Experiment No. 11
15 puzzle problem
Date of Performance:
Date of Submission:

Experiment No. 11

Title: 15 Puzzle

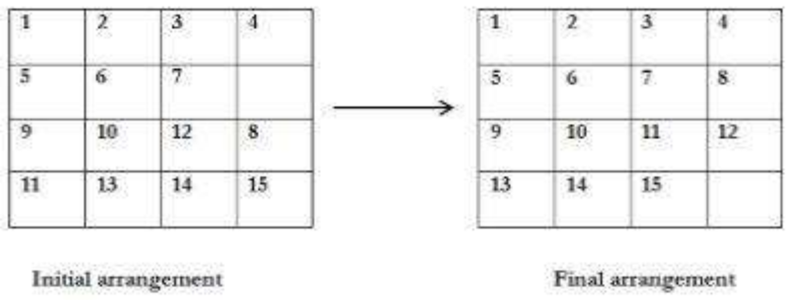
Aim: To study and implement 15 puzzle problem

Objective: To introduce Backtracking and Branch-Bound methods

Theory:

The 15 puzzle problem is invented by Sam Loyd in 1878.

- In this problem there are 15 tiles, which are numbered from 0 – 15.
- The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.
- The initial and goal arrangement is shown by following figure.



1	2	3	4
5	6	7	
9	10	12	8
11	13	14	15

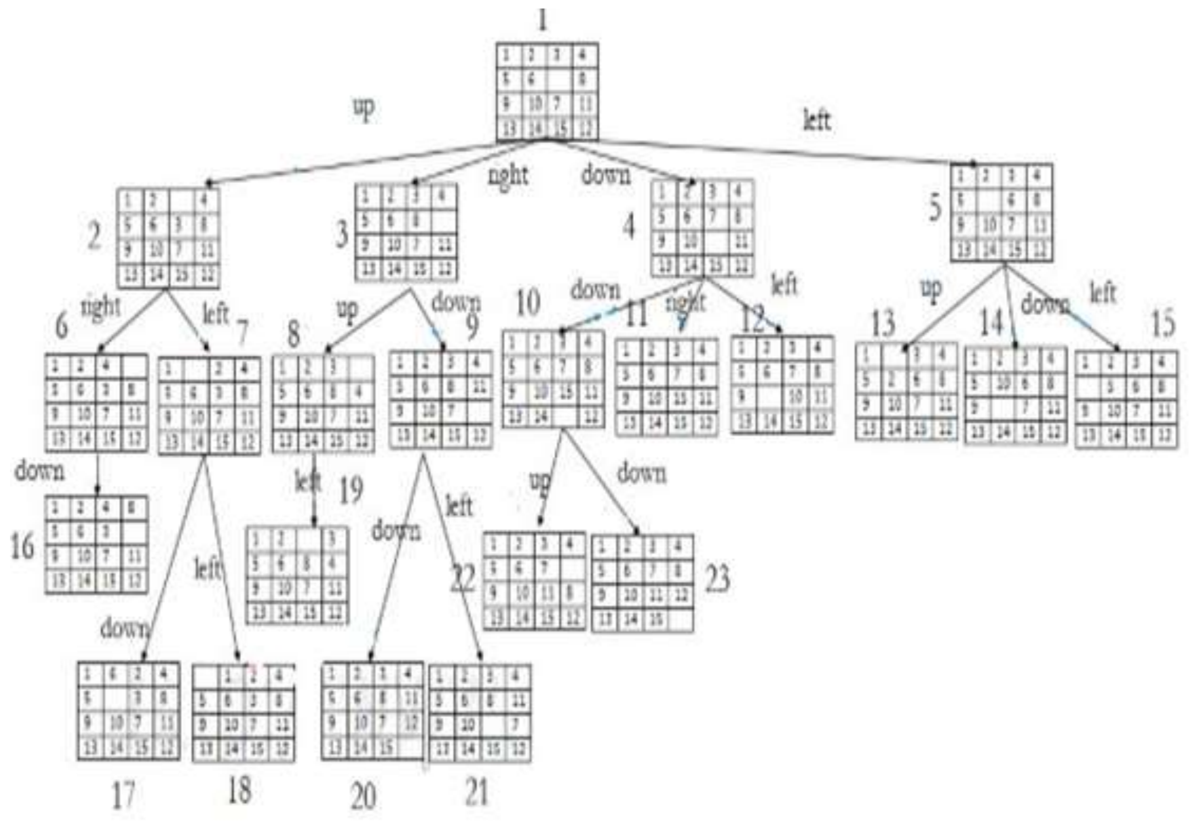
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 12

- There is always an empty slot in the initial arrangement.
- The legal moves are the moves in which the tiles adjacent to ES are moved to either left, right, up or down.
- Each move creates a new arrangement in a tile.
- These arrangements are called as states of the puzzle.
- The initial arrangement is called as initial state and goal arrangement is called as goal state.
- The state space tree for 15 puzzle is very large because there can be 16! Different arrangements.
- A partial state space tree can be shown in figure.

- In state space tree, the nodes are numbered as per the level.
- Each next move is generated based on empty slot positions.
- Edges are label according to the direction in which the empty space moves.
- The root node becomes the E – node.
- The child node 2, 3, 4 and 5 of this E – node get generated.
- Out of which node 4 becomes an E – node. For this node the live nodes 10, 11, 12 gets generated.
- Then the node 10 becomes the E – node for which the child nodes 22 and 23 gets generated.
- Finally we get a goal state at node 23.
- We can decide which node to become an E – node based on estimation formula.

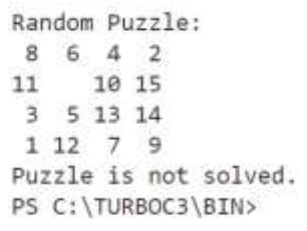
Example:



Implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 4

// Function to print the puzzle
void printPuzzle(int puzzle[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (puzzle[i][j] == 0)
                printf("  ");
            else
                printf("%2d", puzzle[i][j]);
            printf(" ");
        }
        printf("\n");
    }
}

// Function to check if the puzzle is solved
int isSolved(int puzzle[SIZE][SIZE]) {
    int count = 1;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (i == SIZE - 1 && j == SIZE - 1)
                return 1; // Last cell should be empty
            if (puzzle[i][j] != count++)
                return 0;
        }
    }
    return 1;
}

// Function to generate a solvable random puzzle
void generateRandomPuzzle(int puzzle[SIZE][SIZE]) {
    int numbers[SIZE * SIZE];
    int k = 0;

    // Fill the array with numbers from 1 to SIZE*SIZE
    for (int i = 0; i < SIZE * SIZE; i++) {
        numbers[i] = i;
    }

    // Shuffle the numbers array
    for (int i = 0; i < SIZE * SIZE; i++) {
        int temp = numbers[i];
        int randomIndex = rand() % (SIZE * SIZE);
        numbers[i] = numbers[randomIndex];
        numbers[randomIndex] = temp;
    }

    // Fill the puzzle with the shuffled numbers
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            puzzle[i][j] = numbers[k++];
        }
    }

    // Check if the generated puzzle is solvable, if not, regenerate
    while (!isSolvable(puzzle)) {
        generateRandomPuzzle(puzzle);
    }
}

// Function to count inversions in the puzzle
int countInversions(int puzzle[SIZE][SIZE]) {
    int inversions = 0;
    int array[SIZE * SIZE];
    int k = 0;

    // Convert puzzle into a 1D array
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            array[k++] = puzzle[i][j];
        }
    }
    // Count inversions
    for (int i = 0; i < SIZE * SIZE - 1; i++) {
        for (int j = i + 1; j < SIZE * SIZE; j++) {
            if (array[i] != 0 && array[j] != 0 && array[i] > array[j])
                inversions++;
        }
    }

    return inversions;
}

// Function to check if the puzzle is solvable
int isSolvable(int puzzle[SIZE][SIZE]) {
    int inversions = countInversions(puzzle);
    int blankRow = 0;

    // Find the row index of the blank tile
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (puzzle[i][j] == 0) {
                blankRow = i + 1;
                break;
            }
        }
    }

    // If the puzzle size is odd
    if (SIZE % 2 != 0) {
        return inversions % 2 == 0; // Solvable if the number of
inversions is even
    } else {
        // If the blank tile is on an even row from the bottom, the puzzle
is solvable
        return (blankRow + inversions) % 2 != 0;
    }
}

int main() {
    int puzzle[SIZE][SIZE];
    srand(time(NULL)); // Seed for random number generation
    generateRandomPuzzle(puzzle);

    printf("Random Puzzle:\n");
    printPuzzle(puzzle);

    if (isSolved(puzzle)) {
        printf("Puzzle is solved!\n");
    } else {
        printf("Puzzle is not solved.\n");
    }

    return 0;
}
```

Output:



```
Random Puzzle:
  8  6  4  2
 11    10 15
  3  5 13 14
  1 12  7  9
Puzzle is not solved.
PS C:\TURBOC3\BIN>
```

Conclusion: In conclusion, this C program provides a solution to the 15 puzzle problem, a classic sliding puzzle game. It generates a random solvable puzzle configuration and checks whether the generated puzzle is solvable. The program utilizes concepts such as array manipulation, random number generation, inversion counting, and puzzle solvability checks. By understanding and implementing these concepts, users can gain insight into solving similar combinatorial optimization problems. Additionally, the program demonstrates the application of logic and algorithms in solving puzzles, making it a valuable learning resource for programming enthusiasts and puzzle enthusiasts alike.