**A**

**Project Report**

**On**

**An Artificially Intelligent Tic Tac Toe Player**

**Expert Systems Lab**

**(Semester VII)**

**in pursuit of**

**B.Tech in Computer Science & Engg.**

****

**Submitted To (Project In charge): Submitted By:**

**Er. Nitika Chowdhary Jaspreet Singh Arora**

**Department Of CSE, GNDU 2011 CSA 1448**

**Department of Computer Science & Engineering**

**GURU NANAK DEV UNIVERSITY**

**Regional Campus, Jalandhar**

**ACKNOWLEDGMENT**

It gives me immense pleasure to express my sincere and whole hearted sense of gratitude to our esteemed guide, Er. Nitika Chowdhary for her invaluable and untiring guidance and supervision throughout our work. I would also like to thank my friends Manhar Thakur, Saurabh Sharma and Love Arora, without whose input this project would not have been possible.

My heartiest thanks to our Head of Department, Dr. Jyoteesh Malhotra, who continues to inspire us and nurture such an environment where we can fully exercise our creativity and set an example of going beyond the call of duty when it comes to helping others. All cannot be mentioned but none can be forgotten.

**ABSTRACT**

The project is a virtual implementation of the game Tic Tac Toe. The base of the project is an artificially intelligent computer player, which has certain facts and rules hardcoded into it, on the basis of which, it uses the Minimax algorithm, to infer the next best move for the computer to make the game unbeatable, i.e. either the computer wins the game, or the game is a draw. The Minimax algorithm works on the principle of a zero-sum game theory, to minimize the possible loss scenario for a player in the game. Therefore, this particular implementation of Tic Tac Toe is unbeatable.

**INTRODUCTION**

The field of artificial intelligence was founded on the claim that a central property of humans, intelligence—the [sapience](http://en.wikipedia.org/wiki/Sapience) of [*Homo sapiens*](http://en.wikipedia.org/wiki/Homo_sapiens)—"can be so precisely described that a machine can be made to simulate it."

Artificial intelligence (AI) is the intelligence exhibited by machines or software. It is an academic field of study, which studies the goal of creating intelligence, whether in emulating human-like intelligence or not. Major AI researchers and textbooks define this field as "the study and design of intelligent agents", where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success. John McCarthy, who coined the term in 1955, defines it as "the science and engineering of making intelligent machines".

Mechanical or ["formal" reasoning](http://en.wikipedia.org/wiki/Formal_reasoning) has been developed by philosophers and mathematicians. The study of logic led directly to the invention of the [programmable digital electronic computer](http://en.wikipedia.org/wiki/Computer), based on the work of mathematician [Alan Turing](http://en.wikipedia.org/wiki/Alan_Turing) and others. Turing's [theory of computation](http://en.wikipedia.org/wiki/Theory_of_computation) suggested that a machine, by shuffling symbols as simple as "0" and "1", could simulate any conceivable act of mathematical deduction.This, along with concurrent discoveries in [neurology](http://en.wikipedia.org/wiki/Neurology), [information theory](http://en.wikipedia.org/wiki/Information_theory) and [cybernetics](http://en.wikipedia.org/wiki/Cybernetic), inspired a small group of researchers to begin to seriously consider the possibility of building an electronic brain.

AI research is highly technical and specialized, and is deeply divided into subfields that often fail to communicate with each other. Some of the division is due to social and cultural factors: subfields have grown up around particular institutions and the work of individual researchers. Some subfields focus on the solution of specific problems. Others focus on one of several possible approaches or on the use of a particular tool or towards the accomplishment of particular applications.

The AI field is interdisciplinary, in which a number of sciences and professions converge, including computer science, mathematics, psychology, linguistics, philosophy and neuroscience, as well as other specialized fields such as artificial psychology.

In artificial intelligence, an expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented primarily as if–then rules rather than through conventional procedural code. The first expert systems were created in the 1970s and then proliferated in the 1980s. Expert systems were among the first truly successful forms of AI software.

An expert system is divided into two sub-systems: the inference engine and the knowledge base. The knowledge base represents facts and rules. The inference engine applies the rules to the known facts to deduce new facts. Inference engines can also include explanation and debugging capabilities.

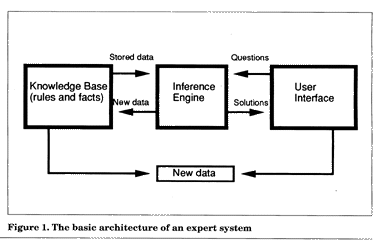
Edward Feigenbaum in a 1977 paper said that the key insight of early expert systems was that "intelligent systems derive their power from the knowledge they possess rather than from the specific formalisms and inference schemes they use" (as paraphrased by Hayes-Roth, et al.).

Expert systems were introduced by the Stanford Heuristic Programming Project led by Feigenbaum, who is sometimes referred to as the "father of expert systems". The Stanford researchers tried to identify domains where expertise was highly valued and complex, such as diagnosing infectious diseases (Mycin) and identifying unknown organic molecules (Dendral).

In addition to Feigenbaum key early contributors were Edward Shortliffe, Bruce Buchanan, and Randall Davis. Expert systems were among the first truly successful forms of AI software.

As Expert Systems evolved many new techniques were incorporated into various types of inference engines. Some of the most important of these were:

* Truth Maintenance. Truth maintenance systems record the dependencies in a knowledge-base so that when facts are altered dependent knowledge can be altered accordingly. For example, if the system learns that Socrates is no longer known to be a man it will revoke the assertion that Socrates is mortal.
* Hypothetical Reasoning. In hypothetical reasoning, the knowledge base can be divided up into many possible views, aka worlds. This allows the inference engine to explore multiple possibilities in parallel. In this simple example, the system may want to explore the consequences of both assertions, what will be true if Socrates is a Man and what will be true if he is not?
* Fuzzy Logic. One of the first extensions of simply using rules to represent knowledge was also to associate a probability with each rule. So, not to assert that Socrates is mortal but to assert Socrates may be mortal with some probability value. Simple probabilities were extended in some systems with sophisticated mechanisms for uncertain reasoning and combination of probabilities.

****

**BACKGROUND**

**TIC TAC TOE**

**Tic-tac-toe** (or **Noughts and crosses**, **Xs and Os**) is a paper-and-pencil game for two players, *X* and *O*, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game.

The following example game is won by the first player, X:

[ame of Tic-tac-toe, won by X](http://en.wikipedia.org/wiki/File:Tic-tac-toe-game-1.svg)

Despite its apparent simplicity, Tic-tac-toe requires detailed analysis to determine even some elementary combinatory facts, the most interesting of which are the number of possible games and the number of possible positions. A position is merely a state of the board, while a game usually refers to the way a terminal position is obtained.

Naive counting leads to 19,683 possible board layouts (39 since each of the nine spaces can be X, O or blank), and 362,880 (i.e. 9[!](http://en.wikipedia.org/wiki/Factorial)) possible games (different sequences for placing the Xs and Os on the board). However, two matters much reduce these numbers:

* The game ends when three-in-a-row is obtained.
* The number of Xs is always either equal to or exactly 1 more than the number of Os (if X starts).

**Number of terminal positions**

When considering only the state of the board, and after taking into account board symmetries (i.e. rotations and reflections), there are only 138 terminal board positions. Assuming that X makes the first move every time:

* 91 unique positions are won by (X)
* 44 unique positions are won by (O)
* 3 unique positions are drawn

**Number of possible games**

Without taking symmetries into account, the number of possible games can be determined by hand with an exact formula that leads to 255,168 possible games. Assuming that X makes the first move every time:

* 131,184 finished games are won by (X)
* 77,904 finished games are won by (O)
* 46,080 finished games are drawn

If board symmetries are taken into account, two games are considered the same if rotating and/or reflecting the board makes one of the games into a copy of the other game. With the use of a computer, Steve Schaeffer found in 2007 that the number of games in these conditions is 26,830.

**Strategy**

The possible moves in a Tic Tac Toe :

1. **Win**: If the player has two in a row, they can place a third to get three in a row.
2. **Block**: If the opponent has two in a row, the player must play the third themselves to block the opponent.
3. **Fork**: Create an opportunity where the player has two threats to win (two non-blocked lines of 2).
4. **Blocking an opponent's fork**:
   * **Option 1**: The player should create two in a row to force the opponent into defending, as long as it doesn't result in them creating a fork. For example, if "X" has a corner, "O" has the center, and "X" has the opposite corner as well, "O" must not play a corner in order to win. (Playing a corner in this scenario creates a fork for "X" to win.)
   * **Option 2**: If there is a configuration where the opponent can fork, the player should block that fork.
5. **Center**: A player marks the center. (If it is the first move of the game, playing on a corner gives "O" more opportunities to make a mistake and may therefore be the better choice; however, it makes no difference between perfect players.)
6. **Opposite corner**: If the opponent is in the corner, the player plays the opposite corner.
7. **Empty corner**: The player plays in a corner square.
8. **Empty side**: The player plays in a middle square on any of the 4 sides.

The first player, who shall be designated "X", has 3 possible positions to mark during the first turn. Superficially, it might seem that there are 9 possible positions, corresponding to the 9 squares in the grid. However, by rotating the board, we will find that in the first turn, every corner mark is strategically equivalent to every other corner mark. The same is true of every edge mark. For strategy purposes, there are therefore only three possible first marks: corner, edge, or center. Player X can win or force a draw from any of these starting marks; however, playing the corner gives the opponent the smallest choice of squares which must be played to avoid losing.

The second player, who shall be designated "O", must respond to X's opening mark in such a way as to avoid the forced win. Player O must always respond to a corner opening with a center mark, and to a center opening with a corner mark. An edge opening must be answered either with a center mark, a corner mark next to the X, or an edge mark opposite the X. Any other responses will allow X to force the win. Once the opening is completed, O's task is to follow the above list of priorities in order to force the draw, or else to gain a win if X makes a weak play.

To guarantee a draw for O, however:

* If X does not play center opening move (playing a corner is the best opening move), take center, and then a side middle. This will stop any forks from happening. If O plays a corner, a perfect X player has already played the corner opposite their first and proceeds to play a 3rd corner, stopping O's 3-in-a-row and making their own fork.
* If X plays center opening move, O should pay attention and not allow a fork. X should play a corner first.
  + If O takes center (best move for them), X should take the corner opposite the original, and proceed as detailed above.
  + If O plays a corner or side-middle first, X is guaranteed to win:
    - If corner, X simply takes any of the other 2 corners, and then the last, a fork.
    - If O plays a side-middle, X takes the only corner that O's blocking won't make 2 in a row. O will block, but the best of the other two will be seen by X, and O is forked. The only way that X must lose is if O plays middle and then a side-middle.

**MINIMAX ALGORITHM**

**Minimax** is a decision rule used in [decision theory](http://en.wikipedia.org/wiki/Decision_theory), statistics and philosophy for *mini*mizing the possible [loss](http://en.wikipedia.org/wiki/Loss_function) for a worst case (*max*imum loss) scenario. In [game theory](http://en.wikipedia.org/wiki/Game_theory) and [economic theory](http://en.wikipedia.org/wiki/Economic_theory), a **zero-sum game** is a [mathematical representation](http://en.wikipedia.org/wiki/Mathematical_model) of a situation in which a participant's gain (or loss) of [utility](http://en.wikipedia.org/wiki/Utility) is exactly balanced by the losses (or gains) of the utility of the other participant(s). If the total gains of the participants are added up and the total losses are subtracted, they will sum to zero. Thus [cutting a cake](http://en.wikipedia.org/wiki/Fair_cake-cutting), where taking a larger piece reduces the amount of cake available for others, is a zero-sum game if all participants value each unit of cake equally.

Originally formulated for two-player [zero-sum](http://en.wikipedia.org/wiki/Zero-sum) [game theory](http://en.wikipedia.org/wiki/Game_theory), covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty.

### Minimax theorem

The minimax theorem states -

For every two-person, [zero-sum](http://en.wikipedia.org/wiki/Zero-sum) game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that

(a) Given player 2's strategy, the best payoff possible for player 1 is V, and

(b) Given player 1's strategy, the best payoff possible for player 2 is −V.

Equivalently, Player 1's strategy guarantees him a payoff of V regardless of Player 2's strategy, and similarly Player 2 can guarantee himself a payoff of −V. The name minimax arises because each player minimizes the maximum payoff possible for the other—since the game is zero-sum, he also minimizes his own maximum loss (i.e. maximize his minimum payoff).

This theorem was first published in 1928 by [John von Neumann](http://en.wikipedia.org/wiki/John_von_Neumann), who is quoted as saying "As far as I can see, there could be no theory of games … without that theorem … I thought there was nothing worth publishing until the *Minimax Theorem* was proved".

The following example of a zero-sum game, where **A** and **B** make simultaneous moves, illustrates *minimax* solutions.

### Example

|  |  |  |  |
| --- | --- | --- | --- |
|  | **B chooses B1** | **B chooses B2** | **B chooses B3** |
| **A chooses A1** | +3 | −2 | +2 |
| **A chooses A2** | −1 | 0 | +4 |
| **A chooses A3** | −4 | −3 | +1 |

Suppose each player has three choices and consider the [payoff matrix](http://en.wikipedia.org/wiki/Payoff_matrix) for **A** displayed at right. Assume the payoff matrix for **B** is the same matrix with the signs reversed (i.e. if the choices are A1 and B1 then **B** pays 3 to **A**). Then, the minimax choice for **A** is A2 since the worst possible result is then having to pay 1, while the simple minimax choice for **B** is B2 since the worst possible result is then no payment. However, this solution is not stable, since if **B** believes **A** will choose A2 then **B** will choose B1 to gain 1; then if**A** believes **B** will choose B1 then **A** will choose A1 to gain 3; and then **B** will choose B2; and eventually both players will realize the difficulty of making a choice. So a more stable strategy is needed.

Some choices are *dominated* by others and can be eliminated: **A** will not choose A3 since either A1 or A2 will produce a better result, no matter what **B** chooses; **B** will not choose B3 since some mixtures of B1 and B2 will produce a better result, no matter what **A** chooses.

**A** can avoid having to make an expected payment of more than 1∕3 by choosing A1 with probability 1∕6 and A2 with probability 5∕6: The expected payoff for **A** would be 3 × (1∕6) − 1 × (5∕6) = −1∕3 in case **B** chose B1 and −2 × (1∕6) + 0 × (5∕6) = −1∕3 in case **B** chose B2. Similarly, **B** can ensure an expected gain of at least 1/3, no matter what **A** chooses, by using a randomized strategy of choosing B1 with probability 1∕3 and B2 with probability 2∕3. These [mixed](http://en.wikipedia.org/wiki/Mixed_strategy) minimax strategies are now stable and cannot be improved.

### Minimax algorithm with alternate moves

A **minimax algorithm** is a recursive [algorithm](http://en.wikipedia.org/wiki/Algorithm) for choosing the next move in an n-player [game](http://en.wikipedia.org/wiki/Game_theory), usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a [position evaluation function](http://en.wikipedia.org/wiki/Evaluation_function) and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is **A**'s turn to move, **A** gives a value to each of his legal moves.

A possible allocation method consists in assigning a certain win for **A** as +1 and for **B** as −1. An alternative is using a rule that if the result of a move is an immediate win for **A** it is assigned positive infinity and, if it is an immediate win for **B**, negative infinity. The value to **A** of any other move is the minimum of the values resulting from each of **B**'s possible replies. For this reason, **A** is called the *maximizing player* and **B** is called the *minimizing player*, hence the name *minimax algorithm*. The above algorithm will assign a value of positive or negative infinity to any position since the value of every position will be the value of some final winning or losing position. Often this is generally only possible at the very end of complicated games such as [chess](http://en.wikipedia.org/wiki/Chess), since it is not computationally feasible to look ahead as far as the completion of the game, except towards the end, and instead positions are given finite values as estimates of the degree of belief that they will lead to a win for one player or another.

This can be extended if we can supply a [heuristic](http://en.wikipedia.org/wiki/Heuristic) evaluation function, which gives values to non-final game states without considering all possible following complete sequences. We can then limit the minimax algorithm to look only at a certain number of moves ahead. This number is called the "look-ahead", measured in "[plies](http://en.wikipedia.org/wiki/Ply_(chess))". For example, the chess computer [Deep Blue](http://en.wikipedia.org/wiki/IBM_Deep_Blue) (that beat [Garry Kasparov](http://en.wikipedia.org/wiki/Garry_Kasparov)) looked ahead at least 12 plies, then applied a heuristic evaluation function.

The algorithm can be thought of as exploring the [nodes](http://en.wikipedia.org/wiki/Node_(computer_science)) of a [*game tree*](http://en.wikipedia.org/wiki/Game_tree). The *effective*[*branching factor*](http://en.wikipedia.org/wiki/Branching_factor) of the tree is the average number of [children](http://en.wikipedia.org/wiki/Child_node) of each node (i.e., the average number of legal moves in a position). The number of nodes to be explored usually [increases exponentially](http://en.wikipedia.org/wiki/Exponential_growth) with the number of plies (it is less than exponential if evaluating [forced moves](http://en.wikipedia.org/wiki/Forced_move) or repeated positions). The number of nodes to be explored for the analysis of a game is therefore approximately the branching factor raised to the power of the number of plies. It is therefore [impractical](http://en.wikipedia.org/wiki/Computational_complexity_theory#Intractability) to completely analyze games such as chess using the minimax algorithm.

The performance of the naïve minimax algorithm may be improved dramatically, without affecting the result, by the use of [alpha-beta pruning](http://en.wikipedia.org/wiki/Alpha-beta_pruning). Other heuristic pruning methods can also be used, but not all of them are guaranteed to give the same result as the un-pruned search.

The minimax function returns a heuristic value for [leaf nodes](http://en.wikipedia.org/wiki/Leaf_nodes) (terminal nodes and nodes at the maximum search depth). Non leaf nodes inherit their value, *bestValue*, from a descendant leaf node. The heuristic value is a score measuring the favor-ability of the node for the maximizing player. Hence nodes resulting in a favorable outcome (such as a win) for the maximizing player have higher scores than nodes more favorable for the minimizing player. For non terminal leaf nodes at the maximum search depth, an evaluation function estimates a heuristic value for the node. The quality of this estimate and the search depth determine the quality and accuracy of the final minimax result.

### Pseudocode

The [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) for the depth limited minimax algorithm is given below.

**function** minimax(node, depth, maximizingPlayer)

**if** depth = 0 **or** node is a terminal node

**return** the heuristic value of node

**if** maximizingPlayer

bestValue := -∞

**for each** child of node

val := minimax(child, depth - 1, FALSE)

bestValue := max(bestValue, val)

**return** bestValue   
**else**

bestValue := +∞

**for each** child of node

val := minimax(child, depth - 1, TRUE)

bestValue := min(bestValue, val)

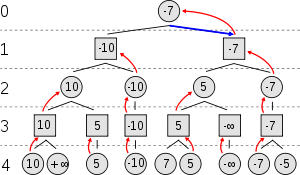
**return** bestValue

minimax(origin, depth, TRUE) *(\* Initial call for maximizing player \*)*

### Example

Suppose the game being played only has a maximum of two possible moves per player each turn. The algorithm generates the [tree](http://en.wikipedia.org/wiki/Game_tree) on the right, where the circles represent the moves of the player running the algorithm (*maximizing player*), and squares represent the moves of the opponent (*minimizing player*). Because of the limitation of computation resources, as explained above, the tree is limited to a *look-ahead* of 4 moves.

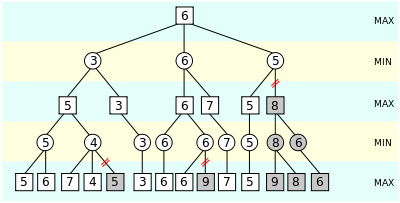
The algorithm evaluates each [*leaf node*](http://en.wikipedia.org/wiki/Leaf_node) using a heuristic evaluation function, obtaining the values shown. The moves where the *maximizing player* wins are assigned with positive infinity, while the moves that lead to a win of the *minimizing player* are assigned with negative infinity. At level 3, the algorithm will choose, for each node, the **smallest** of the [*child node*](http://en.wikipedia.org/wiki/Child_node) values, and assign it to that same node (e.g. the node on the left will choose the minimum between "10" and "+∞", therefore assigning the value "10" to itself). The next step, in level 2, consists of choosing for each node the **largest** of the *child node* values. Once again, the values are assigned to each [*parent node*](http://en.wikipedia.org/wiki/Parent_node). The algorithm continues evaluating the maximum and minimum values of the child nodes alternately until it reaches the [*root node*](http://en.wikipedia.org/wiki/Root_node), where it chooses the move with the largest value (represented in the figure with a blue arrow). This is the move that the player should make in order to *minimize* the *maximum* possible [loss](http://en.wikipedia.org/wiki/Loss_function).

[](http://en.wikipedia.org/wiki/File:Minimax.svg)

**Alpha Beta Pruning**

**Alpha–beta pruning** is a [search algorithm](http://en.wikipedia.org/wiki/Search_algorithm) that seeks to decrease the number of nodes that are evaluated by the [minimax algorithm](http://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves) in its [search tree](http://en.wikipedia.org/wiki/Game_tree). It is an adversarial search algorithm used commonly for machine playing of two-player games ([Tic-tac-toe](http://en.wikipedia.org/wiki/Tic-tac-toe), [Chess](http://en.wikipedia.org/wiki/Chess), etc.). It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

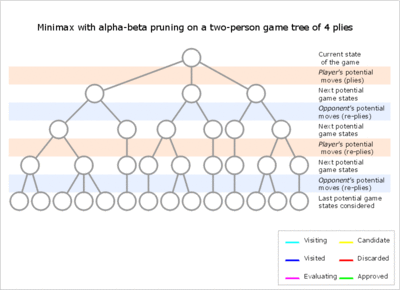
The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the [branch and bound](http://en.wikipedia.org/wiki/Branch_and_bound) class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

[](http://en.wikipedia.org/wiki/File:AB_pruning.svg)

An illustration of alpha–beta pruning. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

Normally during alpha–beta, the subtrees are temporarily dominated by either a first player advantage (when many first player moves are good, and at each search depth the first move checked by the first player is adequate, but all second player responses are required to try to find a refutation), or vice versa. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them.

The algorithm maintains two values, alpha and beta, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity, i.e. both players start with their lowest possible score. It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta<=alpha). If this is the case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored.

[](http://en.wikipedia.org/wiki/File:Minmaxab.gif)

## Pseudocode

**function** alphabeta(node, depth, α, β, maximizingPlayer)

**if** depth = 0 **or** node is a terminal node

**return** the heuristic value of node

**if** maximizingPlayer

**for each** child of node

α := max(α, alphabeta(child, depth - 1, α, β, FALSE))

**if** β ≤ α

**break** *(\* β cut-off \*)*

**return** α

**else**

**for each** child of node

β := min(β, alphabeta(child, depth - 1, α, β, TRUE))

**if** β ≤ α

**break** *(\* α cut-off \*)*

**return** β

***(\* Initial call \*)*** alphabeta(origin, depth, -[∞](http://en.wikipedia.org/wiki/Infinity), +[∞](http://en.wikipedia.org/wiki/Infinity), TRUE)

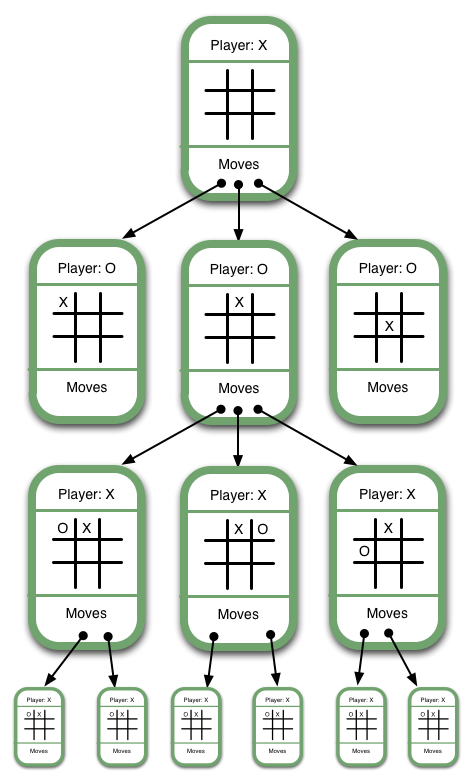
**DESIGN**

The design was started by defining what it means to play a perfect game of tic tac toe:

If I play perfectly, every time I play I will either win the game, or I will draw the game. Furthermore if I play against another perfect player, I will always draw the game.

Representing Moves with the Game Tree Data Structure

Here’s an example of a Game Tree for tic-tac-toe:



Note that this isn’t a full game tree. A full game tree has hundreds of thousands of game states. Some of the discrepancies between the example above and a fully-drawn game tree include:

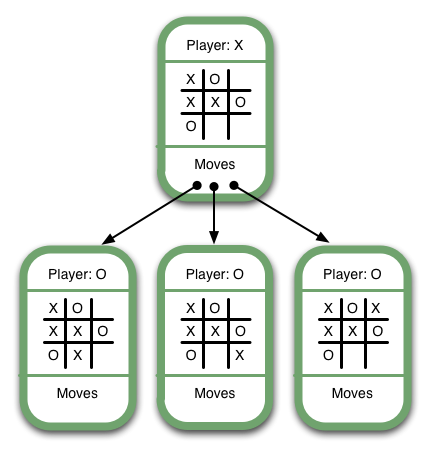
* The first Game State would show nine moves descending from it, one for each of the empty spaces on its board
* Similarly, the next level of Game States would show eight moves descending from them, and so on for each Game State

Representing the game as a game tree allows the computer to evaluate each of its current possible moves by determining whether it will ultimately result in a win or a loss.

Ranking Game States

The basic approach is to assign a numerical value to a move based on whether it will result in a win, draw, or loss. We’ll begin illustrating this concept by showing how it applies to final game states, then show how to apply it to intermediate game states.

Have a look at this Game Tree:

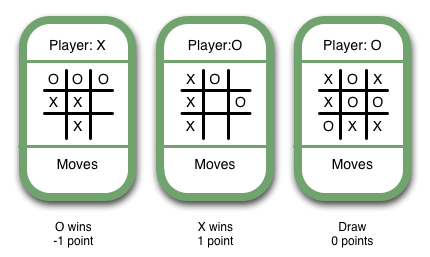


It’s X’s turn, and X has three possible moves, one of which (the middle one) will lead immediately to victory. It’s obvious that an AI should select the winning move. The way we ensure this is to give each move a numerical value based on its board state. Let’s use the following rankings:

* Win: 10
* Draw: 0
* Lose: -10

These rankings are arbitrary. What’s important is that winning corresponds to the highest ranking, losing to the lowest, and a draw’s between the two.

Since the lowest-ranked moves correspond with the worst outcomes and highest-ranked moves correspond with the best outcomes, we should choose the move with the highest value. This is the “max” part of “minimax”. Below are some more examples of final game states and their numerical values:



So now we have a situation where we can determine a possible score for any game end state.

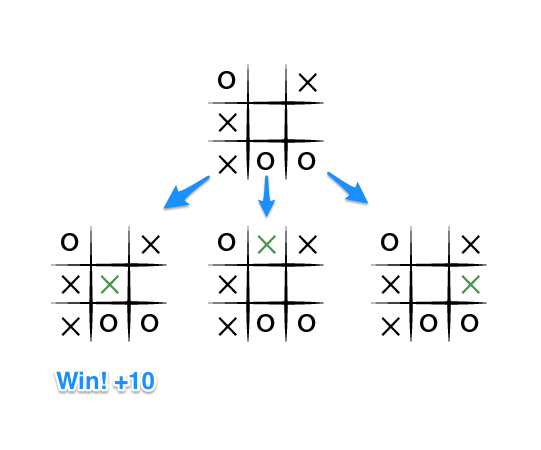
To implement a minimax algorithm for Tic Tac Toe, we keep track of the following:

1. The board state. In this case, where the X’s and O’s are.
2. The current player – the player who will be making the next move.
3. The next available moves. For humans, a move involves placing a game token. For the computer, it’s a matter of selecting the next game state. As humans, we never say, “I’ve selected the next game state”, but it’s useful to think of it that way in order to understand the minimax algorithm.
4. The game state – the grouping of the three previous concepts.

So, a Game Tree is a structure for organizing all possible (legal) game states by the moves which allow you to transition from one game state to the next. This structure is ideal for allowing the computer to evaluate which moves to make because, by traversing the game tree, a computer can easily “foresee” the outcome of a move and thus “decide” whether to take it.

**Looking at a Brief Example**

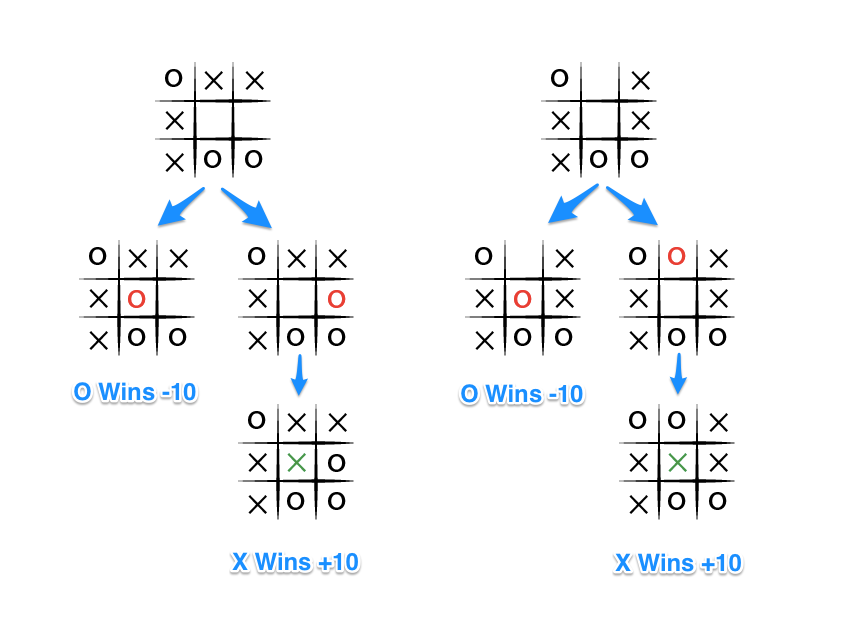
To apply this, let's take an example from near the end of a game, where it is my turn. I am X. My goal here, obviously, is to maximize my end game score.



If the top of this image represents the state of the game I see when it is my turn, then I have some choices to make, there are three places I can play, one of which clearly results in me wining and earning the 10 points. If I don't make that move, O could very easily win. And I don't want O to win, so my goal here, as the first player, should be to pick the maximum scoring move.

## But What About O?

We should assume that O is also playing to win this game, but relative to us, the first player, O wants obviously wants to chose the move that results in the worst score for us, it wants to pick a move that would minimize our ultimate score. Let's look at things from O's perspective, starting with the two other game states from above in which we don't immediately win:



The choice is clear, O would pick any of the moves that result in a score of -10.

## Describing Minimax for Tic Tac Toe

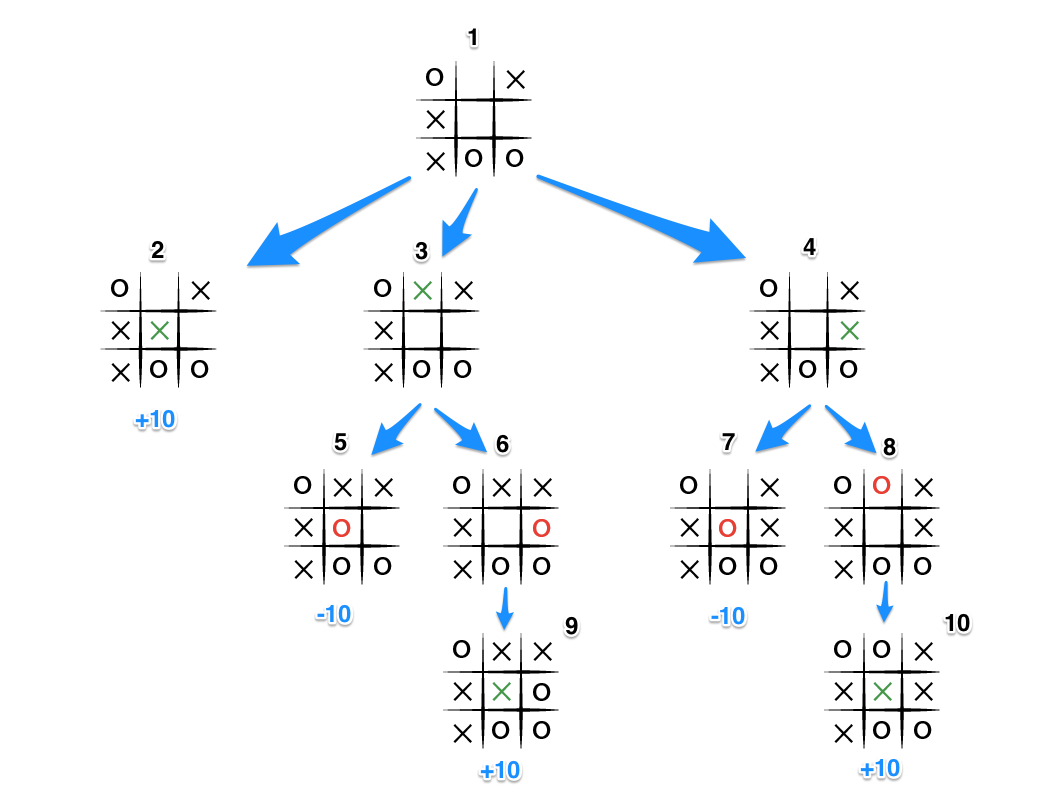
The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn taking player," would look something like:

* If the game is over, return the score from X's perspective.
* Otherwise get a list of new game states for every possible move
* Create a scores list
* For each of these states add the minimax result of that state to the scores list
* If it's X's turn, return the maximum score from the scores list
* If it's O's turn, return the minimum score from the scores list

Let's walk through the algorithm's execution with the full move tree, and show why, algorithmically, the instant winning move will be picked:

* It's X's turn in state 1. X generates the states 2, 3, and 4 and calls minimax on those states.
* State 2 pushes the score of +10 to state 1's score list, because the game is in an end state.
* State 3 and 4 are not in end states, so 3 generates states 5 and 6 and calls minimax on them, while state 4 generates states 7 and 8 and calls minimax on them.
* State 5 pushes a score of -10 onto state 3's score list, while the same happens for state 7 which pushes a score of -10 onto state 4's score list.
* State 6 and 8 generate the only available moves, which are end states, and so both of them add the score of +10 to the move lists of states 3 and 4.
* Because it is O's turn in both state 3 and 4, O will seek to find the minimum score, and given the choice between -10 and +10, both states 3 and 4 will yield -10.
* Finally the score list for states 2, 3, and 4 are populated with +10, -10 and -10 respectively, and state 1 seeking to maximize the score will chose the winning move with score +10, state 2.



**CODING & IMPLEMENTATION**

The game of tic tac toe, along with the artificially intelligent tic tac toe player, as well as the minimax algorithm are implemented in Java. The implementation is divided into 2 parts:

1. The Tic Tac Toe GUI (graphic user interface)
2. The artificially intelligent computer player

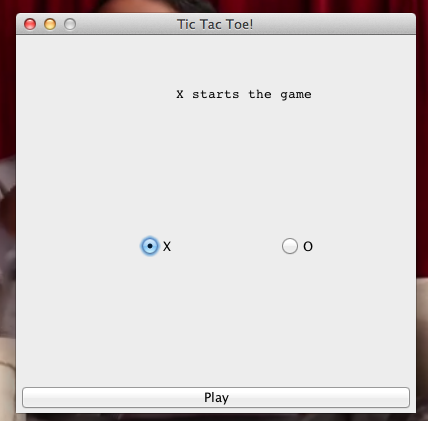
**The Tic Tac Toe GUI**

The Tic Tac Toe GUI has been implemented using the Swing library for Java. S**wing** is a [GUI](http://en.wikipedia.org/wiki/Graphical_user_interface) [widget toolkit](http://en.wikipedia.org/wiki/Widget_toolkit) for [Java](http://en.wikipedia.org/wiki/Java_(programming_language)). It is part of [Oracle](http://en.wikipedia.org/wiki/Oracle_Corporation)'s [Java Foundation Classes](http://en.wikipedia.org/wiki/Java_Foundation_Classes) (JFC) — an [API](http://en.wikipedia.org/wiki/Application_programming_interface) for providing a [graphical user interface](http://en.wikipedia.org/wiki/Graphical_user_interface)(GUI) for Java programs. Swing is a platform-independent, [*Model-View-Controller*](http://en.wikipedia.org/wiki/Model-View-Controller) [GUI](http://en.wikipedia.org/wiki/GUI) framework for Java, which follows a single-[threaded](http://en.wikipedia.org/wiki/Thread_(computing)) programming model. Additionally, this framework provides a layer of abstraction between the code structure and graphic presentation of a Swing-based GUI.

The GUI has been implemented in a single public class, TicTacToe.

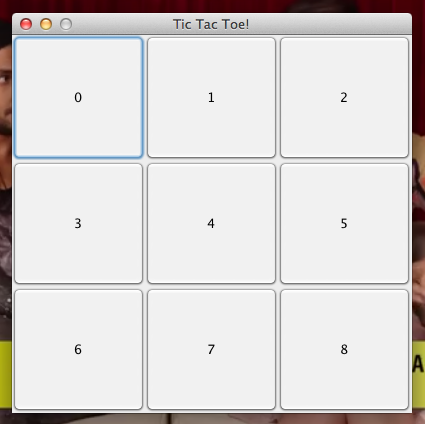
It includes –

**1. The Startup Screen**

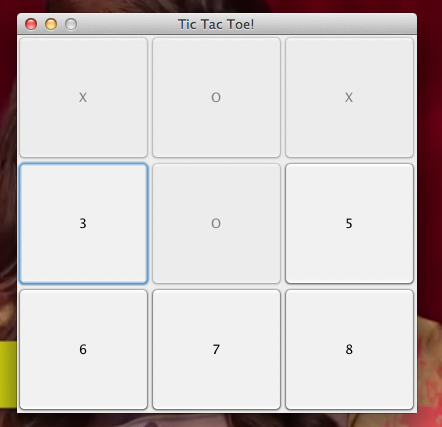


The player can choose to play as ‘X’ or ‘O’. As stated, the first move in the game is played by the user if he chooses ‘X’ or by the computer if he chooses ‘O’ and ‘X’ is allotted to the computer. If the user chooses ‘O’, the first move for the computer is chosen randomly, using Java’s inbuilt random number generator.

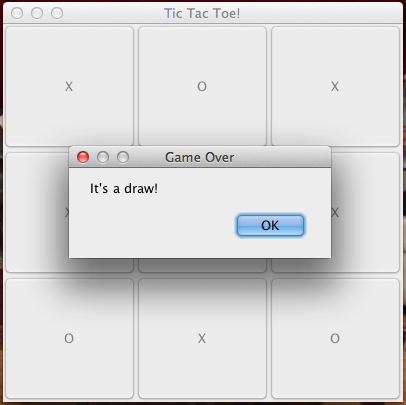
**2. The Game Board**



The 8 buttons represent a standard Tic Tac Toe game board. The player can make a move in the game by clicking any of the 9 buttons. The clicked button is deactivated, and displays the symbol of the user who clicked.



If any player wins, or there’s a tie, a notification is displayed, and the game is restarted.



As in this case, the user is playing against a computer player, every time the user clicks a button to make a move in the game, the move is completed, and after checking for any wins/losses/draws, the control is passed to the computer player to make his move. The computer player makes his move, and the move symbol is changed before returning control to the user.

**The AI computer player**

The AI computer player has been implemented in the public class MiniMax. It receives a character array from the gameboard GUI, depicting the state of the gameboard at that particular instant in the game. The AI works on the principle of Minimax Algorithm along with alpha-beta pruning to find the most optimum next move for the computer player. The Minimax algorithm has been implemented using a function minimax, which keeps track of the path of the solution space currently being explored, on the basis of the index number chosen in the state array, for the insertion of the move symbol. It iteratively scans the game board for empty tiles, and checks whether it has reached a final state based on the hard coded states, which specify if a particular player has won the game.

The computer tries all possible paths that the move symbol can be inserted by iteratively scanning the entire state array for any empty tiles and spaces. As soon as it finds an empty tile, it inserts the move symbol into the empty tile, and checks whether the board has reached a final state. If the computer wins, the function returns immediately to finalize this particular move. Otherwise it passes the new state of the board to a minimize function. This function keeps track of all the possible paths explored for the various possibilities of the game, and returns an index which should be chosen for the computer’s next move, by maximizing the score of all the possible paths explored. All the paths and the scores obtained from these paths is stored in another array, which is traversed to find the maximum score, to return the index with the maximum possible score.

The minimize function explores the various moves the user might make to minimize the computer’s score and win the game. It keeps track of all the further paths explored by similarly scanning for empty tiles in the game board. As soon as it finds an empty tile, it inserts the user symbol into it, and checks if a final state is reached. If it reaches a state where the user wins (-10), it immediately returns the minimum score as -10. If another final state is reached, it returns one step up. Otherwise, it calls the maximize function. The minimize function minimizes the score of the computer, and returns the least value of all the different paths it explores.

The maximize function explores the various moves the computer might make to maximize it’s score and beat the user. It also works in a similar manner to minimize, by iteratively scanning the game board for empty tiles, but returns the maximized score at this step of the solution space, and returns immediately if the final state is a win for the computer (+10). If no final state is reached, it calls the minimize function.

Each path is explored depth first, until we reach a final win/lose or draw state.

**Code**

The class TicTacToe consists of:

1. A global character array, which keeps track of the state of the board at any given instant in the game.

Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 2.28.29 AM.png

1. An object/instance of the computer player.

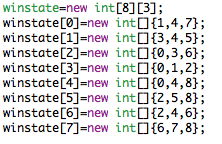
Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 2.32.59 AM.png

1. The GUI builder function for the new game start screen.

1. The GUI builder function for the game board screen.



1. Hard coded states of the board, which represent a win or a loss.



1. An Action Handler class to implement a move when the user clicks a button.



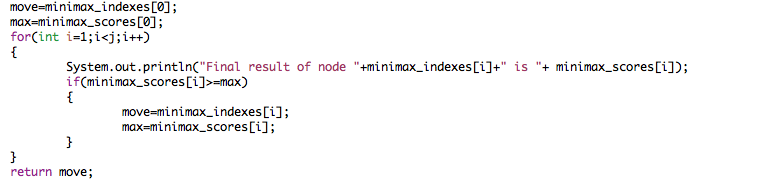
1. A function, which checks to see if the board has reached such a stage after every move.



The class MiniMax consists of:

1. A function minimax which returns the optimum move for the computer





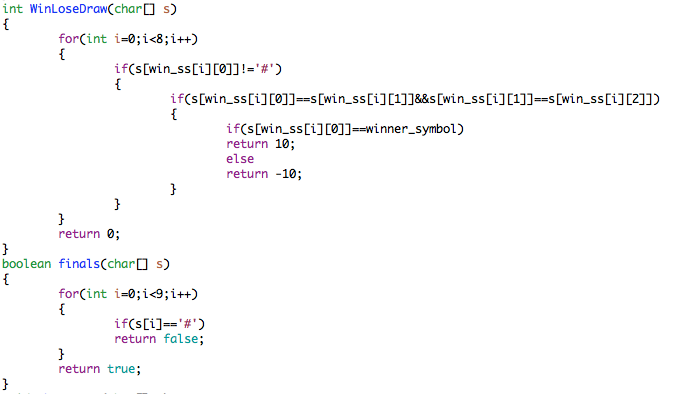
1. A function minimize, which minimizes the score for the computer player



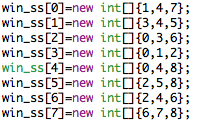
1. A function maximize, which maximizes the score of the computer player



1. Functions to check whether a final state has been raised



1. Hard Coded win states



**A Sample Game**

|  |  |
| --- | --- |
| **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.21.29 AM.png** | **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.21.40 AM.png** |
| **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.21.52 AM.png** | **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.22.02 AM.png** |
| **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.22.15 AM.png** | **Macintosh HD:Users:Perx:Desktop:Screen Shot 2014-11-14 at 12.22.28 AM.png** |
| **CONCLUSION** |  |

Simpler search algorithms tend to utilize a cause-and-effect concept--the search considers each possible action available to it at a given moment; it then considers its subsequent moves from each of those states, and so on, in an attempt to find **terminal states** which satisfy the **goal conditions** it was given. Upon finding a goal state, it then follows the steps it knows are necessary to achieve that state. In a competitive multi-player game, when other agents are involved and have different goals in mind, things get more complicated. Even if a search algorithm can find a goal state, it usually cannot simply take a set of actions, which will reach that state, since for every action our algorithm takes towards its goal, the opposing player can take an action, which will alter the current state.

Thus for these types of two player games like Tic Tac Toe, the **minimax** algorithm is an effective tactic, which uses the fact that the two players are working towards opposite goals to make predictions about which future states will be reached as the game progresses, and then proceeds accordingly to optimize its chance of victory.

A major problem with this approach though, is how pessimistic it is. In trivially small games such as Tic Tac Toe, minimax is useful because it is a reasonable expectation that the computer's opponent can figure out what its best options are; in more complex games, however, this will not be so clear, and a computer running the minimax algorithm may sacrifice major winnings because it assumes its opponent will "see" a move or series of moves which could defeat it, even if those moves are actually quite counterintuitive--in short, the computer assumes it is playing an opponent as knowledgeable as itself.

Still, minimax algorithm proves to be a good choice for implementing an unbeatable Tic Tac Toe player. In all instances, the game was either drawn or won by the computer.

A computer can compute all possible outcomes for a relatively simple game like tic-tac-toe (disregarding symmetric game states, there are 9! = 362,880 possible outcomes), and predict the correct outcome in each case to always play a winning hand.

**FUTURE SCOPE**

In more complex games, an exhaustive use of the minimax algorithm, tends to be hopelessly impractical. Therefore, we use heuristics while implementing the minimax algorithm for more complex games to make it more efficient and reduce redundancy. To limit the amount of information that has to be calculated in more complex games, the depth of the paths explored is limited to a certain number, to make this algorithm practically effective.

Some such techniques are -

**Minimax Regret**

**Regret** (also called **opportunity loss**) is defined as the difference between the actual payoff and the payoff that would have been obtained if a different course of action had been chosen. This is also called **difference regret**. Furthermore, the **ratio regret** is the ratio between the actual payoff and the best one.

The [minimax](http://en.wikipedia.org/wiki/Minimax) regret approach is to minimize the worst-case regret. The aim of this is to perform as closely as possible to the optimal course. Since the minimax criterion applied here is to the regret (difference or ratio of the payoffs) rather than to the payoff itself, it is not as pessimistic as the ordinary minimax approach.

One benefit of minimax (as opposed to expected regret) is that it is independent of the probabilities of the various outcomes: thus if regret can be accurately computed, one can reliably use minimax regret. However, probabilities of outcomes are hard to estimate.

This differs from the standard minimax approach in that it uses *differences* or *ratios* between outcomes, and thus requires interval or ratio measurements, as well as [ordinal measurements](http://en.wikipedia.org/wiki/Ordinal_measurement)(ranking), as in standard minimax.

**Maximin example**

Suppose an investor has to choose between investing in stocks, bonds or the money market, and the total return depends on what happens to interest rates. The following table shows some possible returns:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Return** | **Interest rates rise** | **Static rates** | **Interest rates fall** | ***Worst return*** |
| **Stocks** | −4 | 4 | 12 | *−4* |
| **Bonds** | −2 | 3 | 8 | *−2* |
| **Money market** | 3 | 2 | 1 | *1* |
| *Best return* | *3* | *4* | *12* |  |

The crude maximin choice based on returns would be to invest in the money market, ensuring a return of at least 1. However, if interest rates fell then the regret associated with this choice would be large. This would be −11, which is the difference between the 1 received and the 12 which could have been received if the outturn had been known in advance. A mixed portfolio of about 11.1% in stocks and 88.9% in the money market would have ensured a return of at least 2.22; but, if interest rates fell, there would be a regret of about −9.78.

The regret table for this example, constructed by subtracting best returns from actual returns, is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Regret** | **Interest rates rise** | **Static rates** | **Interest rates fall** | ***Worst regret*** |
| **Stocks** | −7 | 0 | 0 | *−7* |
| **Bonds** | −5 | −1 | −4 | *−5* |
| **Money market** | 0 | −2 | −11 | *−11* |

Therefore, using a minimax choice based on regret, the best course would be to invest in bonds, ensuring a regret of no worse than −5. A mixed investment portfolio would do even better: 61.1% invested in stocks, and 38.9% in the money market would produce a regret no worse than about −4.28.

**Transposition Tables**

In [computer chess](http://en.wikipedia.org/wiki/Computer_chess) and other computer games, **transposition tables** are used to speed up the search of the [game tree](http://en.wikipedia.org/wiki/Game_tree). Transposition tables are primarily useful in [perfect information](http://en.wikipedia.org/wiki/Perfect_information) games, meaning the entire state of the game is known to all players at all times.

The number of positions searched by a computer often greatly exceeds the memory constraints of the system it runs on; thus not all positions can be stored. When the table fills up, less-used positions are removed to make room for new ones; this makes the transposition table a kind of [cache](http://en.wikipedia.org/wiki/Cache_(computing)).

The computation saved by a transposition table lookup is not just the evaluation of a single position. Instead, the evaluation of an entire subtree is avoided. Thus, transposition table entries for nodes at a shallower depth in the game tree are more valuable (since the size of the subtree rooted at such a node is larger) and are therefore given more importance when the table fills up and some entries must be discarded.