

**AGENT BASED INTELLIGENT SYSTEMS**

**CSE-330**

**PROJECT REPORT**

**MAZE SOLVER/ESCAPER**

PROJECT TEAM MEMBERS

1. SAILESH PATRA-13BCE1119
2. CHAVA SAI TEJA-13BCE1032
3. MARLA AKHIL REDDY-13BCE1078

**ABSTRACT :**

This project is basically concerned on making a robot to act as an agent equipped with adequate sensors and programmed with appropriate algorithm to find its way out of a standard maze and then start over from the end by doing some sort of path way processing.

SENSORS USED:

1. Ultrasonic range finder sensors(PING)
2. PID controllers to check the rpm of the motors

**DESCRIPTION:**

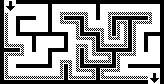
AVAILABLE ALGORITHMS TO SOLVE A MAZE :

There are a number of different maze solving algorithms, that is, automated methods for the solving of mazes.

**Random mouse algorithm:**

This is a trivial method that can be implemented by a very unintelligent robot or perhaps a mouse. It is simply to proceed following the current passage until a junction is reached, and then to make a random decision about the next direction to follow. Although such a method would always eventually find the right solution, this algorithm can be extremely slow.

**Wall follower:**



Traversal using *Right-hand rule*

The wall follower, the best-known rule for traversing mazes, is also known as either the left-hand rule or the right-hand rule. If the maze is simply connected, that is, 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 solver 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 having traversed every corridor next to that connected section of walls at least once.

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. To further this idea, notice that by grouping together connected components of the maze walls, the boundaries between these are precisely the solutions, even if there is more than one solution (see figures on the right).

If the maze is not simply connected (i.e. if the start or endpoints are in the center of the structure surrounded by passage loops, or the pathways cross over and under each other and such parts of the solution path are surrounded by passage loops), this method will not reach the goal.

Wall-following can be done in 3D or higher-dimensional mazes if its higher-dimensional passages can be projected onto the 2D plane in a deterministic manner. For example, if in a 3D maze "up" passages can be assumed to lead northwest, and "down" passages can be assumed to lead southeast, then standard wall following rules can apply. However, unlike in 2D, this requires that the current orientation be known, to determine which direction is the first on the left or right.

**Pledge algorithm :**

[](https://en.wikipedia.org/wiki/File:Cyclope_robot.jpg)

Robot in a wooden maze

Disjoint mazes can still be solved with the wall follower method, if the entrance and exit to the maze are on the outer walls of the maze. If however, the solver starts inside the maze, it might be on a section disjoint from the exit, and wall followers will continually go around their ring. The Pledge algorithm (named after Jon Pledge of Exeter) can solve this problem.

The Pledge algorithm, designed to circumvent obstacles, requires an arbitrarily chosen direction to go toward. When an obstacle is met, one hand (say the right hand) is kept along the obstacle while the angles turned are counted. When the solver is facing the original direction again, and the angular sum of the turns made is 0, the solver leaves the obstacle and continues moving in its original direction.

The hand is removed from the wall only when both "sum of turns made" and "current heading" are at zero. This allows the algorithm to avoid traps shaped like an upper case letter "G". Assuming the algorithm turns left at the first wall, one gets turned around a full 360 degrees by the walls. An algorithm that only keeps track of "current heading" leads into an infinite loop as it leaves the lower rightmost wall heading left and runs into the curved section on the left hand side again. The Pledge algorithm does not leave the rightmost wall due to the "sum of turns made" not being zero at that point. It follows the wall all the way around, finally leaving it heading left outside and just underneath the letter shape.

This algorithm allows a person with a compass to find their way from any point inside to an outer exit of any finite two-dimensional maze, regardless of the initial position of the solver. However, this algorithm will not work in doing the reverse, namely finding the way from an entrance on the outside of a maze to some end goal within it.

## Trémaux's algorithm:

## Trémaux's algorithm, invented by Charles Pierre Trémaux, is an efficient method to find the way out of a maze that requires drawing lines on the floor to mark a path, and is guaranteed to work for all mazes that have well-defined passages. A path is either unvisited, marked once or marked twice. Every time a direction is chosen it is marked by drawing a line on the floor (from junction to junction). In the beginning a random direction is chosen (if there is more than one). On arriving at a junction that has not been visited before (no other marks), pick a random direction that is not marked (and mark the path). When arriving at a marked junction and if your current path is marked only once then turn around and walk back (and mark the path a second time). If this is not the case, pick the direction with the fewest marks (and mark it, as always). When you finally reach the solution, paths marked exactly once will indicate a direct way back to the start. If there is no exit, this method will take you back to the start where all paths are marked twice. In this case each path is walked down exactly twice, once in each direction. The resulting walk is called a bidirectional double-tracing.

Essentially, this algorithm, which was discovered in the 19th century, has been used about a hundred years later as depth-first search.

## Dead-end filling

Dead-end filling is an algorithm for solving mazes that fills all dead ends, leaving only the correct ways unfilled. It can be used for solving mazes on paper or with a computer program, but it is not useful to a person inside an unknown maze since this method looks at the entire maze at once. The method is to 1) find all of the dead-ends in the maze, and then 2) "fill in" the path from each dead-end until the first junction is met. Note that some passages won't become parts of dead end passages until other dead ends are removed first. A video of dead-end filling in action can be seen here:.

Dead-end filling cannot accidentally "cut off" the start from the finish since each step of the process preserves the topology of the maze. Furthermore, the process won't stop "too soon" since the end result cannot contain any dead-ends. Thus if dead-end filling is done on a perfect maze (maze with no loops), then only the solution will remain. If it is done on a partially braid maze (maze with some loops), then every possible solution will remain but nothing more.

**Shortest path algorithm:**



A maze with many solutions and no dead-ends, where it may be useful to find the shortest path

When a maze has multiple solutions, the solver may want to find the shortest path from start to finish. There are several algorithms to find shortest paths, most of them coming from graph theory. One possible algorithm finds the shortest path by implementing a [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search), while another, the [A\* algorithm](https://en.wikipedia.org/wiki/A*_algorithm), uses a [heuristic](https://en.wikipedia.org/wiki/Heuristic) technique. The breadth-first search algorithm uses a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)) to visit cells in increasing distance order from the start until the finish is reached. Each visited cell needs to keep track of its distance from the start or which adjacent cell nearer to the start caused it to be added to the queue. When the finish location is found, follow the path of cells backwards to the start, which is the shortest path. The breadth-first search in its simplest form has its limitations, like finding the shortest path in weighted graphs.

**PROPOSED ALGORITHM:**

The proposition of this project is in two phase. In the first phase the robot follows wall following algorithm and finds a way out of the maze. But the sensors aboard will keep recording the pathway. While returning back, a preprocessing will be done on the matrix of pathway and the shortest path will be figured out.

**Source Code-**

int pingPin1=7;

int pingPin2=8;

int motorpin1=12;

int motorpin2=13;

int reverse\_switch=10;

int temp;

int pathway[2][20];

int k=0;

int i=0;

int count=0;

int one\_time=0;

int pointer=0;

void setup()

{

Serial.begin(9600);

pinMode(motorpin1,OUTPUT);

pinMode(motorpin2,OUTPUT);

pinMode(reverse\_switch,INPUT);

}

void loop()

{

temp=digitalRead(reverse\_switch);

if(temp==1)

{

long duration1;

long duration2;

int cm1;

int cm2;

pinMode(pingPin1, OUTPUT);

pinMode(pingPin2, OUTPUT);

digitalWrite(pingPin1, LOW);

digitalWrite(pingPin2, LOW);

delayMicroseconds(2);

digitalWrite(pingPin1, HIGH);

digitalWrite(pingPin2, HIGH);

delayMicroseconds(5);

digitalWrite(pingPin1, LOW);

digitalWrite(pingPin2, HIGH);

pinMode(pingPin1, INPUT);

pinMode(pingPin2, INPUT);

duration1 = pulseIn(pingPin1, HIGH);

duration2 = pulseIn(pingPin2, HIGH);

cm1 = microsecondsToCentimeters(duration1);

cm2 = microsecondsToCentimeters(duration2);

Serial.print(cm1);

Serial.print("cm");

Serial.print(" ");

Serial.print(cm2);

Serial.print("cm");

Serial.println();

if((cm1==1)&&(cm2>1))

turn\_straight();

if((cm1==1)&&(cm2==1))

turn\_right();

if((cm1>1)&&(cm2>1))

turn\_left();

delay(10);

}

if(temp==0)

{

if(one\_time==0)

{

maze\_solver();

one\_time=1;

}

for(j=0;j<k;j++)

{

while(pointer<=pathway[1][j])

{

digitalWrite(motorpin1,HIGH);

digitalWrite(motorpin2,HIGH);

delay(10);

pointer++;

}

pointer=0;

if(pathway[0][j]==-1)

{

turn\_left();

delay(10);

}

if(pathway[0][j]==1)

{

turn\_right();

delay(10);

}

}

}

}

void turn\_straight()

{

digitalWrite(motorpin1,HIGH):

digitalWrite(motorpin2,HIGH);

count++;

}

void turn\_left()

{

digitalWrite(motorpin1,LOW):

digitalWrite(motorpin2,HIGH);

pathway[0][i]=-1;

pathway[1][k]=count;

i++;

k++;

count=0;

}

void turn\_right()

{

digitalWrite(motorpin1,HIGH):

digitalWrite(motorpin2,LOW);

pathway[0][i]=1;

pathway[1][k]=count;

i++;

k++;

count=0;

}

void maze\_solver()

{

int mat[4];

mat[0]=-1;

mat[1]=1;

mat[2]=1;

mat[3]=-1;

int pointer1=0;

int z=0;

for(z=0;z<k;z++)

{

if((pathway[0][z]==-1)&&(pathway[0][z+1]==1)&&(pathway[0][z+2]==1)&&(pathway[0][z+3]==-1))

{

pathway[1][z]=pathway[1][z]+pathway[1][z+1]+pathway[1][z+2]+pathway[1][z+3];

}

}

}

long microsecondsToCentimeters(long microseconds)

{

return microseconds / 29 / 2;

}

ADVANTAGES :

1. It is fast and a direct method to solve a maze.
2. It involves minimum utilization of sensors and hence power consumption.
3. It can solve any type of maze.