Homework 5 (final)

CS:2230 Computer Science II: Data Structures

Deadline: 5 December 2015 (Saturday) at 11 pm.

Instructions

The contents necessary to complete the assignment will be covered until slightly after its announcement time (but before one week before the deadline) to motivate attention in class. Rules:

- Submission through ICON, not email, of the source files (**.java**) only.
- Submissions must be of **high quality**, without errors.
- The homework is to be completed alone. You can still talk with peers about solution ideas and general concepts.
- Do not use libraries nor additional classes unless permitted by the problems.
- Any fields or methods that are not included in the provided specification must be left **private** or protected.
- **Late or improper submissions will receive 0 points.**

CHOOSE ONE. Each problem outlined in this statement involves a considerable amount of work. Choose only one problem and work on it thoroughly.

Want to work in pairs? This is allowed, but you have to go considerably further. Each problem has an extended version for pairs, because, obviously, much more is expected from the work of a team.

Anounce pairs by Friday. If you are going to work with another student, you must send the TAs and the professor (yes, to all 3 of us) an email stating so. After stating that you are working alone or in pairs, you cannot go back on your word.

Want to work on a different problem? You can negotiate this with the professor. An acceptable problem for this homework must involve intelligent and efficient data organization and manipulation, and should not be easier than the problems presented here.

Outstanding homework. If the homework provided is outstanding (perfectly implemented and doing substantially more than required), you can be awarded additional points for your overall homework grade.

Implementation of trees. Trees should be searched using recursion and smartly, especially if they are search trees. (What is the purpose of data structures if they are facilitate performance, anyway?)

Provide documentation. Besides the source code you will submit, you must also submit a .pdf file explaining the structure of your program and any algorithmic and/or design decisions with respect to the data structures and their implementation.

1 Visualizer of Data Structures

Implement a visualizer of data structures, showing their evolution as operations are performed on them. The flow of the application you are to implement is as follows:

1. After opening, the user is to choose an implementation of an abstract data type from a menu.

2. After choosing the ADT implementation, its internal structure is displayed by the application.

3. Then, the user should choose an action (e.g., inserting a number or String) on the chosen ADT, and its updated structure should be displayed.

4. Step 3 should be repeated until the user decides to close the application.

Expected work. The number of implementations of ADTs and the type of user interface depends on whether you work alone or in pairs:

	User interface	Total ADTs	Tree ADTs
Alone	Text	3	1
Pair	Graphical	5	1

Suggested ADT implementations. The following is the list of implementations of ADTs that I suggest you to implement:

1. Stack as circular array. Methods: push(String), and pop(). The array must have initial size of 4 and duplicate its size as more capacity is needed.

2. Stack as linked list. Methods: push(String), and pop().

3. Queue as a circular array. Methods: enqueue(String), and dequeue(). The array must have initial size of 4 and duplicate its size as more capacity is needed.

4. Queue as a linked list. Methods: enqueue(String), and dequeue().

5. Deque as circular array. Methods: addFirst(String), addLast(String), removeFirst(), removeLast(). The array must have initial size of 4 and duplicate its size as more capacity is needed.

6. Deque as [double] linked list. Methods: addFirst(String), addLast(String), removeFirst(), removeLast().

7. List as array. Methods: add(String), add(index,String), remove(index). The array must have initial size of 4 and duplicate its size as more capacity is needed.

8. List as linked list. Methods: add(String), add(index,String), remove(index).

9. Binary Search Tree. Methods: add(String), remove(String).

10. AVL BST. Methods: add(String), remove(String).

Note: the application should not crash upon invalid input or operations. It must notify the user and continue normally. For example, dequeueing from an empty queue should only produce a notification to the user ("cannot dequeue from empty queue" or something alike), and ask the user for more operations on the queue.

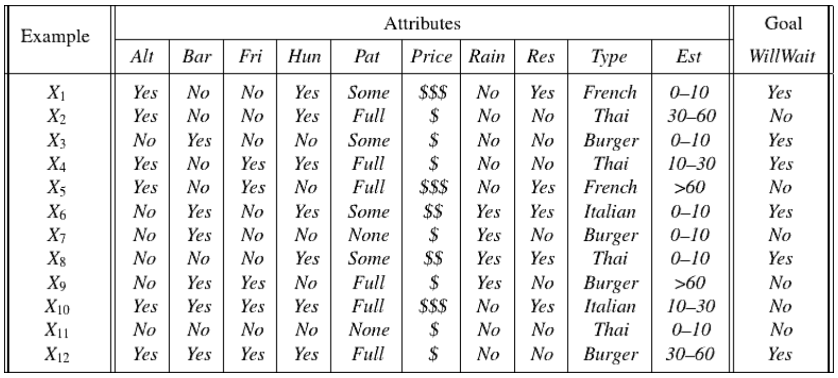
2 Data Mining

For this problem, you are to implement a few *classifiers* and an interface for loading data (.csv files).

Classification is the task of predicting a class for a set of data points or *instances*. The table below exemplifies six instances, with their attributes and their respective classes (*study* home or *play* tennis):

outlook	windy	humidity	temperature	*class*
rain	false	high	hot	*study*
sunny	false	normal	normal	*play*
overcast	true	low	cold	*play*
overcast	false	low	normal	*play*
rain	true	high	hot	*play*
sunny	true	high	normal	*study*

Classification is very important in practice. It has applications to medical diagnosis, insurance, marketing, quality control, etc., including every day decision making (example below).

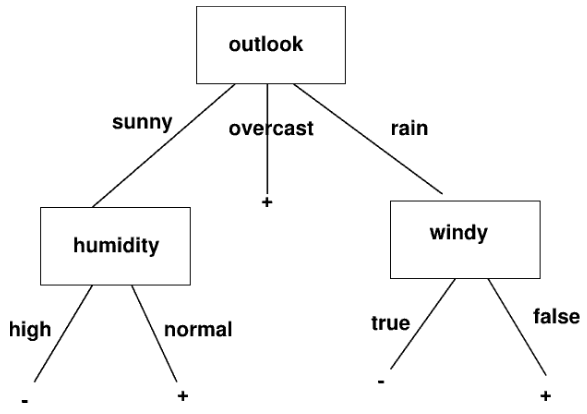


Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

We are interested in the problem of predicting the *class* of each instance as well as possible. *Prediction* means that our predictions are tested against data not used in *training*. The task of *training* consists in adapting a data structure using data. This is normally done with 90% of the data. Then, testing is done with the remaining 10% of it.

Why not just memorize the data? Memorizing usually yields bad predictions. Data normally contains meaningful patterns and random noise. Memorization, unfortunately, does not distinguish between them, giving too much importance to noise.

Example classifier. For the *study/play* example, the following brief decision tree predicts the *class* of each instance by looking at the least number of attributes at once. (In the data mining literature, an attribute is usually called a *feature*.)



Note that this is not a binary search tree. In this tree, each node represents an attribute. Each node models a different attribute than its ancestors. Each child of a node models a different value for the attribute of the node.

Implementation suggestions. You only need to concern with categorical data, meaning, you will not deal with numbers. An attribute can only assume a finite set of values. For example, *outlook* $\in$ {*sunny*, *overcast*, *windy*}. Even if the data contained numbers, you should treat the numbers as categories.

Describe each instance as an array of Strings. An attribute is, then, represented by an index. Also, the class is not part of the instance. In spite that you could store it with the instance, do not consider the class to be an attribute that can be used in prediction! Obviously, the class is unknown at the time of prediction.

When dividing the data into a testing and training set, you can use an index to keep track of the instance number. (For example, you could load the data to an array or a java.util.LinkedList/ArrayList.) Then you can use the multiples of 10 as testing set and the rest as training set.

You can also assume that the class is binary. Treat labels “true”, “yes”, “1”, “+”, “positive” as *true*, and the rest as *false*.

Expected work. You are to implement a few classifiers and provide a little interface for loading data, and presenting the results of your classifiers on the training set, compared through accuracy. What is *accuracy*? Accuracy is the rate of correctly predicted classes. If you've got 315 correctly predicted instances out of 380, accuracy is 82.89%. (Use floating number division to report accuracy.)

If you work alone, you should implement three classifiers. If you work in pairs, you should implement five classifiers, and provide a graphical user interface for the program. Regardless of working alone or in pairs, you should implement at least one tree classifier.

Suggested classifiers.

- **Decision BST**. Implement this tree as a binary search tree, where you compare instances component-wise. For example, for instances A={”rain”,”hot”,”high”}, B={”overcast”,”hot”,”normal”}, C={”rain”, ”cold”,”high”}, you have that A<B, because A[0].compareTo(B[0])<0 (i.e., “overcast”.compareTo(“rain”)>0), but note that A>B, because A[0].equals(B[0])[1] and A[1].compareTo(B[1])>0.
  Now, on each node you would store a copy of the instance (String[]), but you would also store two counters: the number of times such instances are in the *true* class, and the number of times they are in the *false* class.
  How to classify an instance? When searching an instance through the tree, if you reach the desired node, return the most encountered class. However, if you could not find the instance or the number of observations was too small (less than 20 or less 10% of the training set), you can seek the closest instances and combine their class counts. Then, answer the majority of the combined nodes.

[1]For two Strings A,B, both A.equals(B) and A.compareTo(B)==0 denote logical String equality.

- **Mismatch BST**. The previous BST is just memorization, although this is partially palliated by allowing the search to combine the answers of nodes. Well, this BST always does that. Training: for an instance {x,y,z}, for instance, you feed {"*",y,z}, {x,"*",z},{z,y,"*"} to the tree. (I.e., transform an instance of k attributes to k different instances.) Prediction: for an instance {x,y,z}, search for {"*",y,z}, {x,"*",z},{z,y,"*"}, combining all the counts for the *true* class and all the counts for the *false* class. Predict *true* when it surpasses *false*. This is an optimized ranged *nearest-neighbors* classifier. (Note: adapt reasonably the idea for when there are more attributes.)

- **Majority mismatch BST**. It is just as above, but change the prediction rule. Prediction: for an instance {x,y,z}, search for {"*",y,z}, {x,"*",z},{z,y,"*"}, obtaining the most frequent class for each. Then, predict the majority class. (Note: adapt reasonably the idea for when there are more attributes.)

- **Random Forest**. For a training set of n instances with k attributes, you are to divide the k attributes in 15 groups of $\max(2, k/15)$ attributes [almost] randomly. Then, train 15 decision trees, one for each subset of attributes. What class to predict? Whenever predicting an instance, use the most frequently predicted class among the 15 decision trees. The decision trees can be implemented using any of the techniques above.

- **Decision Stumps**. Suppose that most instances with outlook=*overcast* have class *true*, that most instances with humidity=*high* have class *false*, and that most instances with temperature=*hot* have class *true*. Then an instance {"overcast","high","hot"} can be reduced to *true*, *false*, *true*, of which the majority is *true*, so we predict class=*true*. (Do not look back to the training set for this classifier; choose a data structure wisely to perform this task.)

- **Majority Rule**. The simplest classifier. It is useful to determine how bad can other classifiers get. Just predict the most common class encountered in the training set. (If another classifier performs worse than the majority rule, then it is worthless.)

3 Game Battler

This problem is actually about combinatorial optimization and artificial intelligence. Alas, it is also interesting for data structures.

For this problem, you are to implement a turn-based battle system. In a battle, there can only be two teams: the user and the computer. Each team should be composed of the same number of fighters, and the fighters can be of several types. During a turn, a team can only have one active fighter, and only one action can be performed: either an active fighter's action, or two change the active fighter within a team. (So far, this is pretty much like Pokemon.) Also, the moves a fighter has are determined by its type, and they can affect the fighter itself, its team, or the opposing active fighter, or its team. Each fighter has health points (HP), that decrease with the opponent's aggression. A fighter with 0 HP cannot be active any longer; if there are remaining fighters, the team can have a different active fighter without losing turns. The battle is lost by the team whose fighters have 0 HP. What is the problem? To make the *computer* smart, i.e., to let it try to outsmart the user.

A tiny example. Let us consider a more specific example. (You could just program this one.) Each team is composed of three fighters, each being either a *knight* or a *wizard*. Both types have 100 health points, but knights take half damage from physical attacks and wizards take half damage from magic. Their movesets are:

- *Knight*
  - Melee. Take 40 points out of the active opponent. (Physical.)
  - Sacrifice. Both active fighters' HP is set to 0.
  - Heal. Recover 30 HP.
- *Wizard*
  - Flame. Take 40 points out of the active opponent. (Magic.)
  - Fireworks. Take 20 points out of each fighter of the opposing team. (Magic.)
  - Group heal. Each member of own team recovers 15 HP.

    – Fumes. Each member of the opposing team is poisoned; they lose 5 HP for 4 of their turns.

The problem with such a game is that it requires the computer to be an strategical adversary. Basically, the AI has to think about future moves and their outcomes.

Expected work. If you work alone, a simple example like the above is enough to get the job done. However, you are still to supply a user interface for playing such a game. Using text should be enough. Also, you are two implement two AI engines.

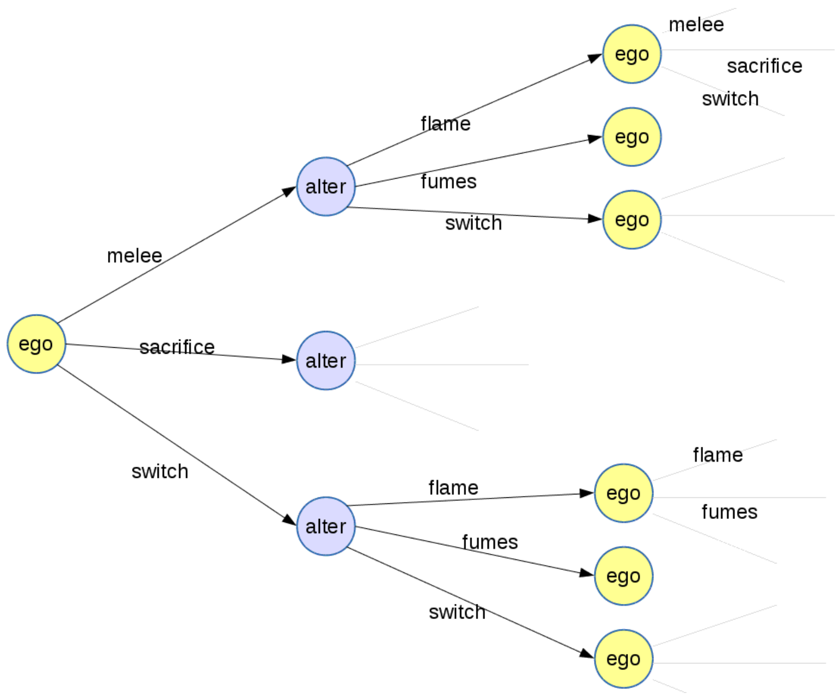
If you work in pairs, you must introduce more types of fighters and more moves. Make sure that the moves are somewhat balanced, i.e., that no one move is dominated by another rule. (For example, having another Melee rule for the knight such that the other rule just means taking less HP of the opponent.) Also, for pairs, the user interface must be graphical. And you will have to implement four AI engines.

Regardless if you work alone or in pairs, you must implement one tree-based AI.

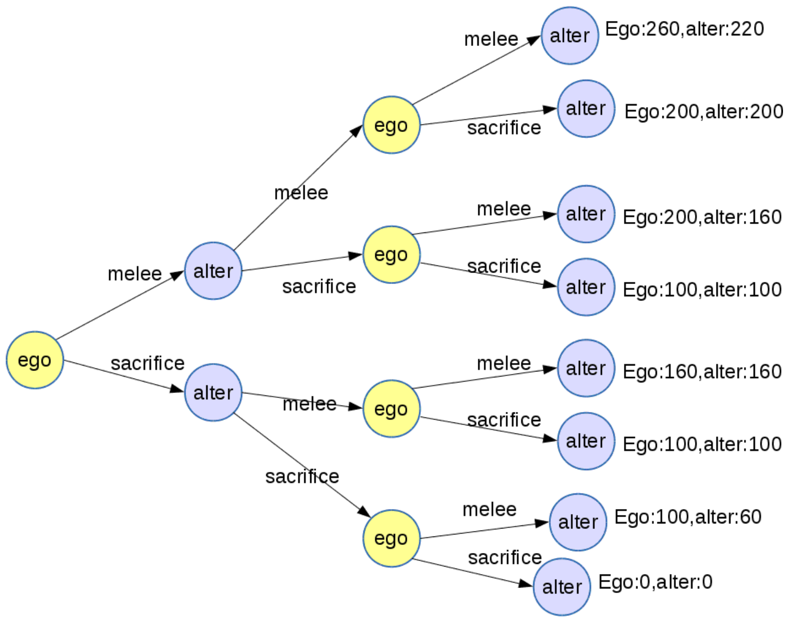
User interface. The game should, at all moments, inform about the HP and types of the fighters of each team. Naturally, the active fighters of each team should be depicted clearly. Also, the UI should allow the user to choose the members of both teams, and the AI engine for the computer.

AI engines. The first AI engine is mandatory to program.

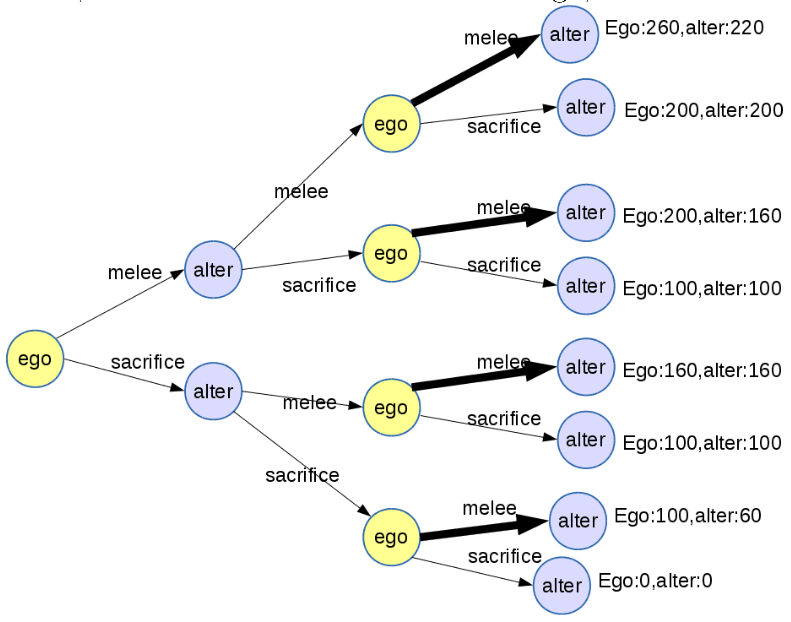
- **Delta HP**. For this one, you have to implement a tree representing the actions of both teams. All possible actions. The tree should have height 8. You must choose the action that leads to the best outcome in this tree. In this tree, a node represents a turn, so the root starts from the computer's turn (ego). Each branch defines a decision, which are the moves of the active fighter and the decision to change the active fighter. The children represent the user's turn, and the branches represent the possible actions taken by the user.



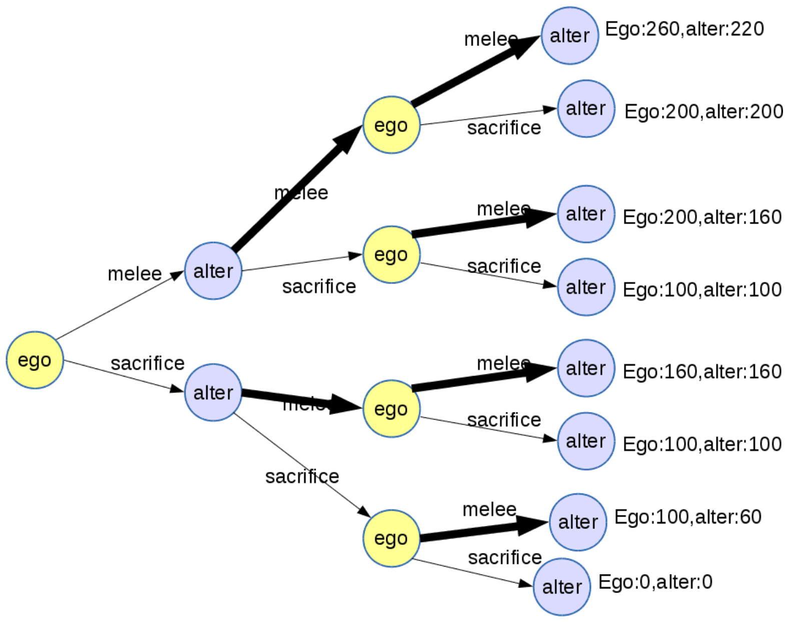
(The tree above depicts a simplified tree representing a subset of the possible futures foreseen by the engine.)
How to choose the best decision? There are many ways to do so. For the Delta HP rule, the best path is the one that maximizes the HP of the computer's team (ego) minus the HP of the user's team (alter). If two options have the same Delta HP outcome, the actor at hand (computer/user) chooses what is best for itself. But note that the computer will choose the decision that is best for the computer, but the user will choose what is worst for the computer.
If we consider the case of teams of size 3, composed of knights, and only thinking of the moves *melee* and *sacrifice*, and for three turns only, a tiny tree could be processed as follows. First, we compute the final HP of both teams after all decisions.



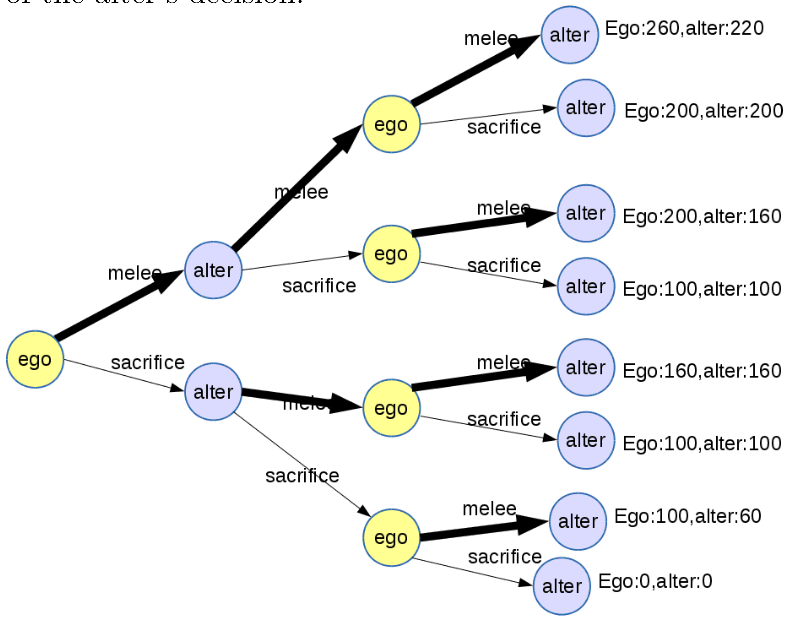
Then, we make the best choice for the ego, around the leaves.



But the alter chooses what is worst for the ego.



Finally, the first decision by the ego is the one that is best in spite of the alter's decision.



- **HP Ratio**. As the rule for the above tree, the ego should seek the moves that maximize its HP ratio to the alter's, i.e., HP(ego)/HP(alter).

The alter seeks to minimize this ratio. Note: to avoid division by zero, you can replace HP(alter) by 1+HP(alter).

- **Survival**. The ego should seek the moves that maximize its HP over time, while the alter seeks the moves that minimize the ego's HP.
- **Kill alter**. The ego should seek the moves that minimize the alter's HP, while the alter seeks the moves that minimize the ego's HP (as above).
- **Other criteria? Heuristics?** You can propose alternative criteria as long as they are reasonable.
- **Random**. When the *computer*'s turn comes, it chooses a random move from the active fighter. If the active fighter hit 0 HP, the computer chooses the remaining fighter to activate at random.

Suggestion: with a clean object-oriented design, this problem can be relatively easy. As a suggestion, you should create a class to represent the state of the game (e.g., GameState), which should return the available actions (e.g., GameState.getActions), update its state by choosing actions (e.g., GameState.applyAction), and is clonable (e.g., GameState.clone()). Cloning is key for building a tree structure that simulates future states of the game BUT does not affect the current state at all.