Vishwakarma Institute of Technology
Department of Computer Science Engineering (CS)

Name	Vaishnavi Vijay Arthamwar
Division	CS-A
Batch	B3
Roll No.	16
PRN No.	12220242
Department	Computer
Subject	Artificial Intelligence
Assignment No.	3

Assingment No. 3

Title: Implementation of Informed strategies

Informed Search:

Informed search algorithms are a type of search algorithm that uses heuristic functions to guide the search process. For example, a heuristic function calculates the cost of moving from a starting state to a goal state. By employing this assessment, informed search algorithms can select search paths more likely to lead to the desired state. As a result, the search process is improved, making it quicker and more accurate for AI systems to make decisions.

A* search, iterative deepening A*, and best first search in artificial intelligence are examples of informed search algorithms. These algorithms are frequently employed in AI applications and have successfully resolved challenging issues.

Heuristics function:

Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$h(n) <= h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search there are two main algorithms which are given below:

1. **Best First Search Algorithm(Greedy search)**
2. **A* Search Algorithm**

A. Best First Search Algorithm(Greedy search)

The Best First Search algorithm in artificial intelligence, sometimes called Greedy Search, is an intelligent search algorithm. Because it uses heuristic functions to choose which path to explore first, it is known as an informed search. According to this technique, the next node to be explored is selected based on how close it is to the target. It starts by exploring the node with the shortest predicted distance.

Steps to perform a Best First Search in artificial intelligence:

1. Start node added to the open list.
2. While the open list is not empty: a. Eliminate the node from the open list with the shortest calculated distance. b. Provide the solution if the node is the goal node. c. Generate the children of the current node. d. Estimate the children's travel time to the objective after adding them to the open list.
3. There is only a solution if the open list is empty.

Advantages:

- If a decent heuristic function is applied, it finds a solution rapidly.
- It can be applied to a variety of issues.
- Implementing it is simple.

Disadvantages:

- As it only considers the expected distance to the target and not the actual cost of getting there, it could only sometimes come up with the best solution.
- It can become trapped in a local minimum and stop looking for alternate routes.
- To perform properly, it needs a good heuristic function.

Example: The algorithm selects the node with the lowest heuristic value as the next node to expand.

Time complexity: The worst-case time complexity of the Greedy best-first search in artificial intelligence is $O(bm)$.

Space Complexity: The worst-case space complexity of Greedy best-first search in artificial intelligence is $O(bm)$, where m is the maximum depth of the search space.

B. A* algorithm

To determine the best route from the starting node to the goal node, the A* Search Algorithm combines the cost function and the heuristic function. $f(n)=g(n)+h(n)$, where $g(n)$ is the

cost from the starting node to node n and $h(n)$ is the anticipated cost from node n to the goal node, expands the one with the lowest overall cost.

Steps of A* Search Algorithm:

1. Set the starting node's g-value to 0 and its h-value to the expected cost of getting there from the starting node.
2. Expand the node with the lowest total f(n) cost.
3. Stop the search and return the route from the starting node to the goal node if the expanded node is the desired node.
4. If not, determine the g-value and h-value, and update the f-value for each neighbour of the expanded node. Finally, add the neighbour to the open list if it's not already there.
5. Until the goal node is located or the open list is empty, repeat steps 2-4.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Complete:

- A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

Consistency: Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d).

Shortest path using Best First Search

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <algorithm> // Include for std::reverse

using namespace std;

struct Node {
    int value;
    int heuristic;

    Node(int v, int h) : value(v), heuristic(h) {}

    bool operator>(const Node& other) const {
        return heuristic > other.heuristic;
    }
};

unordered_map<int, vector<int>> graph = {
    {0, {1, 2, 3}},
    {1, {0, 4}},
    {2, {0, 4, 5}},
    {3, {0, 5}},
    {4, {1, 2, 6}},
    {5, {2, 3, 6}},
    {6, {4, 5, 7}},
    {7, {}}
};

unordered_map<int, int> calculateManhattanHeuristics(int goal) {
    unordered_map<int, int> heuristicValues;

    unordered_map<int, pair<int, int>> nodeCoordinates = {
        {0, {0, 0}},
        {1, {5, 8}},
```

```
        {2, {8, 5}},
        {3, {12, 2}},
        {4, {3, 3}},
        {5, {7, 7}},
        {6, {11, 11}},
        {7, {15, 15}}
    };

    pair<int, int> goalCoordinates = nodeCoordinates[goal];

    for (auto& entry : nodeCoordinates) {
        int node = entry.first;
        pair<int, int> coordinates = entry.second;
        int distance = abs(coordinates.first - goalCoordinates.first) +
abs(coordinates.second - goalCoordinates.second);
        heuristicValues[node] = distance;
    }

    return heuristicValues;
}

vector<int> reconstructPath(const unordered_map<int, int>& parent, int goal) {
    vector<int> path;
    while (parent.find(goal) != parent.end()) {
        path.push_back(goal);
        goal = parent.at(goal);
    }
    path.push_back(goal);
    reverse(path.begin(), path.end()); // Using std::reverse
    return path;
}

vector<int> bestFirstSearch(int start, int goal, const unordered_map<int,
int>& heuristicValues) {
    priority_queue<Node, vector<Node>, greater<Node>> pq;
    unordered_map<int, int> parent;
    unordered_set<int> visited;

    pq.push(Node(start, heuristicValues.at(start)));

    while (!pq.empty()) {
        Node current = pq.top();
        pq.pop();
        int currentValue = current.value;

        if (currentValue == goal) {
            return reconstructPath(parent, currentValue);
        }
```

```
        visited.insert(currentValue);

        for (int neighbor : graph[currentValue]) {
            if (visited.find(neighbor) == visited.end()) {
                parent[neighbor] = currentValue;
                pq.push(Node(neighbor, heuristicValues.at(neighbor)));
            }
        }
    }

    return {};
}

int main() {
    int startState, goalState;
    cout << "Enter the start state: ";
    cin >> startState;
    cout << "Enter the goal state: ";
    cin >> goalState;

    auto heuristicValues = calculateManhattanHeuristics(goalState);
    auto path = bestFirstSearch(startState, goalState, heuristicValues);

    cout << "Heuristic Values: " << endl;
    for (const auto& entry : heuristicValues) {
        cout << "Node " << entry.first << ": " << entry.second << endl;
    }

    cout << "Path: ";
    for (int node : path) {
        cout << node << " -> ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
PS D:\sem2-2023\AI\pracical 3 Informed search\final> cd "d:\sem2
ath.cpp -o bfsforShortestPath } ; if ($?) { .\bfsforShortestPath
Enter the start state: 1
Enter the goal state: 5
Heuristic Values:
Node 0: 14
Node 1: 3
Node 2: 3
Node 3: 10
Node 7: 16
Node 6: 8
Node 5: 0
Node 4: 8
Path: 1 -> 4 -> 2 -> 5 -> Reached
PS D:\sem2-2023\AI\pracical 3 Informed search\final>
```

Shortest Path using A* Algorithm:

Code:

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <algorithm> // For std::find and std::reverse

using namespace std;

class Node {
public:
    string name;
    int h;
    int fScore; // Add fScore here

    unordered_map<Node*, int> neighbours;

    Node(string name, int h) : name(name), h(h), fScore(0) {}

    void addNeighbour(Node* neighbour, int distance) {
        if (this == neighbour)
            return;
        neighbours[neighbour] = distance;
    }

    int Distance(Node* neighbour) {
        return neighbours[neighbour];
```

```
    }
};

class CompareFScore {
public:
    bool operator()(const Node* a, const Node* b) {
        return a->fScore > b->fScore;
    }
};

class AStar {
public:
    static vector<Node*> aStarSearch(Node* start, Node* goal);
    static vector<Node*> finalPath(unordered_map<Node*, Node*>& parent, Node*
current, unordered_map<Node*, int>& gScore,
            unordered_map<Node*, int>& fScore);
};

vector<Node*> AStar::aStarSearch(Node* start, Node* goal) {
    unordered_map<Node*, Node*> parent;
    unordered_map<Node*, int> gScore;
    unordered_map<Node*, int> fScore;
    vector<Node*> openList; // Use a separate vector to track nodes in the
open list
    vector<Node*> closedList;

    gScore[start] = 0;
    fScore[start] = start->h;
    start->fScore = start->h;
    openList.push_back(start);

    while (!openList.empty()) {
        // Find the node with the lowest fScore in the open list
        Node* current = openList.front();
        int minFScore = fScore[current];
        for (Node* node : openList) {
            if (fScore[node] < minFScore) {
                current = node;
                minFScore = fScore[node];
            }
        }
        openList.erase(remove(openList.begin(), openList.end(), current),
openList.end());

        if (current == goal) {
            return finalPath(parent, current, gScore, fScore);
        }
```

```
        closedList.push_back(current);

        for (auto& neighbour : current->neighbours) {
            Node* neighbourNode = neighbour.first;
            int distance = neighbour.second;

            if (find(closedList.begin(), closedList.end(), neighbourNode) !=
closedList.end()) {
                continue; // Ignore the neighbour which is already evaluated
            }

            int tentativeGScore = gScore[current] + current-
>Distance(neighbourNode);

            if (find(openList.begin(), openList.end(), neighbourNode) ==
openList.end() || tentativeGScore < gScore[neighbourNode]) {
                parent[neighbourNode] = current;
                gScore[neighbourNode] = tentativeGScore;
                fScore[neighbourNode] = gScore[neighbourNode] + neighbourNode-
>h;
                neighbourNode->fScore = fScore[neighbourNode];

                if (find(openList.begin(), openList.end(), neighbourNode) ==
openList.end()) {
                    openList.push_back(neighbourNode);
                }
            }
        }
    }
    return vector<Node*>();
}

vector<Node*> AStar::finalPath(unordered_map<Node*, Node*>& parent, Node*
current, unordered_map<Node*, int>& gScore,
        unordered_map<Node*, int>& fScore) {
    vector<Node*> path;
    while (current != nullptr) {
        path.push_back(current);
        current = parent[current];
    }
    reverse(path.begin(), path.end());
    return path;
}

int main() {
    Node* A = new Node("A", 14);
    Node* B = new Node("B", 12);
    Node* C = new Node("C", 11);
```

```
    Node* D = new Node("D", 6);
    Node* E = new Node("E", 4);
    Node* F = new Node("F", 11);
    Node* Z = new Node("Z", 0);
    A->addNeighbour(B, 4);
    A->addNeighbour(C, 3);
    B->addNeighbour(E, 12);
    B->addNeighbour(F, 5);
    C->addNeighbour(D, 7);
    C->addNeighbour(E, 11);
    D->addNeighbour(E, 2);
    F->addNeighbour(Z, 16);
    E->addNeighbour(Z, 5);

    vector<Node*> path = AStar::aStarSearch(A, Z);
    for (int i = 0; i < path.size(); i++) {
        cout << path[i]->name;
        if (i < path.size() - 1) {
            cout << "->";
        }
    }
    return 0;
}
```

Output:

```
PS D:\sem2-2023\AI\pracical 3 Informed search> cd "d:\sem2-2023\AI
 -o shortestpathusingAstar } ; if ($?) { .\shortestpathusingAstar }
 A->C->D->E->Z
PS D:\sem2-2023\AI\pracical 3 Informed search>
```

8-puzzle problem using A* algorithm:

An 8 puzzle is a simple game consisting of a 3 x 3 grid/matrix (containing 9 squares). One of the squares is empty. The object is to move squares around into different positions and have the numbers displayed in the "goal state".

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

#define n 3
const bool SUCCESS = true;

class state {
```

```
public:
  int board[n][n], g, f, h; // Added 'h' to store heuristic value
  state* came_from;
  state () {
    g = 0;
    f = 0;
    h = 0; // Initialize h to 0
    came_from = NULL;
  }
  static int heuristic (state from, state to) {
    int ret = 0;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (from.board[i][j] != to.board[i][j])
          ret++;
    return ret;
  }
  bool operator == (state a) {
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (this->board[i][j] != a.board[i][j])
          return false;
    return true;
  }
  void print () {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
        cout << board[i][j] << " ";
      cout << endl;
    }
    cout << "g = " << g << " | h = " << h << " | f = " << f << endl; // Print
heuristic value (h)
  }
};

vector <state> output;
bool lowerF (state a, state b) {
  return a.f < b.f;
}

bool isinset (state a, vector <state> b) {
  for (int i = 0; i < b.size(); i++)
    if (a == b[i])
      return true;
  return false;
}
```

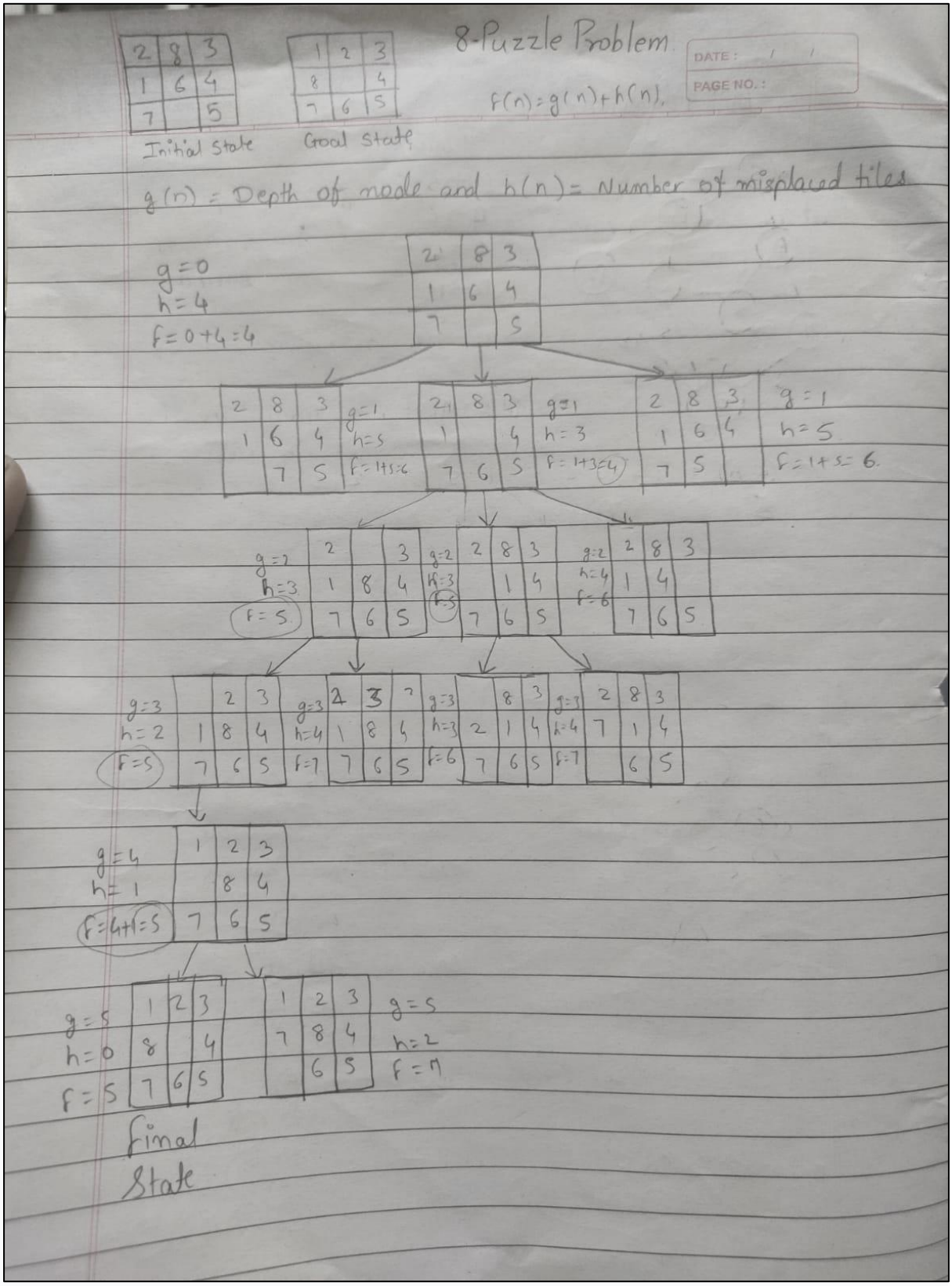
```
void addNeighbor (state current, state goal, int newi, int newj, int posi, int
posj, vector <state>& openset, vector <state> closedset) {
  state newstate = current;
  swap (newstate.board[newi][newj], newstate.board[posi][posj]);
  if (!isinset(newstate, closedset) && !isinset(newstate, openset)) {
      newstate.g = current.g + 1;
      newstate.h = state::heuristic(newstate, goal); // Calculate heuristic
(h)
      newstate.f = newstate.g + newstate.h; // Update f value
      state* temp = new state();
      *temp = current;
      newstate.came_from = temp;
      openset.push_back(newstate);
  }
}

void neighbors (state current, state goal, vector <state>& openset, vector
<state>& closedset) {
  int i, j, posi ,posj;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (current.board[i][j] == 0) {
        posi = i;
        posj = j;
        break;
      }
  i = posi, j = posj;
  if (i - 1 >= 0)
    addNeighbor(current, goal, i - 1, j, posi, posj, openset, closedset);
  if (i + 1 < n)
    addNeighbor(current, goal, i + 1, j, posi, posj, openset, closedset);
  if (j + 1 < n)
    addNeighbor(current, goal, i, j + 1, posi, posj, openset, closedset);
  if (j - 1 >= 0)
    addNeighbor(current, goal, i, j - 1, posi, posj, openset, closedset);
}

bool reconstruct_path(state current, vector<state> &came_from) {
    state *temp = &current;
    while(temp != NULL) {
        came_from.push_back(*temp);
        temp = temp->came_from;
    }
    return SUCCESS;
}

bool astar (state start, state goal) {
  vector <state> openset;
```

```
  vector <state> closedset;
  state current;
  start.g = 0;
  start.h = state::heuristic(start, goal); // Calculate heuristic for start
state
  start.f = start.g + start.h; // Update f value for start state
  openset.push_back(start);
  while (!openset.empty()) {
    sort(openset.begin(), openset.end(), lowerF);
    current = openset[0];
    if (current == goal)
      return reconstruct_path(current, output);
    openset.erase(openset.begin());
    closedset.push_back(current);
    neighbors(current, goal, openset, closedset);
  }
  return !SUCCESS;
}

int main () {
  state start, goal;

  // Initial state
  start.board[0][0] = 2; start.board[0][1] = 8; start.board[0][2] = 3;
  start.board[1][0] = 1; start.board[1][1] = 6; start.board[1][2] = 4;
  start.board[2][0] = 7; start.board[2][1] = 0; start.board[2][2] = 5;

  // Goal state
  goal.board[0][0] = 1; goal.board[0][1] = 2; goal.board[0][2] = 3;
  goal.board[1][0] = 8; goal.board[1][1] = 0; goal.board[1][2] = 4;
  goal.board[2][0] = 7; goal.board[2][1] = 6; goal.board[2][2] = 5;

  if (astar(start, goal) == SUCCESS) {
    for (int i = output.size() - 1; i >= 0; i--)
      output[i].print();
    cout << "SUCCESS!! GOAL STATE REACHED." << endl;
  }
  else cout << "FAIL" << endl;
  return 0;
}
```

Output:

```
2 8 3
1 6 4
7 0 5
g = 0 | h = 5 | f = 5
2 8 3
1 0 4
7 6 5
g = 1 | h = 3 | f = 4
2 0 3
1 8 4
7 6 5
g = 2 | h = 4 | f = 6
0 2 3
1 8 4
7 6 5
g = 3 | h = 3 | f = 6
1 2 3
0 8 4
7 6 5
g = 4 | h = 2 | f = 6
1 2 3
8 0 4
7 6 5
g = 5 | h = 0 | f = 5
SUCCESS!! GOAL STATE REACHED.
```

Example:


8-Puzzle Problem
f(n) = g(n) + h(n),
Initial State
Goal State
g(n) = Depth of node and h(n) = Number of misplaced tiles
g=0
h=4
f=0+4=4
Final State

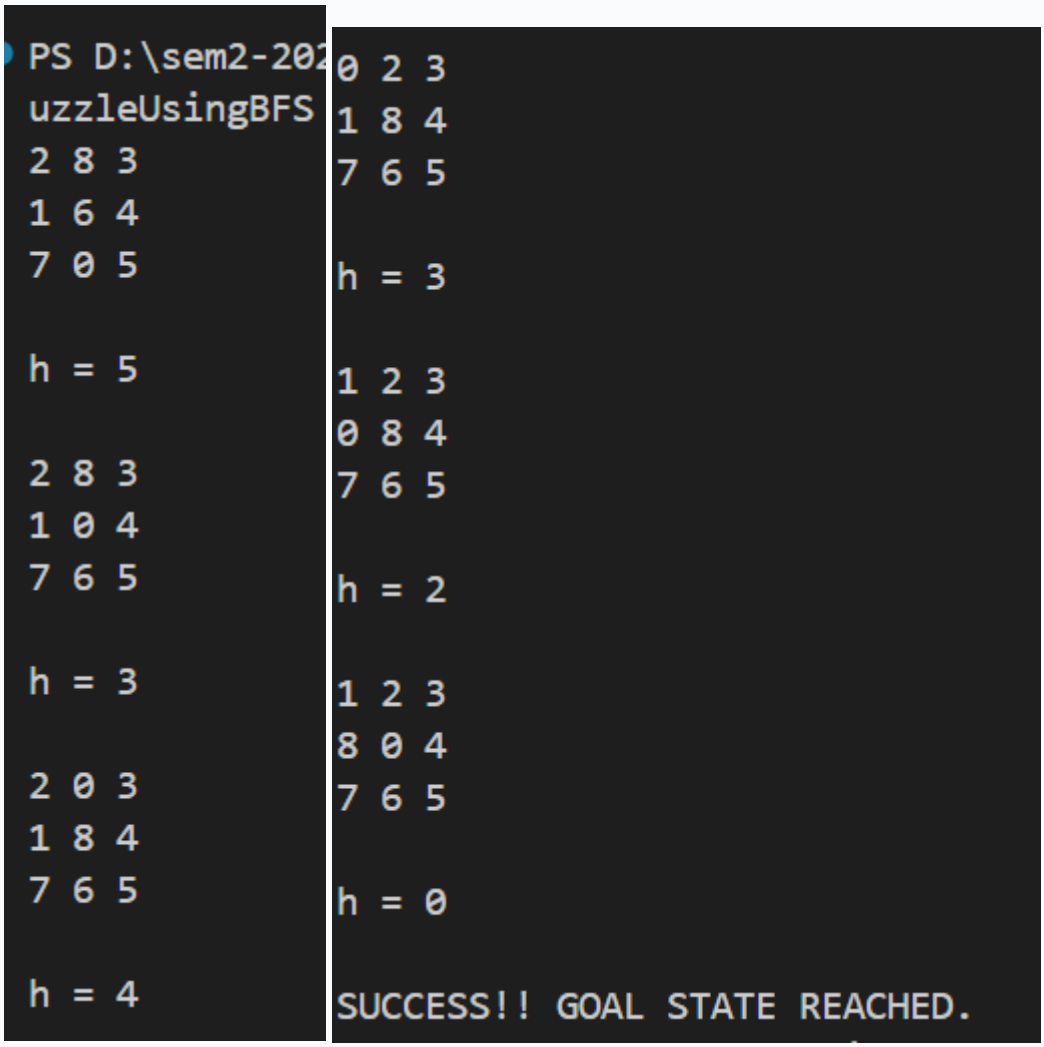
8-puzzle using BFS

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

#define n 3
const bool SUCCESS = true;

class state {
public:
    int board[n][n], h; // Removed g, f; kept only h for heuristic value
    state* came_from;
    state () {
        h = 0; // Initialize h to 0
        came_from = NULL;
    }
    static int heuristic (state from, state to) {
        int ret = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (from.board[i][j] != to.board[i][j])
                    ret++;
        return ret;
    }
    bool operator == (state a) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (this->board[i][j] != a.board[i][j])
                    return false;
        return true;
    }
    void print () {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                cout << board[i][j] << " ";
            cout << endl;
        }
        cout <<endl<< "h = " << h << endl << endl; // Print heuristic value
(h) with an extra newline
    }
};

vector <state> output;
bool lowerH (state a, state b) {
    return a.h < b.h;
}
```

```
bool isinset (state a, vector <state> b) {
    for (int i = 0; i < b.size(); i++)
        if (a == b[i])
            return true;
    return false;
}

void addNeighbor (state current, state goal, int newi, int newj, int posi, int
posj, vector <state>& openset, vector <state> closedset) {
    state newstate = current;
    swap (newstate.board[newi][newj], newstate.board[posi][posj]);
    if (!isinset(newstate, closedset) && !isinset(newstate, openset)) {
        newstate.h = state::heuristic(newstate, goal); // Calculate heuristic
(h)
        state* temp = new state();
        *temp = current;
        newstate.came_from = temp;
        openset.push_back(newstate);
    }
}

void neighbors (state current, state goal, vector <state>& openset, vector
<state>& closedset) {
    int i, j, posi ,posj;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (current.board[i][j] == 0) {
                posi = i;
                posj = j;
                break;
            }
    i = posi, j = posj;
    if (i - 1 >= 0)
        addNeighbor(current, goal, i - 1, j, posi, posj, openset, closedset);
    if (i + 1 < n)
        addNeighbor(current, goal, i + 1, j, posi, posj, openset, closedset);
    if (j + 1 < n)
        addNeighbor(current, goal, i, j + 1, posi, posj, openset, closedset);
    if (j - 1 >= 0)
        addNeighbor(current, goal, i, j - 1, posi, posj, openset, closedset);
}

bool reconstruct_path(state current, vector<state> &came_from) {
    state *temp = &current;
    while(temp != NULL) {
        came_from.push_back(*temp);
        temp = temp->came_from;
```

```
    }
    return SUCCESS;
}

bool best_first_search (state start, state goal) {
    vector <state> openset;
    vector <state> closedset;
    state current;
    start.h = state::heuristic(start, goal); // Calculate heuristic for start
state
    openset.push_back(start);
    while (!openset.empty()) {
        sort(openset.begin(), openset.end(), lowerH);
        current = openset[0];
        if (current == goal)
            return reconstruct_path(current, output);
        openset.erase(openset.begin());
        closedset.push_back(current);
        neighbors(current, goal, openset, closedset);
    }
    return !SUCCESS;
}

int main () {
    state start, goal;

    // Initial state
    start.board[0][0] = 2; start.board[0][1] = 8; start.board[0][2] = 3;
    start.board[1][0] = 1; start.board[1][1] = 6; start.board[1][2] = 4;
    start.board[2][0] = 7; start.board[2][1] = 0; start.board[2][2] = 5;

    // Goal state
    goal.board[0][0] = 1; goal.board[0][1] = 2; goal.board[0][2] = 3;
    goal.board[1][0] = 8; goal.board[1][1] = 0; goal.board[1][2] = 4;
    goal.board[2][0] = 7; goal.board[2][1] = 6; goal.board[2][2] = 5;

    if (best_first_search(start, goal) == SUCCESS) {
        for (int i = output.size() - 1; i >= 0; i--)
            output[i].print();
        cout << "SUCCESS!! GOAL STATE REACHED." << endl;
    }
    else cout << "FAIL" << endl;
    return 0;
}
```

Output:


PS D:\sem2-202
uzzleUsingBFS
2 8 3
1 6 4
7 0 5
h = 5
2 8 3
1 0 4
7 6 5
h = 3
2 0 3
1 8 4
7 6 5
h = 4
0 2 3
1 8 4
7 6 5
h = 3
1 2 3
0 8 4
7 6 5
h = 2
1 2 3
8 0 4
7 6 5
h = 0
SUCCESS!! GOAL STATE REACHED.

Difference between Informed and Uninformed search

Informed search	Uninformed search
Also known as Heuristic search	Also known as Blind search
Requires information to perform search	Do not require information to perform search.
Quick solution to problem.	May be time comsuming.
Cost is low.	Comparitively high in cost.
It can be both complete and incomplete.	It is always bound to complete.
The AI gets suggestions regarding how and where to find a solution to any problem.	The AI does not get any suggestions regarding what solution to find and where to find it. Whatever knowledge it gets is out of the information provided.
Eg. Greedy Search A* Search AO* Search Hill Climbing Algorithm	Eg. Depth First Search (DFS) Breadth First Search (BFS) Branch and Bound