

Name	Parth Gandhi
UID No.	2021300033
Class & Division	S.E. COMPS A (BATCH B)
Experiment No.	8

Aim: Experiment on solving 15 puzzle problem using branch and bound strategy.

Theory:

The 15-puzzle problem is a classic puzzle where a player must rearrange 15 tiles numbered from 1 to 15 on a 4x4 grid by sliding them around, leaving one empty space, until they are in numerical order. This puzzle can be solved using a search algorithm like the branch and bound strategy. The branch and bound strategy is a method for solving combinatorial optimization problems by recursively branching on the search space and pruning branches that cannot lead to an optimal solution. The idea is to explore the search space systematically, evaluating and bounding the cost of each partial solution found, and discarding partial solutions that are worse than the best found so far.

In the context of the 15-puzzle problem, the search space consists of all possible configurations of the puzzle. Each configuration represents a node in the search tree, and the edges represent the possible moves that can be made to reach a new configuration.

To apply the branch and bound strategy to the 15 puzzle problem, we can define a cost function that measures the distance between the current configuration and the goal configuration. One common choice for the cost function is the sum of the Manhattan distances between the tiles and their goal positions.

At each step of the search, we evaluate the cost of the current partial solution and compare it with the best solution found so far. If the cost of the current partial solution is higher than the best solution found so far, we can prune that branch of the search tree because it cannot lead to an optimal solution. If the cost of the current partial solution is equal to the cost of the best solution found so far, we update the best solution.

To reduce the search space, we can use heuristics to guide the search towards more promising branches of the search tree. One common heuristic is to use the number of tiles out of place as an estimate of the cost to reach the goal state. We can use this heuristic to prioritize exploring configurations that are closer to the goal state.

Overall, the branch and bound strategy is an effective approach for solving the 15-puzzle problem. By systematically exploring the search space and pruning unpromising branches, we can find the optimal solution efficiently.

Algorithm:

1. Initialize the puzzle board with an initial state of the puzzle and the goal state.
2. Define a variable "f" to indicate if the goal state has been reached or not. Also, define a variable "moves" to count the number of moves taken to reach the goal state.
3. Define a function "printsol" to print the goal state and the number of moves taken to reach it.
4. Define a function "calculatecost" to calculate the cost of a particular state of the puzzle by comparing it with the goal state.
5. Define a function "solve" that takes the current state of the puzzle, the goal state, the

direction of the last move, and the position of the empty slot (represented by "-1") on the board. The function is called recursively until the goal state is reached.

6. Inside the "solve" function, check if the goal state has been reached. If yes, call the "printsol" function and set "f" to 1 to stop further recursion.
7. If the goal state has not been reached, create a 4x4 vector "cost" and initialize it with a large integer value.
8. Check the valid moves that can be made from the current state of the puzzle, i.e., up, down, left, or right. Calculate the cost of each resulting state of the puzzle and store it in the "cost" vector. Also, store the direction of the move, the row, and column of the empty slot for each move.
9. Sort the "cost" vector based on the cost of each resulting state in ascending order.
10. Choose the move with the lowest cost and make that move on the puzzle board. Update the direction of the last move and the position of the empty slot.
11. If the resulting state of the puzzle is the goal state, call the "printsol" function and set "f" to 1 to stop further recursion.
12. If the resulting state is not the goal state, call the "solve" function recursively with the resulting state, the goal state, the direction of the last move, and the position of the empty slot.
13. After the "solve" function returns, undo the last move made on the puzzle board to backtrack and explore other possible moves.
14. In the main function, take the input puzzle board from the user and call the "calculatecost" function to check if it is already in the goal state. If yes, print that the input puzzle is already in the goal state. If not, call the "solve" function with the input puzzle board, the goal state, and the position of the empty slot.
15. Print the number of moves taken to reach the goal state.

Code:

```
#include <bits/stdc++.h>
using namespace std;
```

```
int f = 0, moves = 0;
void printsol(int a[4][4]){
    cout << "\nGoal state reached. Moves taken: " << moves << endl;
    for (int i = 0; i < 4; i++){
        for (int j = 0; j < 4; j++){
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
    f = 1;
}
```

```
int calculatecost(int a[4][4], int goal[4][4]){
    int cost = 0;
    for (int i = 0; i < 4; i++){
        for (int j = 0; j < 4; j++){
            if (a[i][j] != goal[i][j])
                cost++;
        }
    }
    return cost;
}
```

```

void solve(int a[4][4], int goal[4][4], int direction, int x, int y){
    moves++;
    if (f == 0){
        int left, right, top, bottom;
        vector<vector<int>> cost(4, vector<int>(4, INT_MAX));
        if (direction != 3 && x - 1 >= 0){
            int b[4][4];
            copy(&a[0][0], &a[0][0] + 16, &b[0][0]);
            swap(b[x][y], b[x - 1][y]);
            cost[0][0] = calculatecost(b, goal);
            cost[0][1] = 1;
            cost[0][2] = x - 1;
            cost[0][3] = y;
        }
        if (direction != 4 && y + 1 < 4){
            int b[4][4];
            copy(&a[0][0], &a[0][0] + 16, &b[0][0]);
            swap(b[x][y], b[x][y + 1]);
            cost[1][0] = calculatecost(b, goal);
            cost[1][1] = 2;
            cost[1][2] = x;
            cost[1][3] = y + 1;
        }
        if (direction != 1 && x + 1 < 4){
            int b[4][4];
            copy(&a[0][0], &a[0][0] + 16, &b[0][0]);
            swap(b[x][y], b[x + 1][y]);
            cost[2][0] = calculatecost(b, goal);
            cost[2][1] = 3;
            cost[2][2] = x + 1;
            cost[2][3] = y;
        }
        if (direction != 2 && y - 1 >= 0){
            int b[4][4];
            copy(&a[0][0], &a[0][0] + 16, &b[0][0]);
            swap(b[x][y], b[x][y - 1]);
            cost[3][0] = calculatecost(b, goal);
            cost[3][1] = 4;
            cost[3][2] = x;
            cost[3][3] = y - 1;
        }
        sort(cost.begin(), cost.end());
        direction = cost[0][1];
        if (direction == 1)
            swap(a[x][y], a[x - 1][y]);
        if (direction == 2)
            swap(a[x][y], a[x][y + 1]);
        if (direction == 3)
            swap(a[x][y], a[x + 1][y]);
        if (direction == 4)
            swap(a[x][y], a[x][y - 1]);
        if (cost[0][0]==0)
            printsol(a);
        solve(a, goal, direction, cost[0][2], cost[0][3]);
    }
}

```

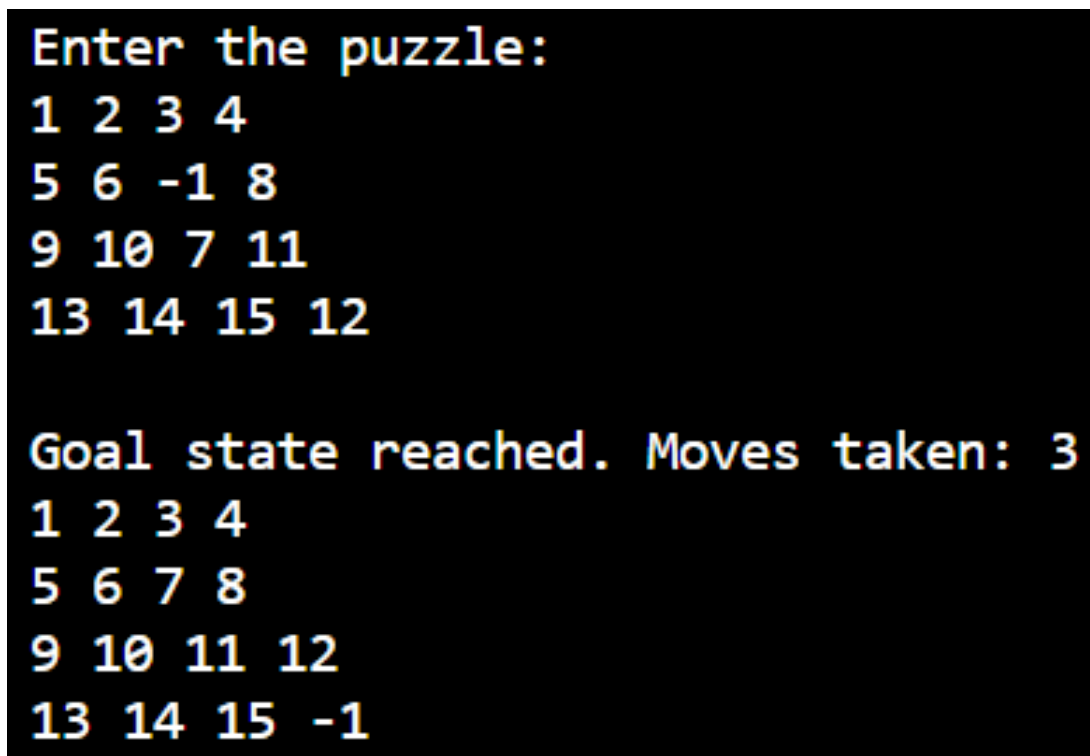
```

    }
}

int main(){
    int a[4][4];
    int goal[4][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, -1}};
    cout << "Enter the puzzle:" << endl;
    int x, y;
    for (int i = 0; i < 4; i++){
        for (int j = 0; j < 4; j++){
            cin >> a[i][j];
            if (a[i][j] == -1){
                x = i;
                y = j;
            }
        }
    }
    int flag = calculatecost(a, goal);
    if (flag == 0)
        cout<<"Entered puzzle is already in goal state";
    else
        solve(a, goal, 0, x, y);
    return 0;
}

```

Output:



```

Enter the puzzle:
1 2 3 4
5 6 -1 8
9 10 7 11
13 14 15 12

Goal state reached. Moves taken: 3
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -1

```

Conclusion: Successfully wrote a program to implement 15 puzzle problem using branch and bound strategy.