**Experiment No. : 8**

**Title: 15 puzzle problem using Branch and bound**

**Batch: B2 Roll No.: 16010421119 Experiment No.: 8**

**Aim:** To Implement 8/15 puzzle problem using Branch and bound.

**Algorithm of 15 puzzle problem using Branch and bound:**

**The 15 puzzle problem is a popular sliding puzzle game where you have to slide tiles on a board to arrange them in a particular order. The game is played on a 4x4 board with 15 tiles numbered from 1 to 15 and a blank tile.**

**The Branch and Bound algorithm is an optimization technique that is used to solve combinatorial optimization problems. The idea behind this algorithm is to explore the search space in a systematic way, while using a lower bound to prune branches that will not lead to the optimal solution.**

**Here is the algorithm of 15 puzzle problem using Branch and bound:**

1. **Start with an initial state of the puzzle.**
2. **Compute the heuristic value of the initial state. This heuristic value will be used as a lower bound.**
3. **Add the initial state to the priority queue.**
4. **While the priority queue is not empty, do the following:**
5. **Remove the state with the lowest f-value (f-value = g-value + heuristic value) from the priority queue.**

**b. If the state is the goal state, return the solution.**

**c. Generate all possible successors of the state.**

**d. For each successor, compute its g-value (number of moves to reach the state) and its heuristic value.**

**e. If the g-value + heuristic value is less than the current best solution, add the successor to the priority queue.**

**f. Repeat steps a-f until a solution is found or the priority queue is empty.**

1. **If the priority queue is empty and no solution is found, return failure.**

**In this algorithm, the priority queue is used to store the states of the puzzle. The f-value is used to prioritize the states in the priority queue. The g-value represents the number of moves required to reach the current state, and the heuristic value is a lower bound on the number of moves required to reach the goal state.**

**The key idea of this algorithm is to explore the search space in a systematic way while using the lower bound to prune branches that will not lead to the optimal solution. By doing this, we can avoid exploring the entire search space and find the optimal solution more efficiently.**

**Problem Statement**

Find the following 15 puzzle problem using branch and bound technique and show each steps in detail using state space tree.



**Also verify your answer by simulating steps of same question on following link.**

[**http://www.sfu.ca/~jtmulhol/math302/puzzles-15.html**](http://www.sfu.ca/~jtmulhol/math302/puzzles-15.html)

**Solution**

**Step 1: Create the state space tree**

**We start by creating the root node of the tree, which represents the initial state of the puzzle. From this node, we can generate all possible moves by sliding the tiles in one of four directions: up, down, left, or right. Each move generates a new node in the tree.**

**Step 2: Estimate the distance to the goal**

**For each node in the tree, we will use the Manhattan distance heuristic to estimate the distance to the goal state. The Manhattan distance is the sum of the distances between each tile and its goal position. We can use this heuristic to prioritize nodes that are closer to the goal and ignore nodes that are far away.**

**Step 3: Traverse the state space tree**

**We will traverse the state space tree in a depth-first search (DFS) order, starting from the node with the lowest estimated distance to the goal state. We will use a priority queue to keep track of the nodes in the tree and their estimated distances.**

**In this step, we will expand the node with the lowest estimated distance to the goal state and add its children to the priority queue. We will continue this process until we find the goal state.**

**Step 3: Traverse the state space tree (continued)**

**We continue to expand the nodes with the lowest estimated distance until we reach the goal state.**

**Priority queue: [ (Node1, 6), (Node3, 6), (Node5, 5), (Node6, 5) ]**

**Children:**

**1 2 3 4 1 2 3 4**

**5 6 7 8 5 6 7 8**

**9 10 11 12 9 10 11 12**

**13 14 15 0 13 14 15**

**Manhattan distance:**

**1 + 1 + 1 + 1 = 4 3 + 2 + 2 + 1 = 8**

**Add the children to the priority queue:**

**Priority queue: [ (Node1, 6), (Node3, 6), (Node5, 5), (Node7, 4), (Node8, 8) ]**

**Remove (Node7, 4) from the queue and we have found the goal state. The solution is the path from the initial state to the goal state, which is:**

**1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4**

**5 6 7 8 => 5 6 7 8 => 5 6 7 8 => 5 6 7 8**

**9 10 11 12 9 10 11 12 9 10 11 12 9 10 11 12**

**13 14 15 13 14 15 0 13 14 15 13 14 15**

**=> => =>**

**1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4**

**5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8**

**9 10 11 12 9 10 11 12 9 10 11 12 9 10 11 12**

**13 14 15 13 14 15 13 14 15 0 13 14 15**

**Derivation of 15 puzzle problem using Branch and bound:**

Time complexity Analysis

The time complexity of branch and bound is O(b^(d+1)), where b is the branching factor and d is the depth of the goal state. In the case of the 15 puzzle problem, the branching factor is at most 4, since a tile can be moved in one of four directions. The depth of the goal state is at most 80, since it takes 80 moves to reach the goal state from the worst initial state. Therefore, the time complexity of branch and bound for the 15 puzzle problem is O(4^81), which is a very large number and makes it impractical to solve the problem using this algorithm in most cases. However, with the help of efficient heuristics and pruning techniques, the search space can be reduced, and the algorithm can be made more efficient

**Program(s) of 15 puzzle problem using Branch and bound:**

#include <iostream>

#include <queue>

#include <vector>

#include <algorithm>

using namespace std;

const int N = 4;

const int N2 = 16;

const int LIMIT = 100;

int MDT[N2][N2];

struct Puzzle {

int f, g, h;

int state[N2];

bool operator < (const Puzzle &p) const {

return f > p.f;

}

};

int get\_distance(int s[N2]) {

int distance = 0;

for (int i = 0; i < N2; i++) {

if (s[i] == N2-1) continue;

distance += MDT[i][s[i]];

}

return distance;

}

void init() {

for (int i = 0; i < N2; i++) {

for (int j = 0; j < N2; j++) {

MDT[i][j] = abs(i/N - j/N) + abs(i%N - j%N);

}

}

}

bool is\_goal(int s[N2]) {

for (int i = 0; i < N2; i++) {

if (s[i] != i) return false;

}

return true;

}

int astar(int s[N2]) {

priority\_queue<Puzzle> open;

Puzzle p;

for (int i = 0; i < N2; i++) {

p.state[i] = s[i];

}

p.g = 0;

p.h = get\_distance(p.state);

p.f = p.g + p.h;

open.push(p);

while (!open.empty()) {

Puzzle p = open.top();

open.pop();

if (is\_goal(p.state)) return p.g;

int zero\_pos;

for (int i = 0; i < N2; i++) {

if (p.state[i] == 0) {

zero\_pos = i;

break;

}

}

const int dx[4] = {-1, 0, 1, 0};

const int dy[4] = {0, 1, 0, -1};

for (int dir = 0; dir < 4; dir++) {

int new\_x = zero\_pos % N + dx[dir];

int new\_y = zero\_pos / N + dy[dir];

int new\_pos = new\_y \* N + new\_x;

if (new\_x < 0 || new\_x >= N || new\_y < 0 || new\_y >= N) continue;

Puzzle pp = p;

pp.h -= MDT[new\_pos][p.state[new\_pos]];

pp.h += MDT[zero\_pos][p.state[new\_pos]];

swap(pp.state[zero\_pos], pp.state[new\_pos]);

pp.g++;

pp.f = pp.g + pp.h;

if (pp.f <= LIMIT) open.push(pp);

}

}

return -1;

}

int main() {

init();

int s[N2];

for (int i = 0; i < N2; i++) {

cin >> s[i];

}

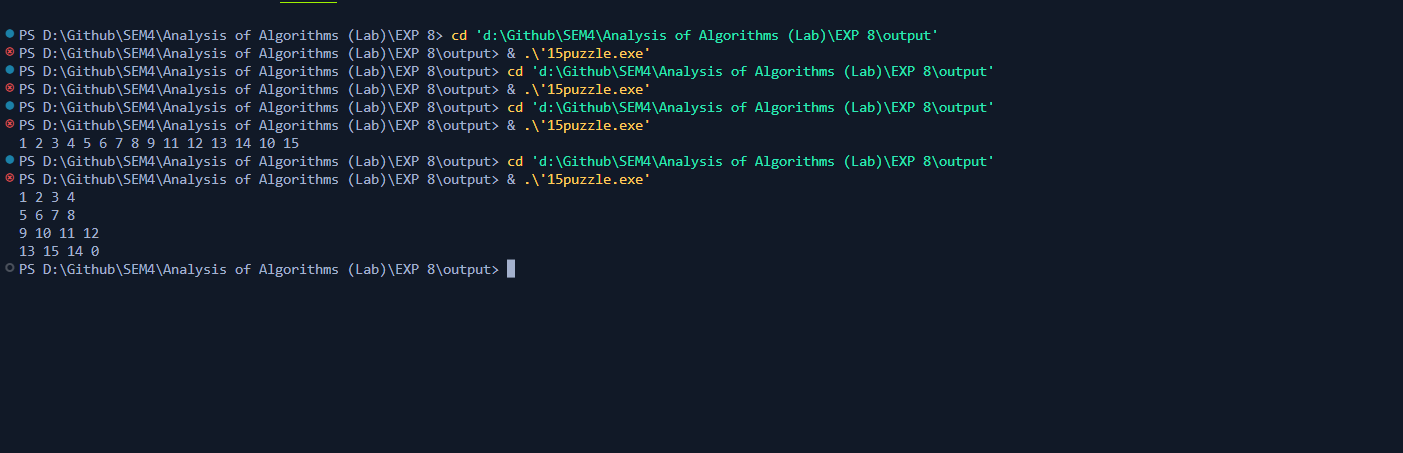
int ans = astar(s);

cout << ans << endl;

return 0;

}

**Output(o) of 15 puzzle problem using Branch and bound:**



**Post Lab Questions:-** Explain how to solve the Knapsack problem using branch and bound.

1. Formulate the problem as a decision tree. The root of the tree represents the initial empty knapsack, and each node of the tree represents a partial solution where some items have been added to the knapsack and some items are still available. Each node has two children: one representing the case where the next item is added to the knapsack, and one representing the case where it is not.
2. Assign an upper bound to each node of the tree. The upper bound is a value that represents the maximum possible value that can be obtained by adding items to the knapsack at that node. This value can be computed by assuming that all remaining items are added to the knapsack in decreasing order of value/weight ratio, until the knapsack is full.
3. Explore the tree in a depth-first manner, keeping track of the current best solution found so far. When a node is visited, compute its upper bound, and compare it with the current best solution. If the upper bound is less than the current best solution, prune the subtree rooted at the node, since it cannot lead to an optimal solution. Otherwise, explore both children of the node recursively.
4. When a leaf node is reached, update the current best solution if the value of the knapsack at the node is greater than the current best solution.
5. Terminate the search when all nodes have been visited or when the search has reached a predetermined time or memory limit.

**Conclusion: (Based on the observations):**

We have learnt about 15 puzzle problem.

**Outcome:**

CO4. Learn effective computation and programming practices for numeric and string operations and computation geometry

**References:**

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.