A project report submitted in partial fulfilment of the award of degree of B.Tech on

APPLICATION OF GENETIC ALGORITHM IN WALL FOLLOWING

GROUP NO.66

Group Members:
KunalChawla(071251)
NileshBansal(071318)

Project Supervisor:
Mr.Pradeep Kumar

JAYPEE UNIVERSITY OF INFORMATION AND TECHNOLOGY
WAKNAGHAT, SOLAN (H.P)

Table of Contents:

WAKNAGHAT
SOLAN, HIMACHAL PRADESH

Date: 23-5-2011

CERTIFICATE

This is to certify that the work titled "APPLICATION OF GENETIC ALGORITHM IN WALL FOLLOWING" submitted by "Nilesh Bansal and Kunal Chawla" in partial fulfilment for the award of degree of B. Tech of Jaypee University of Information Technology; Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature in full of Supervisor: Pardeep Kumar.

Name in Capital block letters: PARDEEP KUMAR

Designation: Sr. Lecturer

ACKNOWLEDGEMENT

It gives us great pleasure in presenting the project report for our project on '**APPLICATION OF GENETOC ALGORITHM IN WALL FOLLOWING**'. We would like to take this opportunity to thank our project guide **Mr. Pardeep Kr. Gupta** for giving us all the help and guidance we needed. We are really grateful to him for his kind support throughout the analysis and design phase. We are also grateful to Brig. S.P. Ghrera, Head of CSE/I.T Department, Jaypee University of Information Technology and other staff members for giving important suggestions.

Signature of the student	[illegible]	Nilesh
Name of Student	KUNAL CHAWLA	NILESH BANSAL
Date	23/5/2011	

SUMARY

This project demonstrates the use of genetic programming (GP) for the development of maze wall-following behaviors. Algorithms are developed for a simulated maze that uses an array of range finders for navigation. Navigation algorithms are tested in a variety of differently shaped environments to encourage the development of robust solutions, and reduce the possibility of solutions based on memorization of a fixed set of movements. A brief introduction to GP is presented. A typical wall-following maze is analyzed, and results are presented. GP is shown to be capable of producing maze wall-following algorithms that perform well in each of the test environments used

Signature of Student: [illegible]	Signature of Student Nilesh
Name: KUNAL CHAWLA	Name NILESH BANSAL
Date 23/5/2011	Date 23/5/2011

Signature of Supervisor Pardeep Kumar.

Name PARDEEP KUMAR

Date 23-05-2011

PROBLEM STATEMENT

- We intend to build a wall following algorithm which will enable the object to successfully traverse the given maze.
- The object will be able to deviate from its current path as soon as it encounters a wall or you may say an obstruction.
- The object has to enter from a certain entrance and has to successfully traverse the whole path and find its way out of the maze.
- We intend to do this using Genetic Algorithm to carry out the various results possible in various generations.
- The various solutions or the successful traversal of the maze will give the fitness function of the algorithm.
- The output or the solution will be taken as the chromosomes which will be changed in each run or you may say in each generation.

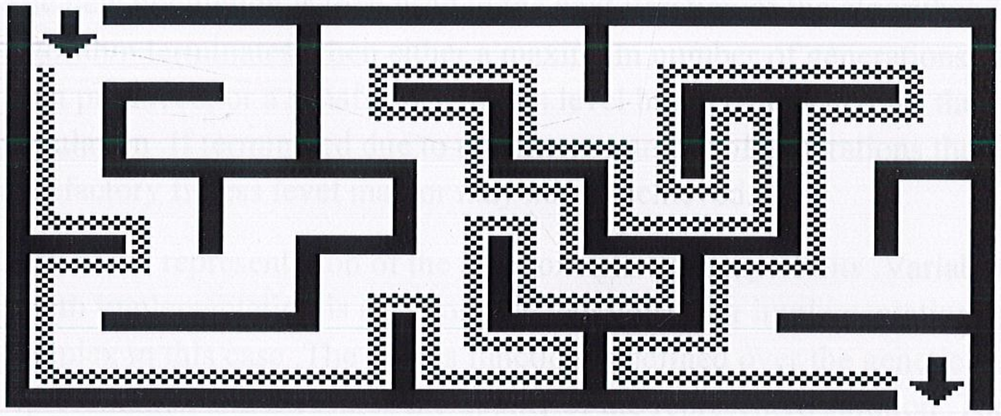
DEFINITION

The wall follower, the best-known rule for traversing mazes,it is also known as either the *left-hand rule* or the *right-hand rule*.

If the maze is *simply connected*, means all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze the player is guaranteed not to get lost and will reach a different exit if there is one; otherwise, he or she will return to the entrance.

This strategy works best when implemented immediately upon entering the maze.Another perspective into why wall following works is topological.If the walls are connected, then they may be deformed into a loop or circle.Then wall following reduces to walking around a circle from start to finish.

If the maze is not simply connected this method will not be guaranteed to help the goal to be reached.



GENETIC ALGORITHM

The Genetic Algorithms a search technique that mimics the process of natural evolution.This technique is generally used to generate useful solutions to optimization and search problems.In GA a string of chromosomes evolves towards better solutions.Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encoding are also possible.

A typical genetic algorithm requires:-

- ❖ A genetic representation of the solution domain.
- ❖ A fitness function to evaluate the solution domain.

The evolution usually starts from a population of randomly generated individuals and happens in generations .In each generation the fitness of every individual in the population is evaluated .Multiple individuals are stochastically selected from the current population and are modified to form a new population.

The new population is then used in the next iteration of the algorithm.The algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population .If terminated due to maximum number of generations then the satisfactory fitness level may or may not be achieved.

A standard representation of the solution is as an array of bits .Variable length implementation is also possible, but crossover implementation is more complex in this case .The fitness function is defined over the genetic representation and measures the quality of the represented solution .The fitness function is always problem dependent.
Initially many individual solutions are randomly generated to form an initial population .The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions .The population is generated randomly, covering the entire range of possible solutions.

OPERATORS OF GENETIC ALGORITHM

SELECTION:

-Selection is the stage of a genetic algorithm in which individual chromosomes are chosen from a population for later breeding (recombination or crossover).

The fitness function is evaluated for each individual, providing fitness values, which are then normalized .Normalization means dividing the fitness value of each individual by the sum of all fitness values, so that the sum of all resulting fitness values equals 1.

The population is sorted by descending fitness values .Accumulated normalized fitness values are computed .The accumulated fitness of the last individual should of course be 1.A random number *R* between 0 and 1 is chosen .The selected individual is the first one whose accumulated normalized value is greater than *R*.

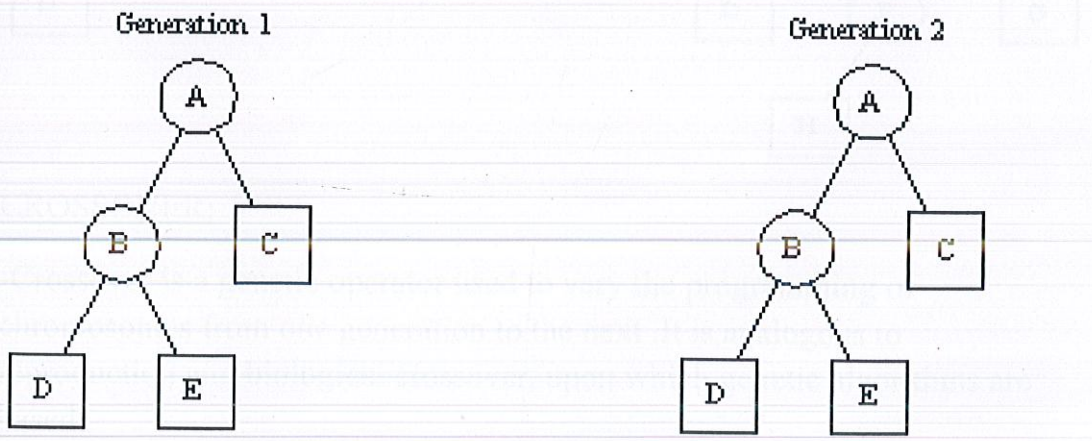
REPRODUCTION:

-It is of two types

-Mutation

-Crossover

Reproduction is simply the copying of an individual from the previous generation into the next generation without any modification of its structure.



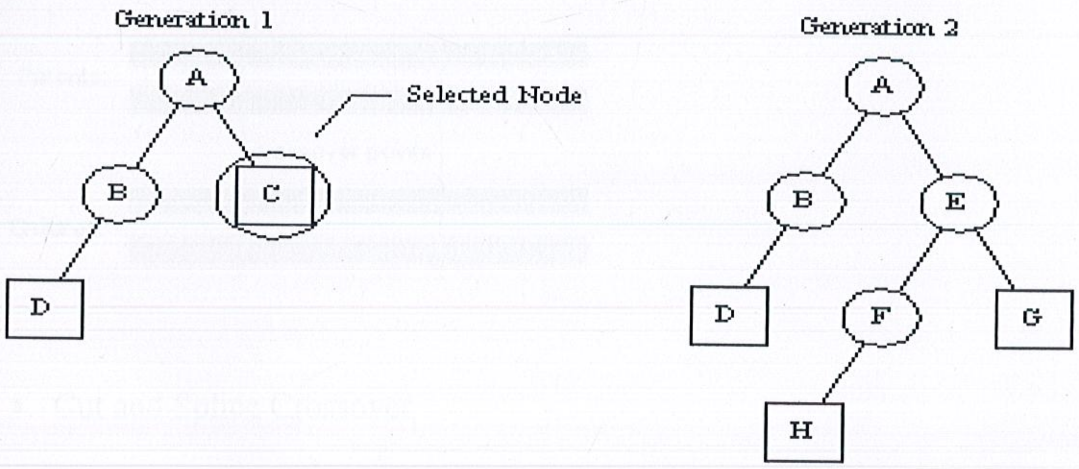
Mutation:

-Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of algorithm chromosomes to the next.It is analogous to biological mutation.

A common method of implementing the mutation operator involves generating a random variable for each bit in a sequence.This random variable tells whether or not a particular bit will be modified.This mutation procedure, based on the biological point mutation, is called single point mutation.

When the gene encoding is restrictive as in permutation problems, mutations are swaps, inversions and scrambles. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution.

Mutation is performed by randomly selecting a node in an individual tree structure, and removing that node along with any sub-tree that may exist below it. A new sub-tree is then generated randomly and "grafted in" at the position where the original node was removed.

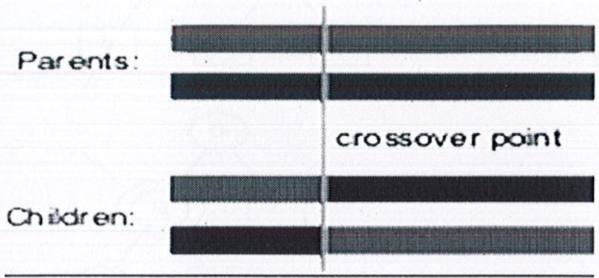


CROSSOVER:

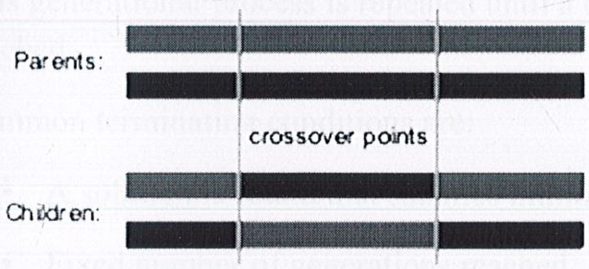
-Crossover is a genetic operator used to vary the programming of chromosomes from one generation to the next .It is analogous to reproduction and biological crossover, upon which genetic algorithms are based.

Types of Crossover

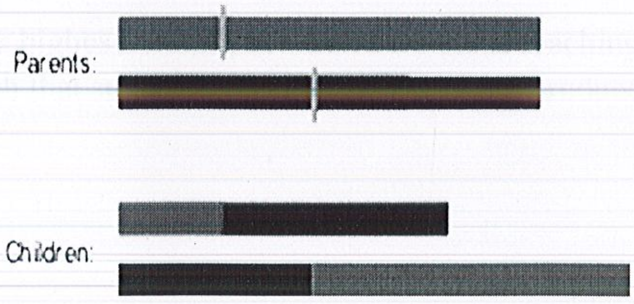
- One Point Crossover



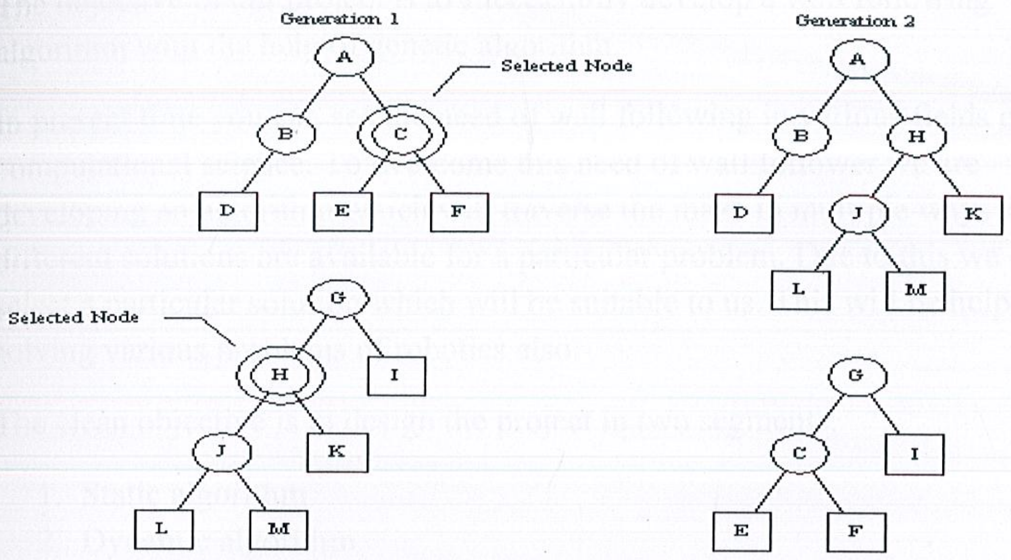
- Two Point Crossover



- Cut and Splice Crossover



Crossover involves selecting two individuals and selecting a node at random in each of them. The selected nodes, along with any sub-trees that exist below them, are exchanged between the two individuals.



TERMINATION:

This generational process is repeated until a termination condition has been reached.

Common terminating conditions are:

- A solution is found that satisfies minimum criteria.
- Fixed number of generations reached.
- Allocated budget (computation time/money) reached.
- Manual inspection.
- The highest ranking solution's fitness is reaching or has reached a level such that successive iterations no longer produce better results.

OBJECTIVE AND SCOPE

The objective of our project is to successfully develop a wall following algorithm with the help of genetic algorithm.

In present time you can see the need of wall following in various fields of computational science. To overcome this need of wall follower we are developing an algorithm which will traverse the maze in multiple ways so that different solutions are available for a particular problem. Due to this we can select a particular solution which will be suitable to us. This will be helpful in solving various problems of robotics also.

The clean objective is to design the project in two segments.

1. Static algorithm
2. Dynamic algorithm

In static algorithm we will have a fixed defined maze and solutions will be generated upon the fix given maze.

In dynamic algorithm we will have variable maze which will be generated randomly by the user and various solutions will be produced based on those different mazes.

The various solutions will help in developing the fitness function of the algorithm.

METHODOLOGY

In a genetic algorithm, a population of strings (called chromosomes or the genotype of the genome), which encode candidate solutions (called individuals, creatures) to an optimization problem, evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

A typical genetic algorithm requires:

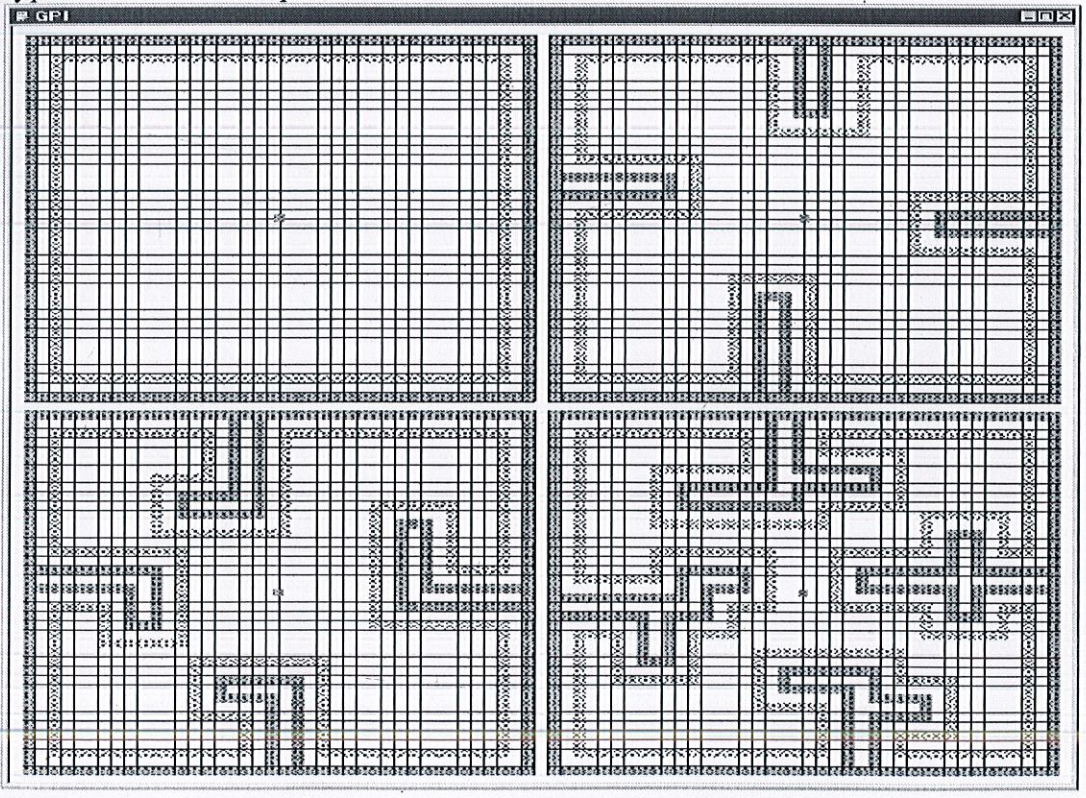
1. A genetic representation of the solution domain,
2. A fitness function to evaluate the solution domain.

A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. Variable length representations may also be used, but crossover implementation is more complex in this case.

The fitness function is defined over the genetic representation and measures the *quality* of the represented solution. The fitness function is always problem dependent. For instance, in the Travelling Salesman Problem one wants to maximize the total value of cities that can be put in the problem of some fixed capacity. A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the city is in the problem. Not every such representation is valid, as the number of cities may exceed the capacity of the problem. The *fitness* of the solution is the sum of paths of all cities in the Travelling Salesman Problem if the representation is valid or 0 otherwise.

EVOLVING WALL FOLLOWING BEHAVIOURS

We conduct an experiment to find out what are the effects that can be seen from the results of the GP runs when more complex wall shapes are introduced. The experiment involves an object moving in a world that consists of a 16 x 16 cells map. The outer cells represent the walls of the room and cannot be occupied by the object. The inner cells adjacent to the walls signify the cells where the object is able to pick up fitness points. The robot is free to occupy the rest of the cells.
For each iteration within a generation, the object is randomly placed within the allowable cells. In order to have a repeatable experiment, we may place the object on any allowable cell at our discretion instead of being randomly placed. The experiments were run subsequently on four different wall types. Each of the wall type is added with an extrusion starting with none for the first wall and ending with four extrusions for the fourth wall.
These experiments were conducted to see whether the addition of complexities on the walls would affect the GP algorithm adversely. This may show the scalability of the GP algorithm for this problem domain .Four different wall types used in the experiments



As you can see the complexity is increasing in each progressing maze.

The centrally placed red dot is our object which has to follow the walls of the given mazes.

We run the GE again and again and observe different results. These results will be stored as chromosomes and will vary in each run or you may say in each generation.

The main aim is to enter a given maze and trace the way out. After approaching walls the object should deviate from its path and should go on the path which it can traverse. This will help in giving the solution in form of a chromosome which can be stored in a linked list and can be modified in each generation.

ALGORITHMS

The software implementation of the maze is important for the proper and efficient working of the maze, as much as its hardware. As the rules of the competition suggests it is not just solving the maze that counts. The algorithm should solve the maze in the shortest time possible giving it the edge over the other competitors participating in the competition. So a proper selection and implementation and even improvisation of algorithm is important for obtaining expected results.

Most of the algorithms used nowadays in maze solving competitions are based on logical synthesis design rules. The algorithms which were used in the beginning years of this competition were simple logical algorithms which became inefficient as the maze itself became complex. The Algorithms used for maze solving competition are

Wall follower Algorithm
Depth first search Algorithm
Flood fill Algorithm
Modified Flood fill Algorithm

Wall follower Algorithm

The wall following algorithm is the simplest of the maze solving techniques. Basically, the mouse follows either the left or the right wall as a guide around the maze. And if the mouse encounters an opening in any of the walls picked up by its sensors, the mouse will stop and take a turn in that direction and then move forward sensing the walls gain. Thus keeping the walls as a guide the micromouse hopes to solve maze rather than actually solving it. The steps involved in following the right wall is given below

```
The right wall following routine:
Upon arriving in a cell:
If there is an opening to the right
Rotate right

Else if there is an opening ahead
Do nothing
Else if there is an opening to the left
Rotate left
```

```
Else
Turn around
End If
Move forward one cell
```

Although this Algorithm was efficient in solving the maze of the beginning years it is not used nowadays. This is because the maze used in competitions nowadays are constructed in such a way that the wall follower algorithm will never solve it. Such mazes have bottleneck regions which will cause the algorithm to fail.

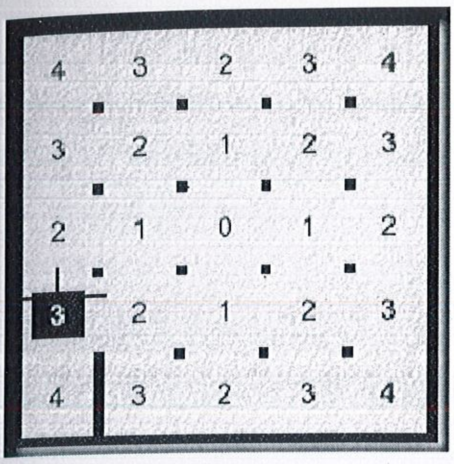
Depth First Search Algorithm

The depth first search is an intuitive method of searching a maze. Basically, the mouse simply starts moving. When it comes to an intersection in the maze, it randomly chooses one of the paths. If that path leads to a dead end, the mouse backtracks to the intersection and chooses another path. This forces the robot to explore every possible path within the maze. By exploring every cell within the maze the mouse will eventually find the center.

Even though this algorithm succeeds in solving the maze completely it need not necessarily be the shortest path. In addition to that the mice explores the entire maze for solving the maze leading to long latency. Cases have occurred in competitions where the competitors using this algorithm had to change their ba tteries in the middle as the mice took too much time in solving the maze. Hence algorithm is also not used in modern competitions.

Flood Fill Algorithm

The introduction of flood fill algorithm in maze solving methods paved new ways by which modern complex mazes can be solved without any bottlenecks. This algorithm is derived from the Bellman Ford Algorithm coming under the field of logic synthesis techniques. Other algorithms in this field are Djikstra's Algorithm, Johnson's Algorithm etc. These algorithms find applications in various fields like Communication, design softwares etc.



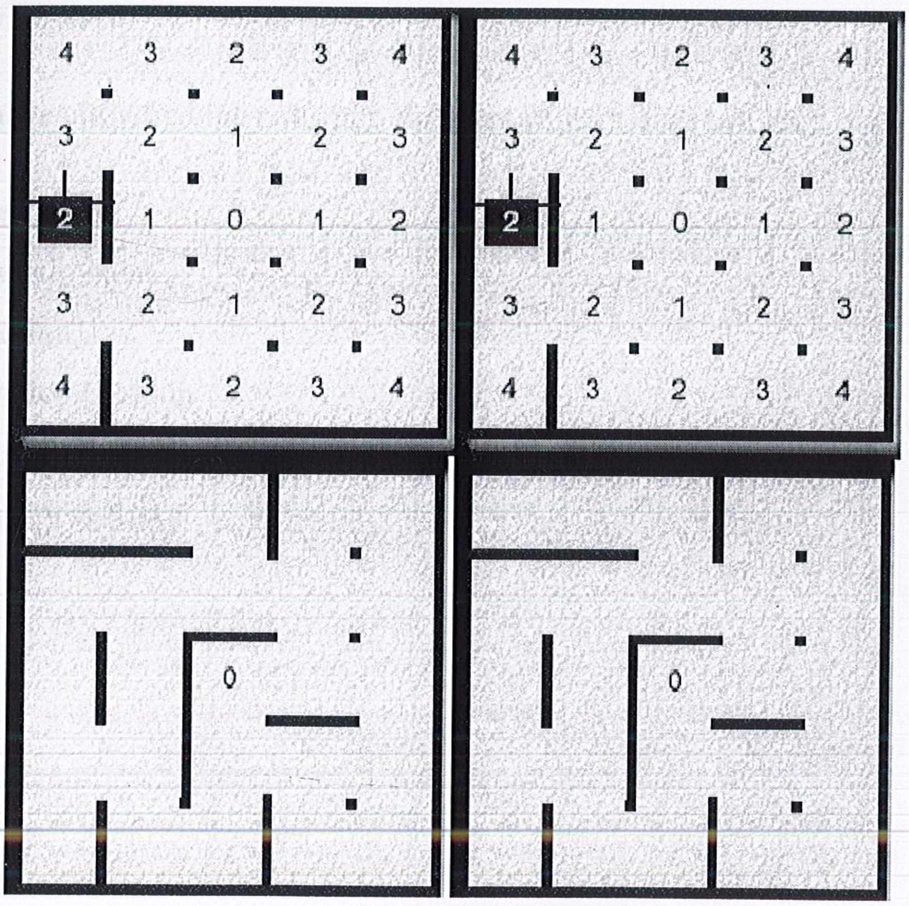
11

The flood- fill algorithm involves assigning values to each of the cells in the maze where these values represent the distance from any cell on the maze to the destination cell. The destination cell, therefore, is assigned a value of 0. If the mouse is standing in a cell with a value of 1, it is 1 cell away from the goal. If the mouse is standing in a cell with a value of 3, it is 3 cells away from the goal. Assuming the robot cannot move diagonally, the values for a 5X5 maze without walls would look like this:
Of course for a full sized maze, you would have 16 rows by 16 columns = 256 cell
values. Therefore you would need 256 bytes to store the distance values for a complete maze.

When it comes time to make a move, the robot must examine all adjacent cells which are not separated by walls and choose the one with the lowest distance value. In our example above, the mouse would ignore any cell to the West because there is a wall, and it would look at the distance values of the cells to the North, East and South since those are not separated by walls. The cell to the North has a value of 2, the cell to the East has a value of 2 and the cell to the South has a value of 4. The routine sorts the values to determine which cell has the lowest distance value. It turns out that both the North and East cells have a distance value of 2. That means that the mouse can go North or East and traverse the same number of cells on its way to the destination cell. Since turning would take time, the mouse will choose to go forward to the North cell.

Now the mouse has a way of getting to center in a maze with no walls. But real mazes have walls and these walls will affect the distance values in the maze so we need to keep track of them. Again, there are 256 cells in a real maze so another 256 bytes will be more than sufficient to keep track of the walls. There are 8 bits in the byte for a cell. The first 4 bits can represent the walls leaving you with another 4 bits for your own use. Remember that every interior wall is shared by two cells so when you update the wall value for one cell you can update the wall value for its neighbor as well.

So now we have a way of keeping track of the walls the mouse finds as it moves about the maze. But as new walls are found, the distance values of the cells are affected so we need a way of updating those. Returning to our example, suppose the mouse has found a wall.



RISK MANAGEMENT

Risk is a possibility of loss or injury. The definition of risk in the software engineering environment that we will use is exposure to harm or loss, as this includes not only the possibility of risks, but their impact as well. Using risk management techniques, we alleviate the harm or loss in a software project. All risks cannot be avoided, but by performing risk management, we can attempt to ensure that the right risks are taken at the right time. "...Risk taking is essential to progress and, and failure is often a key part of learning".

Software Risk Management is an iterative process. About two iterations are both feasible and useful. We use the risk table to identify risks and briefly describe them.

We have classified the risks into different categories. They are:

1. Technical
2. Requirement
3. Design
4. Implementation
5. Post Release
6. Business Risks

Risk Table

No.	Risk	Category	Probability
1	Changes in Design regarding choice of underlying data structure.	Technical – Design	30%
2	Lack of communication among team members due to clashes in schedule	Project	5%
3	Ambiguous Requirements that may change	Project	60%
4	Performance. (Something taking too long or too heavy on resources)	Technical	40%
5	Release of a similar product by another team	Business	10%
6	Experience with development language/ platform/ tools	Project	20%

OVERVIEW OF RISK MITIGATION,MONITORING AND MANAGEMENT

Project scope is vast with limited time, and disaster can be taken care of by risk avoidance using a proactive approach. This can be done by developing a risk mitigation plan.

Small staff size and staff inexperience can be taken care of in risk monitoring stage by the project group members having a good relationship with one another. The members jell with each other well and there must be proper co-ordination among the team members. Also by arranging meetings with people who are experienced in the field and have complete knowledge about the field will help us.

PROJECT SCHEDULE

Sr. No	Name of task	Subtask	
1	**Problem Identification and Information Gathering**	1. **Problem Definition:** • Collecting detailed problem definition of the system to be implemented	16/07/2010 To 31/07/2010
		2. **Literature Survey:** • Visiting different websites. • Studying existing system with it's limitation • Going through Journals, magazines • Studying the reference books	25/07/2009 To 15/08/2010
2	**Analysis**	1. **Project Plan:** • Preparing for complete project plan	1/08/2010 To 15/08/2010

		2. **Requirement Analysis** • Software requirements • Hardware requirements • Databases	16/08/2010 To 30/08/2010
3	Design	1. **Architectural design:** • Describing relationships between modules and sub modules	25/08/2010 To 10/09/2010
4	**Output Screen formats**	**Output Screens:** • Preparing for detailed output screens describing output formats	22/11/2010
5	**Development**	1. **Coding:** • Implementing design details Using programming language C/C++	01/11/2010 To 31/012011

		2. Adding graphics to the final output screen	01/02/2011 To 20/04/2011
6	**Testing**	Testing the system for expected results	25/04/2011 To 05/05/2011

RESOURCES AND LIMITATIONS

- C language

  We know that it is the most efficient and the most powerful language for coding.
  The use of the c language will be done to develop the genetic algorithm module which will help in giving a new maze or you may say graph in the subsequent stages of the project.
  The genetic module that will be generated will create a maze of the give shape and size by the user and will have a room for the complexity from which the user can set the complexity of the maze from a level of 1-10. Here 1 means least complexity and 10 means highest complexity.

- C++ Language

  We know that C++ is a very powerful object oriented language for programming.
  In our project we will use C++ for coding the main wall following code which will basically decide the path of the object which it will be guiding.
  We chose C++ also because it supports graphics so the wall following procedure will be easily demonstrated.
  The maze following will give the result in terms of the chromosomes which will be given to the genetic module developed in C and it will be reproduced again in various forms and will returned giving the new solutions to the given problem or maze. This reproduced chromosome will help us in developing the fitness function for our problem and will describe the success we achieve in developing the project or you may say how efficiently we have done the project of application of genetic algorithm in wall following .As far as the limitations are concerned our algorithm is not flawless. It can be modified as the requirements of the developer as and when he starts a new project.

SAMPLE CODE

```
#include <iostream>

#include <stdlib.h>

#include <conio.h>

#include <vector>

using namespace std;

#define NULL        0


#define LEFT        3

#define UP          2

#define RIGHT       1

#define     DOWN      0


#define FOUND_NOWAY             0

#define       FOUND_EXIT        1

#define FOUND_NOEXIT           -1


int setCoord(int &nrows,int &ncols);

void fillMaze(char ** maze,int nrows,int ncols);

void printMaze(char ** maze,int nrows,int ncols);

int findEntry(char ** maze,int nrows,int ncols,int ** npos,int &num);

int findExit(char ** maze,int nrows,int ncols,int ** xpos,int &num);

void valMaze(char ** maze,int nrows,int ncols,int ** npos,int ** xpos);
```

```
int solveMaze(char ** maze,int ** cell_p,std::vector<int> &x,std::vector<int>
&y);

void refinePath(std::vector<int> &x,std::vector<int> &y);

void printRefine(char ** maze,int nrows,int ncols,std::vector<int>
&x,std::vector<int> &y);

void noSolution(void);


int main(int argc,char *argv[])

{


        int nrows = 0, ncols = 0;

        setCoord(nrows,ncols);

        //------------------------------------------------------------

        char ** maze = new char * [nrows];

        if (maze == NULL) {

                cout << "Couldn't allocate memory for the maze *_*"

                        <<      "press any key to exit..";

                _getch();

                exit(EXIT_FAILURE);

        }

        for (int i = 0; i < nrows; i++)

                maze[i] = new (nothrow) char[ncols];

        //------------------------------------------------------------

        int ** npos = new int * [2];

        if (npos == NULL) {
```

```
        cout << "Couldn't allocate memory for the entry position *_*"
              <<     "press any key to exit..";
        _getch();
        exit(EXIT_FAILURE);
}
for (int i = 0; i < 2; i++)
        npos[i] = new (nothrow) int;
//------------------------------------------------------
int ** xpos = new int * [2];
if (xpos == NULL) {
        cout << "Couldn't allocate memory for the exit position *_*"
              << "press any key to exit..";
        _getch();
        exit(EXIT_FAILURE);
}
for (int i = 0; i < 2; i++)
        xpos[i] = new (nothrow) int;
//------------------------------------------------------
fillMaze(maze,nrows,ncols);

valMaze(maze,nrows,ncols,npos,xpos);
cout << "Your maze has been accepted\n\n";

while(true) {
        char answer = 0;

        cout << "What do you want to do now?\n"

                <<      "\t1\xF9 Print the maze before solving\n"

                <<      "\t2\xF9 Solve the maze then print it\n"

                <<      "\t3\xF9 Exit\n"

                <<      "1,2,3,4 to select an option: ";

        while (answer < '1' || answer > '3') {

                answer = _getch();

        }

        cout << answer << "\n\n";

        if (answer == '1') {

                printMaze(maze,nrows,ncols);

        } else if (answer == '2') {

                //----------------------------------------------------

                int ** cell_p = new int * [3];

                if (cell_p == NULL) {

                        cout << "Couldn't allocate memory for the cell
position *_*"

                                <<      "press any key to exit..";

                        _getch();

                        exit(EXIT_FAILURE);

                }

                for (int i = 0; i < 3; i++)

                        cell_p[i] = new (nothrow) int;

                //----------------------------------------------------
```

```
*cell_p[0] = *npos[0];

*cell_p[1] = *npos[1];

if (*npos[0] == ncols-1)

        *cell_p[2] = LEFT;

else if (*npos[1] == nrows-1)

        *cell_p[2] = UP;

else if (*npos[0] == 0)

        *cell_p[2] = RIGHT;

else if (*npos[1] == 0)

        *cell_p[2] = DOWN;


vector<int> x;

vector<int> y;

x.push_back(*npos[0]);

y.push_back(*npos[1]);

solveMaze(maze,cell_p,x,y);

if (x.size() < 3500) {

        x.push_back(*xpos[0]);

        y.push_back(*xpos[1]);

} else {

        cout << "The vectors allocated have drained\n"

                <<      "press any key to exit..";

        _getch();
```

```
                        exit(EXIT_FAILURE);
                    }
                    printMaze(maze,nrows,ncols);
                    cout << "Press any key now to refine the path we
found\n\n";
                    _getch();
                    refinePath(x,y);
                    printRefine(maze,nrows,ncols,x,y);
                    cout << "Press any key now to exit..";
                    _getch();
                    delete [] cell_p;
                    break;
                } else if (answer == '3') {
                    break;
                }
            }
            delete(maze);
            delete(npos);
            delete(xpos);
            exit(EXIT_SUCCESS);
            return 0;
}
int setCoord(int &nrows,int &ncols) {
        if (ncols <= 0 || ncols > 40) {
                cout << "Enter maze width: ";
        cin >> ncols;
    }
    if (nrows <= 0 || nrows > 40) {
        cout << "Enter maze height: ";
        cin >> nrows;
    }
    if (nrows <= 0 || nrows > 40 || ncols <= 0 || ncols > 40) {
        cout << "Invalid width or height detected\n\n";
        setCoord(nrows,ncols);
    }
    return (nrows,ncols);
}
void fillMaze(char ** maze,int nrows,int ncols) {
    cout << "\n\n" << "Enter maze as you would like it to appear\n"
        <<    "[Wall: 1] [Exit: 2] [Path: 3] [Entry: 4]\n"
        <<    "Example:\n\n"
        <<    "111111\n"
        <<    "433132\n"
        <<    "131131\n"
        <<    "133331\n"
        <<    "111111\n\n";
    int col = 0;
    char c;
    for (int row = 0; row < nrows; row++)
```

```
        {
                c = 0;
                while (col <= ncols) {
                        c = _getch();
                        if (c > '0' && c < '5' && col < ncols) {
                                cout << c;
                                maze[row][col] = c;
                                col++;
                        } else if (c == 0x0d && col == ncols) {
                                cout << "\n";
                                maze[row][col] = 0;
                                col = 0;
                                break;
                        } else if (c == 0x08) {
                                putchar(0x08);
                                putchar(0xFF);
                                putchar(0x08);
                                if (col != 0)
                                        col--;
                        }
                }
        }
        cout << "\n";
}
```

```
void printMaze(char ** maze,int nrows,int ncols) {
        for (int i = 0; i < nrows; i++) {
                for (int j = 0; j < ncols; j++) {
                        cout << maze[i][j];
                }
                cout << "\n";
        }
        cout << "\n";
}
int findEntry(char ** maze,int nrows,int ncols,int ** npos,int &num) {
        int i;
        num = 0;
        for (i = 1; i < ncols-1; i++) {
                if (maze[0][i] == '4' && num == 0) {
                        *npos[1] = 0;
                        *npos[0] = i;
                        num++;
                } else if (maze[0][i] == '4' && num != 0) {
                        num++;
                } else if (maze[nrows-1][i] == '4' && num == 0) {
                        *npos[1] = nrows-1;
                        *npos[0] = i;
                        num++;
                } else if (maze[nrows-1][i] == '4' && num != 0) {
```

```
                num++;
            }
        }
        for (i = 1; i < nrows-1; i++) {
            if (maze[i][0] == '4' && num == 0) {
                *npos[1] = i;
                *npos[0] = 0;
                num++;
            } else if (maze[i][0] == '4' && num != 0) {
                num++;
            } else if (maze[i][ncols-1] == '4' && num == 0) {
                *npos[1] = i;
                *npos[0] = ncols-1;
                num++;
            } else if (maze[i][ncols-1] == '4' && num != 0) {
                num++;
            }
        }
        return num;
}

int findExit(char ** maze,int nrows,int ncols,int ** xpos,int &num) {
        int i;
        num = 0;
```

```
for (i = 1; i < ncols-1; i++) {

        if (maze[0][i] == '2' && num == 0) {

                *xpos[1] = 0;

                *xpos[0] = i;

                num++;

        } else if (maze[0][i] == '2' && num != 0) {

                num++;

        } else if (maze[nrows-1][i] == '2' && num == 0) {

                *xpos[1] = nrows-1;

                *xpos[0] = i;

                num++;

        } else if (maze[nrows-1][i] == '2' && num != 0) {

                num++;

        }

}

for (i = 1; i < nrows-1; i++) {

        if (maze[i][0] == '2' && num == 0) {

                *xpos[1] = i;

                *xpos[0] = 0;

                num++;

        } else if (maze[i][0] == '2' && num != 0) {

                num++;

        } else if (maze[i][ncols-1] == '2' && num == 0) {

                *xpos[1] = i;
```

```
                    *xpos[0] = ncols-1;

                    num++;

               } else if (maze[i][ncols-1] == '2' && num != 0) {

                    num++;

               }

          }

          return num;

     }

     void valMaze(char ** maze,int nrows,int ncols,int ** npos,int ** xpos) {

          int     i, j, w_ok, w = 0, e = 0, num;

          for (i = 0; i < ncols; i++) {

               if (maze[0][i] == '1')

                    w++;

               if (maze[nrows-1][i] == '1')

                    w++;

          }

          for (i = 1; i < nrows-1; i++) {

               if (maze[i][0] == '1')

                    w++;

               if (maze[i][ncols-1] == '1')

                    w++;

          }

          w_ok = nrows * ncols - (nrows-2) * (ncols-2) - 2;

          if (w_ok != w) {
            cout << "\t\xF9 Ambiguous error detected with your walls\n\t"
                    <<      "make sure you have 1 entry and 1 exit points\n\t"
                    <<      "all other wall cells must be equal '1'\n";
            e++;
        }
        findEntry(maze,nrows,ncols,npos,num);
        if (*npos[0] == NULL && *npos[1] == NULL) {
            cout << "\t\xF9 No entry point was detected and NO, parachutes
are not allowed!\n";
            e++;
        } else if (num != 1) {
            cout << "\t\xF9 Multiple entry points were detected\n\t"
                    <<      "maze entry, exit maze, maze exit, entry maze... I'm
lost *_*\n";
            e++;
        }
        findExit(maze,nrows,ncols,xpos,num);
        if (*xpos[0] == NULL && *xpos[1] == NULL) {
            cout << "\t\xF9 No exit point was detected and NO, digging is not
allowed!\n";
            e++;
        } else if (num != 1) {
            cout << "\t\xF9 Multiple exit points were detected\n\t"
                    <<      "maze entry, exit maze, maze exit, entry maze... I'm
lost *_*\n";
            e++;
```

```
    }
    for (i = 1; i < nrows-1; i++) {
        for (j = 1; j < ncols-1; j++) {
            if (maze[i][j] == '4' || maze[i][j] == '2') {
                cout << "\t\xF9 Ambiguous error detected with your
maze\n\t"
                    <<    "make sure it doesn't contain any entry or
exit points inside\n";
                e++;
                i = nrows;
                j = ncols;
            }
        }
    }
    cout << "\n";
    if (e != 0) {
        cout << "\t\xF9 Errors were detected with your maze design\n\t"
            <<    "thus no further operations are allowed\n\t"
            <<    "press any key to exit the program..";
        _getch();
        exit(EXIT_FAILURE);
    } else {
        cout << "\t\xF9 No errors were detected with your maze design\n\t"
            <<    "GOD speed :)\n\n";
    }
```

```
}
int solveMaze(char ** maze,int ** cell_p,std::vector<int> &x,std::vector<int>
&y) {
        int n, i;

        switch(*cell_p[2])
        {
        case LEFT:
                *cell_p[0] -= 1;
                break;
        case UP:
                *cell_p[1] -= 1;
                break;
        case RIGHT:
                *cell_p[0] += 1;
                break;
        case DOWN:
                *cell_p[1] += 1;
        }

        switch(maze[*cell_p[1]][*cell_p[0]])
        {
        case '1':
                return FOUND_NOWAY;
        case '2':
          return FOUND_EXIT;
    case '4':
          return FOUND_NOEXIT;
    }

    *cell_p[2] -= 1;
    if (*cell_p[2] < 0)
          *cell_p[2] += 4;

    for (i = 0; i < 4; i++) {
          if (maze[*cell_p[1]][*cell_p[0]] == '3' ||
maze[*cell_p[1]][*cell_p[0]] == '@') {
          //POSITION
    maze[*cell_p[1]][*cell_p[0]] = '@';
                printf("\n%d\t%d\n",*cell_p[1],*cell_p[0]);

     if (x.size() < 3500) {
                      x.push_back(*cell_p[0]);
                      y.push_back(*cell_p[1]);
                } else {
                      cout << "The vectors allocated have drained\n"
                            <<      "press any key to exit..";
                      _getch();
                      exit(EXIT_FAILURE);
                }
```

```
}

n = solveMaze(maze,cell_p,x,y);
if (n) {
        if (n == FOUND_NOEXIT) {
                maze[*cell_p[1]][*cell_p[0]] = '3';
                x.pop_back();
                y.pop_back();
        }
        return n;
} else {
        switch(*cell_p[2])
        {
        case LEFT:
                *cell_p[0] += 1;
                break;
        case UP:
                *cell_p[1] += 1;
                break;
        case RIGHT:
                *cell_p[0] -= 1;
                break;
        case DOWN:
                *cell_p[1] -= 1;
```

```
                    }
                }

                *cell_p[2] += 1;
                if (*cell_p[2] > 3)
                        *cell_p[2] -= 4;
        }

        maze[*cell_p[1]][*cell_p[0]] = '@';
        ////////////////////////////////////

        //////////////////////////////
        if (x.size() < 3500) {
                x.push_back(*cell_p[0]);
                y.push_back(*cell_p[1]);
        } else {
                cout << "The vectors allocated have drained\n"
                        <<      "press any key to exit..";
                _getch();
                exit(EXIT_FAILURE);
        }

        return 0;
}
```

```
void refinePath(std::vector<int> &x,std::vector<int> &y) {

        for (int i = 1; i < x.size()-1; i++) {

                for (int j = i+1; j < x.size()-1; j++) {

                        if (x[i] == x[j] && y[i] == y[j]) {

                                x.erase(x.begin()+i,x.begin()+j);

                                y.erase(y.begin()+i,y.begin()+j);

                        }

                }

        }

}

void printRefine(char ** maze,int nrows,int ncols,std::vector<int>
&x,std::vector<int> &y) {

        int i, j, k;

        for (i = 1; i < nrows-1; i++) {

                for (j = 1; j < ncols-1; j++) {


                        if (maze[i][j] != '3' && maze[i][j] != '1') {

                                for (k = 1; k < x.size()-1; k++) {

                                        if (x[k] == i && y[k] == j)

                                                maze[i][j] = '3';

                                        else

                                                maze[i][j] = '@';

                                }

                        }
                }
        }
        printMaze(maze,nrows,ncols);
}
void noSolution() {
        cout << "No first move is allowed unless you want to hit a solid wall\n"
                <<      "no solution\n"
                <<      "press any key to exit..";
        _getch();
        exit(EXIT_SUCCESS);
}
```

RESULTS AND CONCLUSION

These experiments show that GP is capable of generating wall-following algorithms using the function set provided. Although a 100% solution (passing through all possible corridor grids) was never attained, most learning cycles produced algorithms that exhibited the desired behaviour. The convex corners proved to be particularly troublesome and warrant further analysis. This is probably a sufficiency issue and resolution should enhance performance, possibly leading to 100% solutions.

The algorithms generated are robust in that they perform well across the four environments used in these experiments. It seems reasonable to expect that the algorithms will also perform well in additional environments of similar complexity, but this hypothesis remains untested. The original intent was to test the algorithms on a physical robot equipped with ultrasonic range finders, but such a robot was unavailable during the required time interval. Follow-on work will focus on employing GP-developed algorithms on physical robots.

This project demonstrates the feasibility of using genetic programming to develop mobile robot navigation algorithms. It lays the foundation for planned follow-on projects of maze traversal, map generation and full-coverage area traversal. The results of this project show sufficient promise to warrant further research into these more complex tasks.

REFRENCES

- J.R. Koza, *Genetic programming: On the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, 1992.

- J.H. Holland, *Adaptation In Natural And Artificial Systems,* University of Michigan Press, Ann Arbor, MI, 1975.

- C.W. Reynolds, *Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions*, Advances in Genetic Programming, MIT Press, Cambridge, MA, pp. 221-241, 1994.

- R.A. Dain,*An Overview of Genetic Programming with Machine Vision Examples,* Northwest Artificial Intelligence Forum Journal, Volume 4, pp. 21-30, 1995.

- Bhanzaf W., Nordin P., and Olmer M., Generating Adaptive Behaviour for a Real Robot using Function Regression within Genetic Programming.Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H and Riolo, Rick L., editors. 35-43 1997: Proceedings of the 2nd Annual Conference, Stanford University, 1997.

- Daida J., Ross S., McClain J., Ampy D. and Holczer M., Challenges with Verification, Repeatability, and Meaningful Comparisons in Genetic Programming. Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H et al., editors. Genetic Programming 1997: Proceedings of the Second Annual Conference. San Francisco,CA: Morgan Kaufmann Publishers, Inc., 1997.

- Hu H., Kostiadis K. and Liu Z., Coordination and Learning in a team of Mobile Robots, Proceedings of the IASTED Robotics and Automation Conference, ISBN 0-88986-265-6, pages 378-383, Santa Barbara, CA, USA, 28-30 October 1999.

- Ross S. J., Daida J. M., Doan, C. M., Bersano-Begey and McClain, J. J., Variations in Evolution of Subsumption Architectures Using Genetic Programming: The Wall Following Robot Revisited. Koza JR, Goldberg DE, Fogel DB, Riolo RL, editors. 191-199. 1996. Cambridge, MA, The

MIT Press. Genetic Programming 1996: Proceedings of the 1st Annual Conference, Stanford University, July 28-31, 1996

- Kostiadis K., Hunter M. and Hu H., The Use of Design Patterns for Building Multi-Agent Systems, IEEE Int. Conf. On Systems, Man and Cybernetics, Nashville, Tennessee, USA, 8-11 October 2000.

- Goldberg D.E., Genetic Algorithms in Search, Optimisation, and Machine Learning. Reading, MA: Addison-Wesley, 1989.

BIO DATA

KUNAL CHAWLA
D-2/2105, Vasant Kunj
New Delhi – 110 070
Tel®: +91 11 2612 5198
Mobile: +91 9816 334 995 **E-mail:** chawlakunal@ymail.com

CAREER OBJECTIVE

To grow intellectually and enhance my skills and abilities while sharing a mutually beneficial relationship with the organization I serve.

EDUCATION

Standard	College/School	Year	CGPA/Percentage
B.Tech (CSE)	JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, SOLAN.	2011 (Pursuing)	5.9/66.0% (Up till 7th sem)
12th (C.B.S.E)	THE AIRFORCE SCHOOL,NEW DELHI	2007	72.2%
10th (C.B.S.E)	THE AIRFORCE SCHOOL,NEW DELHI	2005	77.6%

INTERNSHIP/TRAINING PROGRAM

Engineers India Limited, New Delhi. 31st May – 6th July 2010

Successfully completed summer training of six weeks at Engineers India Limited (EIL), New Delhi. During the training period, I worked on a project **"System for Storing and Maintaining Data"** at EIL's Information Technology Services Division. Comprehensive report was prepared and submitted to the Division.

TECHNICAL SKILLS

- Languages: C, C++

PROJECTS UNDERTAKEN:

- Working on Application of Genetic Algorithms in Wall Following.
- Developed a website on "**jal board management system**".

EXTRA CURRICULAR ACTIVITIES

- Good athelete and won medals at zonal and state level.
- Passed the Certificate 'A' Examination of National Cadet Corps in 2004 under the authority of Ministry of Defence, Government of India.
- Participated in Quiz contests like Petroleum Conservation for Economy and Environment and also made outstanding contribution towards social welfare activities like curbing vehicular pollution, care for the elderly (Help Age India) etc.
- Awarded with merit scholarship for four years by Engineers India Ltd., New Delhi for B.Tech studies.

HOBBIES

- Travelling to new places
- Listening to music.

STRENGTHS

- Responsible and confident
- Quick in learning from mistakes
- Ability to manage demanding situation deftly & cheerfully.

PERSONAL

Date of Birth	: 22nd june 1989
Father's Name	: Mr. R.S Chawla
Mother's Name	: Mrs. Saroj Chawla
Permanent Address	: D-2/2105, Vasant Kunj New Delhi 110-070

NILESH BANSAL
Jaypee University of Information Technology
Waknaghat, District Solan,
Himachal Pradesh-173215.
DOB: 2nd February, 1990.
Mobile: +91-9736111656
Email: nileshbansal318@gmail.com

OBJECTIVE

To achieve excellence in I.T. industry through continuous learning and innovative work. Also to build a highly motivated team of people working in harmony, so that, set organizational goals and objectives are not only met but also surpassed.

ACADEMIC SUMMARY

Examination	Board/University	Year	CGPA / Percentage
B.Tech (C.S.E.)	Jaypee University of Information Technology Waknaghat, Solan	2011	**5.5 (62.76%)** up till 7th Sem.
12th	Vivekanand School D-Block Anand Vihar C.B.S.E.	2007	**82.0%** First class with Distinction
10th	Ryan International School Sect-39 Noida C.B.S.E.	2005	**80.0%** First class with Distinction

SKILLS

- **Computer Languages** — C, C++, HTML,
- **Operating Systems** — Microsoft Windows XP/Vista/7
- **Languages Known** — English(Read & Write) Hindi (Read & Write)
- **Technical** — Object Oriented Programming, Data Structures.

INDUSTRIAL / SUMMER TRAINING

- Summer Training on "Reliance Infra Messaging Dashboard" at **Reliance Infrastructure Limited, Noida** for a period of 6 weeks. The project was basically to design a dashboard for each division for better connectivity of the various departments of a same EPC division.

PROJECTS UNDERTAKEN

- Developed a **Tic-Tac-Toe** using **Data Structures** in C language. The code used a tree data structure and implemented functions such as add, search and delete.

- Developed web based software using **ASP.NET** on **ONLINE SHOPPING.** The application performed all functions required by operator to successfully handle customer requirements. An extensive database was also maintained for data to be managed and handled for all related queries.

Final Year Project

- **Application of Genetic Algorithm In Wall Following**

Intend to build an application for wall following using genetic algorithm.
The main aim of the project is to develop an algorithm which can have various possible results in different chromosomes so as to have genetic diversity. We have to develop an algorithm which successfully traverses the maze and find its way out. The application of this project is in bioinformatics, computational science and phylogenetics.

ACHIEVEMENTS

- Secured **Third position** in Inter-House **Extempore (English)** in the year 2005.
- Participated in various quizzes and Olympiads on school level.
- Active member of the Event Management Club for consecutively three years (2007-2010).
- Won various **Lan Gaming** tournaments in various colleges.
- Won debate competitions at the School level.
- Distinctive Performance in various technical competitions.

WORKSHOPS

- Attended a 48 hour workshop on **Object Oriented Programming Using C++** at NIIT New Delhi.
- Workshop of IBM on their database DB2 (**IBM DB2**).

AREAS OF INTEREST / HOBBIES

- Listening Music
- Sports (Cricket, Table Tennis, Chess)
- Social Networking.
- Lan Gaming

Date: 23-05-2011

[illegible]

(Nilesh Bansal)