Course Assignment 1
Search Algorithms

Artificial Intelligence

By Dr. Masoud Shariat Panahi

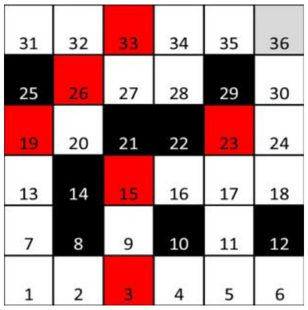
Author: Emil Keshishian

Student ID: 810602120

Spring 2024

Problem Definition

A 6 by 6 numbered grid with obstacle and penalty cells is defined, starting from cell #1 an imaginary robot should traverse and reach the end cell #36. The following rules are imposed upon the robot:



1. Only up, down, right, and left moves are permitted (no diagonal moves are allowed)
2. Traversing to a previously visited cell is prohibited.
3. The black cells are blocked and cannot be passed through
4. Traversing to any white cell costs one point and traversing to any red cells costs 2 points.
5. All moves should be one cell apart meaning no jumping or teleporting is allowed

The following search algorithms are implemented in C++ and their performance is evaluated using a total score calculated by the combination of the path score gained by each algorithm and the search score of each method.

- Greedy Algorithm
- Depth-First Algorithm
- Breadth-First Algorithm
- A* Algorithm
- Local-Search Algorithm

General overview of the developed program structure for testing search algorithms

The program is based on a class based structure where there's a group of graphics classes which are responsible for rendering graphical objects, these classes are the graphical object class and shader class. These classes utilize the OpenGL graphics API for rendering. For implementing the search algorithms there's a class called robot which is responsible for creating a robot instance and computing the path of the robot to the end goals given to each instance. The search algorithms are written as child functions of the robot class.

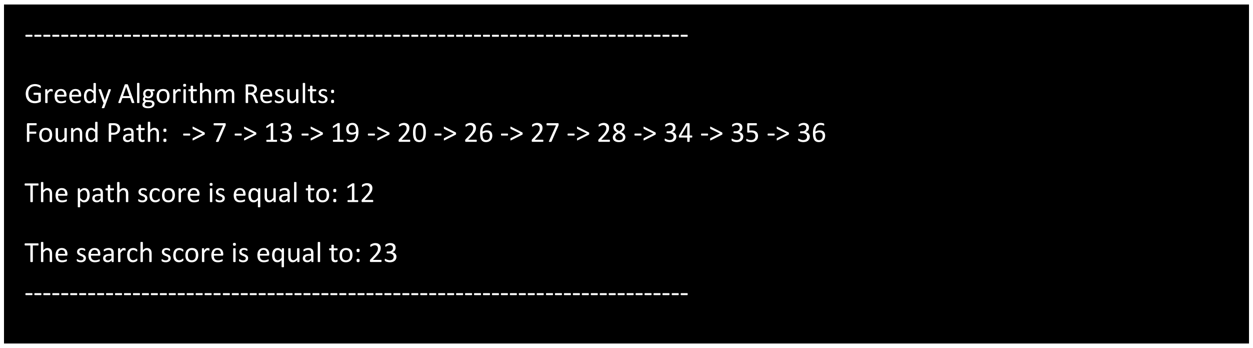
Greedy Algorithm

The Greedy algorithm is one of the simplest pathfinding algorithms in the computational sciences. The rational behind it is simple yet effective. A greedy pathfinder first checks it's neighbors and evaluates all valid neighbors for their distance to goal, which in my implementation is done using a function called getDistance which receives to integer arguments as inputs, it's inputs indicate the beginning and ending cells of the vector from which the distance is calculated , then the cell with the lowest distance is chosen as the next cell to travel to. The C++ implementation of the greedy algorithm is provided in snippet 1:

```cpp
// Greedy Search Algorithm
vector<int> robot::SolveGreedy(int begin, int end)
{
        SearchScore = 0;
        score = 0;
        int tries{0}, oldNum{ 0 }, nextNum;
        float dist{ 0.0f }, LowestDist{ 100.0f };
        vector<int> steps;

        cout << endl << "--------------------------------------------------------------------------" << endl;
        cout << "Greedy Algorithm Results:" << endl << endl;

        while (CurrentPosNum != 36 || tries > 100)
        {
                vector neighbors = getValidNeighors(CurrentPosNum);
                tries++;

                for (int i : neighbors)
                {
                        dist = getDistance(i, end);

                        if (dist < LowestDist)
                        {
                                LowestDist = dist;
                                nextNum = i;
                        }
                        else if (dist == LowestDist)
                        {
                                nextNum = i > oldNum ? i : oldNum;
                        }
                        SearchScore++;
                }

                steps.push_back(nextNum);

                if (nextNum - CurrentPosNum == 1)
                {
                        move_right();
                }
                else if (nextNum - CurrentPosNum == -1)
                {
                        move_left();
                }
                else if (nextNum - CurrentPosNum == 6)
                {
                        move_up();
                }
                else if (nextNum - CurrentPosNum == -6)
                {
                        move_down();
                }

                if (CurrentPosNum == 36)
                {
                        cout << "Found Path: ";
                        for (int num : steps) cout << " -> " << num;
                        cout << endl;
                        return steps;
                }

        }
}
```

Snippet 1 – Greedy Algorithm

The above mentioned code outputs a vector indicating the path cells in order from begin to end. The output of the code is available in output1. The path score is calculated with a function called CalcScore (snippet 2) which computes the path score from the vector outputted by each algorithm. The search score is computed in the function of each search algorithm, it indicated the number of times that the program has searched through cells and has executed search loops.



```
----------------------------------------------------------------------

Greedy Algorithm Results:
Found Path:  -> 7 -> 13 -> 19 -> 20 -> 26 -> 27 -> 28 -> 34 -> 35 -> 36

The path score is equal to: 12

The search score is equal to: 23
----------------------------------------------------------------------
```

Output1 – Greedy Algorithm results

```
void robot::CalcScore(vector<int> seq)
{
        int TempScore{ 0 };

        if (seq.size() > 0)
        {
                for (int i : seq)
                {
                        if (std::find(grid.PenalizedCells.begin(), grid.PenalizedCells.end(), i)
== grid.PenalizedCells.end())
                                TempScore++;
                        else
                                TempScore+=2;
                }
                score += TempScore;
        }
}
```

Snippet 2 – CalcScore function

Depth-First Algorithm

The depth-first algorithm is another very simple algorithm which searches each branch of the grid graph until it either reaches the end cell or it encounters a blockage. The following code provided in snippet 2 is my implementation of the algorithm.

```
// Depth-First Search Algorithm
vector<int> robot::SolveDepthFirst(int begin, int end)
{
        SearchScore = 0;
        score = 0;

        int next_cell{ begin }, i{ 0 }, n{ 0 }, counter{ 0 }, SearchCost{ 0 }, temp_cell,
crnt_cell;
        vector<int> neighbors, PassedCells, path;
        bool EndCellReached{ false }, stuck{ false };

        cout << endl << "-----------------------------------------------------------------------
---" << endl;
        cout << "Depth-First Algorithm Results:" << endl << endl;

        while (SearchCost < 200 && !EndCellReached)
        {
                crnt_cell = next_cell;
                neighbors = getValidNeighors(crnt_cell);
                std::sort(neighbors.begin(), neighbors.end());

                if (neighbors.size() == 1)
                {
                        stuck = true;
                        PassedCells.push_back(crnt_cell);
                }

                else if (!stuck)
                {
                        for (int n : neighbors)
                        {
                                        if (n == end)
                                        {
                                                if (std::find(PassedCells.begin(),
PassedCells.end(), crnt_cell) == PassedCells.end())
                                                        PassedCells.push_back(crnt_cell);
                                                next_cell = n;
                                                path.push_back(next_cell);
                                                EndCellReached = true;
                                                counter = 0;
                                                break;
                                        }

                                        else if (std::find(PassedCells.begin(),
PassedCells.end(), n) == PassedCells.end())
                                        {
                                                if (std::find(PassedCells.begin(),
PassedCells.end(), crnt_cell) == PassedCells.end())
                                                        PassedCells.push_back(crnt_cell);
                                                next_cell = n;
                                                path.push_back(next_cell);
                                                counter = 0;
                                                break;
                                        }
                        }
```

Snippet 3 – Depth-First Algorithm

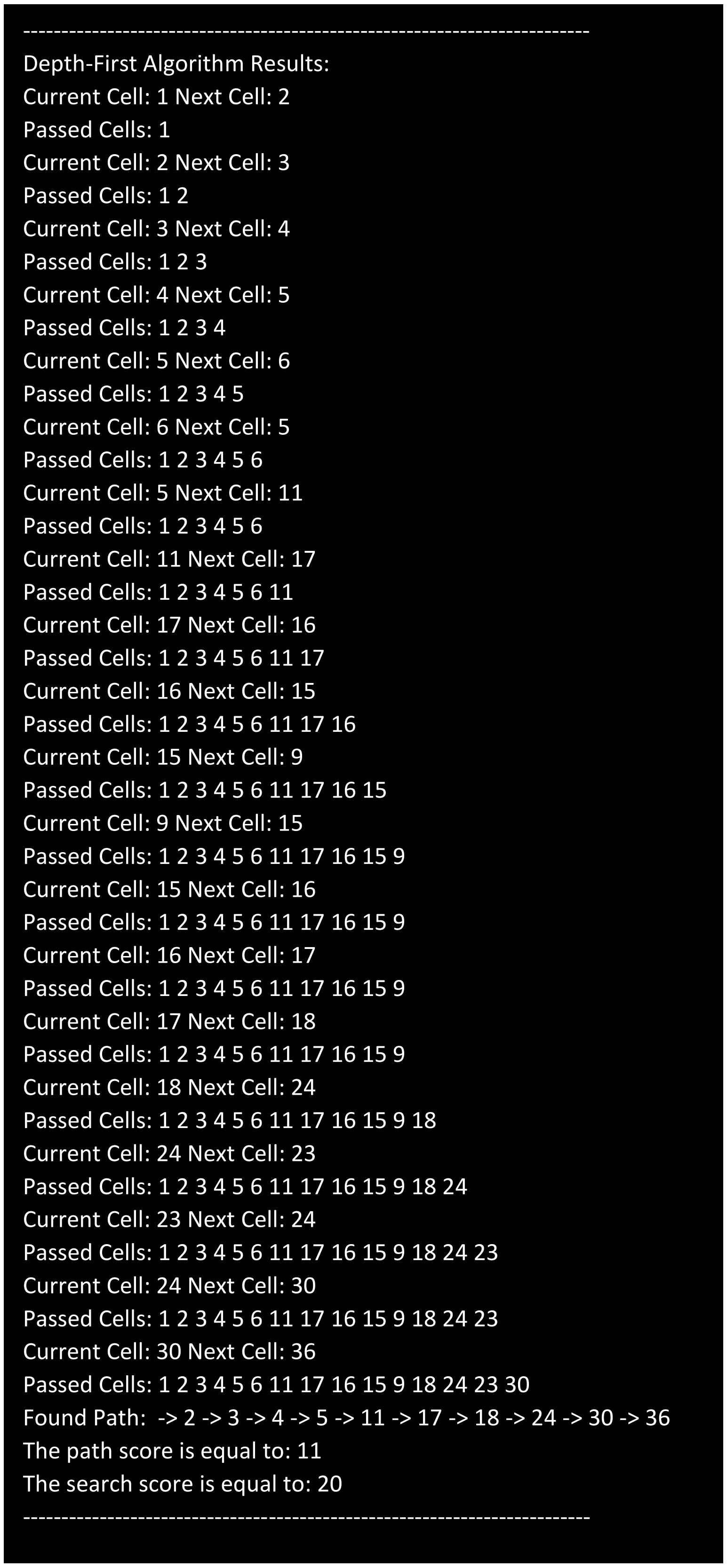
```
                        if (crnt_cell == next_cell)
                        {
                                stuck = true;
                                if (std::find(PassedCells.begin(), PassedCells.end(), crnt_cell)
== PassedCells.end())
                                        PassedCells.push_back(crnt_cell);
                        }
                }

                if (stuck)
                {
                        for (int n : neighbors)
                        {
                                if (std::find(PassedCells.begin(), PassedCells.end(), n) ==
PassedCells.end())
                                {
                                        stuck = false;
                                        if (std::find(PassedCells.begin(), PassedCells.end(),
crnt_cell) == PassedCells.end())
                                                PassedCells.push_back(crnt_cell);
                                        next_cell = n;
                                        path.push_back(next_cell);
                                        break;
                                }

                        }

                        if (counter <= PassedCells.size() && stuck)
                        {
                                next_cell = *(PassedCells.end() - (counter + 2));
                                counter++;
                                path.pop_back();
                        }

                }

                // Printing each step for debugging purposes
                cout << "Current Cell: " << crnt_cell << " Next Cell: " << next_cell << endl;
                cout << "Passed Cells: ";
                for (int i : PassedCells) cout << i << " ";
                cout << endl << endl;

                // Calculating the search score
                SearchCost++;
        }

        SearchScore = SearchCost;

        cout << "Found Path: ";
        for (int num : path) cout << " -> " << num;
        cout << endl;
        return path;
}
```

Snippet 3 – Depth-First Algorithm

Depth-First search algorithm assumptions:

- Each time in the intersections the neighboring cell with the lowest cell number is chosen
- Upon reaching the end goal the search operation is terminated and alternate paths are not considered although implementation of the said system is trivial

Results of the depth-first algorithm are presented in output 2.



```
-------------------------------------------------------------------------
Depth-First Algorithm Results:
Current Cell: 1 Next Cell: 2
Passed Cells: 1
Current Cell: 2 Next Cell: 3
Passed Cells: 1 2
Current Cell: 3 Next Cell: 4
Passed Cells: 1 2 3
Current Cell: 4 Next Cell: 5
Passed Cells: 1 2 3 4
Current Cell: 5 Next Cell: 6
Passed Cells: 1 2 3 4 5
Current Cell: 6 Next Cell: 5
Passed Cells: 1 2 3 4 5 6
Current Cell: 5 Next Cell: 11
Passed Cells: 1 2 3 4 5 6
Current Cell: 11 Next Cell: 17
Passed Cells: 1 2 3 4 5 6 11
Current Cell: 17 Next Cell: 16
Passed Cells: 1 2 3 4 5 6 11 17
Current Cell: 16 Next Cell: 15
Passed Cells: 1 2 3 4 5 6 11 17 16
Current Cell: 15 Next Cell: 9
Passed Cells: 1 2 3 4 5 6 11 17 16 15
Current Cell: 9 Next Cell: 15
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9
Current Cell: 15 Next Cell: 16
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9
Current Cell: 16 Next Cell: 17
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9
Current Cell: 17 Next Cell: 18
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9
Current Cell: 18 Next Cell: 24
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9 18
Current Cell: 24 Next Cell: 23
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9 18 24
Current Cell: 23 Next Cell: 24
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9 18 24 23
Current Cell: 24 Next Cell: 30
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9 18 24 23
Current Cell: 30 Next Cell: 36
Passed Cells: 1 2 3 4 5 6 11 17 16 15 9 18 24 23 30
Found Path:  -> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36
The path score is equal to: 11
The search score is equal to: 20
-------------------------------------------------------------------------
```

Output 2 – Depth-First Algorithm results

Breadth-First Algorithm

The Breadth-first algorithm is very similar to the depth-first algorithm, the difference is instead of following each branch to the end it does so for all branches simultaneously in a parallel manner where in each stage of the search, we're at the same depth in all branches. My C++ implementation of the breadth first algorithm is presented in snippet 4.

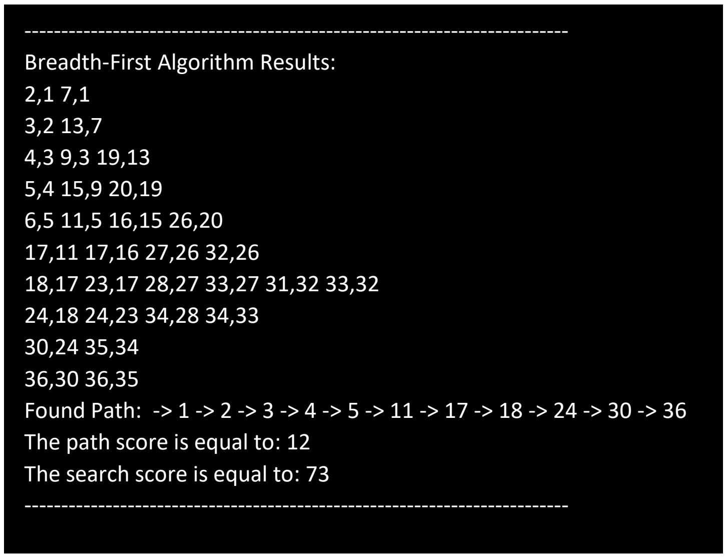
```cpp
// Breadth-First Search Algorithm
vector<int> robot::SolveBreadthFirst(int begin, int end, bool DebugPrintingOn)
{
        int i{ 0 }, j{ 0 }, CellParent;
        vector<int> neighbors, path, tempPath, PassedCells;
        vector<vector<int>> AllPaths;
        vector<array<int,2>> NextLayer, CrntLayer, flaggedCells;
        vector<vector<array<int, 2>>> AllLayers;
        bool EndCellReached{ false }, AllPathsNotFound{ true };

        NextLayer.push_back({ begin, begin });
        PassedCells.push_back(begin);

        if (DebugPrintingOn)
                cout << endl << "-----------------------------------------------------------------
-----------" << endl << "Breadth-First Algorithm Results:" << endl << endl;

        while (!EndCellReached)
        {
                CrntLayer = NextLayer;

                NextLayer.clear();
                for (array<int,2> cell: CrntLayer)
                {
                        neighbors = getValidNeighbors(cell[0]);
                        //std::sort(neighbors.begin(), neighbors.end());

                        for (int n : neighbors)
                        {
                                if (cell[0] == end)
                                {
                                        EndCellReached = true;
                                        // Extracting all paths
                                        PassedCells.clear();
                                        CrntLayer.clear();
                                        NextLayer.clear();
                                        CellParent = end;
                                        path.insert(path.begin(), end);

                                        for (i = AllLayers.size() - 1; i >= 0; i--)
                                        {
                                                CrntLayer = AllLayers.at(i);

                                                for (array<int, 2> cell : CrntLayer)
                                                {
                                                        if (cell[0] == CellParent &&
std::find(flaggedCells.begin(), flaggedCells.end(), cell) == flaggedCells.end())
                                                        {
                                                                CellParent = cell[1];
                                                                path.insert(path.begin(),
CellParent);
                                                        }

                                                        SearchScore++;
                                                }
                                        }
```

Snippet 4 – The Breadth-First Algorithm

```
                    // For debugging

                    if (DebugPrintingOn)
                    {
                        cout << endl << "Found Path: ";
                        for (int num : path) cout << " -> " << num;
                        cout << endl;
                    }

                    return path;
                }

                if (std::find(PassedCells.begin(), PassedCells.end(), n) ==
PassedCells.end() && std::find(NextLayer.begin(), NextLayer.end(), array { n,cell[0]}) ==
NextLayer.end())
                    NextLayer.push_back({ n,cell[0]});
            }

            AllLayers.push_back(NextLayer);
        }

        for (array<int,2> n : NextLayer)
        {
            if (n[0] > 0)
            {
                if (std::find(PassedCells.begin(), PassedCells.end(), n[0]) ==
PassedCells.end())
                {
                    PassedCells.push_back(n[0]);
                }
            }
        }

        // For debugging
        if (DebugPrintingOn)
        {
            for (array<int, 2> n : NextLayer)
                cout << n[0] << "," << n[1] << " ";
            cout << endl;
        }

    }


}
```

Snippet 4 – The Breadth-First Algorithm

Results of the Breadth-first algorithm are presented in output 3.



```
-------------------------------------------------------------------------
Breadth-First Algorithm Results:
2,1 7,1
3,2 13,7
4,3 9,3 19,13
5,4 15,9 20,19
6,5 11,5 16,15 26,20
17,11 17,16 27,26 32,26
18,17 23,17 28,27 33,27 31,32 33,32
24,18 24,23 34,28 34,33
30,24 35,34
36,30 36,35
Found Path:  -> 1 -> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36
The path score is equal to: 12
The search score is equal to: 73
-------------------------------------------------------------------------
```

Output 3 – Breadth-First Algorithm Results

Now we get to the interesting parts ☺ The A* Algorithm

Suffice to say, A* … it's probably the most famous search algorithm around.

In my implementation of the A* algorithm the heuristic function is chosen to be the lowest path score computed by breadth-first algorithm for every neighbor of the current cell, rinse and repeat … .

My implementation of the A* algorithm in C++ is presented in snippet 5.

```
vector<int> robot::SolveAStar(int begin, int end)
{

        int CrntCell{ begin }, NextNum, NextCell{ begin }, counter{ 0 };
        float g{ 0 }, h{ 0 }, TotalScore{ 1000 };
        vector<int> neighbors, pathtillnow, path, PassedCells;
        bool GoalReached{ false };

        while (!GoalReached && counter < 100)
        {
                CrntCell = NextCell;
                neighbors = getValidNeighors(CrntCell);

                for (int cell : neighbors)
                {
                        if (std::find(PassedCells.begin(), PassedCells.end(),
cell)==PassedCells.end())
                        {
                                if (cell == end)
                                {
                                        NextCell = cell;
                                        path.push_back(NextCell);
                                        cout << "Found Path: ";
                                        for (int num : path) cout << " -> " << num;
                                        return path;
                                }
```

Snippet 5 – The A* Algorithm

```
                                CalcScore(SolveBreadthFirst(cell, end));
                                h = this->score;

                                path.push_back(cell);
                                CalcScore(path);
                                g = score;
                                path.pop_back();

                                if (TotalScore > g + h)
                                {
                                        TotalScore = g + h;
                                        NextCell = cell;
                                }

                                PassedCells.push_back(cell);
                        }
                        score = 0;
                }

                TotalScore = 1000;
                path.push_back(NextCell);
                counter++;
        }
}
```

Snippet 5 – The A* Algorithm

The Results of the A* algorithm are presented in output 4.

```
-----------------------------------------------------------------------

A* Algorithm Results:
Found Path:  -> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36

The path score is equal to: 11

The search score is equal to: 613
-----------------------------------------------------------------------
```

Output 4 – A* Algorithm Results

Local-Search Algorithm

A probability based algorithm. My implementation of the local-search algorithm is presented in snippet 6.

Snippet 6 – The Local-Search algorithm

```
vector<int> robot::SolveLocalSearch(int begin, int end)
{
        vector<int> path, neighbors, PassedCells, NonPassedNeighbors;
        vector<float> costs, chances;
        int NextCell{ begin }, counter{ 0 }, backprop{ 1 };
        float sum{ 0 }, p{ 0 };
        bool CellFound{ false };

        PassedCells.push_back(begin);

        while(counter < 100)
        {
                NonPassedNeighbors.clear();
                costs.clear();
                chances.clear();
                p = 0;
                sum = 0;
                CellFound = false;

                if (NextCell == end)
                {
                        return path;
                }

                neighbors = getValidNeighors(NextCell);

                for (int cell : neighbors)
                {
                        if(std::find(PassedCells.begin(), PassedCells.end(),
cell)==PassedCells.end())
                        {
                                score = 0;
                                CalcScore({ cell });
                                cout << "Cell: " << cell << " Score: " << score << endl;
                                costs.push_back(float(1.0 / score));
                                sum += float(1.0 / score);
                                NonPassedNeighbors.push_back(cell);
                                CellFound = true;
                                backprop = 1;
                        }
                        SearchScore++;
                }

                if(!CellFound)
                {
                        if (std::find(PassedCells.begin(), PassedCells.end(), NextCell) ==
PassedCells.end())
                                PassedCells.push_back(NextCell);
                        NextCell = *(PassedCells.rbegin() + backprop);
                        path.push_back(NextCell);
                        counter++;
                        backprop++;
                        continue;
                }
                for (int i = 0; i < costs.size(); i++)
                {
                        if (std::find(NonPassedNeighbors.begin(), NonPassedNeighbors.end(),
neighbors.at(i)) != NonPassedNeighbors.end())
                        {
                                chances.push_back((1.0f / costs[i]) / sum);
                        }
                        SearchScore++;
                }

                std::mt19937 gen(std::random_device{}());
                std::discrete_distribution<> d{ chances.begin(), chances.end() };

                PassedCells.push_back(NextCell);
                NextCell = NonPassedNeighbors.at(d(gen));

                path.push_back(NextCell);
                counter++;
        }
}
```

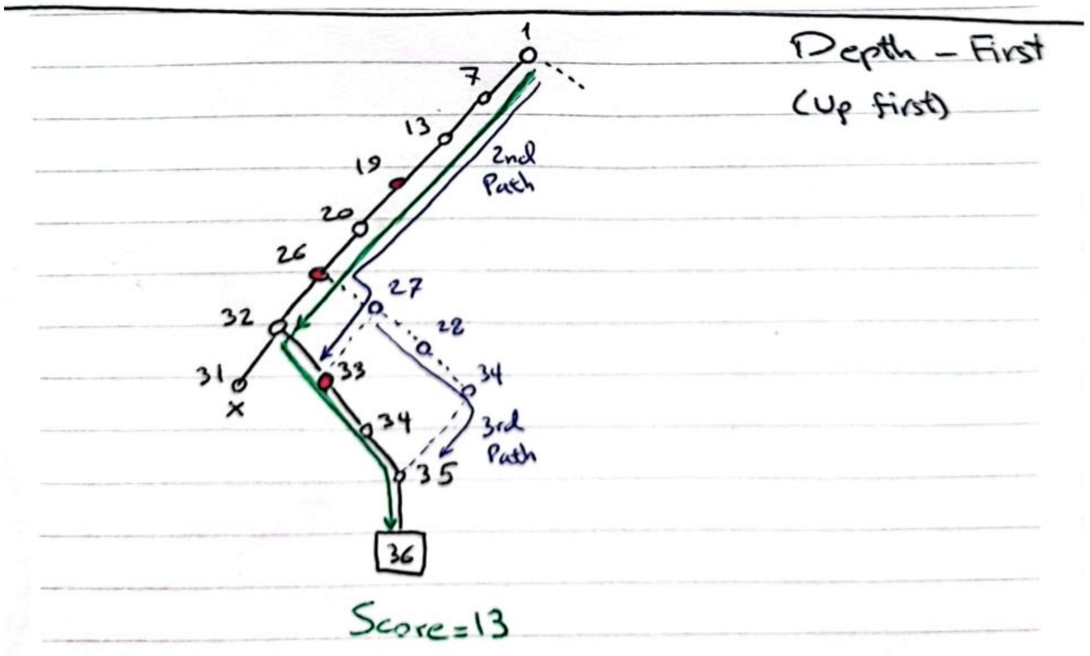
Snippet 6 – The Local-Search algorithm

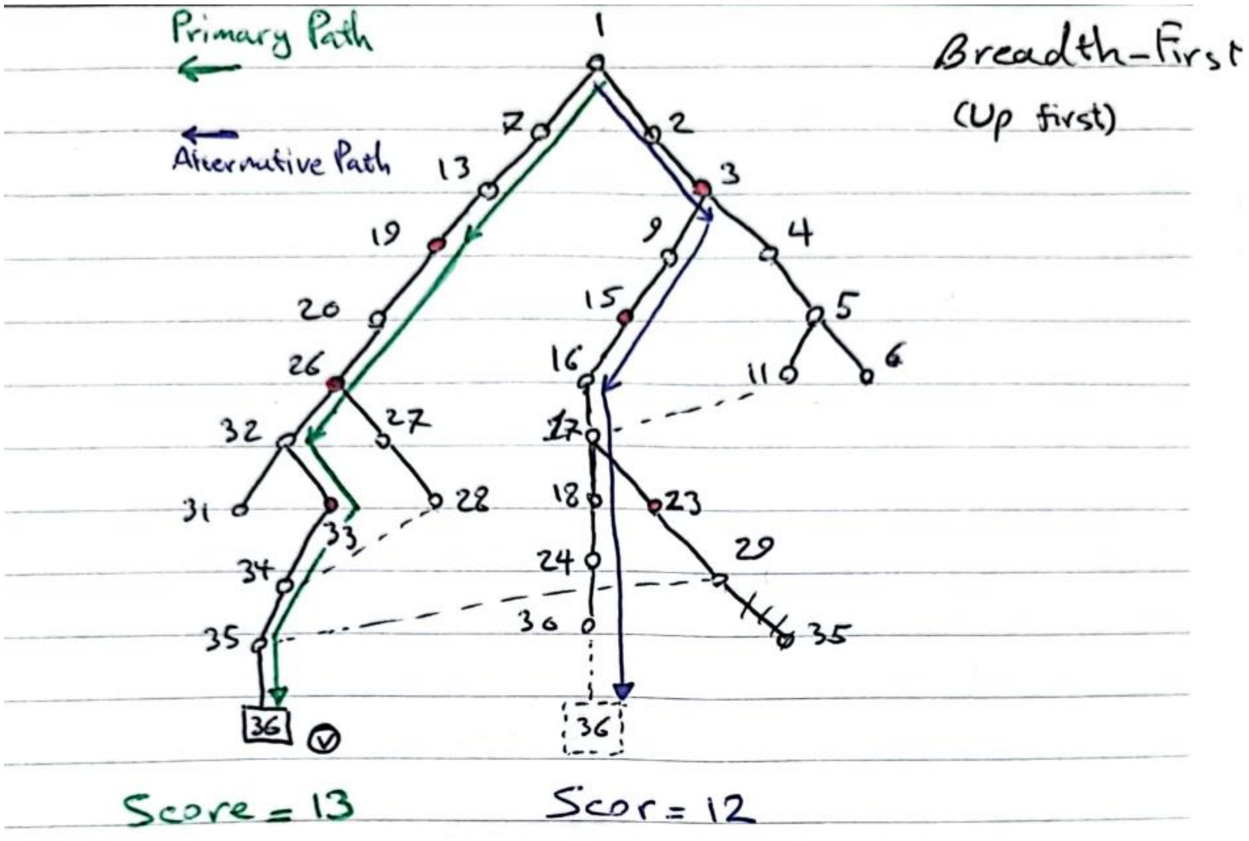
Local-Search algorithm results are presented in output 5.

```
------------------------------------------------------------------------

 Local Search Algorithm Results:

 Found Path:  -> 7 -> 13 -> 19 -> 20 -> 26 -> 27 -> 28 -> 34 -> 33 -> 32 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36
The path score is equal to: 21

The search score is equal to: 56

 ------------------------------------------------------------------------
```

Output 5 – Local-Search Algorithm Results

Hand calculations for depth-first and breadth-first algorithms





Many more functionalities were developed for the program for future usecases which are available in the provided source code.

All search algorithms are evaluated in the table below:

Algorithm	*Path*	*Path Score*	*Search Score*	*Total Score*
Greedy	-> 7 -> 13 -> 19 -> 20 -> 26 -> 27 -> 28 -> 34 -> 35 -> 36	12	21	33
Depth-First	-> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36	11	27	38
Breadth-First	-> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36	12	73	85
*A**	-> 2 -> 3 -> 4 -> 5 -> 11 -> 17 -> 18 -> 24 -> 30 -> 36	11	613	624
Local-Search	-> 7 -> 13 -> 19 -> 20 -> 26 -> 27 -> 28 -> 34 -> 33 -> 34 -> 35 -> 36	21	56	77

*The Local-Search algorithm output was containing duplicate cells hence the path and path score were corrected.

Hence the greedy algorithm is chosen for this particular grid.