

# Inside the ER1 Mapper

By: Abraham L. Howell

## Abstract:

The ER1 Mapper (ER1M) is a software application that allows the ER1 user to create and save two-dimensional maps, solve for the path, and finally command an ER1 robot to traverse the solution path. The ER1 is a fully autonomous mobile robot manufactured by Evolution Robotics, Inc. In this paper the architecture of the ER1M will be discussed along with the specific implementation of the A\* Algorithm and actual code.

## ER1M Architecture:

The ER1M builds off the Robot Control Center (RCC), a Graphical User Interface (GUI) that is supplied with the ER1 robot. The RCC is great for the first time robot enthusiast, but the more advanced user will quickly lose interest. Also, the RCC doesn't provide any means for robot path planning or mapping. What is good about the RCC is that it provides access to a robot Application Programmers Interface (API). The ER1 API is easily accessed through TCP Sockets and can be programmed in the following languages: Visual Basic, Visual C++, Python, Java, or any language that supports TCP Sockets. The diagram below illustrates the ER1M architecture.

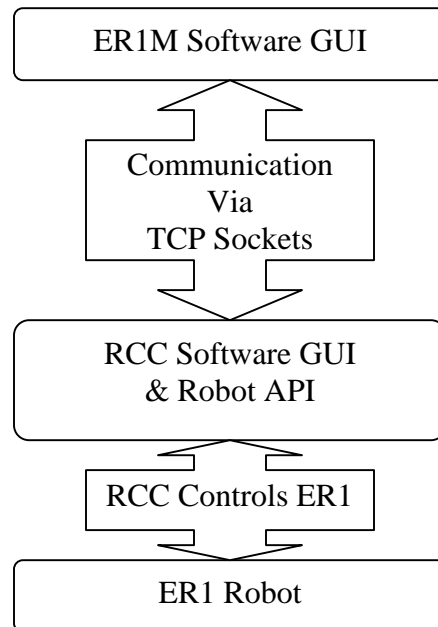


Figure 1. Architecture of ER1M.

It can be seen from figure 1 that the ER1M sits above the RCC and has access to the ER1 Robot through this TCP Socket connection. The RCC has a direct connection to the ER1 robot and can request sensor readings, camera images or command the ER1 to move.

Once a TCP Socket connection has been properly established with the RCC, the ER1M has the ability to command the ER1 robot to move along a specified path or in this case follow the solution path for a user created map. Since the ER1M was written using Visual Basic 6, creating the TCP Socket connection with the RCC was relatively simple. All one needs to do is hit Ctrl+T to enter the Project Components menu in the Visual Basic 6 IDE. Next, scroll down to find the Microsoft Winsock Control 6 (SP5), check the box and click OK. Now simply drag and drop the Winsock control to your project's form. You now have access to the Winsock control in your project. Later on I will discuss the proper configuration and use of this control.

### **The A\* Algorithm:**

The ER1M utilizes the A\* algorithm to solve for the desired path around obstacles within a created or opened map. Numerous role-playing computer games make use of the A\* algorithm. The current implementation of the A\* algorithm in the ER1M makes use of two heuristics: the distance from the start cell to the current cell, and the distance from the current cell to the goal cell. Setting one of the heuristics to be dominant will change the solution behavior. For example, by favoring the first heuristic,  $D_g$ , the solver will search more cells and try to find the optimal path. Whereas when favoring the second heuristic,  $D_h$ , the solver will explore fewer cells, find a path quicker, but in most cases it will not find the optimal path. In this paper the optimal path is defined as the path with the shortest length.

The A\* algorithm essentially utilizes lists to explore possible solution paths. The two lists used are called Open and Closed. The algorithm begins by adding the start cell to the Closed list and then proceeds to investigate the four surrounding cells: north, south, east, and west. If any of the four surrounding cells are a possible path and aren't already on the Open list or Closed list they will be added to the Open list and the cell's parent will be set to the currently investigated cell or in this case the start cell. Next, the cell on the Open list with the lowest F score will be removed from the Open list and added to the Closed list. A cell's F score is composed of two components: the G score and H score. The H score for a cell remains constant throughout the solution process and is calculated using the Manhattan distance, which is simply the sum of the x and y distances from the current cell to the goal cell. On the other hand, the G score for a cell can change during the solution. The G score is calculated by taking the sum of the x and y distances from the current cell to the start cell. However, the G score of a cell will change if the G score is lower when using the currently investigated cell's path. If this is the case, the G score for that particular cell must be updated. The  $d_g$  and  $d_h$  heuristics will directly affect the G and H scores since they are used in the respective calculations. At the end of one iteration the A\* algorithm will check to see if it has found the goal cell and if not it will loop back and investigate the just chosen cell with the lowest F score. If the goal cell were indeed found then the solution would be displayed to the user. Simply starting with the goal cell

and then proceeding to the parent cell until the start cell is discovered will back track the solution path. A diagram below helps to explain the A\* algorithm solution process.

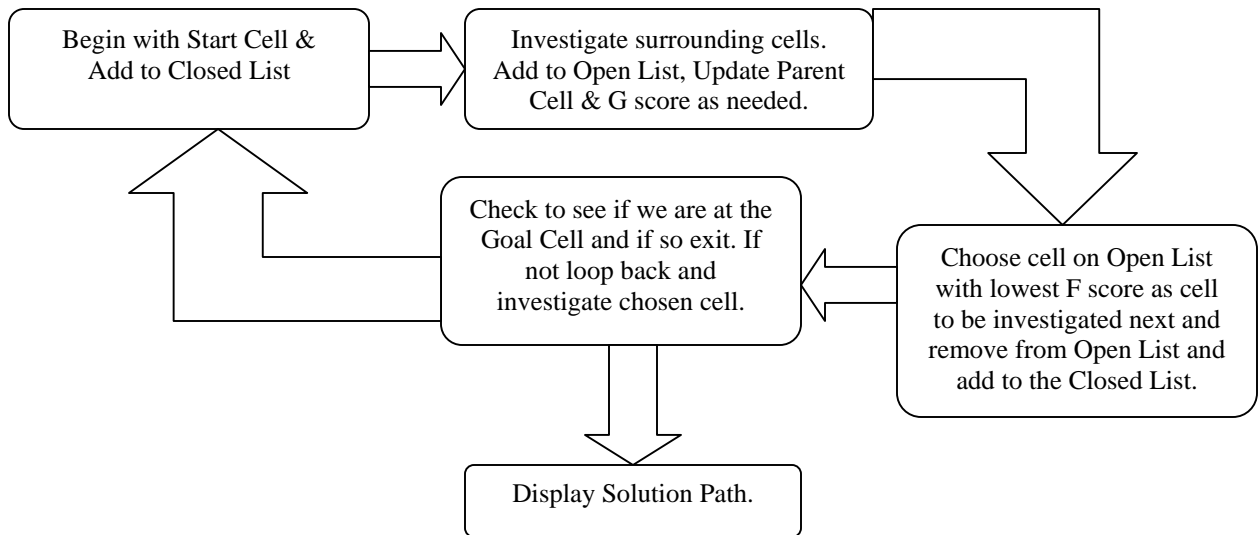


Figure 2. Diagram of the A\* Algorithm.

### Actual Code:

The entire ER1M application has been written using Visual Basic 6, but as was previously mentioned this could have been written using any language that supports TCP Sockets. Visual Basic 6 was chosen due to its ease of use with regards to the Microsoft Winsock Control, which allows you to configure and create a TCP Socket connection. Once the socket control has been added to your project it must be properly configured before it can connect with the RCC. To do this you must add two lines of code to the Form\_Load Sub and they are as follows.

```

Winsock1.RemoteHost = "localhost"
Winsock1.RemotePort = "9000"

```

The first line will set the remote host to be the laptop or computer that is directly connected to the ER1 robot. The second line sets the remote port to be 9000, which is the port that the RCC is listening to and waiting for a connection to be requested. Connecting to the RCC is simple and can be accomplished in one line of code.

```

Winsock1.Connect

```

However, to be sure that a proper connection has been established we will loop until the connection is made or a timer times out. The code is listed below.

```

Do Until Winsock1.State = sckConnected
DoEvents

```

```

If Connect_TimedOut = True Then
    GoTo Connection_TimeOut
End If
Loop

```

Now that we've established a successful connection to the RCC we have access to all the API functions. We will only be using two commands: move and events. The move command can be used to move the ER1 robot linearly or angularly. To move the ER1 linearly you must supply the move distance, move units, and a line feed carriage return (vbCrLf). To move the robot forward 12 inches you would send the following command through the TCP Socket connection.

```
Winsock1.SendData "move 12 i" & vbCrLf
```

We must also send the events command because we want to know when the ER1 finishes the requested command or movement.

```
Winsock1.SendData "events" & vbCrLf
```

To monitor when a requested event has finished we need to create a Sub to receive the data from the socket connection. Once the data has been received we still need to parse it to find out if the request was successful or not.

```

Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
Dim strData, mbr As String
Winsock1.GetData strData, vbString
Dim StartPos, n(2) As Integer
lblER1Messages.Caption = strData

```

```

If (InStr(1, UCase(strData), "DONE", vbTextCompare) > 0) Then
    ER1MoveDone = True
    Exit Sub
End If

```

```

'Now parse the received data for any error messages
If InStr(1, UCase(strData), "ERROR", vbTextCompare) > 0 Then
    ER1MoveError = True
    'Then an Error has occured and we need to determine which error!
    If InStr(1, UCase(strData), "CANNOT CONNECT TO THE ROBOT HARDWARE", vbTextCompare) >
0 Then
        mbr = MsgBox("Cannot connect to the ERI hardware. Check to be sure that the ERI is powered up.",
vbCritical, "Connection Error!")
    End If
End If
Exit Sub
End Sub

```

The A\* algorithm code was placed in the MapFunctions Module so that it would be more modular and easily understood. The first several functions and subs deal with either initialization, clearing or drawing the grid, and drawing a loaded map file. I will not go

into any detail regarding these simple functions or subs since they are fairly straightforward and easy to understand. The actual algorithm is contained within the SolvePath function, which will return a string that explains whether or not the solution was successful and if so the details regarding the solution. Looking through the SolvePath function code you will see that it follows the general flow of figure 2, which outlines the A\* algorithm. Additionally, the code in this function has been well documented and should be easy to follow, but let's start at the beginning of the SolvePath function and go into a little more detail.

You will first notice that all the variables and arrays are initialized. The first sub routine call is made to Calculate\_Cell\_Heuristics. In this sub the two heuristics are calculated for each of the 2500 cells. This is where the previously mentioned G and H scores originate from and make use of the dg and dh heuristics. The next step involves setting the current cell to the start cell using the following code.

```
'Begin with the startcell  
curCell.X = mvarStartCell.X  
curCell.Y = mvarStartCell.Y
```

Once the current cell has been set we jump inside the do while loop, where we will remain until either the goal cell is found or the Open list is empty. The first task to accomplish once inside the loop is to determine the north, south, east, and west cells bordering the current cell and then check each one to see if its unblocked and also if its on the Open or closed list. We determine which cells border the current cell with the following code.

```
Ncell.X = curCell.X  
Ncell.Y = curCell.Y - 1
```

```
Scell.X = curCell.X  
Scell.Y = curCell.Y + 1
```

```
Wcell.X = curCell.X - 1  
Wcell.Y = curCell.Y
```

```
Ecell.X = curCell.X + 1  
Ecell.Y = curCell.Y
```

We start with the north cell and first check to be sure that it is within range of our map grid. If the cell isn't in range we simply jump to check the south cell.

```
If ((Ncell.X < 0) Or (Ncell.Y < 0) Or (Ncell.X > 50) Or (Ncell.Y > 50)) Then  
    GoTo SouthCell  
End If
```

However, if the north cell is within range we must check to see if it's already on the Closed list. Then if it's not on the Closed list we check to see if it's on the Open list. If not on the Open list we first add it and then reset its parent cell, but if its already on the Open list we check to see if the path from the current cell yields a lower F score and if so we recalculate and reset the parent cell. The code is listed below.

```

If ((CellStatus(Ncell.X, Ncell.Y) = False) And (InClosedList(Ncell.X, Ncell.Y) = False)) Then
    'if its not already in the openlist then add it and set the parent
    If (InOpenList(Ncell.X, Ncell.Y) = False) Then
        AddTo_OpenList Ncell.X, Ncell.Y
        ParentCell(Ncell.X, Ncell.Y) = curCell
    Else
        'Check to see if the path from the current cell has a lower G value
        If (G(curCell.X, curCell.Y) + Dg < G(Ncell.X, Ncell.Y)) Then
            'If so, set the parent to be the current cell
            'and recalculate F & G scores for the Ncell
            ParentCell(Ncell.X, Ncell.Y) = curCell
            G(Ncell.X, Ncell.Y) = G(curCell.X, curCell.Y) + Dg
            F(Ncell.X, Ncell.Y) = G(Ncell.X, Ncell.Y) + H(Ncell.X, Ncell.Y)
        End If
    End If
End If

```

After completing the above for the north cell we simply do this for the remaining south, west, and east cells. Once this is complete we need to check the Open list to see if it's empty or not. We've exhausted all possible solution paths and must exit if the Open list is empty, but if not we must continue on.

```

If OpenListEmpty = True Then
    Exit Do
End If

```

Next, we need to pick a cell from the Open list, which has the lowest F score and this cell will be investigated during the next loop. After selecting this cell it must be removed from the Open list and added to the Closed list.

```

curCell.X = OpenList(UBound(OpenList)).X
curCell.Y = OpenList(UBound(OpenList)).Y

For indx = 0 To UBound(OpenList)
    If F(OpenList(indx).X, OpenList(indx).Y) < F(curCell.X, curCell.Y) Then
        curCell = OpenList(indx)
    End If
Next

AddTo_ClosedList curCell.X, curCell.Y
RemoveFrom_OpenList curCell.X, curCell.Y

```

Finally, we must check to see if we've actually found the Goal cell and if so exit the loop and display the solution path on the grid.

```
If ((curCell.X = mvarGoalCell.X) And (curCell.Y = mvarGoalCell.Y)) Then  
    'we've found a suitable path to the goal cell!  
    GoTo Show_Path  
End If
```

If the Goal cell wasn't found then highlight the cell that was just added to the Closed list and loop back for another run.

```
If (((curCell.X > 0) And (curCell.X < 51)) And ((curCell.Y > 0) And (curCell.Y < 51)))  
Then  
    If (ShowInvestigatedCells = True) Then  
        xy1.X = (curCell.X) * 180 + 98  
        xy1.Y = (curCell.Y) * 180 + 98  
        frmMap.ForeColor = vbButtonFace  
        frmMap.FillColor = vbYellow  
        frmMap.FillStyle = 0  
        frmMap.Circle (xy1.X, xy1.Y), 75  
    End If  
End If
```

This will conclude my discussion on the ER1M. For information regarding the use of this application please check out "Using the ER1 Mapper".

**References:**

<http://theory.stanford.edu/~amitp/GameProgramming/>

<http://www.policyalmanac.org/games/aStarTutorial.htm>

<http://www.evolution.com>