```cpp
/**
    Robust Energy Efficient Multi-path Routing Protocol for Wireless Sensor Networks
*/

#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
using namespace std;

//Global variables
struct Node
{
    int sid;    //Sensor ID
    float x,y;  //Co-ordinates
    float R_EN; //Residual Energy
    float BUF_C; //Remaining Buffer Capacity

};
int n;  //No. of sensors
float TH;   //Threshold Value
int MaxSNR;   //Max Value of SNR
int ReqSNR;     //Required value of SNR
float T_Range;  //Transmission range
float dist; //Distance
int Source;    //Source ID
int Sink;   //Sink ID
int pathcount; //path count
int H_Msg; //Hello Message
float **N_Table;  //Neighbor Table
Node *S;    //Sensors

void End()
//The only way for program to End
{
    cout<<"No More Paths Found"<<endl;
    exit(69);
}

int find_index(int id)
//Returns array index of node, given the sensor ID
{
    int s_index;
    for(int i=0;i<n;i++)
    {
        if(S[i].sid == id)
        {
            s_index = i;
            break;
        }
    }
    return s_index;
}

void Send_RREQ(int i,int j)
//Sends RREQ from node index i to j
{
    if(S[j].sid != Source)
        N_Table[j][n] = 1;
    if(i == find_index(Sink))
        cout<<"\nPath No. "<<pathcount+1<<"\n";

    cout<<i<<"->"<<j<<"\n";
}

int Best_link(int sid)
```

```cpp
//returns ID of best quality node out of all available neighbor nodes
{
    int nid,s_index;
    float val=0;
    s_index = find_index(sid);

    for(int i=0;i<n;i++)
    {
        if( (N_Table[s_index][i] != 0) && (N_Table[i][n] == 0) )
        {
            if(val < N_Table[s_index][i])
            {
                val = N_Table[s_index][i];
                nid = S[i].sid;
            }
        }
    }
    if(val == 0)
    {
        return -1;
    }
    return nid;
}

void Pathselection()
//Selects a path for Data Transmission
{
    int sid,s_index,check;
    cout<<"\nPath Selection Started"<<endl;
    s_index = find_index(Sink);

    if(N_Table[s_index][n]==0)
    {
        while(true)
        {
            check = 0;
            for(int i=0;i<n;i++)
            {
                if(N_Table[s_index][i]!=0)
                {
                    if(S[i].sid == Source)
                    {
                        check = 1;
                        Send_RREQ(s_index,i);
                        break;
                    }
                }
            }
            if(check == 1)
            {
                pathcount++;
                break;
            }
            else
            {
                N_Table[s_index][n] = 1;
                sid = Best_link(S[s_index].sid);
                Send_RREQ(s_index,find_index(sid));
                s_index = find_index(sid);
            }
        }
    }
    else
    {
        int nid;
        while(S[s_index].sid != Source)
        {
            nid = Best_link(S[s_index].sid);
            if(nid == -1)
                End();
            Send_RREQ(s_index,find_index(nid));
            s_index = find_index(nid);
        }
        pathcount++;
    }
}

void DisplayNTable()
//Displays the Neighbor Table
{
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            cout<<N_Table[i][j]<<"\t\t";
        }
        cout<<"\n";
    }
    cout<<"\n";

    while(true)
        Pathselection();
}

float HELLO(int i, int j, int Hello)
//Sends Hello Message from node index i to j
{
    float SNR,Val;
    int temp = rand();
    SNR = ((temp%(MaxSNR-ReqSNR)+ReqSNR)/(float)MaxSNR)*100;
    Val = (SNR + S[j].BUF_C + S[j].R_EN)/3.0;
    return Val;
}

float Dist(int i, int j)
//Returns distance between 2 given nodes
{
    float d;
    d = sqrt(pow((S[i].x-S[j].x),2)+pow((S[i].y-S[j].y),2));
    return d;
}

void NeighborSelection()
//Finds which nodes are neighbors of each other
{
    cout<<"Neighbor Selection Started\n\n";
    int visited[n] = {0};
    int Q[n];
    int top = -1;
    int point = 0;
    int index;
    int check=1;

    Q[++top] = find_index(Source);

    while(check)
    {
        index = Q[point++];
        visited[index] = 1;

        for(int i=0;i<n;i++)
        {
            if(!visited[i])
            {
                dist = Dist(index,i);
                if(dist < T_Range)
                {
                    float val = HELLO(index,i,H_Msg);
                    if(val > TH)
                    {
                        N_Table[index][i] = val;
                        N_Table[i][index] = val;
                        Q[++top] = i;
                    }
                }
            }
        }
        check = 0;
        for(int i=0;i<n;i++)
        {
            if(visited[i]==0)
            {
                check = 1;
                break;
            }
        }
    }
    cout<<"Neighbor Selection Ended\n\n";
    DisplayNTable();
}

void inputfromfile()
//Inputs Sensor Information from infile.txt
{
    ifstream infile;
    infile.open("infile.txt");
    for(int i=0;i<n;i++)
    {
        infile>>S[i].sid;
        infile>>S[i].x;
        infile>>S[i].y;
        infile>>S[i].R_EN;
        infile>>S[i].BUF_C;
    }
    infile>>Source;
    infile>>Sink;

    cout<<"Input Complete\n\n";

    NeighborSelection();
}

void Initialization()
//Inputs Basic Information from initfile.txt and allocates memory to arrays
{
    ifstream infile;
    infile.open("initfile.txt");
    infile>>n;
    infile>>T_Range;
    infile>>MaxSNR;
    infile>>ReqSNR;
    infile>>TH;

    cout<<"Initialization Complete\n\n";

    //Memory Allocation for Arrays:
    S = new Node[n];
    N_Table = new float*[n];
    for(int i = 0; i < n; ++i)
        N_Table[i] = new float[n+1];

    //Other Initializations :

    pathcount = 0;

    for(int i=0;i<n;i++)
        for(int j=0;j<=n;j++)
            N_Table[i][j] = 0;

    //Calling the input function
    inputfromfile();
}

int main()
//Just Calls Initialization Method
{
    Initialization();
    return 0;
}
```