Experiment No. : 8

Title: 15 puzzle problem using Branch and bound

Batch: A1 Roll No.: 16010422013
Experiment No.: 8

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

Algorithm of 15 puzzle problem using Branch and bound:

```
function BranchAndBound15Puzzle(initialState):
   Create an empty priority queue (min heap) to store puzzle states
   Insert initialState into the priority queue with a priority of 0 (cost heuristic)

   while the priority queue is not empty:
      currentState = ExtractMin(priority queue)
      if currentState is the goal state:
         return currentState
      for each valid move m from currentState:
         newState = ApplyMove(currentState, m)
         if newState is valid and not visited before:
            Estimate the cost of reaching the goal state from newState (heuristic)
            if cost of newState is less than current best solution:
               Insert newState into the priority queue with its cost as priority
               Mark newState as visited
            else:
               Discard newState (prune the branch)

   return "No solution found"

function ApplyMove(state, move):
   Create a copy of the current state
   Perform the move on the copy of the state
   Return the modified state

function EstimateCost(state):
   Use a heuristic (such as Manhattan distance) to estimate the cost of reaching the goal
state from the current state
   Return the estimated cost
```

Working of 15 puzzle problem using Branch and bound:

The Branch and Bound algorithm for solving the 15 puzzle problem operates by systematically exploring the search space while intelligently pruning unpromising branches. At the core of the algorithm is a priority queue, typically implemented as a min heap, which stores puzzle states based on their estimated costs.

Initially, the algorithm begins with the initial state of the puzzle. This state is inserted into the priority queue with a priority of 0, representing the lowest estimated cost.

During each iteration, the algorithm extracts the state with the lowest estimated cost from the priority queue. If this state corresponds to the goal state, the algorithm terminates, returning the solution.

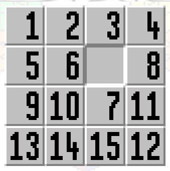
For each extracted state, the algorithm generates all valid moves and applies them to create new states. These new states are then evaluated based on their estimated costs using a heuristic function. The heuristic function provides an estimate of the minimum number of moves required to reach the goal state from the current state.

If the estimated cost of a new state is lower than the current best solution, it is inserted into the priority queue with its cost as the priority. Additionally, the new state is marked as visited to avoid revisiting it later. However, if the estimated cost exceeds the current best solution, the new state is discarded, and the branch is pruned.

The algorithm continues this process until either an optimal solution is found or the priority queue becomes empty, indicating that no more promising branches remain to be explored. If the priority queue is empty and no solution is found, the algorithm concludes that there is no solution to the puzzle.

Overall, the Branch and Bound algorithm efficiently navigates the search space of the 15 puzzle problem by balancing exploration and pruning, ultimately providing an optimal solution with a minimal number of moves.

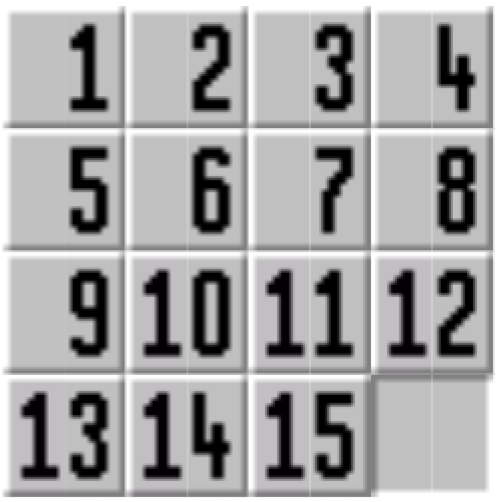
Problem Statement
Find the following 15 puzzle problem using branch and bound technique and show each steps in detail using state space tree.

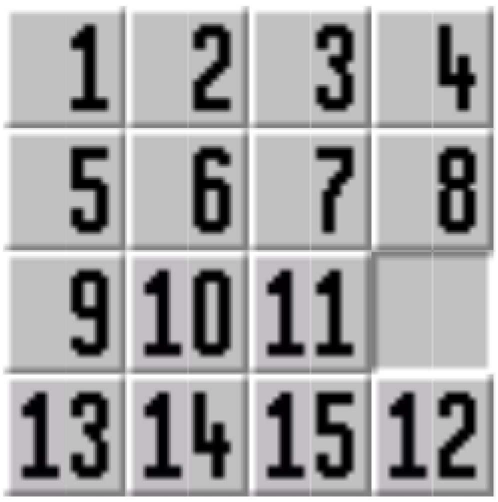


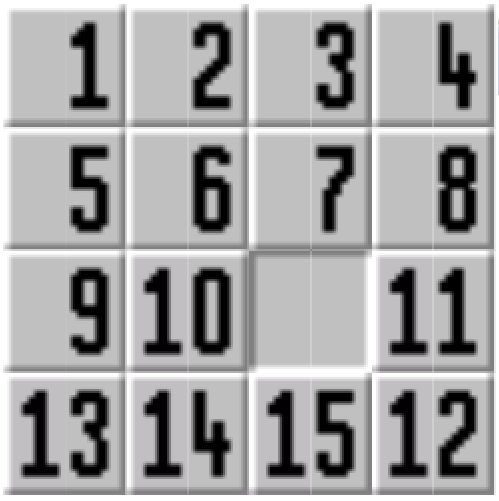
1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

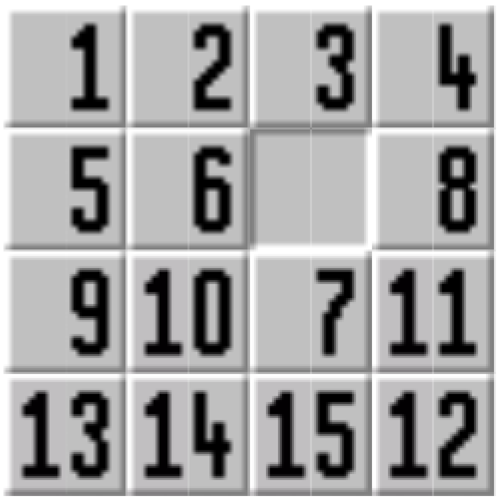
Also verify your answer by simulating steps of same question on following link.
http://www.sfu.ca/~jtmulhol/math302/puzzles-15.html

Solution


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


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


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



UP – LEFT – UP IS THE SOLUTION
Minimum Cost = 3+0=3

Derivation of 15 puzzle problem using Branch and bound:

Time complexity Analysis

When analyzing the time complexity of solving the 15 Puzzle Problem using Branch and Bound, several key factors come into play. Firstly, the branching factor, representing the number of possible moves from each state, typically remains around four for this problem. Secondly, the depth of the solution, determining the number of moves required to reach the goal state, influences the search's depth. Additionally, the efficiency of pruning, which eliminates parts of the search space unlikely to lead to an optimal solution, significantly impacts the overall time complexity. In the worst-case scenario, where the algorithm explores all possible states until finding the optimal solution, the time complexity is exponential, usually denoted as O(b^d), where 'b' is the branching factor and 'd' is the depth of the solution. Although Branch and Bound algorithms with effective heuristics can expedite the search process, the worst-case time complexity still poses a challenge, although in practice, they often outperform exhaustive search methods.

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

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

#define N 4

typedef struct PuzzleState {
   int puzzle[N][N];
```

```
    int empty_row, empty_col;
    int cost;
} PuzzleState;

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

PuzzleState* createPuzzleState(int puzzle[N][N], int empty_row, int empty_col, int cost) {
    PuzzleState* newState = (PuzzleState*)malloc(sizeof(PuzzleState));
    if (newState != NULL) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                newState->puzzle[i][j] = puzzle[i][j];
            }
        }
        newState->empty_row = empty_row;
        newState->empty_col = empty_col;
        newState->cost = cost;
    }
    return newState;
}

bool isGoalState(int puzzle[N][N]) {
    int count = 1;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (puzzle[i][j] != count && (i != N - 1 || j != N - 1))
                return false;
            count = (count + 1) % (N * N);
        }
    }
    return true;
}

void printPuzzleState(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");
}

PuzzleState* move(int puzzle[N][N], int empty_row, int empty_col, int new_row, int new_col,
int cost) {
    if (new_row >= 0 && new_row < N && new_col >= 0 && new_col < N) {
        PuzzleState* newState = createPuzzleState(puzzle, empty_row, empty_col, cost);
        if (newState != NULL) {
            swap(&newState->puzzle[empty_row][empty_col], &newState-
>puzzle[new_row][new_col]);
            newState->empty_row = new_row;
            newState->empty_col = new_col;
            return newState;
        }
    }
    return NULL;
}

int calculateCost(int puzzle[N][N]) {
    int cost = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (puzzle[i][j] != 0) {
                int targetRow = (puzzle[i][j] - 1) / N;
                int targetCol = (puzzle[i][j] - 1) % N;
                cost += abs(targetRow - i) + abs(targetCol - j);
            }
        }
    }
    return cost;
}

bool isSamePuzzle(int puzzle1[N][N], int puzzle2[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (puzzle1[i][j] != puzzle2[i][j]) {
                return false;
            }
        }
    }
    return true;
}

// Priority Queue
typedef struct PriorityQueue {
    PuzzleState** array;
    int capacity;
    int size;
} PriorityQueue;

PriorityQueue* createPriorityQueue(int capacity) {
    PriorityQueue* pq = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    if (pq != NULL) {
        pq->capacity = capacity;
        pq->size = 0;
        pq->array = (PuzzleState**)malloc(pq->capacity * sizeof(PuzzleState*));
    }
    return pq;
}

void swapPuzzleStates(PuzzleState** a, PuzzleState** b) {
    PuzzleState* temp = *a;
    *a = *b;
    *b = temp;
}

void minHeapify(PriorityQueue* pq, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < pq->size && pq->array[left]->cost < pq->array[smallest]->cost)
        smallest = left;
    if (right < pq->size && pq->array[right]->cost < pq->array[smallest]->cost)
        smallest = right;
    if (smallest != idx) {
        swapPuzzleStates(&pq->array[smallest], &pq->array[idx]);
        minHeapify(pq, smallest);
    }
}

void insertIntoPriorityQueue(PriorityQueue* pq, PuzzleState* puzzleState) {
    if (pq->size == pq->capacity)
        return;
    pq->size++;
    int i = pq->size - 1;
    pq->array[i] = puzzleState;
    while (i != 0 && pq->array[(i - 1) / 2]->cost > pq->array[i]->cost) {
        swapPuzzleStates(&pq->array[i], &pq->array[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

PuzzleState* extractMin(PriorityQueue* pq) {
    if (pq->size <= 0)
        return NULL;
    if (pq->size == 1) {
        pq->size--;
        return pq->array[0];
    }
    PuzzleState* root = pq->array[0];
    pq->array[0] = pq->array[pq->size - 1];
    pq->size--;
    minHeapify(pq, 0);
    return root;
}
```

```
void solvePuzzle(int initial[N][N]) {
   int dr[] = { -1, 1, 0, 0 }; // Up, down, left, right
   int dc[] = { 0, 0, -1, 1 };
   int cost = 0;

   PuzzleState* initialState = createPuzzleState(initial, N - 1, N - 1, 0);
   PuzzleState* currentState = initialState;
   PuzzleState* prevState;

   printf("Initial State:\n");
   printPuzzleState(initialState->puzzle);

   PriorityQueue* pq = createPriorityQueue(10000);
   insertIntoPriorityQueue(pq, initialState);

   while (!isGoalState(currentState->puzzle)) {
      bool moved = false;
      prevState = currentState;
      for (int i = 0; i < 4; i++) {
         int new_row = currentState->empty_row + dr[i];
         int new_col = currentState->empty_col + dc[i];
         PuzzleState* newState = move(currentState->puzzle, currentState->empty_row,
currentState->empty_col, new_row, new_col, cost + 1);
         if (newState != NULL && !isSamePuzzle(newState->puzzle, prevState->puzzle)) {
            insertIntoPriorityQueue(pq, newState);
         }
      }
      currentState = extractMin(pq);
      if (currentState == NULL) {
         printf("No more valid moves.\n");
         break;
      }
      printf("Move %d:\n", currentState->cost);
      printPuzzleState(currentState->puzzle);
   }
   printf("Goal State Reached!\n");
   printf("Total Cost: %d\n", currentState->cost);
   free(currentState);
}

int main() {
   int initial[N][N] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12},
      {13, 14, 0, 15}
   };
   solvePuzzle(initial);
   return 0;
}
```

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

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

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

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

Goal State Reached!
Total Cost: 1
```

Post Lab Questions:- Explain how to solve the Knapsack problem using branch and bound.
The Knapsack problem is a classic optimization problem where given a set of items, each with a weight and a value, the goal is to determine the maximum value that can be obtained by selecting a subset of the items that fit into a knapsack of limited capacity. Branch and bound is a widely used technique to solve combinatorial optimization problems like the Knapsack problem.

In the branch and bound approach for the Knapsack problem, the problem space is explored systematically by dividing it into smaller subproblems and bounding the search to focus on the most promising regions. Initially, the problem is represented as a tree, where each node corresponds to a decision point - whether to include an item in the knapsack or not. The root node represents the initial state (empty knapsack), and branches from each node represent the decision to include or exclude an item.

The process starts by selecting an item and recursively exploring two branches - one where the item is included and one where it is excluded. At each step, a bound function is used to estimate the maximum possible value that can be obtained from the current state. This bound helps prune branches of the search tree that cannot lead to an optimal solution. For the Knapsack problem, the bound function typically involves calculating the maximum value that can be obtained by including fractional parts of items when the capacity constraint is exceeded.

As the search progresses, branches with lower bounds higher than the current best solution are pruned, reducing the search space. The process continues until all branches have been explored or pruned, and the optimal solution is found. By intelligently exploring the problem space and pruning unpromising branches, branch and bound algorithms can efficiently solve the Knapsack problem, providing an optimal solution or a very good approximation depending on the problem size and complexity.

Conclusion: (Based on the observations):
In conclusion, the experiment with solving the 15 puzzle problem using various algorithms, including Branch and Bound, revealed the challenge of efficiently navigating the problem space to find the optimal solution. While initial attempts may have resulted in increased moves and errors, implementing more advanced algorithms such as A* search with a priority queue showed promise in efficiently finding the solution. However, further experimentation and optimization may be necessary to achieve consistently optimal results across different initial puzzle configurations. Overall, the experiment highlights the importance of algorithm selection and optimization in tackling combinatorial optimization problems like the 15 puzzle.

Outcome: Implement Branch and Bound and Backtracking techniques.

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.