A.I.

10
Artificial Intelligence

David: Martin is Mommy and Henry's real son. After I find the Blue Fairy then I can go home. Mommy will love a real boy. The Blue Fairy will make me into one.
Gigolo Joe: Is Blue Fairy Mecha, Orga, man or woman?
David: Woman.
Gigolo Joe: Woman? I know women! They sometimes ask for me by name. I know all about women. About as much as there is to know. No two are ever alike, And after they've met me, no two are ever the same. And I know where most of them can be found.
David: Where?
Gigolo Joe: Rouge City. Across the Delaware.

Dialog between two Artificial Intelligence entities: Gigolo Joe (played by Jude Law) and David (played by Haley Joel Osment) in the movie, *Artificial Intelligence*, Directed by Steven Speilberg, Warner Bros., 2001.

Opposite page: A.I. Artificial Intelligence
From the movie poster. Warner Bros., 2001.

The Question of Intelligence

The quest for the understanding of intelligence probably forms the oldest and yet to be fully understood human inquiry. With the advent of computers and robots the question of whether robots and computers can be as intelligent as humans has driven the scientific pursuits in the field of *Artificial Intelligence (AI)*. Whether a computer can be intelligent was lucidly discussed by Professor Alan Turing in 1950. To illustrate the issues underlying machine intelligence, Turing devised a thought experiment in the form of an *imitation game*. It is played with three people, a man, a woman, and an interrogator. They are all in separate rooms and interact with each other by typing text into a computer (much like the way people interact with each other over IM or other instant messaging services). The interrogator's task is to identify which person is a man (or woman). To make the game interesting, either player can try and be deceptive in giving their answers. Turing argues that a computer should be considered intelligent if it could be made to play the role of either player in the game without giving itself away. This *test* of intelligence has come to be called the *Turing Test* and has generated much activity in the community of AI researchers (see Exercises). The dialog shown above, from the movie *Artificial Intelligence*, depicts an aspect of the test of intelligence designed by Alan Turing. Based on the exchange between Gigolo Joe and David, can you conclude that they are both intelligent? Human?

After over five decades of AI research the field has matured and evolved in many ways. For one, the focus on intelligence is no longer limited to humans: insects and other forms of animals with varying degrees and kinds of intelligence have been the subject of study within AI. There has also been a fruitful exchange of ideas and models between AI scientists, biologists, psychologists, cognitive scientists, neuroscientists, linguists and philosophers. You saw examples of such an influence in the models of Braitenberg vehicles introduced earlier. Given the diversity of researchers involved in AI there has also been an evolution of what AI itself is really about. We will return to this later in the chapter. First, we will give you a few examples of models that could be considered *intelligent* that are commonly used by many AI scientists.

Language Understanding

One aspect of intelligence acknowledged by many people is the use of language. People communicate with each other using a language. There are many (several thousand) languages in use on this planet. Such languages are called *natural languages*. Many interesting theories have been put forward about the origins of language itself. An interesting question to consider is: Can people communicate with computers using human (natural) languages? In other words, can a computer be made to understand language? Think about that for a few moments.

To make the question of language understanding more concrete, think of your Scribbler robot. So far, you have controlled the behavior of the robot by writing C++ programs for it. Is it possible to make the Scribbler understand English so that you could interact with it? What would an interaction with Scribbler look like? Obviously, you would not expect to have a conversation with the Scribbler about the dinner you ate last night. However, it would probably make sense to ask it to move in a certain way. Or to ask whether it is seeing an obstacle ahead.

Do this: Write down a series of short 1-word commands like: `forward`, `right`, `left`, `stop`, etc. Create a vocabulary of commands and then write a program that inputs a command at a time interprets it and makes the Scribbler carry it out. For example:

```
You: forward
Scribbler: starts moving forward…
You: right
Scribbler starts turning right…
You: stop
…
```

Experiment with the behavior of the robot based on these commands and think about the proper interpretation that may make its behavior more natural.

You will find yourself making several assumptions about interpretation of even the simplest commands in the exercise above. For example, what happens when after you command the Scribbler to move forward, you ask it to turn right? Should the Scribbler stop going forward or should it stop and then start turning?

Decisions like these also give deep insights into our own abilities of understanding language. You can also see that, as in the case of visual perception, processing of language (or text) begins at a very primitive level: words. If the input is speech, the basic units are electrical signals, perhaps coming from a microphone. Just like processing individual pixels to try and understand the contents of an image, one has to start at a low level of representation for beginning to understand language.

Researchers working in the field of *computational linguistics* (or *natural language understanding*) have proposed many theories of language processing that can form the basis of a computational model for a Scribbler to understand a small subset of the English language. In this section, we will examine one such model which is based on the processing of syntax and semantics of language interaction. Imagine, interacting with the Scribbler using the following set of sentences:

```
You: do you see a wall?
Scribbler: No

You: Beep whenever you see a wall.
You: Turn right whenever you see a wall to your left.
You: Turn left whenever you see a wall to your right.
You: Move for 60 seconds.

[The Scribbler robot moves around for 60 seconds turning
whenever it sees a wall. It also beeps whenever it sees a
wall.]
```

Earlier, you have written C++ programs that perform similar behaviors. However, now imagine interacting with the robot in the fashion described. From a physical perspective, imagine that you are sitting in front of a computer, and you have a Bluetooth connection to the robot. The first question then becomes: Are you actually speaking or typing the above commands? From an AI perspective, both modalities are possible: You could be sitting in front of the computer and speaking into a microphone; or you could be typing those commands on the keyboard. In the first instance, you would need a speech understanding capability. Today, you can obtain software (commercial as well as freeware) that will enable you to do this. Some of these systems are capable of distinguishing accents, intonations, male or female voices etc. Indeed, speech and spoken language understanding is a fascinating field of study that combines knowledge from linguistics, signal processing, phonology, etc.

You can imagine that the end result of speaking into a computer is a piece of text that transcribes what you said. So, the question posed to the Scribbler above: *Do you see a wall?* will have to be processed and then transcribed into text. Once you have the text, that is, a string `"Do you see a wall?"` it can be further processed or analyzed to understand the *meaning* or the content of the text. The field of *computational linguistics* provides many ways of syntactic parsing, analyzing, and extracting meaning from texts. Researchers in AI itself have developed ways of representing knowledge in a computer using symbolic notations (e.g. *formal logic*). In the end, the analysis of the text will result in a `getIR()` or `getObstacle()` command to the Scribbler robot and will produce the response shown above.

Our goal of bringing up the above scenario here is to illustrate to you various dimensions of AI research that can involve people from many different disciplines. These days, it is entirely possible even for you to design and build computer programs or systems that are capable of interacting with robots using language.

Game Playing

In the early history of AI, scientists posed several challenging tasks which if performed by computers could be used as a way of demonstrating the feasibility of machine intelligence. It was common practice to think of games in this realm. For example, if a computer could play a game, like chess, or checkers, at the same level or better than humans we would be convinced into thinking that it was indeed feasible to think of a computer as a possible candidate for machine intelligence. Some of the earliest demonstrations of AI research included attempts at computer models for playing various games. Checkers and chess seemed to be the most popular choices, but researchers have indulged themselves into examining computer models of many popular games: Poker, Bridge, Scrabble, Backgammon, etc.

Two-person zero-sum games: Chess, Tic Tac Toe, Konane

In many games, it is now possible for computer models to play at the highest levels of human performance. In Chess, for example, even though the earliest programs handily beat novices in the 1960's, it wasn't until 1996 when an IBM computer Chess program, named Deep Blue, beat the world champion Gary Kasparov at a tournament-level game, though Kasparov did manage to win the match 4-2. A year later, in New York, Deep Blue beat Kasparov in a 6 game match representing the very first time a computer defeated the best human player in a classical style game of Chess. While these accomplishments are worthy of praise it also now clear that the quest for machine intelligence is not necessarily answered by computer game playing. This has resulted in much progress in game playing systems and game playing technology which is now a multi-billion dollar industry.

It turns out that in many Chess-like games the general strategy for a computer to play the game is very similar. Such games are classified as two-person zero-sum games: two people/computers play against each other and the result of the game is either a win for one player and loss for the other, or it is a draw.

In many such games, the basic strategy for making the next move is simple: look at all the possible moves I have and for each of them all the possible moves the other player might have and so on until the very end. Then, trace back from wins (or draws) and make the next move based on those desirable outcomes. You can see this even in simple games like Tic Tac Toe where it is easy to mentally look ahead future moves and then make more informed decisions about what to do next. The best way to understand this is to actually write a program that plays the game.

Tic Tac Toe

Also known as *Noughts and Crosses* or *Hugs and Kisses*, Tic Tac Toe is a popular children's game (see description on Wikipedia under Tic-tac-toe). We will develop a program that can be used to play this game against a person. Almost any board game can be programmed using the basic loop shown below:

```
void play() {
    // Initialize board
    Board board = makeBoard();

    // set who moves first/next: X always moves first
    char player = 'X';

    // Display the initial board
    display(board);

    // The game loop
    while (winner(board) == " " &&
           !gameOver(board, player)) {
        move(board, player);
        display(board);
        player = opponent(player);
    }

    // game over, show outcome
    string winningPiece = winner(board);
    if (winningPiece != "Tie")
        cout << winningPiece << " won.\n";
    else
        cout << "It is a tie.\n";
}
```

The function above can be used to play a round of any two-person board game. The variable `player` is the player (or piece) whose move is next. We are already using the Tic Tac Toe piece ‘X’ in the function above. Six basic functions (shown highlighted above) make up the basic building blocks of the game. For Tic Tac Toe, they can be defined as:

1. **`makeBoard()`**: Returns a fresh new board representing the start of the game. For Tic Tac Toe, this function will return an empty board representing the nine squares.
2. **`displayBoard(board)`**: Displays the board on the screen for the user to see. The display can be as simple or elaborate as you wish. It is good to start with the easiest one you can write. Later you can make it fancier.
3. **`opponent(player)`**: Returns the opponent of the current player/piece. In Tic Tac Toe, if the player is X, it will return an O, and vice versa.
4. **`move(board, player)`**: Updates the board by making one move for the player. If the player is the user, it will input the move from the user. If the player is the computer, it will decide how to make the best move. This is where the smarts will come in.
5. **`gameOver(board)`**: Returns `true` if there are no more moves left to be made, `false` otherwise.
6. **`winner(board)`**: Examines the board and returns the winning piece or that the game is not yet over, or that it is a tie. In Tic Tac Toe, it will return either an X, O, a blank (representing game is not over yet), or a TIE.

We will need a few more functions to complete the game but these six form the basic core. We will write them first.

The board itself consists of nine squares that start out being empty. Players choose a game piece: an 'X' or an 'O'. We will assume that 'X' always goes first. The first thing to do then is to design a representation of the game board. We will use the following simple representation:

```
typedef vector<char> Board;
Board board (9, ' ');
```

The first line allows `Board` to be used as an abbreviation for the type `vector<char>`. The second line defines `board` to be a vector of 9 single characters initialized to blanks. Note that we are using this linear representation of the board instead of a 2-dimensional one. However, as you will see, this representation makes it easier to do many manipulations for the game. During play, the board can be displayed in its natural format. Below, we show two functions: one creates a fresh new board each time it is called; and one displays it:

```
Board makeBoard() {
    /* A 3x3 board is represented as a list of 9 elements.
     * We will use the following numbering to locate a square
     *  0 | 1 | 2
     * ---|---|---
     *  3 | 4 | 5
     * ---|---|---
     *  6 | 7 | 8
     */
    return Board (9, ' ');
}

void display (Board board) {
    for (int i = 0; i < 9; i += 3) {
        if (i > 0)
            printf ("---|---|---\n");
        printf (" %c | %c | %c \n",
          board[i], board[i+1], board[i+2]);
    }
    cout << endl;
}
```

One advantage of writing the display function as shown is that it gives us a quick way of creating and displaying the game. Later, when you are done, you can write a fancier version that displays the game graphically (see Exercises). With the above functions, we can easily create a fresh new board and display it as follows:

```
Board board = makeBoard();
display(board);
```

To determine the opponent of a given piece is simple enough:

```
char opponent(char player) {
    if (player == 'X')
        return 'O';
    else
        return 'X';
}
```

Next, we have to write the function `move`. It first determines whose move it is. If it is the user's move, it will input the move from the user. Otherwise, it will determine the best move for the computer. Then it will actually make the move. As a first design, we will let move make a random choice out of the possible moves for the computer. Later, we will make a more informed decision.

```
#include <cstlib>
using namespace std;

char You = 'X';
char Me = 'O';

void move(Board& board, char player) {
    int square;

    if (player == You) {   // user's move?
        cout << "Enter your move: ";
        cin >> square;
        square--;
    } else                 // my turn
```

```
        // player is the computer, make a random choice
        square = choice(possibleMoves(board, player));
    // place player's piece on the chosen square
    applyMove(board, player, square);
}
```

Notice the ampersand `&` after type `Board` in the first line of `move`. This means that when we call `move`, the computer will not make a fresh copy of the board for `move` to use (which is what usually happens), but will pass a *reference* to (the address of) the board in the caller. This is because the purpose of `move` is to make a move, which changes the board, and so we want to change the original game board and not a local copy of it. This way of passing a parameter to a function is called *pass by reference* (as opposed to the usual *pass by value*).

We will leave it as an exercise for you to define an integer function `choice(vec)` that returns a random element of the integer vector `vec`.

We have set the global variables `You` and `Me` to specific pieces. This is a simplification for now. Later you can come back and rewrite them so that for each game, the user gets to select their piece. In Tic Tac Toe, X always moves first. So by making the user's piece X, we have made an assumption that the user always goes first. Again, later we can come back and modify this (see exercises). Also notice that we are not doing any error checking in user input to ensure that a legal move was input (see exercises).

The user inputs their move by entering a number from 1..9 where the square numbering is as shown below. This is slightly different from our internal numbering of squares and more natural for people. In the `move` function above, we subtract 1 from the input number so it maps to the proper square in our internal scheme.

```
 1 | 2 | 3
---|---|---
 2 | 5 | 6
---|---|---
 7 | 8 | 9
```

Again, we have simplified the interface for now. The exercises suggest how to improve on this design.

The `move` function defined above requires two additional functions (shown highlighted). These are also core functions in any board-based game and are described below:

7. **`possibleMoves(board, player)`**: Returns a list of possible moves for the given player.
8. **`applyMove(board, player, square)`**: Given a specific square and a player/piece, this function actually applies the move on the board. In Tic Tac Toe all one has to do is actually place the piece in the given square. In other games, like Chess or Checkers, there may be pieces that get removed.

In Tic Tac Toe all empty squares are possible places where a player can move. Below, we write a function that, given a board, returns a list of all the possible locations where a piece can be placed:

```
vector<int> possibleMoves(Board board, char player) {
    vector<int> moves (0);
    for (int i = 0; i < 9; i++)
        if (board[i] == ' ')
            moves.push_back(i);
    return moves;
}
```

To complete the game playing program, we need to write two more functions defined above. The `winner` function examines the board and determines who won. It returns the winning piece (an `"X"`, `"O"`) or the string `"Tie"`. In case the game is not yet over, it returns `" "`. Below, we first define all the winning positions in Tic Tac Toe, based on our board representation. Next, we define the function itself to examine the board.

```
// These square triples represent wins (three in a row).

int Wins[][3] = {{0, 1, 2},{3, 4, 5},{6, 7, 8}, // the rows
                 {0, 3, 6},{1, 4, 7},{2, 5, 8}, // the columns
                 {0, 4, 8},{2, 4, 6}};          // diagonals

string winner(Board board) {
    for (int win = 0; win < 8; win++) {
        char posWinner = board[Wins[win][0]];
        if (posWinner != ' ' &&
            posWinner == board[Wins[win][1]] &&
            posWinner == board[Wins[win][2]])
            return string(1, posWinner);
    }

    // No winner yet, are there empty squares left?
    for (int i = 0; i < 9; i++)
        if (board[i] == ' ')
            return " ";
    // The board is full and no one has three in a row
    return "Tie";
}
```

The declaration of `Wins` illustrates a C++ feature that you have seen before (ch. 7), but uses it in a new way. It declares `Wins` to be a two-dimensional array; `[][3]` means that it has three columns but the number of rows is determined by the following initialization, which lists the contents of the array row by row. In the body of `winner`, expressions such as `Wins[win][1]` refer to row `win` and column `1` of `Wins`.

Last, the `gameOver` function can be written either by relying on the fact that winner returns `" "` when the game is not yet over. Alternately, we can write it using `possibleMoves` as follows:

```
bool gameOver(Board board, char player) {
    return possibleMoves(board).size() == 0;
}
```

Implement `applyMOVE` and you have all the ingredients for a program to play a game of Tic Tac Toe. The version above is simplified in many ways, and yet it captures the essential elements of playing a round of Tic Tac Toe.

Do This: Implement the program above and play a few rounds of Tic Tac Toe. When you play against the computer, you are anticipating possible moves down the road and then playing your own moves with those in mind.

You probably had no problems beating the computer at the game. That is because the computer is just picking its moves by random from a set of possible moves:

```
// player is the computer, make a random choice
square = choice(possibleMoves(board, player));
```

This is the line in `move` where we can put some intelligence into the program. However, the basic question that needs to be asked is: which of the possible moves is the best one for me? Think about this for yourself a little. In Tic Tac Toe, you play in two modes: defensively so as not to lose after the next move, or offensively to try and win. If a win is imminent for you, you will of course make that move, but if it isn't (and neither is a loss) how do you select your move? We will try and generalize this idea next in a way that will also be applicable to other board games.

Let us delegate the responsibility of finding the best move to a function: `bestMove` that will return the best move from among a set of possible moves. That is, we can change the line in `move` as follows:

```
// player is the computer, make the best choice
square = bestMove(board, player,
                  possibleMoves(board, player));
```

If `bestMove` were to make a random choice (as above) we could write it as:

```
int bestMove(Board board, char player, vector<int> moves) {
    return choice(moves);
}
```

Imagine that you decided to go watch two players playing a game of Chess (or Tic Tac Toe). However, you get there in the middle of the game. You walk into the room, look at the board, and are able to evaluate how each player is doing without knowing anything about how the board came to be. For example, take a look at the Tic Tac Toe boards:

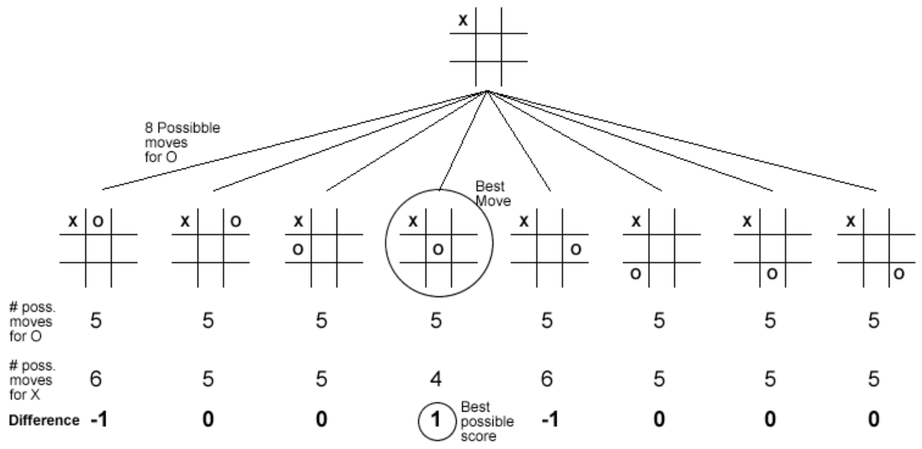
```
 X | O | X        O | X | X        X | O | X        X |   |
---|---|---      ---|---|---      ---|---|---      ---|---|---
   | O |            |   |          O | X | O          |   |
---|---|---      ---|---|---      ---|---|---      ---|---|---
 O | X | X        X |   | O          | X |            |   |

     1                2                3                4
```

In all the cases above, the computer is playing O and it is O's turn next. Examine each board and think about what the best move for O would be.

In case 1, the computer has to play defensively and place an O in square 6 to avoid losing the game. In case 2, it can be on the offensive and recognize that it will win the game by placing an O in square 5. In case 3, it has to play to avoid losing. In the last case, the board is wide open. Of the eight possible choices, is there a best move? From experience, you are probably going to place the O in the center square (square 5). Why? Let us elaborate this by looking ahead (see Figure on next page).

Think about how we could quantify each of the above board positions with respect to O's chances of winning. For each board, count the number of possible winning positions still remaining for O and compare those for X. For example, in the first board, O has 5 possible winning positions, and X has 6. We could say that O is 5 – 6 = –1 has one less possibility than X. If you did this for the second and the third boards, you will see that both X and O are even in the number of possible remaining winning positions (5 each). However, in the fourth case, O has 5 possible winning positions remaining, and X has only 4. Try and work out the remaining board positions to confirm the scores shown above. Clearly, you can see that placing O in square 5 is the best of all the possibilities. Try and elaborate the 5 possible moves for O in



case 2 above. You will find that one of the five leads to a win for O. Similarly, when you elaborate the two options for O in case 1, you will notice that one leads to a tie and the other to a loss for O. We can capture all these situations in a function that takes a look at the board and returns a number representing a score in favor of O: the higher the value returned, the more desirable the board.

$$\texttt{evaluate(board)} = \begin{cases} \infty, & \text{if board is a win for computer} \\ -\infty, & \text{if board is a loss for computer} \\ \texttt{openWins(computer)} - \texttt{openWins(user)}, & \text{otherwise} \end{cases}$$

That is, a win for the computer receives a high score, a loss receives a low score. Otherwise, it is the difference in the number of open wins remaining for each. We can capture this in C++ as follows:

```
int INFINITY = 10;
int evaluate(Board board, char player) {
    // if board is a win for player, return INFINITY
    char piece = winner(board, player);
    if (piece == Me)
        return INFINITY;
    else if (piece == You)
        return -INFINITY;
    else
        return openWins(board,Me) - openWins(board,You);
}
```

We define `INFINITY` as 10, a large enough number relative to the other values that might be returned by `evaluate`. `openWins` looks at each winning triple on the board and counts the number of openings for the given player.

```
int openWins(Board board, char player) {
    int possWins = 0;
    for (int pos = 0; pos < 8; pos++) {
        int n = 0;
        for (int i = 0; i < 3; i++)
            if ((board[Wins[pos][i]] == player) ||
                (board[Wins[pos][i]] == ' '))
                n++;
        if (n == 3) possWins++;
    }
    return possWins;
}
```

Do This: Implement the two functions above and then test them. Create several board positions (use the ones from examples above) and confirm that the board is being evaluated to the correct values.

Next, we can rewrite `bestMove` to take advantage of `evaluate`:

```
int bestMove(Board board, char player, vector<int> moves) {
    vector<int> scores (0);
    for (m = 0; m < moves.size(); m++) {
        Board b (board);
        applyMove(b, player, moves[m]);
        scores.push_back(evaluate(b));
    }
    return moves[maxIndex(scores)];
}
```

The declaration `Board b (board)` initializes `b` to be a fresh copy of `board`, so that we won't affect the original board while we are trying out possible moves. Notice how `bestMove` takes each possible move, creates a new board with that move, and then evaluates the score of the resulting board. It pushes each evaluation on the back (end) of the `scores` vector, which is initially empty. Finally, it returns the move with the highest score as the best move. Modify your program from above to use this version of `bestMove` (you will need to program `maxIndex(V)`, which returns the index (position) of the maximum element of `V`). Play the game several times. You will notice that there is a significant improvement in the computer's playing ability. Can you measure it in some way? (See Exercises).

The above rewrite of `bestMove` will make the program play significantly better but there is still more room for improvement. In most board games good players are able to mentally picture the game several moves ahead. In many games, like Chess, certain recognizable situations lead to well determined outcomes and so a great part of playing a successful game also relies on the ability to recognize those situations.

Looking ahead several moves in a systematic manner is something computers are quite capable of doing and hence anyone (even you!) can turn them into fairly good players. The challenge lies in the number of moves you can look ahead with limited memory capacity and, if time to make the next move is limited, how to choose among the best available options. These decisions lend interesting character to computer game programs and continue to be a constant source of fascination for many people. Let's us look at how your Tic Tac Toe program can easily look at all the possible moves all the way to the end of game in determining its next move (which, in most situations leads to a draw, given the simplicity of the game).

When you look ahead a few moves, you take into account that your opponent is going to try and beat you at every move. Your program, in trying to select the best move, can look ahead at the opponent's moves and take that into consideration when choosing its best move. In fact, it can go further, all the way to the end. The `evaluate` function we wrote above can be used effectively to evaluate future board situations by assuming that when it is the computer's move, it will always try to pick the move that promises the highest score in the future. However, when it examines the opponent's moves, it has to assume that the opponent is going to make the move that is worst for the computer. In other words, when looking ahead, the computer is going to maximize its possible score while the opponent is going to minimize the computer’s chances to win. This can be captured in the following:

```
int lookahead(Board board, char player, bool MAX, int level) {

   vector<int> moves = possibleMoves(board, player);
   if (level <= 0 || moves.size()==0) // limit of look ahead
      return evaluate(board, player);

   int V;
   if (MAX) {                       // computer’s move
      V = -INFINITY;
      for (int m = 0; m < moves.size(); m++) {
         Board b (board);
         b[moves[m]] = player;
         V = max(V,
              lookahead(b, opponent(player), !MAX, level-1));
      }
    } else {                        // opponent’s move
      V = INFINITY;
      for (int m = 0; m < moves.size(); m++) {
         Board b (board);
         b[moves[m]] = player;
         V = min(V,
              lookahead(b, opponent(player), !MAX, level-1));
      }
    }
    return V;
}
```

The `lookahead` function defined above takes the current board, the player whose turn it is, whether the player is the computer (one trying to maximize its outcome) or the opponent (one trying to minimize the computer’s outcomes), and the levels still to look ahead, and computes a score based on examining all the moves going forward to the limit of look ahead. In the above function when `MAX` is `true` it represents the computer and `false` represents its opponent. Thus, depending on the value of `MAX`, the evaluation is minimized or maximized accordingly. Each time it looks ahead further, the level is reduced by 1. The final value returned by `lookahead` can be used by `bestMove` as follows:

```
int LEVEL = 9;
int bestMove(Board board, char player, vector<int> moves) {

    vector<int> scores (0);
    for (int m = 0; m < moves.size(); m++) {
        Board b (board);
        b[moves[m]] = player;
        scores.push_back(
          lookahead(b, opponent(player), false, LEVEL-1));
    }

    return moves[maxIndex(scores)];
}
```

As before, the move with the highest value is considered the best move. We have set the value of `LEVEL` above at 9 (i.e. look 9 moves ahead!) implying that each time it will look as far as the end of the game before making the decision. There can only be a maximum of 9 moves in a Tic Tac Toe game. The quality of the computer's decision maker can in fact be adjusted by lowering the value of `LEVEL`. At `LEVEL = 1`, it will be equivalent to the version we wrote earlier that only used the `evaluate` function.

How many levels ahead one looks in a game like this can depend on the game itself. Can you guess how many board situations the computer will have to look at in doing a look ahead at `LEVEL = 9` after the user's first move? It will be 40,320 different board situations! Why? Additionally, by the time it is the computer's second move, it will only need to look at 720 board positions. This is because, in Tic Tac Toe, as the board gets filled, there are fewer possible moves remaining. In fact, by the time it is the computer's third move, it only needs to look at a total of 24 boards. And, if the computer makes it to its fourth move, it will only have to look at two possible moves. Thus, an exhaustive search for all the possible board positions until the end of the game each time the computer has to make a move it will be examining a total of 41,066 board positions. However, if you consider a typical game of Chess, in which each player makes an average of 32 moves and the number of feasible moves available at any time averages around 10, you would soon realize that the computer would have to examine something of the order of 10^{65} board positions before making a move! This, even for the fastest computers available today, will take several gazillion years! More on that later. But, to play an interesting two-person zero-sum game, it is not essential to look so far ahead. You can vary the amount of look ahead by adjusting the value of `LEVEL` in such programs.

Do This: Implement `lookahead` as described above and compare how well the computer plays against you. Try and vary the levels from 1, 2, …, to see if there is any improvement in the computer's play. Would you consider this program intelligent?

The exercises at the end of the chapter will guide you in transforming the above program into a more robust and even efficient game playing program for Tic Tac Toe. However, study the program structure carefully and you will be able to use the same strategy, including much of the core of the program, to play many other two-person board games.

Smarter Paper Scissors Rock

In Chapter 7, you saw an example of a program that played the game of Paper Scissors Rock against a human user. In that version, the program's choice strategy for picking an object was completely random. We reproduce that section of the program here:

```
…
string items[] = {"Paper", "Scissors", "Rock"};

…
// Computer makes a selection
string myChoice = items[rand() % 3];
…
```

In the above program segment, `myChoice` is the program's choice. As you can see, the program uses a random number to select its object. That is, the likelihood of picking any of the three objects is 0.33 or 33%. The game and winning strategies for this game have been extensively studied. Some strategies rely on detecting patterns in human choice behavior. Even though we may not realize it there are patterns in our seemingly random behavior. Computer programs can easily track such behavior patterns by keeping long histories of player's choices, detect them, and then design strategies to beat those patterns. This has been shown to work quite effectively. It involves recording player's choices and searching through them. Another strategy is to study human choice statistics in this game. Before we present you with some data, do the exercise suggested below:

Do This: Play the game against a few people, Play several dozen rounds. Record the choices made by each player (just write a P/S/R in two columns). Once done, compute the percentages of each object picked. Now read on.

It turns out that most casual human players are more prone towards picking Rock than Paper or Scissors. In fact, various analyses suggest that 36% of the time people tend to pick Rock, 30% Paper, and 34% Scissors. This suggests that RPS is not merely a game of chance there is room for some strategies at winning. Believe it or not, there are world championships of RPS held each year. Even a simple game like this has numerous possibilities. We can use some of this information, for instance, to make our program smarter or better adept at playing the game. All we have to do is instead of using a fair 33% chance of selecting each object we can skew the chances of selection based on people's preferences. Thus, if 36% of the time people tend to pick Rock, it would be better for our program to pick Paper 36% of the time since Paper beats Rock. Similarly, our program should pick Scissors 30% of the time to match the chance of beating Paper, and pick Rock 34% of the time to match the chances of beating Scissors. We can bias the random number generator using these percentages as follows:

First generate a random number in the range 0..99
If the number generated is in the range 0..29, select Scissors (30%)
If the number generated is in the range 30..63, select Rock (34%)
If the number generated is in the range 64..99, select Paper (36%)

The above strategy of biasing the random selection can be implemented as follows:

```
string mySelection() {

    // First generate a random number in the range 0..99
    int n = rand() % 100;

    // If the n is in range 0..29, select Scissors
    if (n <= 29)
        return "Scissors";
    else if (n <= 63)
        // if n in range 30..63, select Rock
        return "Rock";
    else
        return "Paper";
}
```

Do This: Modify your RPS program from Chapter 7 to use this strategy. Play the game several times. Does it perform much better that the previous version? You will have to test this by collecting data from both versions against several people (make sure they are novices!).

Another strategy that people use is based upon the following observation:

After many rounds, people tend to make the move that would have beaten their own previous move.

Say a player picks Paper. Their next pick will be Scissors. A computer program or a player playing against this player should then pick Rock to beat Scissors. Note that the relationship between the choices is cyclical. Paper beats Rock, Rock beats Scissors, and Scissors beat Paper. Therefore, since the player's previous move was Paper, your program can pick Rock in anticipation of the player's pick of Scissors. Try to think over this carefully and make sure your head is not spinning by the end of it. If a player can spot this they can use this as a winning strategy. We will leave the implementation the of this strategy as an exercise. The exercises also suggest another strategy.

The point of the above examples is that using strategies in your programs you can make your programs smarter or more intelligent. Deliberately, we have started to use the term intelligence a little more loosely than what Alan Turing implied in his famous essay. Many people would argue that these programs are not intelligent in the ultimate sense of the word. We agree. However, writing smarter programs is a natural activity. If the programs incorporate strategies or heuristics that people would use when they are doing the same activity, then the programs have some form of artificial intelligence in them. Even if the strategy used by the program is nothing like what people would use, but it would make the program smarter or better, we would call it artificial intelligence. Many people would disagree with this latter claim. To some, the quest for figuring out intelligence is limited to the understanding of intelligence in humans (and other animals). In AI both points of view are quite prevalent and make for some passionate debates among scholars.

Discussion

The very idea of considering a computer as an intelligent device has its foundations in the general purpose nature of computers. By changing the program the same computer can be made to behave in many different ways. At the core of it a computer is just a symbol manipulator: manipulating encodings for numbers, or letters, or images, etc. It is postulated that the human brain is also a symbol manipulator. The foundations of AI lie in the fact that most intelligent systems are physical symbol systems and since a computer is a general purpose symbol manipulator, it can be used for studying or simulating intelligence.

Myro Review

No new Myro functions were introduced in the chapter.

C++ Review

`TYPE & NAME`
Indicates that the specified parameter is to be passed by reference (rather than passed by value), which allows it to be modified from within the function.

`typedef TYPE NAME`
Allows `NAME` to be used as an abbreviation for `TYPE`.

Exercises

1. Read Alan Turing's paper Computing Machinery and Intelligence. You can easily find a copy of it by searching on the web.

2. Do a web search for "Searle Chinese Room argument" to locate Philosopher John Searle's arguments that no matter how intelligent a computer or a program gets, it will never have a "mind".

3. Rewrite display for Tic Tac Toe game to display the board graphically.

4. Design a language of one-word English commands for the Scribbler. Write a program to input one command at a time, interpret it, and then execute the command on the Scribbler.

5. Extend the language from Exercise 4 to include queries (e.g. wall?) and then modify your program to incorporate such queries.

6. Do a survey of speech understanding systems.

7. Do a survey of computational linguistics.

8. In the Tic Tac Toe program designed in this Chapter, we assumed that the user always plays an X. Modify your program so that it gives the user a choice at the beginning of the game. Further, at the end of each game, the pieces are swapped.

9. In the function `move` defined for Tic Tac Toe, the program accepts whatever the user inputs for their move. Try the program and instead of entering a valid move, enter your name instead. What happens? Such an error might be easily detected since it will halt the program's execution. However, next try entering a number from 1-9 using a spare position that is already occupied in the board. What happens? Modify the function to accept only correct moves. If the user enters an incorrect move, the program should point that out and give the user another chance.

10. The function `gameOver` can make use of the `winner` function to make its decision in the Tic Tac Toe program. Rewrite `gameOver` to do this.

11. One way to measure how one strategy compares against another is to play it against another strategy over and over again recording the number of wins, losses, and draws. Modify your Tic Tac Toe or RPS program to substitute the second strategy for the user (instead of taking input from the user, it uses a function that implements the second strategy. Add statements to play the game many times (say 1000) and record the wins, losses, and draws. You may also want to suppress all board output since the game is being played completely inside the program. Do a comparison of all the strategies discussed in this Chapter and how they compare against each other.