Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	8
Aim	Implement 15 puzzle problem
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:

The 15-puzzle problem can be solved using the Branch and Bound technique, which is a general algorithm for solving combinatorial optimization problems, including puzzles like the 15-puzzle. The Branch and Bound technique is a systematic approach that involves exploring the search space of possible solutions in a tree-like structure, pruning branches of the tree based on certain criteria to optimize the search process.

Here's an overview of how the Branch and Bound technique can be applied to solve the 15-puzzle problem:

1. Initial State: Start with the initial state of the 15-puzzle, which is the given arrangement of the 15 numbered tiles on the 4x4 grid with one blank tile.

2. Heuristic Function: Define a heuristic function that estimates the cost or distance from the current state to the goal state, which is the arrangement of the tiles in numerical order with the blank tile in the bottom-right corner. A commonly used heuristic for the 15-puzzle is the Manhattan distance, which calculates the sum of the distances between each tile and its desired position.

3. Priority Queue: Maintain a priority queue to store the partial solutions or states of the puzzle, sorted based on their heuristic values. The heuristic value serves as a lower bound on the actual cost of reaching the goal state.

4. Branching and Pruning: Expand the current state of the puzzle by generating all possible moves or transitions to neighboring states, such as sliding a tile into the blank space. For each generated state, calculate its heuristic value and add it to the priority queue.

5. Search: Repeatedly select the state with the lowest heuristic value from the priority queue, and expand it by generating its neighboring states. Prune the search tree by discarding states that have heuristic values higher than the current best solution found so far. Continue this process until the goal state is reached or no more states are left in the priority queue.

6. Solution: If the goal state is reached, the optimal solution to the 15-puzzle is found. Otherwise, if the priority queue is empty and no solution is found, then the 15-puzzle is unsolvable from the given initial state.

7. Optimization: The performance of the Branch and Bound technique can be improved by using various optimization techniques, such as using a more sophisticated heuristic function, exploring the search space in a smart way to avoid redundant states, and parallelizing the search process across multiple processors.

The Branch and Bound technique is a powerful algorithm for solving the 15-puzzle problem optimally, as it guarantees finding the optimal solution with the lowest number of moves. However, it can be computationally expensive for large puzzles, as the size of the search space grows exponentially with the number of tiles. Therefore, efficient implementation and optimization techniques are crucial for solving larger instances of the 15-puzzle within reasonable time limits.

Algorithm:

1. Accept the puzzle input
2. Print the current state of the puzzle grid for debugging purposes.
3. Calculate the Manhattan distance heuristic for a given state of the puzzle. The heuristic is the sum of the distances of each tile from its goal position.
4. Implement a function to check if the puzzle is solved. It checks if the current state of the puzzle grid matches the goal state, where the tiles are arranged in ascending order from left to right, top to bottom, and the blank tile is in the bottom-right corner.
5. Implement a function to generate neighboring states from a given state. It slides the blank tile in all possible directions (up, right, down, left) to create new states by swapping the blank tile with a neighboring tile.
6. Implement the Branch and Bound algorithm as follows:
    - Initialize an empty priority queue to store the states to be explored, with the initial state of the puzzle as the first element in the queue.
    - Initialize an empty set to store the visited states to avoid redundant exploration.
    - While the priority queue is not empty:
        # Pop the state with the lowest estimated cost (heuristic + actual cost) from the priority queue.
        # If the popped state is the goal state, return the solution.
        # Generate neighboring states from the popped state
    - For each neighboring state:
        # If the state has not been visited before, calculate its estimated cost (heuristic + actual cost) and add it to the priority queue.
        # Update the visited set with the new state.
    - If the priority queue becomes empty and no solution is found, return failure.
7. The estimated cost for a state is calculated as the sum of the cost to reach that state from the initial state (actual cost) and the Manhattan distance heuristic of that state. The actual cost is the number of moves made to reach the current state from the initial state, which can be tracked by incrementing a counter each time a move is made.

8. The Branch and Bound algorithm uses a priority queue to explore states in an informed manner, prioritizing states with lower estimated costs first to guide the search towards the goal state more efficiently.

Program:

```
#include <stdio.h>
#include <conio.h>

int m = 0, n = 4;

int cal(int temp[10][10], int t[10][10])
{
    int i, j, m = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (temp[i][j] != t[i][j])
                m++;
        }
    return m;
}

int check(int a[10][10], int t[10][10])
{
    int i, j, f = 1;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (a[i][j] != t[i][j])
                f = 0;
    return f;
}

void main()
{
    int p, i, j, n = 4, a[10][10], t[10][10], temp[10][10], r[10][10];
    int m = 0, x = 0, y = 0, d = 1000, dmin = 0, l = 0;

    printf("\nEnter the matrix to be solved,space with zero :\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
```

```
printf("\nEnter the target matrix,space with zero :\n");
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        scanf("%d", &t[i][j]);

printf("\nEntered Matrix is :\n");
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
        printf("%d\t", a[i][j]);
    printf("\n");
}

printf("\nTarget Matrix is :\n");
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
        printf("%d\t", t[i][j]);
    printf("\n");
}

while (!(check(a, t)))
{
    l++;
    d = 1000;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (a[i][j] == 0)
            {
                x = i;
                y = j;
            }
        }

    // To move upwards
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            temp[i][j] = a[i][j];

    if (x != 0)
    {
        p = temp[x][y];
        temp[x][y] = temp[x - 1][y];
```

```
            temp[x - 1][y] = p;
        }
        m = cal(temp, t);
        dmin = l + m;
        if (dmin < d)
        {
            d = dmin;
            for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                    r[i][j] = temp[i][j];
        }

        // To move downwards
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                temp[i][j] = a[i][j];
        if (x != n - 1)
        {
            p = temp[x][y];
            temp[x][y] = temp[x + 1][y];
            temp[x + 1][y] = p;
        }
        m = cal(temp, t);
        dmin = l + m;
        if (dmin < d)
        {
            d = dmin;
            for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                    r[i][j] = temp[i][j];
        }

        // To move right side
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                temp[i][j] = a[i][j];
        if (y != n - 1)
        {
            p = temp[x][y];
            temp[x][y] = temp[x][y + 1];
            temp[x][y + 1] = p;
        }
        m = cal(temp, t);
        dmin = l + m;
        if (dmin < d)
```

```
        {
            d = dmin;
            for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                    r[i][j] = temp[i][j];
        }

        // To move left
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                temp[i][j] = a[i][j];
        if (y != 0)
        {
            p = temp[x][y];
            temp[x][y] = temp[x][y - 1];
            temp[x][y - 1] = p;
        }
        m = cal(temp, t);
        dmin = l + m;
        if (dmin < d)
        {
            d = dmin;
            for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                    r[i][j] = temp[i][j];
        }

        printf("\nCalculated Intermediate Matrix Value :\n");
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
                printf("%d\t", r[i][j]);
            printf("\n");
        }
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
            {
                a[i][j] = r[i][j];
                temp[i][j] = 0;
            }
        printf("Minimum cost : %d\n", d);
    }
    getch();
}
```

Results:

```
Enter the matrix to be solved,space with zero :
1  2  3  4
5  6  7  8
9  10 11 0
13 14 15 12

Enter the target matrix,space with zero :
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

Entered Matrix is :
1       2       3       4
5       6       7       8
9       10      11      0
13      14      15      12

Target Matrix is :
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      0

Calculated Intermediate Matrix Value :
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      0
Minimum cost : 1
```

Conclusion:

I have understood the 15 puzzle problem and successfully implemented it.