CS5346: Advanced Artificial Intelligence

Project -2 : Spring 2023

SUDOKU-PUZZLE GAME USING A* ALGORITHM

Under the guidance of Dr.Moonis Ali

Submitted By: Sai Pranav Datrika

Table of Contents

1. Problem Description

The goal of the project is to solve the sudoku puzzle game by implementing A* search algorithm .We must be writing the source code using the A* algorithm.Every team member should be creating their own heuristic function.The performance of the algorithms and heuristic functions by running the program for a heuristic function .

The objective of the sudoku game is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 subgrids contain all of the digits from 1 to 9.The problem of solving a Sudoku puzzle can be modeled as a search problem, where the goal is to find a valid assignment of numbers to the empty cells of the grid. The A* search algorithm can be used to efficiently solve the Sudoku puzzle problem by exploring the search space in a heuristic-driven way.

To apply A* search to solve Sudoku puzzles, we can represent the puzzle as a search graph, where each node in the graph represents a partially-filled Sudoku grid, and each edge represents a valid move that leads to a new partially-filled grid. The search algorithm starts from an initial grid and tries to find a path to the goal state, where all the cells are filled with digits from 1 to 9 and the constraints of the game are satisfied.

The A* search algorithm uses a heuristic function to guide the search towards the goal state. The heuristic function estimates the cost of reaching the goal state from the current state. In the case of Sudoku puzzles, the heuristic function can be based on the number of empty cells in the grid, the number of constraints that are violated in the current grid, or a combination of both.

The A* search algorithm works by maintaining a priority queue of nodes to be expanded. Each node in the stack is assigned a priority value, which is the sum of the cost of reaching the node from the start state and the estimated cost of reaching the goal state from the node. The algorithm repeatedly selects the node with the highest priority from the stack, expands it by generating its successor nodes, and adds them to the stack with their priority values.By using A* search with an appropriate heuristic function, we can efficiently solve even the most difficult Sudoku puzzles, where the search space is very large and the constraints of the game are complex.

Here are the rules of the Sudoku puzzle game:

- Each row must contain all the digits from 1 to 9.
- Each column must contain all the digits from 1 to 9.
- Each 3x3 sub-grid must contain all the digits from 1 to 9.

In the implementation of the game using the A* algorithm, the algorithm will continue searching until it finds a valid solution or exhausts all possible combinations of digits for each cell in the grid. Once the algorithm finds a valid solution, it will output the completed Sudoku puzzle grid as the final result.

It's important to note that Sudoku puzzles are designed to have a unique solution, which means that there should only be one possible arrangement of digits that satisfies all the rules of the game. If a Sudoku puzzle has more than one valid solution, it is not considered a valid puzzle.

2. Domain of the problem:

Game playing is a significant application of artificial intelligence. Unlike other domains, games require minimal knowledge - just the rules, available moves, and the criteria for winning or losing. To play games, we perform searches and use various search techniques. However, some algorithms such as BFS can slow down the search process since they generate a large number of successors, and the search continues for all the nodes. Thus, we must employ other search procedures that enhance the generating and testing procedures. The generating procedure should consider only the best move, whereas the testing procedure should choose the best move from the expanded tree. Using artificial intelligence in games increases the probability of the computer winning and makes the game more challenging and engaging for players.

3. Methodologies:

The game program is implemented using A* search algorithm. This algorithm helps in generating the moves and making the best possible move. This algorithm helps to reduce the time in searching and generating the possible nodes.

3.1 A* SEARCH ALGORITHM

The A* search algorithm is a widely used search algorithm that is used in many fields, including artificial intelligence, robotics, and video games. It is an informed search algorithm that uses heuristic information to guide its search towards the goal state.The algorithm maintains two lists of nodes: an open list and a closed list. The open list contains nodes that have been generated but not yet explored, while the closed list contains nodes that have already been explored.

The A* search algorithm uses a cost function that combines the cost to reach a node from the starting node and a heuristic estimate of the cost to reach the goal from that node. The cost function is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach node n from the starting node, and $h(n)$ is the heuristic estimate of the cost to reach the goal from node n.The algorithm starts by adding the starting node to the open list with a cost of zero. It then repeatedly selects the node with the lowest cost from the open list and expands it by generating its successor nodes. For each successor node, the algorithm computes its cost $f(n)$ and checks whether it is already in the closed list or open list. If it is in the closed list, the node is skipped. If it is in the open list and the new cost is lower than its previous cost, the node is updated with the new cost. Otherwise, the node is added to the open list.

The algorithm continues this process until it finds the goal node or the open list is empty. If it finds the goal node, it returns the path from the starting node to the goal node. If the open list is empty and the goal node has not been found, the algorithm terminates and returns failure.

The A* search algorithm is guaranteed to find the optimal path to the goal if the heuristic is admissible, meaning it never overestimates the true cost to reach the goal.

Here's the step-by-step algorithm for A*:

1. Initialize the open list and the closed list.

2. Add the starting node to the open list.

3. While the open list is not empty:

   a. Find the node with the lowest f score in the open list.

   b. Remove that node from the open list and add it to the closed list.

   c. If the current node is the goal node, stop.

   d. Generate the current node's neighbors and calculate their tentative g and f scores.

   e. For each neighbor:

   i. If the neighbor is already in the closed list, skip it.

   ii. If the neighbor is not in the open list, add it to the open list.

   iii. If the neighbor is in the open list and its tentative g score is higher than the current tentative g score, skip it.

   iv. Otherwise, update the neighbor's parent to the current node and update its g and f scores.

4. If there is no path from the start node to the goal node, return failure.

5. Create a path from the goal node to the start node by following each node's parent pointers.

In this algorithm, f(n) represents the total cost of the path from the starting node to the goal node passing through node n, while g(n) represents the cost of the path from the starting node to node n, and h(n) represents the estimated cost of the cheapest path from node n to the goal node (known as the heuristic function). The A* algorithm selects the next node to explore based on the lowest f(n) value, where f(n) = g(n) + h(n).

4. Heuristic Functions

The heuristic function used in A* algorithm for solving Sudoku puzzles is typically a measure of how far the current state is from the goal state. There are several heuristic functions that can be used to estimate the remaining cost to the goal state. Here are a few examples:

1. Minimum Remaining Values (MRV): This heuristic function estimates the number of empty cells in the Sudoku grid. The idea is to prioritize cells with fewer remaining values to try first, which is likely to lead to a solution faster.

2. Degree Heuristic: This heuristic function prioritizes cells that are involved in many constraints with other cells. This is based on the observation that the number of possible values for a cell depends on the values assigned to its neighboring cells.

3. Least Constraining Value (LCV): This heuristic function estimates the number of values that will be eliminated from neighboring cells if a particular value is assigned to a given cell. The idea is to choose a value for a cell that eliminates the fewest possibilities for its neighbors, making it more likely that a solution will be found.

4. Combination of MRV, Degree and LCV heuristics: This approach combines the MRV, Degree and LCV heuristics to estimate the remaining cost to the goal state. This approach can lead to more efficient search for a solution.

It's important to note that the choice of heuristic function can greatly affect the performance of the A* algorithm for solving Sudoku puzzles. A good heuristic function should strike a balance between accuracy and efficiency.

We initially understood the approaches that should be used to resolve the sudoku puzzle when a person thinks to do so before constructing the heuristic function.As a result, we learned that the trick involves setting a number in one position and trying different things at each position. Humans can change their state at any time since they have all of their intuitions stored within them. On the computer, it is difficult to randomly return to a state that has already passed; it is feasible, but it requires more time and space.

Here, we discovered that assigning the numbers to each and every location in a sequence is the sole solution. By doing so, a system that assigns an erroneous number can return to that spot and rectify itself, just like a person would when solving a sudoku puzzle.My heuristic function uses a technique that facilitates assigning values in row-order. As a result, the A* search algorithm will explore a single node at a cell that appears in a series of spaces in a row before moving on to the next row. I have therefore provided heuristic values for each cell in the sudoku grid in order to follow a sequential row process. The grid of heuristic values for each cell is shown below.

```
hvg[9][9]={810 800 790 780 770 760 750 740 730
720 710 700 690 680 670 660 650 640
630 620 610 600 590 580 570 560 550
540 530 520 510 500 490 480 470 460
450 440 430 420 410 400 390 380 370
360 350 340 330 320 310 300 290 280
270 260 250 240 230 220 210 200 190 180
170 160 150 140 130 120 110 100
90 80 70 60 50 40 30 20 10}
```

5. Program Implementation

The C++ programming language is used to implement the software. Graph data structures, vector data structures, and object-oriented programming principles were then applied. A two-dimensional vector has been created to represent the sudoku grid.

6. Source Code

```
#include <bits/stdc++.h>

#include<iostream>

#include <vector>

using namespace std;


/*vector<vector<int>> ssb={

    {5, 3, 0, 0, 7, 0, 0, 0, 0},

    {6, 0, 0, 1, 9, 5, 0, 0, 0},

    {0, 9, 8, 0, 0, 0, 0, 6, 0},

    {8, 0, 0, 0, 6, 0, 0, 0, 3},

    {4, 0, 0, 8, 0, 3, 0, 0, 1},

    {7, 0, 0, 0, 2, 0, 0, 0, 6},

    {0, 6, 0, 0, 0, 0, 2, 8, 0},

    {0, 0, 0, 4, 1, 9, 0, 0, 5},

    {0, 0, 0, 0, 8, 0, 0, 7, 9},
   };*/ vector<vector<int>>
ssb={
   {3, 0, 6, 5, 0, 8, 4, 0, 0},
   {5, 2, 0, 0, 0, 0, 0, 0, 0},
   {0, 8, 7, 0, 0, 0, 0, 3, 1},
   {0, 0, 3, 0, 1, 0, 0, 8, 0},
   {9, 0, 0, 8, 6, 3, 0, 0, 5},
   {0, 5, 0, 0, 9, 0, 0, 0, 0},
   {1, 3, 0, 0, 0, 0, 2, 5, 0},
   {0, 0, 0, 0, 0, 0, 0, 7, 4},
   {0, 0, 5, 2, 0, 6, 3, 1, 0}
}; vector<vector<int>>
ini; class Astar{ public:
int NC; int zero; class
Node{
public: int fv,hv,gv; //f(n)=g(n)
    + h(n)
    int i,j;
    long int nodecount;
    vector<vector<int>> sb; bool
    vis=false; Node(){ for(int
    x=0;x<9;x++){ vector<int> vc;
    for(int y=0;y<9;y++){
    vc.push_back(ssb[x][y]);
            }
       sb.push_back(vc);
         }
    }
  }; vector<pair<int,int>>
path; vector<Node> open;
vector<Node> openchange;
vector<Node> closed;
vector<Node> graph[10000];
int hv[9][9]; int hvc[9][9];
int hvg[9][9]={
   {810,800,790,530,520,510,260,250,240},
   {780,770,760,500,490,480,230,220,210},
   {750,740,730,470,460,450,200,190,180},
   {710,700,690,440,430,420,170,160,150},
   {680,670,660,410,400,390,140,130,120},
    {650,640,630,380,370,360,110,100,90},

    {620,610,600,350,340,330,80,70,60},

    {590,580,570,320,310,300,50,40,30},

    {560,550,540,290,280,270,20,10,0}

};

Astar(){

    //graph.resize(1,vector<Node>(9))

    ; int sum=810; for(int

    l=0;l<9;l++){ for(int

    m=0;m<9;m++){ hv[l][m]=sum;

    sum=sum-10;

        } }

    sum=810

    ;

    for(int l=0;l<9;l++){

        for(int m=0;m<9;m++){

        hvc[m][l]=sum;

        sum=sum-10;

        }

    } for(int x=0;x<9;x++){

    vector<int> vc; for(int
    y=0;y<9;y++){
    vc.push_back(ssb[x][y]);

          }
       ini.push_back(vc);
     }
    Node In;
    In.fv=0;
    In.hv=810;
    In.i=0;
    In.j=0; In.nodecount=0;
open.push_back(In); } bool
rowcheck(Node N,int i,int num){ int
n=0; for(n=0;n<9;n++){
if(N.sb[i][n]!=num) continue; else
break;
     } if(n==9)
   return true;
   return false;
   } bool
   columncheck
```

```
    (Node N,int

    j,int num){

    int n=0;

    for(n=0;n<9;

    n++){

    if(N.sb[n][j]!

    =num)

    continue;

    else break;

      } if(n==9) return true; return false; } bool

gridsolver(Node N,int i,int j,int m,int n,int num){

for(int x=i;x<=m;x++){ for(int y=j;y<=n;y++){

if(N.sb[x][y]==num){ return false;

        }

      }
   } return true; } int gridsol(Node N,int

i,int j,int m,int n){ int count=0; for(int

x=i;x<m;x++){ for(int y=j;y<n;y++){

if(ssb[x][y]==0){ count++;

        }
      } } return count; } int
grids(Node N,int i,int j){
   if((i>=0 && i<=2) && (j>=0 && j<=2))
      return gridsol(N,0,0,2,2);
   else if((i>=0 && i<=2) && (j>=3 && j<=5))
      return gridsol(N,0,3,2,5);
   else if((i>=0 && i<=2) && (j>=6 && j<=8))
      return gridsol(N,0,0,2,2);
   else if((i>=3 && i<=5) && (j>=0 && j<=2))
      return gridsol(N,3,0,5,2);
   else if((i>=3 && i<=5) && (j>=3 && j<=5))
      return gridsol(N,3,3,5,5);
   else if((i>=3 && i<=5) && (j>=6 && j<=8))
      return gridsol(N,3,6,5,8);
   else if((i>=6 && i<=8) && (j>=0 && j<=2))
      return gridsol(N,6,0,8,2);
   else if((i>=6 && i<=8) && (j>=3 && j<=5))
      return gridsol(N,6,3,8,5);
   else if((i>=6 && i<=8) && (j>=6 && j<=8))
      return gridsol(N,6,6,8,8);
```

```
}   bool   gridcheck(Node   N,int   i,int   j,int
num){  if((i>=0  &&  i<=2)  &&  (j>=0  &&
j<=2))
        return gridsolver(N,0,0,2,2,num);
    else if((i>=0 && i<=2) && (j>=3 && j<=5))
          return gridsolver(N,0,3,2,5,num);
    else if((i>=0 && i<=2) && (j>=6 && j<=8))
          return gridsolver(N,0,6,2,8,num);
    else if((i>=3 && i<=5) && (j>=0 && j<=2)) return gridsolver(N,3,0,5,2,num);
    else if((i>=3 && i<=5) && (j>=3 && j<=5))
          return gridsolver(N,3,3,5,5,num);
    else if((i>=3 && i<=5) && (j>=6 && j<=8))
          return gridsolver(N,3,6,5,8,num);
    else if((i>=6 && i<=8) && (j>=0 && j<=2))
          return gridsolver(N,6,0,8,2,num);
    else if((i>=6 && i<=8) && (j>=3 && j<=5))
          return gridsolver(N,6,3,8,5,num);
    else if((i>=6 && i<=8) && (j>=6 && j<=8))
          return gridsolver(N,6,6,8,8,num);
}
```

```
//first heuristic function int heuristic_1(Node
N,int i,int j,int num,int lev){
  // cout<<"4";
   int count=0;
   //for(int y=j;y<9;y++){
           if(ssb[i][j]==0)
          {count=hv[i][j]+num;}
   //} return
   count; } int
   heuristic_2(
   Node N,int
   i,int j,int
   num,int
   lev){ int
   count=0;
   if(ssb[i][j]=
   =0)
   count=hvc[i
   ][j]+num;
   return
```

```
    count; } int
    heuristic_3(
    Node N,int
    i,int j,int
    num,int
    lev){ int
    count=0;
    if(ssb[i][j]=
    =0)
    count=hvg[i
    ][j]+num;
} bool sudokugoal(Node
N){ for(int i=0;i<9;i++){
for(int j=0;j<9;j++){
if(N.sb[i][j]==0) return
false;
        } }
    return
    true;
}
Node successorgenerator(Node N,int i,int j,int lev){
    //cout<<"3"<<" ";

    //vector<int> crct;

    map<int,int> mp;

    bool a,b,c; //

    if(ssb[i][j]==0 |

    if(ssb[i][j]==0){

    for(int m=1;m<=9;m++){

        //rule 1: row Checking

        a=rowcheck(N,i,m);

        b=columncheck(N,j,m);

        c=gridcheck(N,i,j,m);

        if(a && b && c)

        {N.sb[i][j]=m;break;}

      }

    } return N; } void explorenode(Node N){

cout<<"fv="<<N.fv<<",("<<N.i<<N.j<<"),nodecount="<<N.nodecount<<"\n";

}

void subroutine(Node v,int u,int l,int m,int lev){

    graph[u].push_back(v); int tn=0; bool a,b,c;

    for(int x=1;x<=9;x++){
        // Node ne;
         a=rowcheck(v,l,x);
         b=columncheck(v,m,x);
         c=gridcheck(v,l,m,x);
        if(a && b && c)
            {Node ne;
             NC++;tn++;
             //ne=successorgenerator(ne,x,y,lev)
             ; ne.sb[l][m]=x;
             ne.hv=heuristic_2(ne,l,m,x,lev);
             ne.gv=0; ne.fv=ne.hv+ne.gv;
             ne.i=l;ne.j=m; ne.nodecount=NC;
             explorenode(ne);
             graph[u].push_back(ne);}

    } } bool search(Node N){ for(int x=0;x<open.size();x++){
Node t=open[x]; if(t.fv<=N.fv && t.i==N.i && t.j==N.j &&
t.vis==true){ return true;
        } } return false; } bool search_close(Node N){ for(int
x=0;x<closed.size();x++){ Node t=closed[x]; if(t.fv<=N.fv
&& t.i==N.i && t.j==N.j && t.vis==true){ return true;
```

```
        } } return false; }
vector<int> findnextempty(){
vector<int> vc; for(int
x=0;x<9;x++){ for(int
y=0;y<9;y++){
if(ssb[x][y]==0)
            { vc.push_back(x);
                vc.push_back(y);
            }
        } } return vc; }
vector<int> findnextemptyc(){
vector<int> vc; for(int
x=0;x<9;x++){ for(int
y=0;y<9;y++){
if(ssb[y][x]==0)
             { vc.push_back(x);
                vc.push_back(y)
                ; break;
            }
        }
```

```
    } return vc; } vector<int>
findnextemptyg(){ vector<int>
vp; for(int y=0;y<9;y++){
for(int x=0;x<=2;x++){
if(ssb[y][x]==0){
vp.push_back(y);
vp.push_back(x); return vp;
            }
        }
    } for(int y=0;y<9;y++){
    for(int x=3;x<=5;x++){
    if(ssb[y][x]==0){
    vp.push_back(y);
    vp.push_back(x);

                return vp;
            }
        } } for(int
    y=0;y<9;y++){ for(int
    x=6;x<=8;x++){
    if(ssb[y][x]==0){
```

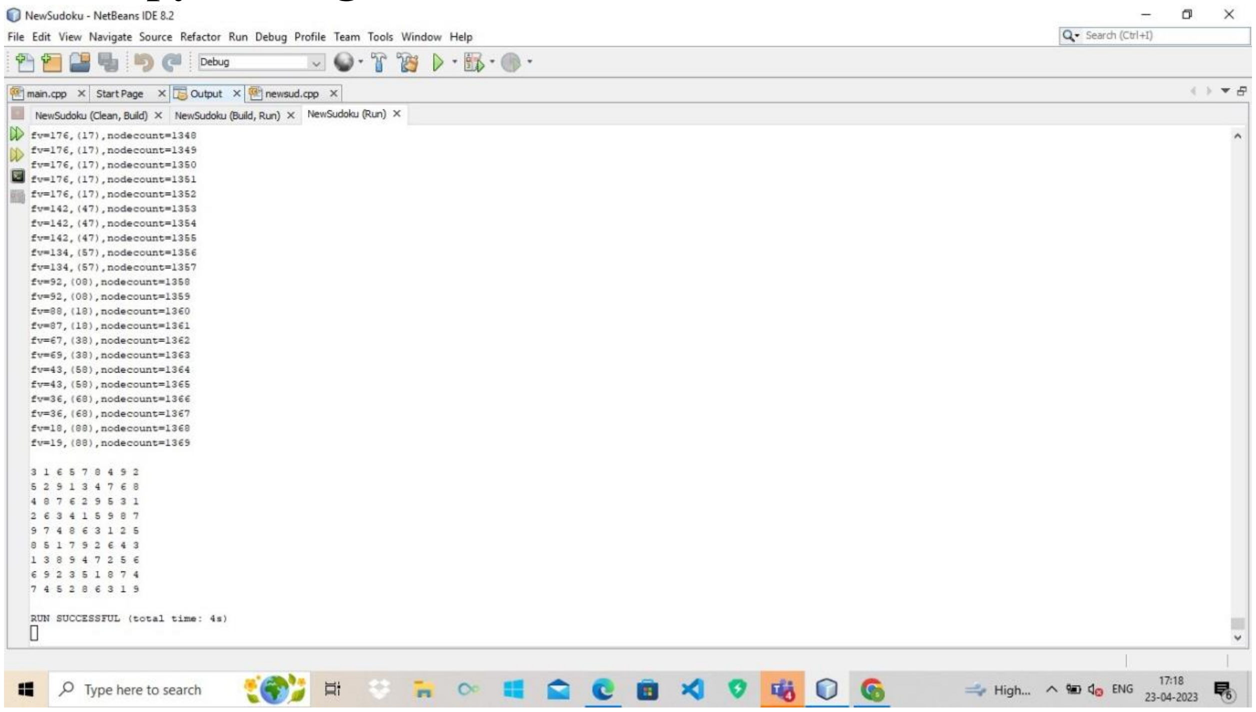
```
  vp.push_back(y);

  vp.push_back(x); return

  vp;

        }

     } } return vp; } void

Astar_1search(int i,int j,int lev){

Node y;

  y=open.back(); for(int

  g=0;g<9;g++){ for(int

  h=0;h<9;h++){

  ssb[g][h]=y.sb[g][h];


        }

     } open.pop_back(); subroutine(y,0,2,0,0);

  vector<Node>::iterator it; Node x; int k=1; int

  u=y.nodecount;

  for(it=graph[u].begin();it!=graph[u].end();it++){

  x=*it; if(k>1)

        {  if(sudokugoal(x))

             {break;}

           if((search(x)==true)||(search_close(x)==true))
```

```
            {continue;}
          else
             {open.push_back(x);}
        }
        k++;
      }
      y.vis=true;
      closed.push_back(y);
} void Astar_Search(){
int lev=0;
while(!open.empty()){
      sort(open.begin(),open.end(),[](const Node& a,const Node& b){return
a.fv<b.fv;});
      Node q;
      q=open.back(); for(int
      g=0;g<9;g++){ for(int
      h=0;h<9;h++){
      ssb[g][h]=q.sb[g][h];
        } }
      open.pop_back();
      int
```

```
 u=q.nodecount;

 lev++;

  //vector<int> vp=findnextempty();       // first heuristic

 vector<int> vp=findnextemptyc();         //second heuristic

 //vector<int> vp=findnextemptyg();   //third heuristic

// subroutine(q,u,vp[0],vp[1],lev); //generating successors

 subroutine(q,u,vp[1],vp[0],lev);   //second heuristic


// subroutine(q,u,vp[0],vp[1],lev);

 q.vis=true;



 vector<Node>::iterator it;

 Node x; int k=1;

 for(it=graph[u].begin();it!=graph[u].end();it++)

 { x=*it; if(k>1)

    { if(sudokugoal(x))

          {break;}

      if((search(x)==true)||(search_close(x)==true))

        {continue;} else

      if(x.sb[x.i][x.j]!=0)
```

```
                {open.push_back(x);
              } else{
              graph[u].erase(it);
              }
            }
            // searching for correct node
            k++;
         }//cout<<"done\n";
           closed.push_back(q);
    } } void displays(Node N){
for(int x=0;x<9;x++){ for(int
y=0;y<9;y++){
cout<<N.sb[x][y]<<" "; }
cout<<"\n";
    } } void
display(){
Node nes;
    for(int i=0;i<9;i++){ for(int
        j=0;j<9;j++){
        if(ssb[i][j]==0){ for(int
        x=1;x<=9;x++){ bool a,b,c;
        a=rowcheck(nes,i,x);
        b=columncheck(nes,j,x);
        c=gridcheck(nes,i,j,x);
                if(a && b && c)
                { ssb[i][j]=x; break;}
              }
          }
        } } cout<<"\n"; for(int
    x=0;x<9;x++){ for(int
    y=0;y<9;y++){
    cout<<ssb[x][y]<<" "; }
    cout<<"\n";
    }
   }
}; int main(int argc, char**
argv) {
    Astar A;
    A.Astar_1search(0,0,0);
    A.Astar_Search();
    A.display();

   // cout<<A.NC<<" ";

    ssb.clear();

    return 0;

}
```

7. A Copy of Program Run



```
fv=176,(17),nodecount=1348
fv=176,(17),nodecount=1349
fv=176,(17),nodecount=1350
fv=176,(17),nodecount=1351
fv=176,(17),nodecount=1352
fv=142,(47),nodecount=1353
fv=142,(47),nodecount=1354
fv=142,(47),nodecount=1355
fv=134,(57),nodecount=1356
fv=134,(57),nodecount=1357
fv=92,(08),nodecount=1358
fv=92,(08),nodecount=1359
fv=88,(18),nodecount=1360
fv=87,(18),nodecount=1361
fv=67,(38),nodecount=1362
fv=69,(38),nodecount=1363
fv=43,(58),nodecount=1364
fv=43,(58),nodecount=1365
fv=36,(68),nodecount=1366
fv=36,(68),nodecount=1367
fv=18,(88),nodecount=1368
fv=19,(88),nodecount=1369

3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9

RUN SUCCESSFUL (total time: 4s)
```

8. Analysis of the Results

Heuristic Functions	ET	NG	NE	D	b*
H1(i,j)=hv[i][j]+num-level	4seconds	2457	1179	48	51
H2(i,j)=hvc[i][j]+num-level	7seconds	2737	1319	48	57

H3(i,j)=hvg[i][j]+num-level	21seconds	5837	2869	48	121

When it comes to the program's time complexity, it is O(N2), where n is the total number of created nodes and space is exponential. We worked on a graph data structure to hold all the nodes in order to arrive at these results.

We gained insight into node storage and path tracing using the graph data structure. Tracing the graph of the nodes that we built yielded the findings in the following table. The longest path in the graph is also the depth of the tree.

9. Conclusions

Finding the best route between two nodes in a network can be done using the straightforward and effective A* Search Algorithm. It will be applied to discover the shortest path. It is a development of the shortest path algorithm by Dijkstra. Additionally, the A* Search Algorithm makes use of a heuristic function that offers more details on our distance from the objective node.In a sudoku puzzle, the target node might be anything because there are numerous possible outcomes.By working on this project, we get to know how code can be developed regardless of the goal node.Finally, the program we have implemented can also be optimized more with using more complex data structures but this was our initial project we just focused on working of A* search rather than on optimization of other data structures we used.

10. References

https://www.geeksforgeeks.org/a-search-algorithm/

https://towardsdatascience.com/solving-sudoku-with-ai-d6008993c7de

https://www.opensourceforu.com/2018/11/solving-sudoku-with-a-simple-algorithm/

https://en.wikipedia.org/wiki/Sudoku

11. Contributions

SRINIDHIJUNUTHULA

- Worked on getting information about the sudoku puzzles, understanding the various techniques of the sudoku solving tricks from the Internet which helped us in building heuristic functions of the program.
- Debugging and testing of the code at different breakpoints which helped us rectify our mistakes.
- Developed and implemented heuristic function 2.

SRINAGA

- Worked on the data structures part of the program which helped us in choosing the write organization of memory of the A* algorithm.
- Analyzing the data structure of the code to sync the graph data structure with A* algorithm.
- Developed and implemented heuristic function 3.

SAIPRANAVDATRIKA

- Designed the object-oriented part of the program, which helped us in building functions and assigning variables at the right part of the function and helped in getting the results of count of the nodes, depth of the tree and branching factor.

- Worked on building different heuristic functions to solve the sudoku puzzle and run it with the A* algorithm.
- Imparted knowledge of heuristic to the team which gave them correct insight to build heuristic functions and get the output for the code.
- Developed and implemented heuristic function 1.