Problem 4 (24 points) Write the procedure:

```
(learn-c4-eval pop-size num-selected num-mutated generations)
```

which should return a list containing the fittest player from each generation. This list should be in "reverse chronological order," i.e., the player from the most recent generation should be first, and the best player from the initial population should be last. There should be `generations` + 1 elements on this list, e.g., if you ran one generation, the last element would be the best player from the initial population and the first element would be the best player from the new population (after selection, crossover, etc.)

You should assume that the `pop-size` minus `num-selected` is an even number (because mutation requires an even number of parents and produces an even number of offspring).

Problem 5 (30 points, written) Run your `learn-c4` procedure for 10 generations. Examine the list of most fit players from each generation and describe similarities and differences that you observe.

Evaluate the (hopefully) increasing ability of these players by playing a game (the learned players get to play `X`) against a set of "reference players" that will be provided. (You will probably find it convenient to write a procedure to do this for you.) Evaluate the match results in the same way as for evaluating player fitness. Report your results and describe and explain the trends you observe.

Problem 6 (36 points, written) For this problem, implement some variation to the basic genetic algorithm described in this assignment. For example, you might write a different mutation, crossover, or fitness evaluation. In addition, you can try different population sizes, alpha-beta minimax search depth, or changing other parameters.

Run your new genetic algorithm and measure its performance the same way you did for Problem 5. Turn in a written report that describes:

- what you did — include a printout of the code you wrote or substantially modified for this problem. (You can simply describe minor changes that you made to support code or code that you wrote for other problems in this assignment.)
- the results of tests you ran to evaluate your work
- analysis of your results

This should definitely not be more than 5 (single-sided) pages (excluding code listings) but should probably be at least 2 pages.