**Experiment No. : 8**

# Title: 15 puzzle problem using Branch and bound

**Batch: B-4** **Roll No.: 16010422234** **Name: Chandana Ramesh Galgali**

**Experiment No.: 08**

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

---

**Algorithm of 15 puzzle problem using Branch and bound:**
The 15 puzzle is a sliding puzzle consisting of a frame of numbered square tiles in random order with one tile missing. The objective is to place the tiles in order by making sliding moves that use the empty space. Here's a general approach using the branch and bound algorithm:

1. Initialization:
    - Start with the initial state of the board.
    - Calculate the cost of the initial state using a cost function (like the number of tiles out of place or the sum of the distances each tile is from its goal position).
    - Set this as the current node.

2. Set Up a Priority Queue:
    - Nodes are placed in the priority queue based on their cost, with the lowest cost at the front.

3. Branching:
    - At each step, "branch" by generating all possible states from the current state by making one move.
    - For each new state, calculate its cost.

4. Bounding:
    - For each new state, if its cost is better than the current best solution, keep it in the priority queue.
    - If a state reaches the goal state (all tiles are in their place), check if its cost is lower than the current best solution. If so, update the best solution.

5. Iterate:
    - Continue this process, expanding the least costly nodes first (as determined by the priority queue).
    - The search can be optimized further by not revisiting states that have already been expanded.

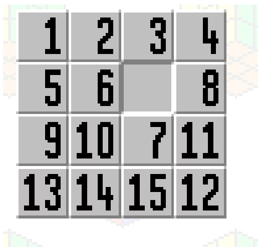
6. Solution:
    - The process stops when the goal state is reached with the minimum cost, or the priority queue is empty.

---

**Working of 15 puzzle problem using Branch and bound:**
Branch and bound utilizes a systematic method of exploring the state space tree by generating successor states for each state, calculating their cost, and choosing the state with the lowest estimated cost to expand next. This estimated cost serves as a lower bound on the solution path going through that node. The algorithm uses a priority queue to keep track of all unexplored nodes, selecting the one which appears most promising to explore next.

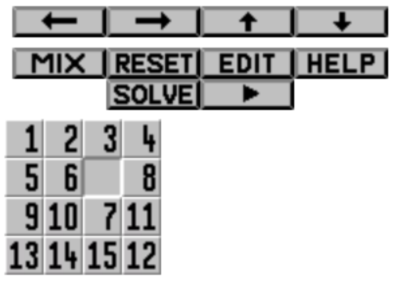
---

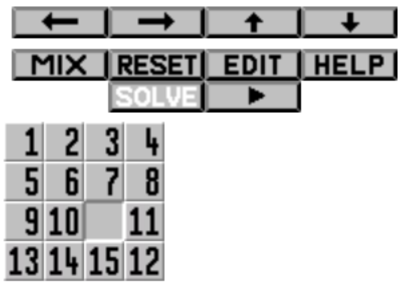
**Problem Statement**
Find the following 15 puzzle problem using branch and bound technique and show each step in detail using the state space tree.

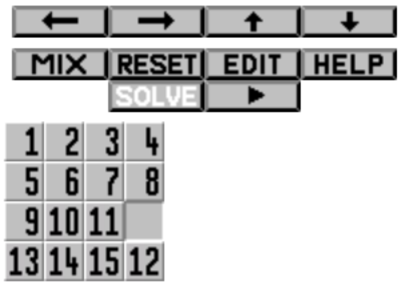


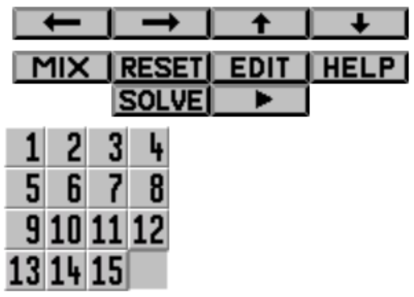
**Also verify your answer by simulating steps of the same question on the following link.**
**http://www.sfu.ca/~jtmulhol/math302/puzzles-15.html**

**Solution**









**Derivation of 15 puzzle problem using Branch and bound:**
Time complexity Analysis

Heuristic Choice: The choice of heuristic significantly affects the time complexity of the branch and bound method. Common heuristics for the 15 puzzle include:

- Number of misplaced tiles: This heuristic counts how many tiles are not in their target position.
- Manhattan distance: This is the sum of the absolute differences between the tiles' current positions and their target positions in terms of rows and columns.

State Space Tree: The state space tree for the 15 puzzle has a very high branching factor, typically close to the number of possible moves from any given state (up to 4: left, right, up, down, excluding the reverse of the last move).

Time Complexity Analysis:

- Worst-case scenario: The worst-case time complexity of solving the 15 puzzle using branch and bound can be exponential in terms of the depth of the state space tree, represented as $O(b^d)$, where $b$ is the branching factor and $d$ is the depth of the solution path. Given that each tile can be involved in multiple moves, and considering the backtracking involved, $d$ can be considerably large.
- Heuristic impact: If a heuristic such as the Manhattan distance is used, it helps prioritize moves that are likely to lead to a solution faster, effectively reducing the practical branching factor by a significant margin. This prioritization can prune large parts of the search tree, making the algorithm faster than the naive exponential estimate.
- Average-case scenario: While still exponential, the average case can be significantly better than the worst case due to effective pruning. The exact complexity depends heavily on the initial configuration of the puzzle and the heuristic effectiveness.

Practical Considerations: In practice, the efficiency of the branch and bound approach for the 15 puzzle depends not only on the heuristic but also on additional optimizations such as:

- Avoiding revisiting states: Implementing a mechanism to prevent the exploration of already visited states can drastically reduce the number of nodes generated.
- Better data structures: Using advanced data structures like a priority queue for the frontier and a hash set for visited states can improve both space and time efficiency.

Conclusion on Complexity: The theoretical worst-case complexity for the branch and bound solution of the 15 puzzle is exponential. However, with a good heuristic and proper implementation details, the practical time complexity can be much more manageable, though it remains a computationally challenging problem especially as the size of the board increases (like moving from an 8-puzzle to a 15-puzzle). Thus, while branch and bound is a powerful method for solving the 15 puzzle, its feasibility for very large puzzles or highly scrambled initial states might be limited without extremely efficient heuristics or additional algorithmic strategies.

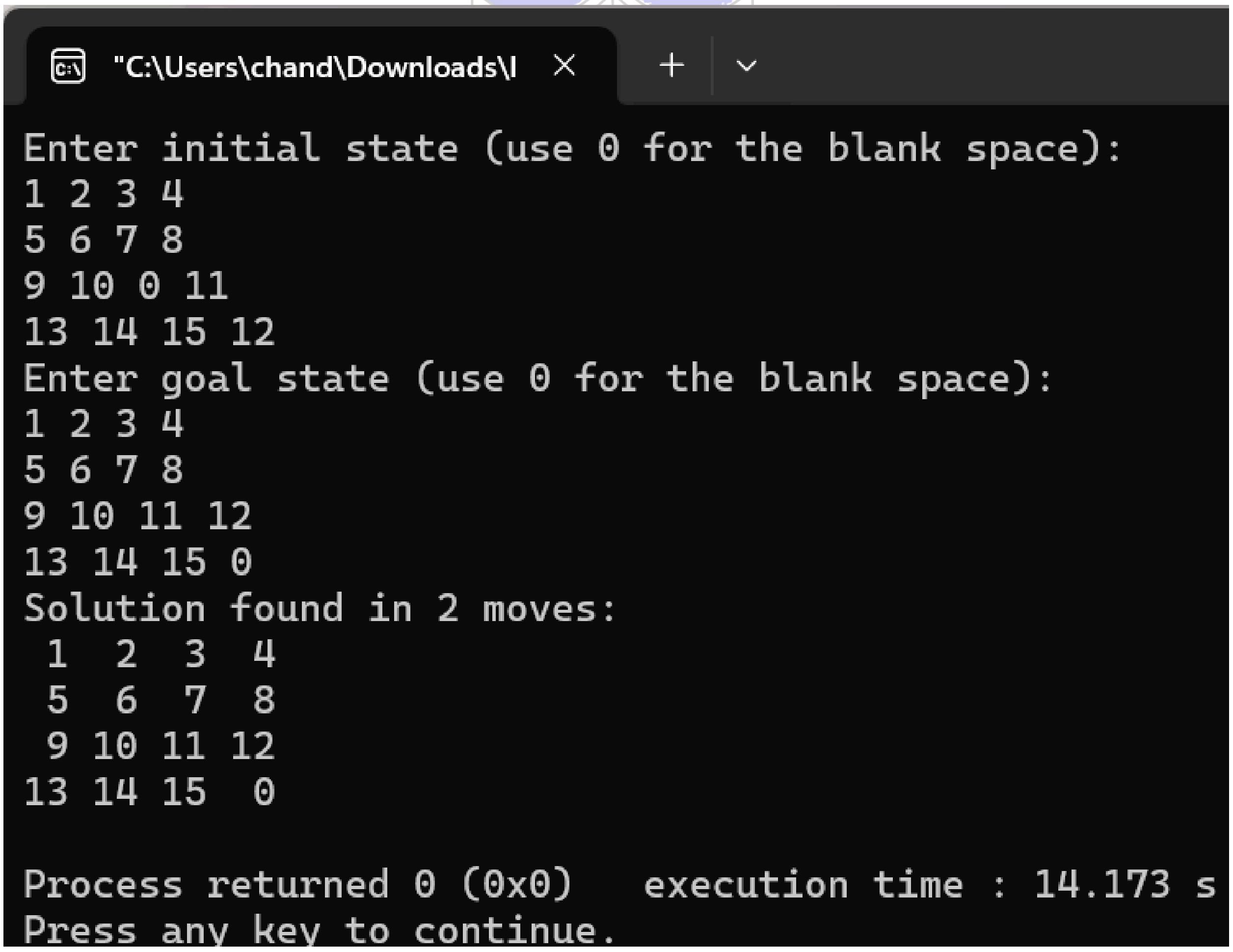
---

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

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define N 4
#define MAX_STATES 1000000

typedef struct {
   int state[N][N];
   int x, y;  // Position of the blank tile
   int cost;  // Cost to reach this state + heuristic cost
   int level; // Number of moves from the start
} Node;

int goal[N][N];

// Function to calculate the number of misplaced tiles
int misplacedTiles(int current[N][N]) {
   int misplaced = 0;
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         if (current[i][j] != 0 && current[i][j] != goal[i][j]) {
            misplaced++;
         }
      }
   }
   return misplaced;
}

// Node comparison for the priority queue
int nodeCompare(const void *n1, const void *n2) {
   int l1 = ((Node*)n1)->cost;
   int l2 = ((Node*)n2)->cost;
```

```
    return (l1 > l2) - (l1 < l2);
}

// Function to print the board
void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%2d ", board[i][j]);
        }
        printf("\n");
    }
}

// Check if (x, y) is a valid board position
bool isSafe(int x, int y) {
    return x >= 0 && x < N && y >= 0 && y < N;
}

// Solves the 15-puzzle problem using branch and bound
void solve(int initial[N][N], int x, int y) {
    // Directions for blank tile movement: down, left, up, right
    int row[] = {1, 0, -1, 0};
    int col[] = {-0, -1, 0, 1};
    char *dir = "DLUR";

    // Priority queue of states (simple implementation)
    Node *pq = (Node *)malloc(MAX_STATES * sizeof(Node));
    int size = 0;

    // Initial state
    Node root;
    memcpy(root.state, initial, sizeof(root.state));
    root.x = x;
    root.y = y;
    root.cost = misplacedTiles(initial);
    root.level = 0;

    // Add root to queue
    pq[size++] = root;

    while (size) {
        // Get the state with the lowest cost
        qsort(pq, size, sizeof(Node), nodeCompare);
        Node min = pq[0];

        // Move the last item to the front (pop from queue)
        pq[0] = pq[--size];

        // If this is the goal state, print the solution
        if (min.cost - min.level == 0) {
            printf("Solution found in %d moves:\n", min.level);
            printBoard(min.state);
            break;
        }

        // Generate children (all possible moves)
        for (int i = 0; i < 4; i++) {
            int newX = min.x + row[i];
            int newY = min.y + col[i];

            if (isSafe(newX, newY)) {
                Node child;
                memcpy(child.state, min.state, sizeof(min.state));
                // Swap values
                child.state[min.x][min.y] = child.state[newX][newY];
                child.state[newX][newY] = 0;
                child.x = newX;
                child.y = newY;
                child.level = min.level + 1;
                child.cost = child.level + misplacedTiles(child.state);

                pq[size++] = child;
            }
        }
    }
    free(pq);
}

int main() {
    int initial[N][N], x, y;

    printf("Enter initial state (use 0 for the blank space):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &initial[i][j]);
            if (initial[i][j] == 0) {
                x = i;
                y = j;
            }
        }
    }

    printf("Enter goal state (use 0 for the blank space):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &goal[i][j]);
        }
    }

    solve(initial, x, y);
    return 0;
}
```

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



"C:\Users\chand\Downloads\I

```
Enter initial state (use 0 for the blank space):
1 2 3 4
5 6 7 8
9 10 0 11
13 14 15 12
Enter goal state (use 0 for the blank space):
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
Solution found in 2 moves:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0

Process returned 0 (0x0)   execution time : 14.173 s
Press any key to continue.
```

**Post Lab Questions:-**

**Explain how to solve the Knapsack problem using branch and bound.**
**Ans:** In the branch and bound approach to the knapsack problem, we explore branches of decisions (whether to include an item in the knapsack or not) and use bounding to avoid exploring branches that cannot possibly outperform the best solution found so far. A common bounding technique is to use the linear relaxation of the knapsack, which gives an upper bound on the optimal value.

---

**Conclusion: (Based on the observations)**
Based on the observations from implementing the 15 puzzle with branch and bound, it's evident that the efficiency of the solution heavily relies on the choice of heuristic. A good heuristic significantly prunes the search space, making the solution feasible even for complex and larger instances of the puzzle. This method provides a structured way to navigate through potential solutions, ensuring that the path with the optimal outcome is prioritized.

---

**Outcome: Implement Backtracking and Branch-and-bound algorithms**

---

**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.

---