Department of Computer Science

Artificial Intelligence
COS314

Project 2: Machine Learning & Meta-Heuristics

Due: Wednesday 25 May, 23:59

You may choose any of the options given below for your project. For each of these projects, note the individual instructions per project. However, the following applies to all of the options:

- The project is done by each student individually, and all code has to be written by yourself.
- You have to upload one compressed archive containing two main folders: `documentation` and `code`. The former will contain a pdf file, with name `????????.pdf`, with the questions marks replaced with your student number. Any other documentation required for your project will also be in this folder. The latter will contain all of your code. Please make sure that the run script for your project is in the root of the `code` folder.
- You may implement the project in any of the following programming languages: C++, C, Java, Python, or Scala, and must run on Linux. You have to provide a Makefile, ant file, or any other alternative project script that will run on Linux. Describe in the pdf file how the project should be compiled and used.
- On the first page of your pdf document, provide your name, surname, and student number, and clearly indicate which option you have implemented.
- Where your program requires user input, these have to be provided as command line arguments.
- Although this is not a programming course, you should always provide well-structured, modular, well-written and documented code.
- Submit your project before the deadline via the online submission system as a compressed tar archive. Late submissions will not be accepted, and emailed submissions will not be accepted.
- A marking rubric for each option will be provided at a later stage.

Option 1: Decision Trees

For this project, you will implement and test a classification tree. In addition to the instructions above, note the following:

- Your program has to accept command line parameters, to be executed as follows:

  `./DecisionTree <options> data.spec data.dat`

  where `DecisionTree` is the name of the program, `<options>` specifies one of a number of options (see the different options below), `data.spec` provides a specification of the attributes and class(es), and `data.dat` is the name of your data file.

- In your pdf document, clearlly indicate which options you have implemented and which not.

- You can use any of the classification data sets available at the UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/
  It will be good if you test on more than one of these data sets.

Your decision tree induction algorithm should be able to do the following:

- Induce a decision tree on a given data set, by making use of the gain ratio criterion as given on slide 67 of the machine learning slides. The given data set has to be randomly split into two subsets: A training set containing 70% of the data patterns and a test set containing 30% of the test set.

- The following options will be used to run your decision tree algorithm:
  - `d`: To induce the decision tree on data sets that have only discrete-valued attributes
  - `c`: To induce a decision tree on data sets that, in addition to discrete-valued attributes, also contain continuous-valued attributes.
  - `md`: To induce a decision tree on data sets that have only discrete-valued attributes, but may contain missing values.
  - `pd`: As for option `d`, but after induction of the tree, you have to prune the tree to prevent overfitting.

  When any of these options do not work (you have not implemented them), then just display an appropriate error message.

- Your decision tree should handle any number of classes.

- After each tree induction, the following output has to be displayed on the screen, and be outputted to the data file `data.out`, in a readible format:
  - The classification error on the training set and the test set.
  - A list of each of the classification rules, in the following format:

    ```
    IF (attr1 relop value) AND (attr2 relop value) AND ... AND (attrN relop value) THEN
       className is classValue
    ```

    where `attr1, attr2, ...` refer to attribute names, `relop` is a relational operator, `value` is the value on which the split is done, `className` is the name of the class, and `classValue` is the value of the class variable.

    Remember that the number of rules is equal to the number of leaf nodes. So, one rule is one traversal from the root to a leaf.
  - For each rule, indicate the classification error on the training set and the test set, and indicate the total number of data patters associated with that rule, and for each class, the number of data patterns associated for that class.
  - For option `pc`, display the above information for both the original tree and the pruned tree.

If you implement pruning of the decision tree, do it as follows: When considering to replace an intermediate (decision) node with a leaf node, calculate the classification error on the test set before the pruning and after the pruning. If the generalization error does not deteriorate more than 5%, then do the pruning. Otherwise, do not prune.

Your program will receive two files, i.e. `data.spec` and `data.dat`. The first file provides a specification of the attributes and classes, while the latter contains the actual data patterns. For the specification file, the format is as follows (note that the files that I will use to test your program will use these formats):

- Each line specifies one attribute.
- The first line specifies the class attribute as follows:

  `className: value1 value2 value3 ... valueN`

  where `className` is the name of the class, and `value1`, ..., `valueN` are the finite set of values that the class attribute can assume.
- The remainder of the lines specify an attribute each, as follows:

  `attributeName: Type`

  where `attributeName` is the name of the attribute and `Type` indicates the type of that attribute. For continuous-valued attributes, the type will be given as `Real`, and for discrete-valued attributes the type will be given as `{ value1, value2, ..., valueM }`, where the M finite values that the attribute can have, are listed.

For the data file, use the following format:

```
attrVal1 attrVal2 ... attrValM classValue
```

where each row contains the values for each attribute in the same order that they are provided in the specification file, and `classValue` is the associated target class. If an attribute has a missing value for a specific pattern, then indicate this with a `?`.

Option 2: Training a Neural Network

For this project you will implement a feedforward neural network, trained using gradient descent, to predict which uppercase letter is represented by an image. You will therefore implement the backpropagation learning algorithm.

The classification problem to solve using a neural network is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique patterns. Each pattern corresponds to one character in some arbitrary font. Each of these patterns was converted into 16 primitive numerical attributes (statistical moments and edge counts) which characterizes the corresponding letter. These numerical values were then scaled to fit into a range of integer values from 0 through 15.

Download the files `letter-recognition.data` and `letter-recognition.names`. The first file provides a description of the attributes used to describe a letter, while the second file contains the actual data. Go through these files so that you are familiar with their contents and the characteristics of each of the attributes.

Find below a pseudocode algorithm to train a feedforward neural network that consists of an input layer, one hidden layer, and an output layer, using gradient descent. Please refer to your notes for the mathematical details. Note that this algorithm implements stochastic learning, and not batch learning.

1. Decide on your NN architecture.
2. Perform pre-processing on the data. That is, do the necessary scaling on input and target values.
3. Shuffle the pre-processed data set, and then divide it into a training set, D_T (60% of the data), validation set, D_V (20%) of the data, and a generalization set, D_G (20%) of the data.
4. Initialize all the weights (including the threshold values) to random values in the range $[-\frac{1}{\sqrt{fanin}}, \frac{1}{\sqrt{fanin}}]$, where $fanin$ is the number of weights leading to the neuron.
5. Initialize values for η (the learning rate), α (the momentum), E_T (the training error), $\xi = 0$ (the epoch counter), ξ_{max} (the maximum number of epochs), and ϵ (the desired training accuracy[1]).
6. Repeat until a convergence criterion has been satisfied

[1] Do not use the mean squared error, but the percentage of correctly classified patterns as the training accuracy.

(a) $E_T = 0$, $\xi++$

(b) for each pattern in the training set D_T,

i. Compute the net input, net_{y_j}, to each hidden unit.

ii. Compute the activation, y_j, of each hidden unit (using the sigmoid activation function).

iii. Compute the net input, net_{o_k}, to each output unit.

iv. Compute the activation, o_k, of each output unit (using the sigmoid activation funciton).

v. Determine if the actual output, a_k, should be 0 or 1, as follows: If $o_k \geq 0.7$, then let $a_k = 1$, meaning that the letter recognized is that represented by the k-th output unit. If $o_k \leq 0.3$, then $a_k = 0$, meaning the letter is not that represented by the k-th output unit. For outputs between 0.3 and 0.7, the network is uncertain about the classification, and you record a classification error for that pattern.

vi. Determine if the target output has been correctly predicted. Let $e = 1$ if the target is correctly predicted (if $t_k = a_k$ for all output units, then the target is correctly predicted); otherwise, $e = 0$.

vii. $E_T += e$

viii. Calculate the error signal for each output

$$\delta_{o_k} = -(t_k - o_k)(1 - o_k)o_k$$

ix. Calculate the new weight values for the hidden-to-output weights

$$\Delta w_{kj}(\xi) = -\eta \delta_{o_k} y_j$$

$$w_{kj}(\xi + 1) = w_{kj}(\xi) + \Delta w_{kj}(\xi) + \alpha \Delta w_{kj}(\xi - 1)$$

x. Calculate the error signal for each hidden unit:

$$\delta_{y_j} = \sum_{k=1}^{K} \delta_{o_k} w_{kj}(1 - y_j)y_j$$

where w_{kj} is the weight between output unit k and hidden unit j.

xi. Calculate the new weight values for the weights between hidden neuron j and input neuron i

$$\Delta v_{ji}(\xi) = -\eta \delta_{y_j} z_i$$

$$v_{ji}(\xi + 1) = v_{ji}(\xi) + \Delta v_{ji}(\xi) + \alpha \Delta v_{ji}(\xi - 1)$$

(c) Calculate the percentage correctly classified patterns as $E_T = E_T/P_T * 100$, where P_T is the total number of patterns in the training set. Do the same for the generalization set, D_G.

(d) Save the epoch number, E_T and E_G to a file.

The instructions below will indicate what information has to be provided in your pdf document. Please provide the items to be included in the document in the same order as specified, and under appropriate section headings.

You need to follow the following steps to complete this project:

1. Study the letter recognition problem and the given data files.

2. Perform the necessary pre-processing on the data to be in a format acceptable for neural network training. Your pre-processing should be such that the training process is optimized. For this you will have to remember the discussions in class. Describe in your document exactly how you have pre-processed the data, and provide motivations.

3. Decide on which stopping conditions you will use, and provide motivations.

4. Experiment 1: For this experiment, you will simply distinguish between one letter and the rest. In other words, if the letter to recognize is 'A', then the classification will be if a pattern corresponds to an 'A' or not. You will not distinguish between all 26 letters. Therefore, your neural network will have only one output unit. If the actual output, $a_k = 1$, then letter 'A' is recognized, otherwise the pattern represents any of the other 25 letters. Your program should be generic to allow the user to specify any letter to be recognized. For this experiment you need to determine the best number of hidden units, and the best value for the learning rate and the momentum. In your document, list these values, and give the training and generalization errors. You also need to provide motivations for why you say these values are best. The way to do this is to provide a table with different values for the number of hidden units, learning rate and momentum that you have tried, and the associated errors. You may even include graphs to illustrate this table. You have to include a graph to illustrate the learning profile, i.e. a line graph where the x-axis represents epoch number, and the y-axis represents accuracy.

5. Experiment 2: Here the task will be to distinguish between vowels (A,E,I,O,U), and non-vowels. Again, use only one output unit. Report the same information as above.

6. Experiment 3: Now, you need to identify the appropriate letter from the 26 letters. In other words, your network will have 26 output units. Report the same information as above.

7. Submission procedure: Submit separate programs for each of the experiments, and name each main program (and the executable) as `experiment?`, where the '?' is replaced with the experiment number. For each experiment, the program must use the original data file provided. All required processing must occur on this file.

For the above, please note that training starts with random weights. This means that you should not make any decisions based only on one training session. To make valid decisions, one should train a neural network on the same problem at least 30 times. The average performance for these 30 networks is then used to make decisions. This process is done as follows: Repeat the training process 30 times starting from step 3 of the training algorithm, and for each of the 30 training sessions record the final network accuracy. Compute the average over these, and use this average to guide your decisions. Be aware that the training will take long, so, allocate enough time for this. State in your report, for each of the experiments, over how many training sessions have you based your decisions (if time becomes a problem, you may use less training sessions, but at least 5; note that the more training sessions you use, the more accurate your conclusions will be).

Option 3: Genetic Algorithm to Evolve Picture Mosaics

For this project you will develop a program to evolve computer generated picture mosaics. A picture mosaic is obtained by sectorizing an original color image into equally sized cells, and then by replacing each cell with a smaller image, selected from a database of images, such that the characteristics of the smaller image closely match that of the original image. You will make use of a genetic algorithm to find the best combination of smaller images to accurately reflect the original image.

The first thing for you to do, is to read about creating picture mosaics, or also referred to as image montages. You may want to look at

- Michael Troebs, *Algorithms to create Image Montages* by Michael Troebs
- ML Michelane, MP Medel, *Understanding Photomosaics*, Dr Dobb's Journal, No 330, Nov 2001, pp 58-63.

You have to find our own database of images to use, but make sure that you have a sufficiently large database. The larger the database, the more accurate the mosaic will be. Also, you will have to write some scripts to covert all of the images to the same size, and to `.jpg` images. Your program will be tested using a `.jpg` image, and a database of `.jpg` images. You can not assume any size of the images in the database that will be used to test your program.

You may make use of libraries for manipulation of the images. You are also allowed to make use of any data structure libraries that you may need. However, you have to write your own genetic algorithm (GA) code. You are not allowed to use any GA libraries, or any GA code not written by yourself. If we do find that you have not written your own GA code, you will get no marks for this project, and plagiarism actions will be followed. Although the objective of this project is to implement the AI (most of the marks will be for the AI), you also have to provide a good GUI.

If you make use if any AI in addition to the specifications here, please describe these in our pdf document. Please do not include your images in the tar file, and no executables. However, you may inlcude example mosaics that you have evolved in the folder `mosaics`. If you do, describe for each example mosaic the GA parameters used to produce it in your pdf document.

Your program should be compiled to the executable, `mosaic`. The program will be executed in the following format:

```
./mosaic image_directory image.jpg x y
```

where

- `image_directory` is the name of the directory where all the images will be stored. Your program should be written such that the images in the given directory is used.

- `image.jpg` is the original image for which a mosaic will be constructed.
- `x` is the number of rows of the grid fitted over `image.jpg`.
- `y` is the number of columns of the grid fitted over `image.jpg`.

The final output, in other words, the mosaic, is to be stored under the file name *image_mosaic.jpg*, where *image* is the name of the image, as provided on the command line.

Now, some guidelines to write the GA to evolve the mosaic:

- Each individual (chromosome) will represent a solution, i.e. a mosaic for the original image. Thus, if the original image is divided into 500 cells, then each individual will consist of 500 images from the image database. That is, each gene of the chromosome will represent one image. You need to define your own mapping from individual to 2D picture mosaic.
- Initialization of the first population: Decide on a number of individuals for the population. Then randomly select from the database an image for each of the genes for each of the individuals. However, you should take note of the following constraints: No pair of adjacent cells of the original image may be replaced with the same image, and you should use as many as possible of the images in the database (try to reduce the number of times that the same image is used in the same chromosome).
- Selection: Use tournament selection as default. For more marks, you may include any other selection method, but the user should be allowed to specify which to use.
- Crossover: Implement two-point cross-over. For more marks, you may include any other selection method, but the user should be allowed to specify which to use. Cross-over occurs at a given probability, which is a value between 0 and 1. If the cross-over probability is 0.9, then after two parents have been selected, then generate a random number between 0 and 1. If this number is less than or equal to 0.9, then perform cross-over. If not, then the parents are not allowed to produce offspring. You may implement additional cross-over operators, from which the user can choose (for extra marks).
- Mutation: This is done by replacing a gene with a new randomly selected image, but such that the constraints above are satisfied. Mutation occurs at a given probability using the same method described above for cross-over. You will find that random mutation on its own will work, but not that well. So, you will need to devise a more efficient way of mutation.
- Selecting the new population: Select the best individuals from the parents and the offspring to form the new population. That is, if the population has 100 individuals, you select the 100 best performing individuals from the parents and the offspring. In addition to this, you are welcome to experiment with different methods.
- And now the fitness function. Here I want you to devise your own fitness function. The fitness function needs to measure how closely the set of images (as given by the genes of the individual) match the original image. For this you need to define a method to measure the distance between the mosaic and the original image. One way of doing this is to calculate for each cell of the original image the average R,G,B values over the pixels of that cell. Then do the same for each image of the chromosome. Then, for each cell use the Euclidean distance or Riemersma's formula (refer to *troeb.pdf* to calculate the similarity between that cell and the corresponding gene (image)). If there are 500 cells, this will result in 500 average values. Now calculate the overage over these to give you a distance between the original and mosaic images. This can serve as a fitness, and the objective is to have this as small as possible. In addition to this, you may want to add penalties if any of the constraints are violated. This is just a suggestion. Play around and see if you can define a better fitness function.

For all of the GA parameters (e.g. population size, cross-over probability, mutation probability, tournament size, etc), you will have to play around to find the best values that will give you a very good mosaic. The user should be allowed to select his own values for these probabilities. You are allowed to add any other operators not mentioned above, as long as it improves the quality of your results.

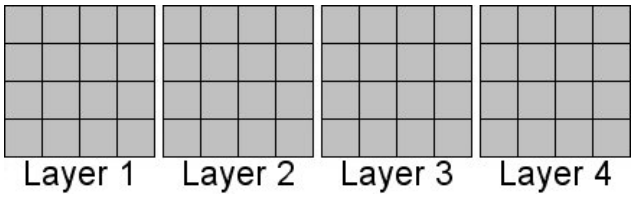
In your pdf file, you need to describe for each of the points above, exactly how you have implemented it. Please give attention to this and do it properly: write enough detail. This will also be used to give you a mark.

Option 4: Co-evolutionary Training of Neural Network for Game-Playing

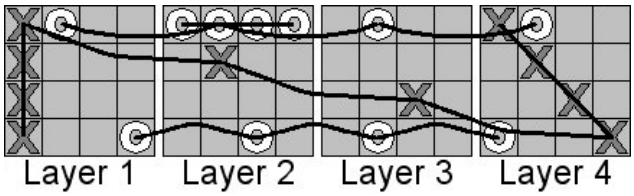
For this project, you will develop an algorithm to evolve an agent to play a probabilistic $4 \times 4 \times 4$ board game. The game is a varation of the classic tic-tac-toe game. The main characteristics of the game are that it is a two-player, turn-based board game, but not deterministic. The decision of where a player may make a move will be probabilistically decided.

In a nutshell, the artificial intelligence approach that you will use to evolve the game agent is competitive coevolution to evolve the evaluation function to be used in an alpha-beta game tree. The evaluation function will be a neural network. The main objective of this approach is therefore to train a neural network to serve as an evaluation function. More detail on this approach is provided below. First, carefully read the description of the game below.

The tic-tac-toe variation described here is a probabilistic two player game, played on four layers consisting of 4×4 grids. Another way to visualize the game board is by seeing it as a three dimensional cube, consisting of 64 smaller separate cube spaces which make up the positions in which a player can place a piece. The figure below shows this.



The game is played similar to the original tic-tac-toe game, with $Player^{1st}$ and $Player^{2nd}$ alternating turns and respectively placing an X or O piece on one of the board layers. A player does not have the freedom though of placing a piece in any of the four available layers. The layer in which a player has to make a move is determined by a "four sided dice". Just before executing a move, the player rolls this dice to determine the level to play. If a player has to play on a layer where all spaces are occupied by pieces, he misses that round and the game moves on to the next player. The game only ends when there are no more empty spaces to place a piece. When the board is full, each player counts the number of rows, columns and diagonals he has completed and gets a point awarded for each successful four pieces placed in sequence. The player with the most points wins the game. If the players have an equal score, the game is a draw. The figure below shows different combinations in which a player can score points. All three dimensions can be used and any four pieces lined up in sequence can score a point.



The table below shows the probability of a win for both $Player^{1st}$ and $Player^{2nd}$, and of a draw for a total of 100000 games that are played randomly by both players.

	Games	%
$Player^{1st}$	50776	50.776
$Player^{2nd}$	44367	44.367
$Draw$	4857	4.857

The advantage of $Player^{1st}$ over $Player^{2nd}$ has been considerably reduced compared to the original tic-tac-toe game. Only a 6.4% winning advantage seperates $Player^{1st}$ from $Player^{2nd}$ in the probabilistic variation.

The game playing agents are represented by standard 3-layer feed forward neural networks, consisting of summation units. The size of the input layer is dependant on the size of the game board. The input layer therefore consists of nine neurons for a standard 3×3 tic-tac-toe game, while 64 neurons are required for a probabilistic $4 \times 4 \times 4$ tic-tac-toe game. The size of the hidden layer varies, depending on the complexity of the game. Only one neuron is used in the output layer. The architecture explained above excludes all bias units. Use the rectified linear activation function in the hidden units, and the sigmoid activation function in the output

neuron. The weights of the neural network are randomly initialized between the range $[\frac{-1}{\sqrt{fanin}}], [\frac{1}{\sqrt{fanin}}]$, where $fanin$ represents the number of incoming weights to the neuron.

The neural network is used to evaluate a given state by accepting the actual state as an input and returning as an output a value that represents how advantageous the state is, with states returning higher values preferred.

Assume that $Player^x$ plays against $Player^y$ and that $Player^x$ needs to plan a new move. Let $State^{current}$ denote the current game state. The following steps are used to determine the next state.

1. Build a game tree with a depth of N, using $State^{current}$ as the root node and by adding all possible moves for $Player^x$ for all odd depths and $Player^y$ for all even depths.
2. Evaluate all leaf nodes by using the neural network as an evaluation function in the following manner:
   i. For all valid positions on the board assign a value of 0.5 for every $Player^x$ piece on the board, a value of -0.5 for every $Player^y$ piece and a value of 0 if there is no piece on a specific position.
   ii. Supply these values as inputs to the neural network and perform a feed forward process to determine the output.
   iii. Assign the value of the neural network output as the evaluation value of the node.
3. Use a minimax game tree with $\alpha\beta$ pruning to determine the most beneficial next move.

Since a neural network is used to evaluate how good a state is, the objective is to find a set of weights which can differentiate between good states and bad states. Usually, supervised training would be used to adjust the neural network weights. In the case of game learning one does not have a training set and the weights can therefore not be adjusted using back propagation or any other supervised training technique. This problem is overcome with the use of coevolution and particle swarm optimization algorithms.

As indicated above, you will use a competitive coevolutionary approach to train a neural network to serve as the evaluation function of leaf nodes of a game tree. Since we will not have a training set of board states and target moves, training can not be supervised, but will rather be unsupervised. We will use a particle swarm optimizer to train the neural network in a competitive coevolutionary manner. The learning model will therefore consist of three components:

- An alpha-beta game tree expanded to a given ply-depth. The root tree represents the current board state, while the other nodes in the tree represent future board states. The objective is to find the next move to take a player maximally closer to its goal, i.e. to win the game. To evaluate the desirability of future board states, an evaluation function is applied to the leaf nodes.
- A neural network evaluation function used to estimate the desirability of board states represented by the leaf nodes. The neural network receives as input the board state, and produces as output a scalar value as the board state desirability.
- A swarm of neural network agents, where each neural network (or particle) is trained in competition with other neural networks.

The training approach is inspired from the work done by Chellapilla and Fogel who developed a coevolutionary game-learning approach to evolve NNs to estimate board state desirability using an evolutionary program. The objective is to evolve a game-playing agent from zero knowledge. That is, no information about playing strategies is provided. The only available information is the rules of the game, and whether a game has been won, lost or drawn.

For the PSO coevolutionary training algorithm, summarized in Algorithm 1, a swarm of particles is randomly created, where each particle represents a single NN. Each NN plays in a tournament against a group of randomly selected opponents, selected from a competition pool (usually consisting of all the current particles of the swarm and all personal best positions). After each NN has played against a group of opponents, it is assigned a score based on the number of wins, losses and draws achieved. These scores are then used to determine personal best and neighborhood best solutions. Weights are adjusted using the position and velocity updates of any PSO algorithm.

Some detail about the particle swarm optimization algorithm is needed:

- Candidate solutions are referred to as particles.
- For this project, a particle will represent one neural network, and use a swarm of 30 particles.

Algorithm 1: Particle Swarm Optimization Coevolutionary Game Training Algorithm

```
Create and randomly initialize a swarm of NNs;
repeat
    Add each personal best position to the competition pool;
    Add each particle to the competition pool;
    for each particle (or NN) do
        Randomly select a group of opponents from the competition pool;
        for each opponent do
            for a number of games do
                Play a game (using game trees to determine next moves) against the opponents, playing
                 as first player;
                Record if game was won, lost or drawn;
                Play a game against same opponent, but as the second player;
                Record if game was won, lost or drawn;
            end
        end
        Determine a score for each particle as the sum over all scores;
        Compute new personal best positions based on scores;
    end
    Compute global best position;
    Update particle velocities;
    Update particle positions;
until stopping condition is true;
Return global best particle as game-playing agent;
```

- The fitness function is the score computed using Algorithm 1.
- Each particle has the following information associated with it:
  - A floating-point position vector, $\mathbf{x}_i$, randomly initialized to values in the range $[-1, 1]$, uniformly sampled.
  - A floating-point velocity, $\mathbf{v}_i$, initialized to zero.
  - A personal best position, $\mathbf{y}_i$, initialized to the initial position
  - A global best position, $\hat{\mathbf{y}}$, computed as the personal best position with the best score
- The personal best position of a particle is set to the new particle position only if this new particle position has a better score than the current personal best position.
- The global best position is the personal best position with the highest score. If there are ties, break the tie by randomly selecting a personal best position.
- Particle velocities are updated using:

$$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)] \tag{1}$$

  where where $v_{ij}(t)$ is the velocity of particle i in dimension $j = 1, \ldots, n_x$ at time step t, $x_{ij}(t)$ is the position of particle i in dimension j at time step t, c_1 and c_2 are positive acceleration constants, and $r_{1j}(t), r_{2j}(t) \sim U(0, 1)$ are random values in the range $[0, 1]$, sampled from a uniform distribution.

  For the purposes of this project, let $w = 0.72$ and $c_1 = c_2 = 1.4$.
- Particle positions are updated using:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2}$$

- The stopping condition is a user specified maximum number of iterations
- The global best particle produced from the algorithm is then used as the game agent, against which you as human will play.

Also implement a GUI to illustrate each of the moves during the game play. Your program should allow for a human to play against the game AI, and also should allow the game AI to play against the game AI, with different game tree ply depths for the different AI players. Remember that all input to the program has to be specified via command line arguments.