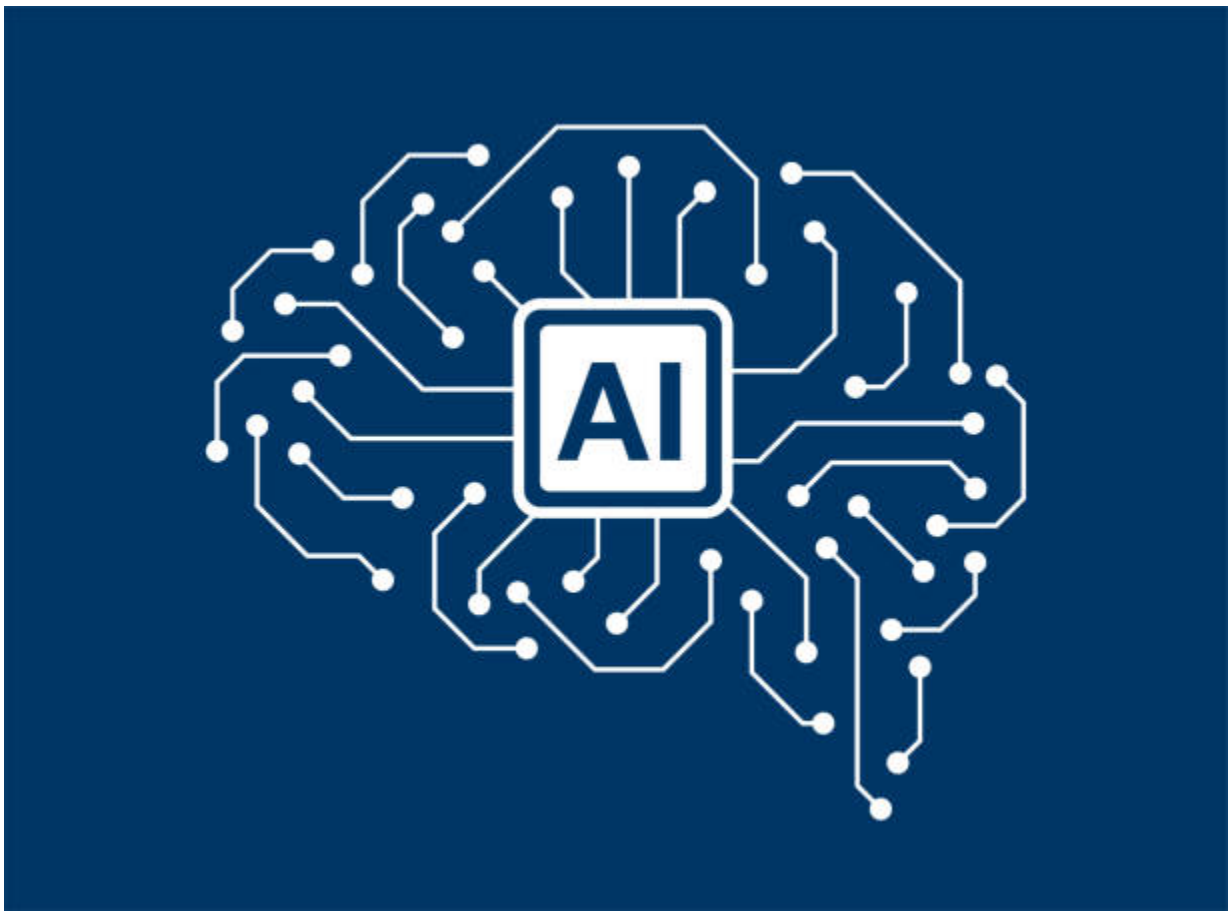
ARTIFICIAL INTELLIGENCE - ITIT - 2203
PROJECT



SUBMITTED BY:
KUNAL JAIN
(2019IMT-052)

SUBMITTED TO:
Dr. PINKU RANJAN

TITLE OF THE PROJECT:

Solving Hitori Grid game using informed and local search algorithms and comparing their time and space complexities.

ABSTRACT:

Hitori is a popular Japanese game, we'll be using various AI algorithms of informed and local search type to solve this game and further we will also be comparing their space and time complexities to find out which algorithm is better than others.

INTRODUCTION:

Hitori (Alone in Japanese) is a type of logic puzzle. Board of the game is a grid of cells containing a number. The goal is not to have more than one identical number in each row or column. Three main rules of Hitori are:

1. There must not be any duplicate numbers in any rows or columns.
2. Black cells cannot be adjacent, although they can be diagonal to one another.
3. There must not be 4 black cells around a white one.

The chosen programming language for this project is C++. It is the fastest computer language, which makes it great for AI programming projects that are time sensitive. It provides faster execution time and has a quicker response time.

METHODOLOGY:

DATA STRUCTURE:

This is an important phase in every project; you have to consider efficiency and clarity of information. The simplest way is to present the game board as a structure of state. For clarity and not mixing the numbers and white and black cells, every cell is represented as a pair of “cell number” and “cell color”. So if we have a game board like:

1	1	3
1	2	3
1	3	3

It can be shown by a 3*3 matrix:

(1,W) (1,W) (3,W)

(1,W) (2 , B) (3,W)

(1 , B) (3,W) (3,W)

INITIAL STATE:

The initial state is a 2D vector, containing each cell represented by a pair of number and color. It is assumed that all cells are white in the initial state.

The results of algorithms analysis are based on two given samples, as our initial states.

GOAL STATE:

The image of the goal state is not exactly clear. It is achieved by satisfying the game's constraints; Since our successor doesn't allow states with black adjacent and isolating white cells to be a part of searching. The "is goal" function only checks if there are any white duplicate numbers in the game board; if not, Goal is achieved.

SUCCESSOR:

Successor function is defined as a function that generates one or more states for a given state. Now, a good successor is the one that only generates states that:

1. Do not break game (or problem) rules.

2. Are possible steps to achieve goal

In this game, considering a game with two rules (no isolated white cells, and no adjacent black cells); the successor can return any possible next state based on the given state. It finds the duplicate cells that both are white, and black each of them in a different state.

HEURISTIC:

The estimation process is done by functions called Heuristic functions. We have 2 heuristics that are explained below:

1. **Heuristic 1 (Duplicates):**

   Based on game rules, the goal is not to have any cells with duplicate values, so this function counts the number of cells that have duplicates. This means the function estimates the remaining cost to reach goal state; So the minimum h is considered the best (nearest to goal).

2. **Heuristic 2 (White Cells):**

   The other point of view is that the less we have white cells, we are more likely to achieve our goal. So function counts white cells and again, less is better.

SEARCH ALGORITHMS:

INFORMED SEARCH ALGORITHMS:

GREEDY

According to the general definition of greedy algorithm, search must be prioritized by heuristic function estimation. Now what it means is that if we have our next steps in a queue, the first one to be chosen is the one with the best h(n).
The problem with this algorithm is that the visionary is all about future "estimation". You need to notice that estimating something is not the exact thing and it could be wrong! This is why greedy can easily trap in loops like DFS.

A STAR

So before we get into explaining A*, we need to know an algorithm named "**Uninformed Cost Search**". This algorithm is a uniformed one and yet, the way it works is like greedy. Except, instead of taking steps based on future estimation, it "computes" the cost of the path it has come from. We show this computing result by f(n). Now what A* does is combining two Greedy and Uninformed Cost Search, So we can have a pretty reliable scale for prioritizing steps by summing h(n) and f(n) for each node. The result not only shows us how much it has cost to be in this state, but also the estimation of how much is left to find the goal.

LOCAL SEARCH ALGORITHMS:

A local algorithm is a distributed algorithm that runs in constant time, independently of the size of the network. Being highly scalable and fault-tolerant, such algorithms are ideal in the operation of large-scale distributed systems. Here are three famous local algorithms, Hill climbing (Simple and random) and simulated annealing.

SIMPLE HILL CLIMBING

Unlike upper algorithms, which give you the optimal result (If they ever get), hill climbing gives you the local minimum. This means though it can give you an answer, it won't be the best answer; just best in that area.
Let's see what it does. Like the act of hill climbing itself, the algorithm will analyze the choices and choose the best one based on Heuristic function; Then it clears all the others choices. In other words, you only choose one path and don't give the chance to other ones. Theoretically with a good Heuristic, Hill Climbing will reach its maximum functionality, but in reality it only works 10-20 percent of the time.

It’s because of certainty and not giving the other states any chance. So, how to fix that?

RANDOM HILL CLIMBING

In simple hill climbing, we confronted a situation that decreased functionality down to 10 percent. So the main idea of this algorithm is to “give a chance to states by the weight of their value”. It means that we don't choose the one that heuristic tells is best. (remember heuristic is also a function based on estimation and it has no certainty.) We choose randomly based on a non-uniformed probability distribution which is given based on their heuristic estimate. In reality, this method increases functionality up to 80%.

DRIVER CODE:

The entire code for the project is pushed to GitHub, below I’ve explained the important functions of the code for Greedy Search Algorithm.

GitHub Repository: https://github.com/kunaljain0212/AI-Project

Creating Matrix from input file:

```
void build_input(state &init) //initial state
{
   ifstream input;
   input.open("sample2.txt");
```

```
    int m, n;
    input >> m; //rows
    input >> n; //columns

    init.matrix.resize(n);
    for (int i = 0; i < n; ++i)
            init.matrix[i].resize(m);

    for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
            {
                    char c;
                    input >> c;
                    init.matrix[i][j].first = c;
                    init.matrix[i][j].second = 'W';
            }
    input.close();
    return;
}
```

Finding successors of current state:

```
vector<state> successor(state S)
{
    vector<state> result;
    state new_state;

    for (int i = 0; i < S.matrix.size(); i++)
            for (int j = 0; j < S.matrix.size(); j++)
            {
                    for (int r = 0; r < S.matrix.size(); r++)
                    {
                            if (r != i)
                                    if (S.matrix[i][j].first == S.matrix[r][j].first)
                                            if (S.matrix[i][j].second != 'B' && S.matrix[r][j].second != 'B')
                                            {
                                                    new_state = color_state(S, i, j);
                                                    if (is_valid(new_state))
                                                    {
                                                            if (search_in_successor(result, new_state) == false)
                                                                    result.push_back(new_state);
                                                    }
                                                    new_state = color_state(S, r, j);
                                                    if (is_valid(new_state))
                                                    {
                                                            if (search_in_successor(result, new_state) == false)
                                                                    result.push_back(new_state);
                                                    }
```

```
                                        }
                        }
                        for (int c = 0; c < S.matrix.size(); c++)
                        {
                                if (c != j)
                                        if (S.matrix[i][j].first == S.matrix[i][c].first)
                                                if (S.matrix[i][j].second != 'B' && S.matrix[i][c].second != 'B')
                                                {
                                                        new_state = color_state(S, i, j);
                                                        if (is_valid(new_state))
                                                        {
                                                                if (search_in_successor(result, new_state) == false)
                                                                        result.push_back(new_state);
                                                        }
                                                        new_state = color_state(S, i, c);
                                                        if (is_valid(new_state))
                                                        {
                                                                if (search_in_successor(result, new_state) == false)
                                                                        result.push_back(new_state);
                                                        }
                                                }
                        }
                }
        return result;
}
```

Functions for finding heuristics:

```
void heuristic_count(state &S)
{
    int Count = 0;
    int c = 0;
    char flag;

    vector<bool> flags;
    flags.resize(9);
    init_false(flags);

    //row
    for (int i = 0; i < S.matrix.size(); i++)
    {
            for (int j = 0; j < S.matrix.size(); j++)
            {
                    flag = S.matrix[i][j].first;
                    if (flags[(flag - '0') - 1] == false)
                    {
                            for (int k = 0; k < S.matrix.size(); k++)
                            {
```

```
                                        if (flag == S.matrix[i][k].first && S.matrix[i][k].second != 'B')
                                                c++;
                                }
                        }
                        if (c != 1)
                                Count += c;
                        c = 0;
                        flags[(flag - '0') - 1] = true;
                }
                for (int i = 0; i < flags.size(); i++)
                        flags[i] = false;
        }

        for (int i = 0; i < flags.size(); i++)
                flags[i] = false;

        //col
        for (int i = 0; i < S.matrix.size(); i++)
        {
                for (int j = 0; j < S.matrix.size(); j++)
                {
                        flag = S.matrix[j][i].first;
                        if (flags[(flag - '0') - 1] == false)
                        {
                                for (int k = 0; k < S.matrix.size(); k++)
                                {
                                        if (flag == S.matrix[k][i].first && S.matrix[k][i].second != 'B')
                                                c++;
                                }
                        }
                        if (c != 1)
                                Count += c;
                        c = 0;
                        flags[(flag - '0') - 1] = true;
                }
                for (int i = 0; i < flags.size(); i++)
                        flags[i] = false;
        }
        S.h = Count;
}


void heuristic_whites(state &S)
{
        int count = 0;
        for (int i = 0; i < S.matrix.size(); i++)
                for (int j = 0; j < S.matrix.size(); j++)
                {
                        if (S.matrix[i][j].second == 'W')
```

```
                {
                        count++;
                }
        }
    S.h = count;
}
```

Functions for checking if a particular state is a valid state or not:

```
bool check_for_duplicates(state S) //returns true if there are duplicates
{
    char flag_number, flag_color;
    //row
    for (int i = 0; i < S.matrix.size(); i++)
    {
            //cout << "*" << endl;
            for (int j = 0; j < S.matrix.size(); j++)
            {
                    for (int r = 0; r < S.matrix.size(); r++)
                    {
                            if (r != i)
                                    if (S.matrix[i][j].first == S.matrix[r][j].first)                                    //it is duplicate
                                            if (S.matrix[i][j].second != 'B' && S.matrix[r][j].second != 'B') //if none of them is black
                                            {
                                                    return false;
                                            }
                    }
                    for (int c = 0; c < S.matrix.size(); c++)
                    {
                            if (c != j)
                                    if (S.matrix[i][j].first == S.matrix[i][c].first)
                                            if (S.matrix[i][j].second != 'B' && S.matrix[i][c].second != 'B')
                                            {
                                                    return false;
                                            }
                    }
            }
    }
    return true;
}


bool check_for_adjacent_blacks(state S) //returns true if there are 2 adjacent blacks
{
    for (int i = 0; i < S.matrix.size(); i++) //two blacks beside in a row
    {
            for (int j = 0; j < S.matrix[i].size(); j++)
            {
                    if (S.matrix[i][j].second == 'B')
                            if (j + 1 < S.matrix.size())
                                    if (S.matrix[i][j + 1].second == 'B')
                                            return true;
            }
```

```
    }
    for (int i = 0; i < S.matrix.size(); i++) //two blacks beside in a column
    {
            for (int j = 0; j < S.matrix[i].size(); j++)
            {
                    if (S.matrix[i][j].second == 'B')
                            if (i + 1 < S.matrix.size())
                                    if (S.matrix[i + 1][j].second == 'B')
                                            return true;
            }
    }
    return false;
}


bool check_for_blocks(state S) //returns true if there are 4 blacks around a white
{
    int Size = S.matrix.size() - 1;
    for (int i = 0; i < S.matrix.size(); i++)
    {
            for (int j = 0; j < S.matrix.size(); j++)
            {
                    if (i == 0 && j == 0) //up left corner
                            if (S.matrix[i][j + 1].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                    return true;
                    if (i == 0 && j > 0 && j + 1 < S.matrix.size()) //up row
                            if (S.matrix[i][j - 1].second == 'B' && S.matrix[i][j + 1].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                    return true;
                    if (i == 0 && j == Size) //up right corner
                            if (S.matrix[i][j - 1].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                    return true;
                    if (i == Size && j == 0) //down left corner
                            if (S.matrix[i - 1][j].second == 'B' && S.matrix[i][j + 1].second == 'B')
                                    return true;
                    if (i == Size && j > 0 && j + 1 < S.matrix.size()) //down row
                            if (S.matrix[i - 1][j].second == 'B' && S.matrix[i][j - 1].second && S.matrix[i][j + 1].second == 'B')
                                    return true;
                    if (i == Size && j == Size) //down right corner
                            if (S.matrix[i - 1][j].second == 'B' && S.matrix[i][j - 1].second == 'B')
                                    return true;

                    if (j == 0 && i > 0 && i + 1 < S.matrix.size()) //leftmost col
                            if (S.matrix[i - 1][j].second == 'B' && S.matrix[i][j + 1].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                    return true;
                    if (j == Size && i > 0 && i + 1 < S.matrix.size()) //rightmost col
                            if (S.matrix[i][j - 1].second == 'B' && S.matrix[i - 1][j].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                    return true;
                    if (j > 0 && i > 0 && j + 1 < S.matrix.size() && i + 1 < S.matrix.size())
                    {
                            if (S.matrix[i][j - 1].second == 'B' && S.matrix[i][j + 1].second == 'B')
                                    if (S.matrix[i - 1][j].second == 'B' && S.matrix[i + 1][j].second == 'B')
                                            return true;
                    }
            }
```

```
    }
    return false;
}
```

Main driver code for Greedy algorithm:

```
state greedy(state &current_state, vector<state> &visited_list, int &max_space, int &no_of_succ)
{
    priority_queue<state, vector<state>, compare_h_for_greedy> state_q;
    vector<state> successors = successor(current_state);
    no_of_succ += successors.size();
    int q_size = 0;

    heuristic(current_state);

    for (int i = 0; i < successors.size(); i++)
    {
            heuristic(successors[i]);
            state_q.push(successors[i]);
    }
    q_size = state_q.size();
    max_space = max(q_size, max_space);

    while (!state_q.empty())
    {
            cout << "h : " << current_state.h << endl;

            if (is_goal(current_state) == true)
            {
                    max_space += visited_list.size();
                    return current_state;
            }

            current_state = state_q.top();
            state_q.pop();
            if (!visited(current_state, visited_list))
            {
                    successors.clear();
                    successors = successor(current_state);
                    no_of_succ += successors.size();
                    for (int i = 0; i < successors.size(); i++)
                    {
                            heuristic(successors[i]);
                            state_q.push(successors[i]);
                    }
```

```
            }
            q_size = state_q.size();
            max_space = max(q_size, max_space);
    }
}
```

COMPARISON OF TIME AND SPACE COMPLEXITIES:

Algorithm	Maximum saved states	Total generated states by successor	Run time (seconds)
Greedy	589	588	2.35
A Star	589	588	2.42
Simple Hill Climbing	80	588	2.49
Random Hill Climbing	57	532	2.23

RESULT:

Through this project we were able to understand the working of the Hitori Grid game and how different AI algorithms can be used to find the goal state of the game.

DISCUSSION:

After completion of this project I have understood how heuristics affect our result and how different algorithms even with same heuristics have different approaches of selecting the next node and maintaining space to improve the goal.

CONCLUSION:

This project helped us gain understanding of the following points:

1. Understanding how Hitori is played and rules associated to it.
2. Understanding what successor function, goal test and other functions are.
3. Understanding how heuristics play an important role for finding the goal faster and optimally.

RESOURCES:

1. https://www.geeksforgeeks.org/
2. https://stackoverflow.com/
3. https://www.conceptispuzzles.com/
4. https://towardsdatascience.com/
5. https://www.researchgate.net/
6. https://math.stackexchange.com/
7. https://dzone.com/