Finding the shortest path for a mobile robot in an unmapped maze from minimum runs

Sameer Singh
email : **sameeer_s@yahoo.com**
83 / ECE / 2000
3rd year, Electronics & Communications Engineering
Netaji Subhas Institute of Technology
New Delhi

B-14, IARI
Pusa Campus
New Delhi - 110012

Abstract
This paper describes a software which can be used to program robots to find the shortest path to any point in an unmapped maze. The software contains integration of various different calculations to decide at each point where to go, and in the end, to use further optimizations to minimise the distance traveled. Also developed is a simulation software with features like single-stepping, etc. and a random maze generator to test the program.

The paper describes how the simple maze solving capability can be extended to other applications.

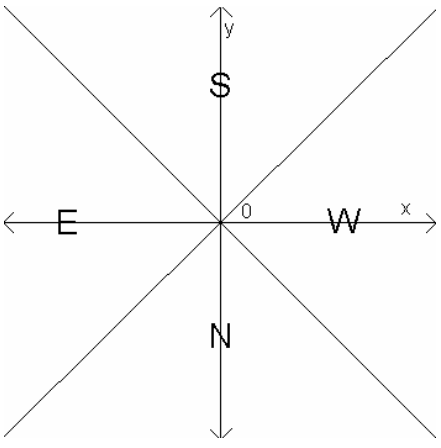
Objective
Robot navigation is a field that is constantly being studied and new ideas and methods are being introduced everyday. Each of these ideas is very important, and their relative importance differs with each application. Navigation of a vehicle on a road is very different from that of a robot inside a mine.

The objective of this paper is to study the navigation of a mobile robot in the most elementary form. A software model, with code, has been designed for this purpose. I have attempted to come up with a solution to a maze-solving problem, which can then later be modified and applied to other applications.

The maze-solving problem comprises of a mobile robot within a maze. The maze is rectangular, with equi-sized square cells, with a wall absent or present on each side. The robot is a small mobile vehicle with sensors to detect the *immediate* presence of a wall. It can turn in any direction. Most importantly, it has no prior knowledge of the structure of the map. It is required to start at any cell, facing any direction, and is required to find the shortest distance to a given point in the maze. It may be allowed any number of runs; runs being each time the robot starts again.

The algorithm presented finds the minimum distance, and the path to be followed, in a very small number of runs, by using different optimizations. The software code can directly be programmed into any robot to enable it to find the shortest route.

Theory
As is common in any such application, recursive logic is used to try and traverse through all the points, and find the shortest distance. In software, this is an acceptable solution, assuming some optimizations, but in hardware, there is more at stake than just time. Each run costs fuel, maintenance cost, etc., which have to be reduced. Thus different optimizations have to be used, to minimise the time required to get the shortest path.



Recursive Logic : According to this logic, each cell visited is stored as a node in a stack. Thus whenever the robot visits a new cell, a new node is generated in the stack. A node is popped from the stack only when all directions from the node have been visited.
Weighted Direction : Each direction is given a certain weight, according to the current cell and the target cell coordinates. This direction helps in deciding which direction to try out first. Initially the four variables (prior[N], prior[S], prior[E] & prior[W]) are set to zero. The weights are calculated as given below.

x = Current_x - Target_x
y = Current_y - Target_y

if x > 0	*:*	*prior[W] += 2 prior[N] += 1 prior[S] += 1*
if x < 0	*:*	*prior[E] += 2 prior[N] += 1 prior[S] += 1*
if y > 0	*:*	*prior[S] += 2 prior[E] += 1 prior[W] += 1*
if y < 0	*:*	*prior[N] += 2 prior[E] += 1 prior[W] += 1*
if x > y	*:*	*prior[W] += 2 prior[N] += 2*
if x < y	*:*	*prior[E] += 2 prior[S] += 2*
if x > -y	*:*	*prior[W] += 2 prior[S] += 2*
if x < -y	*:*	*prior[E] += 2 prior[N] += 2*

These priorities are based on the area shared by each of the direction quadrants, with each of the domains. Half area of the quadrant corresponds to 1 prior unit.

Visited Map : This is a data structure representing the map (and the cells) in a Boolean representation to store which cell has been visited. Thus the cells which have already been popped out of the stack are stored as visited in the map. There are two problems if this is not impleme nted. Firstly, a cell maybe pushed into the stack twice, and result in a indefinite loop. Secondly, the robot may get into a path through which it has already gone and rejected, thus unnecessarily wasting time and energy. Thus this logic takes care of these problems.

After the robot reaches the target cell, it uses two more routines to further minimise the distance.

U Turn Removal : This logic analyses the path attained to find if there are two cells in the path, which are physically adjacent with no wall between them, but in the path are not together. This occurs in case of a *u-turn* situation. If this has happened, then the path between them is redundant and is removed. The shortest distance is recalculated.

Backstepping : The robot tries to retrace one step at a time, and tries to visit cells which hadn't been visited before. If the distance overshoots the already calculated distance, it again retraces one step. Thus suppose a variable *n* is defined as 0. Whenever the robot enters a new cell, n is incremented, and whenever the robot retraces its path, it is decremented. *n* is not allowed ever to have positive values (i.e. the new distance will always be smaller than already calculated). If the robot reaches the target cell, it means a shorter path has been found. The already calculated shortest distance is decremented by *n* and the path is updated.

Thus, even though the prioritizing logic may not be flawless, the following optimizations guarantee a very fast calculation of the shortest path.

Data Structures

A node in this program represents a cell, with the directions a robot will be allowed to go to, arranged according to the priority routine. Thus there is a stack of directions, which store the directions robot should go to from this cell. Each time the robot leaves a node, the direction in which it leaves is popped from the direction stack, since it doesn't need to go there again. Once the direction stack is empty, it denotes the corresponding cell does not figure in the shortest path, and the robot has to leave in the direction it came from. There are also variables in the node storing the cell's coordinates. A node has subroutines to push and pop directions.

Each node (cell) is pushed and popped in to another stack, which represents the current path. Each time a new node is visited, it is added to the stack. Each time the robot runs out of directions to go to from a cell, and has to go back, the topmost cell is popped. Thus when the target cell is reached, this stack corresponds to the shortest path. The path has subroutines to push and pop nodes.

There are other variables like environment variables (Target_x, Target_y, etc.) and the visited map. Subroutines include ones to prioritise, the recursive routine, the backstepping routine and the u-turn removal routine.

All these together build the basic blocks for the program to calculate and guide a robot to the target cell in the shortest time.

Code Segments

This algorithm was implemented using C++ (DJGPP Compiler). The code for the various different classes and sub routines is given below for further explanation of the program.

The classes have been defined as follows.

```
class node
{
  public:
  node *next;
  node *prev;
  int x;
  int y;
  char size;
  char list[4];
  node() { size=0; list[0]=list[1]=list[2]=list[3]='x'; }
  void pop()
  { int i=0; size--;
    for(i=0;i<size;i++) list[i]=list[i+1];
    list[i]='x';
  }
};

class stack
{
  public:
  node *top;
  node *bottom;
  stack() { top=bottom=NULL; }
  void push(int x, int y);
  void pop();
}path;
```

The *prior()* function sets priority to the different directions, setting the least priority to the direction the robot came from, and arranges it in descending order.

```
void prior(node &a)
{
//  according to x and y, and
//  centre x & y, the function
//  sets priority to each direction
//  setting least to ori. It also checks
//  where walls are and inserts the
//  directions into the "list" stack
  float xp=float(x)-target_x;
  float yp=float(y)-target_y;
  char dir[4]={0};
  if(yp>0) { dir[SOUTH]+=2; dir[EAST]++; dir[WEST]++; }
  if(yp<0) { dir[NORTH]+=2; dir[EAST]++; dir[WEST]++; }
  if(xp>0) { dir[WEST]+=2; dir[SOUTH]++; dir[NORTH]++; }
  if(xp<0) { dir[EAST]+=2; dir[SOUTH]++; dir[NORTH]++; }
  if(xp>yp) { dir[WEST]+=2; dir[NORTH]+=2;}
  if(xp<yp) { dir[EAST]+=2; dir[SOUTH]+=2;}
  if(xp>(-yp)) { dir[WEST]+=2; dir[SOUTH]+=2;}
  if(xp<(-yp)) { dir[EAST]+=2; dir[NORTH]+=2;}
  if((dir[NORTH]==dir[EAST])||(dir[NORTH]==dir[WEST]))
       dir[NORTH]++;
  if((dir[SOUTH]==dir[EAST])||(dir[SOUTH]==dir[WEST]))
       dir[SOUTH]++;
  dir[(ori+2)%4]=0; //Setting least priority to back dir
  for(int i=0;i<4;i++)
    if(map[x][y].wall[i]==0)
    {  a.list[a.size]=i; a.size++; }
//arranging the list in descending order
  for(int i=0;i<a.size-1;i++)
  {
    for(int j=i+1;j<a.size;j++)
      if(dir[a.list[i]]<dir[a.list[j]])
        {char t=a.list[i];a.list[i]=a.list[j];a.list[j]=t;}
  }
}
```

The main recursive function is given as follows. Comments explain what each section does.

```
int recur()
{
//  Main prog of the robot
//  It is not truly recursive in the sense that though it
//  is called in a loop, it does not call itself. therefore,
//  "going" to a cell means setting x and y (& ori) and coming out
//  of the recur function
//  1. check if visited if not
//  --- 1. check if target. if yes, return 1
//  --- 2. create new node in stack
//  --- 3. set prioritites to dir by using prior()
//  --- 4. set visited
//  2. if node.size>1 goto next cells acc to list if not visited.
//         -- if visited, pop dir.
//  3. if node.size=1 -- pop the node
```

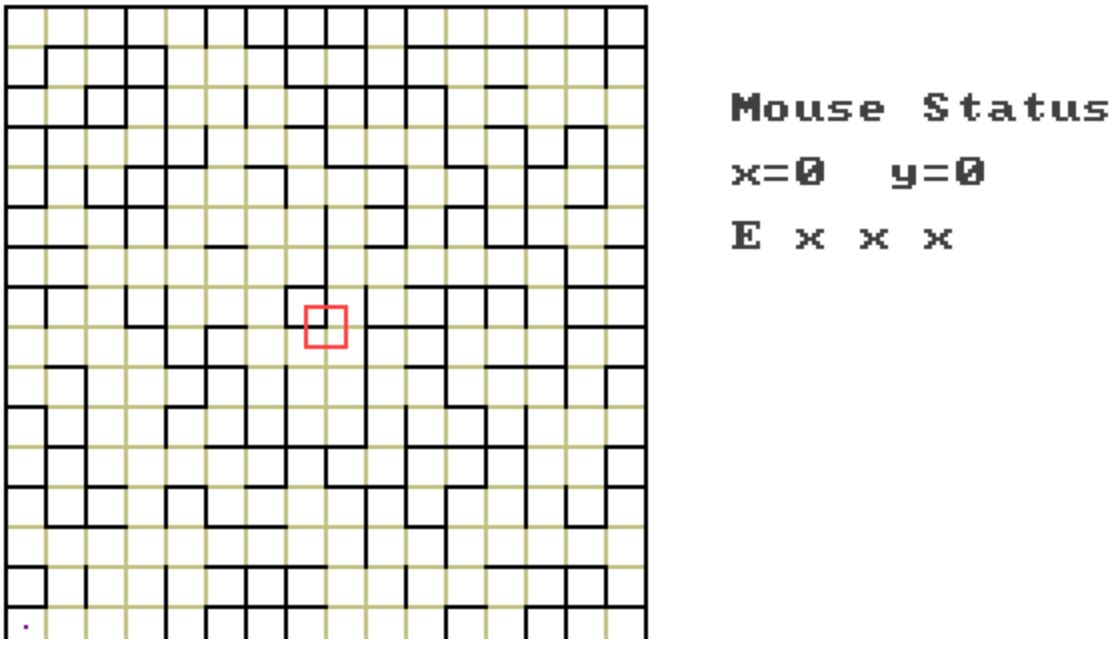
```
//     -- goto cell   -- pop dir. in top->list
//  4. return 0
  if(visited[x][y]==0)
  {
    if((x==target_x)&&(y==target_y)) return 1;
    path.push(x,y);
    prior(*path.top);
    visited[x][y]=1;
  }
  int xt, yt; char c=0,d;
  while((c==0)&&(path.top->size>1))
  {
    xt=x; yt=y; d=path.top->list[0];
    update(xt,yt,d);
    if(visited[xt][yt]==0) c=1;
    else path.top->pop();
  }
  if(c==1)
  { x=xt; y=yt; ori=d; }
  if(path.top->size==1)
  { update(x,y,path.top->list[0]); ori=path.top->list[0];
    path.pop(); path.top->pop();
  }
  if((x==startx)&&(y==starty)&&(path.top->list[0]=='x')) return 2;
  return 0;
}
```

These routines and classes, along with other subroutines, form the software implementation of the algorithm, which can be programmed into a mobile robot.

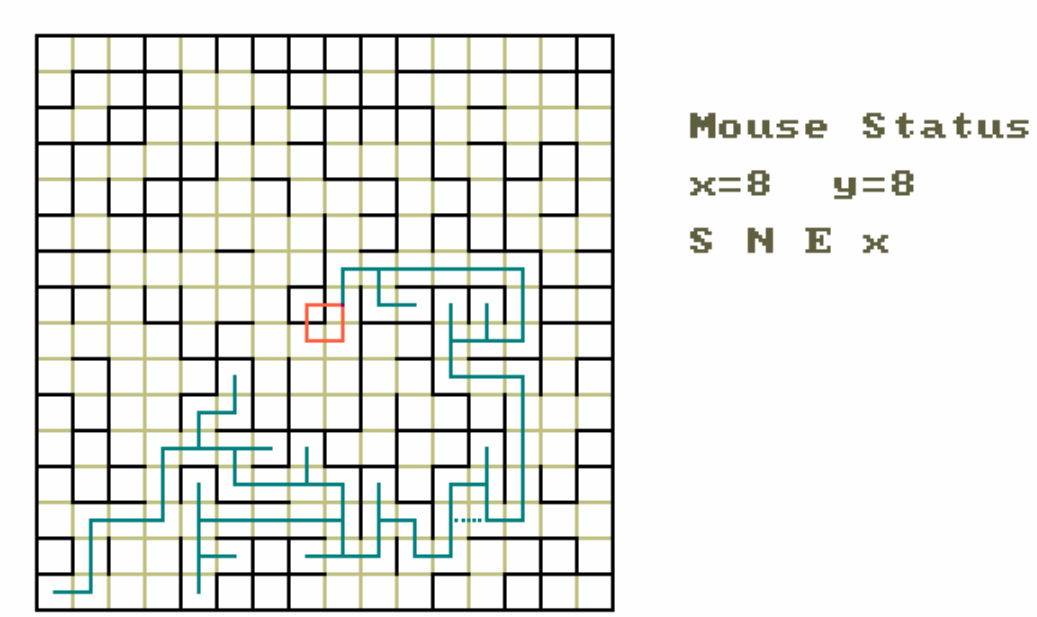
Simulation

A maze simulation program was also written alongside to test the robot with different kinds of mazes. It was also written in C++ (DJGPP Compiler) and had subroutines to input a maze, show the maze, plot any point, plot a path, print robot status and single step through each process.

For the simulation purposes, the maze I have taken has 16X16 cells, with the starting coordinates as (0,0), and the final cell being any of the four in the centre, i.e. (7,7), (7,8), (8,7) and (8,8). (see figure)



The first path is plotted with a distance of 42 steps, using only priority settings in mind.



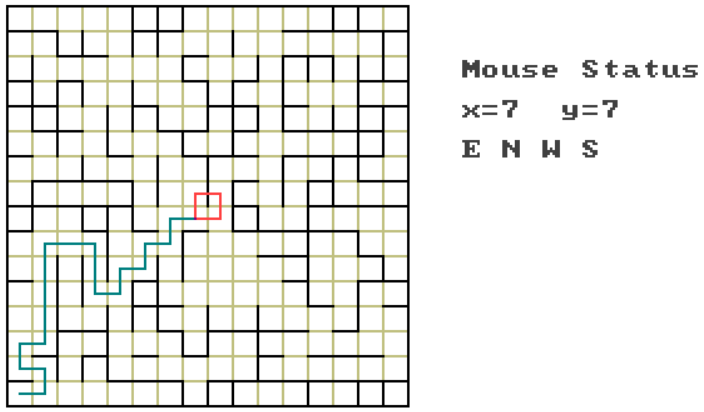
The second path is plotted with u-turn removals implemented (marked as dotted on the figure). The number of steps reduces to 40. The third path is calculated over and over again by using the back stepping technique. This results in same path length in this case. The u-turn removal is applied again to remove u-turns from new path. This is undoubtedly the shortest path to the centre of the given maze.

Thus the shortest distance was reached by not mapping most of the maze, and only the relevant cells were mapped. This saves time and robot memory. The simulation results were as expected, and was verified by using a random maze generator to create mazes of high complexity. This random maze generator, simulator and the code files altogether form a total package.

Here are some more mazes with their solution as simulated. The mazes have been created using random maze generator.

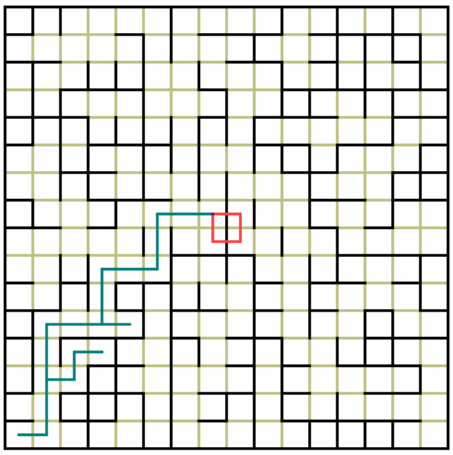
Simulation 1

This is an easy maze used to demonstrate how the prioritizing of each direction results in solutions that are perfect. At each point the robot (“mouse”) takes only the required direction.



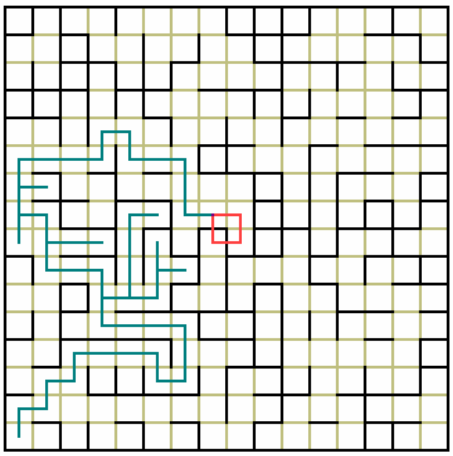
Simulation 2

This is a relatively tougher maze, but again the robot handles it with very little extra time.

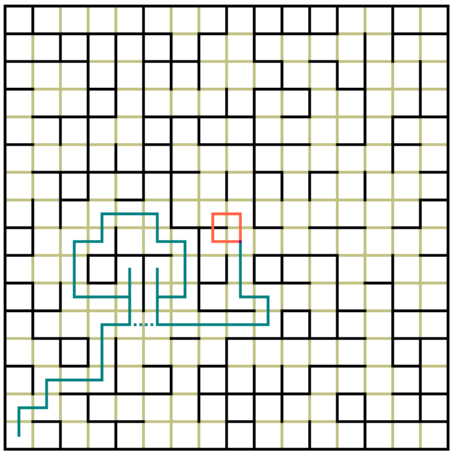


Simulation 3

This simulation contains a very tough maze. The main point to observe here is how the robot is tying to get towards the centre at every opportunity (*probing*), till it finally gets here.

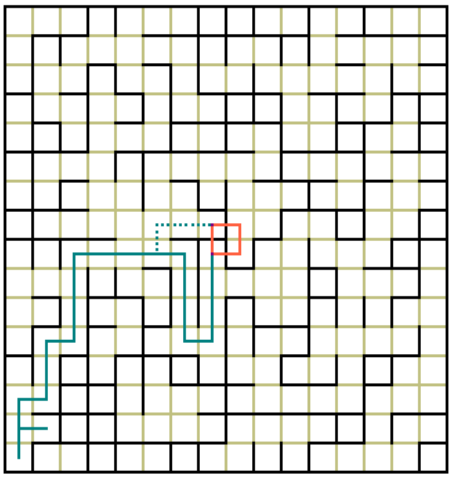


Simulation 4
This simulation shows how many steps can having a u-turn removal save (dotted line).



Mouse Status
x=8 y=7
N S x x

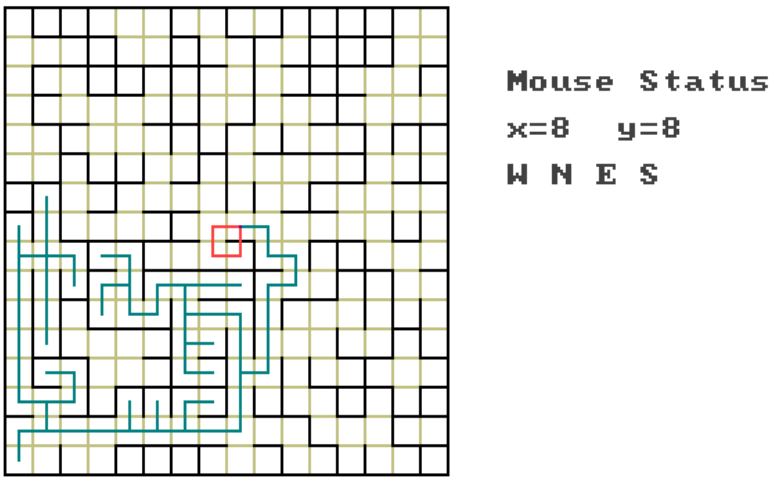
Simulation 5
This simulation demonstrates how many steps can be saved by back stepping (dotted line).



Mouse Status
x=7 y=7
N E S x

Simulation 6

In this simulation, the robot maps much more area than necessary to find the shortest path. This can be counted as a bug, and happens because of the priority code that sets higher priority to north and south over east and west if their priorities come out to be equal. This particular maze exploits this particular assumption. Using a random decider between N/S and E/W can solve this problem most impartially.



Conclusion

In developing this software, I have assumed the mobile robots to be basic vehicles with two degrees of freedom and basic sensing switches to detect the absence and the presence of a wall directly next to the robot. Also cells were assumed to be big enough to fit the robot, and had clearly defined walls. Keeping these assumptions in mind, the algorithm works well. In other applications, this may not be the case. If the algorithm is used for path finding over a rough terrain, then each cell has to be reduced in size for greater precision. Which means greater number of steps. Also, there are no clearly defined obstructions in a rough terrain, and each has to be treated with a factor set to it, which varies from obstruction to obstruction. Thus the algorithm has to be modified to a great degree, and loses the simplicity of operation.

Let us study under sea exploration, which is usually carried out by dividing the seabed into grids, and assigning each cell as an obstruction or not. Thus if a robot needs to navigate on the seabed to reach a particular point, this algorithm can be used.

A very good application of this algorithm can be in the multiple robot system. Here, a scout robot can use the algorithm to determine the most efficient path for the other robots, while they carry out the task at hand. For e.g. a block has to be carried through a factory which is unmapped. A fast scout robot can move ahead plotting the path while others, slowed due to the task at hand, slowly follow the path of the scout.

An extension of this algorithm could be to extend it to the third dimension, so it can be used for aerial navigation and underground work.

Thus there are many applications of this algorithm that extend beyond maze solving.

References

- *Artificial Intelligence*
  - Elaine Rich
  - ***McGraw Hill Publishing House***

- *Algorithms & Data Structures*
  - Niklaus Wirth
  - ***Prentice Hall***

- *Algorithms in C++*
  - Robert Sedgewick
  - ***Addison Wesley***