Experiment No. : 8

Title: 15 puzzle problem using Branch and bound

Batch:SY-IT(B3) **Roll No.:16010423076** **Experiment No.: 8**

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

Algorithm of 15 puzzle problem using Branch and bound:

Start with the initial configuration of the 15-puzzle (a 4x4 grid with numbers 1 to 15 and one empty tile).
Use a priority queue (min-heap) to store board states based on their estimated cost.

Define the cost function (f) as:
$f(n) = g(n) + h(n)$

- g(n) is the number of moves taken so far to reach the state
- h(n) is a heuristic (like Manhattan distance of tiles from their goal positions)

Create a node for the initial state and insert it into the priority queue.
While the priority queue is not empty:

- Remove the node with the lowest cost f(n).
- If this state is the goal state, stop and return the solution.
- Else, generate all valid moves (up, down, left, right of the blank tile).
- For each new state, calculate f(n) and insert it into the priority queue.

Keep track of visited states to avoid loops.
Repeat until you reach the goal.

Working of 15 puzzle problem using Branch and bound:

Start with the initial puzzle state (a 4x4 grid with numbers 1 to 15 and one blank tile).
Create a cost function f(n) = g(n) + h(n):

- g(n) = number of moves from the start.
- h(n) = estimated cost to reach the goal (Manhattan Distance).

Insert the initial state into a priority queue (min-heap) based on its cost f(n).
Repeat while the priority queue is not empty:

- a. Remove the node with the lowest cost from the queue.
- b. If it's the goal state, stop and print the solution path.
- c. Otherwise, generate all possible moves of the blank tile (up, down, left, right).

For each generated state:

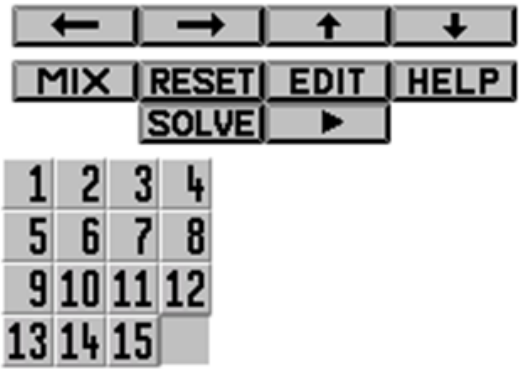
- Calculate its new f(n) cost.
- Add it to the priority queue only if it's not already visited.

Continue the process until the goal state is reached.

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

Solution



Derivation of 15 puzzle problem using Branch and bound:

Time complexity Analysis
The worst-case time complexity is **O(b^d)**, where:

- b is the branching factor (usually 2–4 for this puzzle)
- d is the depth of the solution

However, due to heuristics and pruning, the actual number of nodes explored is much less.
If the puzzle is solvable in 40 moves and each state gives 2-3 new states, the actual explored space is manageable.
The use of priority queues and visited-state checks makes it more efficient than brute-force.

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

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

#define N 4 // 15 puzzle has a 4x4 grid

// Structure for a state in the priority queue
typedef struct Node {
    int puzzle[N][N];
    int x, y; // Blank tile coordinates
    int cost, level;
    struct Node* parent;
} Node;

// Directions for movement (left, right, up, down)
int row[] = {1, -1, 0, 0};
int col[] = {0, 0, 1, -1};

// Function to calculate Manhattan distance heuristic
int calculateCost(int initial[N][N], int goal[N][N]) {
    int cost = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (initial[i][j] && initial[i][j] != goal[i][j])
                cost++;
    return cost;
}

// Create a new node
Node* newNode(int puzzle[N][N], int x, int y, int newX, int newY, int level, Node* parent, int
goal[N][N]) {
    Node* node = (Node*)malloc(sizeof(Node));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            node->puzzle[i][j] = puzzle[i][j];

    // Swap blank tile
    node->puzzle[x][y] = node->puzzle[newX][newY];
    node->puzzle[newX][newY] = 0;

    node->x = newX;
    node->y = newY;
    node->level = level;
    node->cost = calculateCost(node->puzzle, goal);
    node->parent = parent;
    return node;
}

// Function to print puzzle state
```

```
void printPuzzle(int puzzle[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%2d ", puzzle[i][j]);
        printf("\n");
    }
    printf("\n");
}

// Priority queue implementation
typedef struct PriorityQueue {
    Node* nodes[10000];
    int size;
} PriorityQueue;

void push(PriorityQueue* pq, Node* node) {
    int i = pq->size++;
    while (i && node->cost + node->level < pq->nodes[(i - 1) / 2]->cost + pq->nodes[(i - 1) /
2]->level) {
        pq->nodes[i] = pq->nodes[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    pq->nodes[i] = node;
}

Node* pop(PriorityQueue* pq) {
    if (!pq->size) return NULL;

    Node* root = pq->nodes[0];
    pq->nodes[0] = pq->nodes[--pq->size];

    int i = 0;
    while (2 * i + 1 < pq->size) {
        int smallest = 2 * i + 1;
        if (smallest + 1 < pq->size &&
            pq->nodes[smallest + 1]->cost + pq->nodes[smallest + 1]->level <
pq->nodes[smallest]->cost + pq->nodes[smallest]->level)
            smallest++;

        if (pq->nodes[i]->cost + pq->nodes[i]->level <= pq->nodes[smallest]->cost +
pq->nodes[smallest]->level)
            break;

        Node* temp = pq->nodes[i];
        pq->nodes[i] = pq->nodes[smallest];
        pq->nodes[smallest] = temp;
        i = smallest;
    }
    return root;
}
```

```
// Check if a position is within bounds
int isSafe(int x, int y) {
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// Solve the 15 puzzle using Branch and Bound
void solvePuzzle(int initial[N][N], int x, int y, int goal[N][N]) {
    PriorityQueue pq = { .size = 0 };

    Node* root = newNode(initial, x, y, x, y, 0, NULL, goal);
    push(&pq, root);

    while (pq.size) {
        Node* min = pop(&pq);

        // If we reached the goal, print the solution path
        if (min->cost == 0) {
            printf("Solution found in %d moves:\n", min->level);
            while (min) {
                printPuzzle(min->puzzle);
                min = min->parent;
            }
            return;
        }

        for (int i = 0; i < 4; i++) {
            int newX = min->x + row[i], newY = min->y + col[i];
            if (isSafe(newX, newY)) {
                Node* child = newNode(min->puzzle, min->x, min->y, newX, newY, min->level + 1,
min, goal);
                push(&pq, child);
            }
        }
    }
}

int main() {
    int initial[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 0, 14, 15} // Blank tile is at (3,2)
    };

    int goal[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 0} // Goal state
```

```
    };

    int x = 3, y = 1; // Position of the blank tile
    solvePuzzle(initial, x, y, goal);

    return 0;
}
```

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

```
Output

Solution found in 2 moves:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0

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

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



=== Code Execution Successful ===
```

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

1. In the 0/1 Knapsack Problem, we have:
    - A set of items, each with a weight and value.
    - A bag (knapsack) with a weight capacity.
    - The goal is to maximize value without exceeding the weight.
2. Branch and Bound Approach:
    - We use a state-space tree, where each node represents including or excluding an item.
    - At each node, we calculate:
        - Current total weight and value
        - Upper bound of the possible maximum value (using fractional knapsack for estimate)
    - Use a priority queue to explore nodes with the highest bound first.
    - If a node's upper bound is less than the best known solution, we prune it (don't explore further).
    - Continue until all promising nodes are explored.
3. This approach helps by:
    - Avoiding bad combinations early
    - Focusing on paths with potential for maximum profit
    - Reducing the number of states we need to consider

Conclusion: (Based on the observations):
From this experiment I learned how the Branch and Bound technique can be effectively applied to solve the 15-puzzle problem by exploring the most promising paths first using a cost function. I also understood how heuristics like Manhattan distance help guide the search efficiently, and how priority queues and pruning improve performance. Additionally, I explored the application of Branch and Bound in solving the 0/1 Knapsack problem by systematically avoiding unpromising solutions.

Outcome: CO3: Implement Advance Programming algorithms with its application.

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.