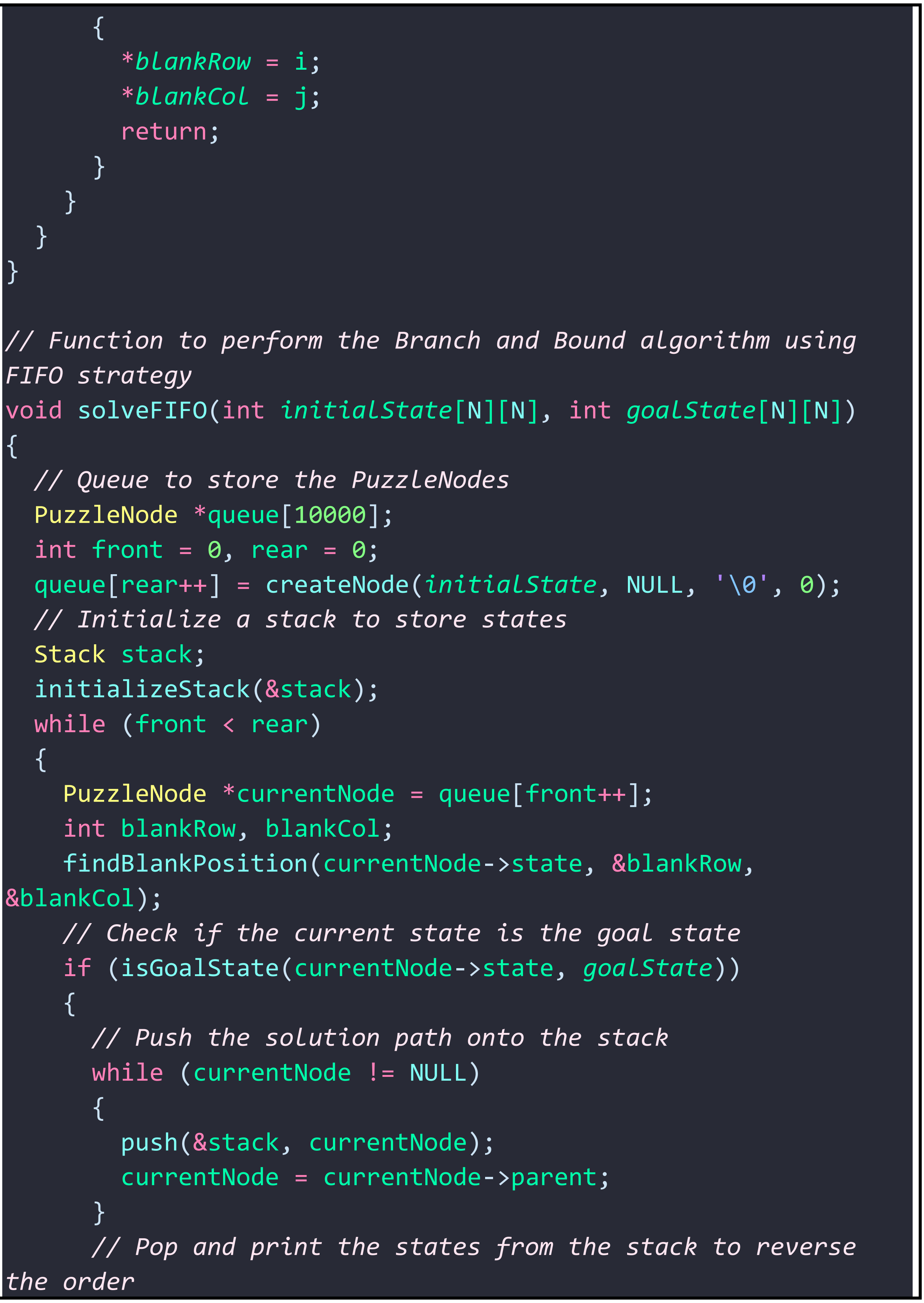
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to Mumbai University)
Department Of Computer Engineering

Name	Manish Shashikant Jadhav
UID	2023301005
Subject	Design and Analysis of Algorithms (DAA)
Experiment No.	9
Aim	To implement Branch and Bound (FIFO and LC).

Code:

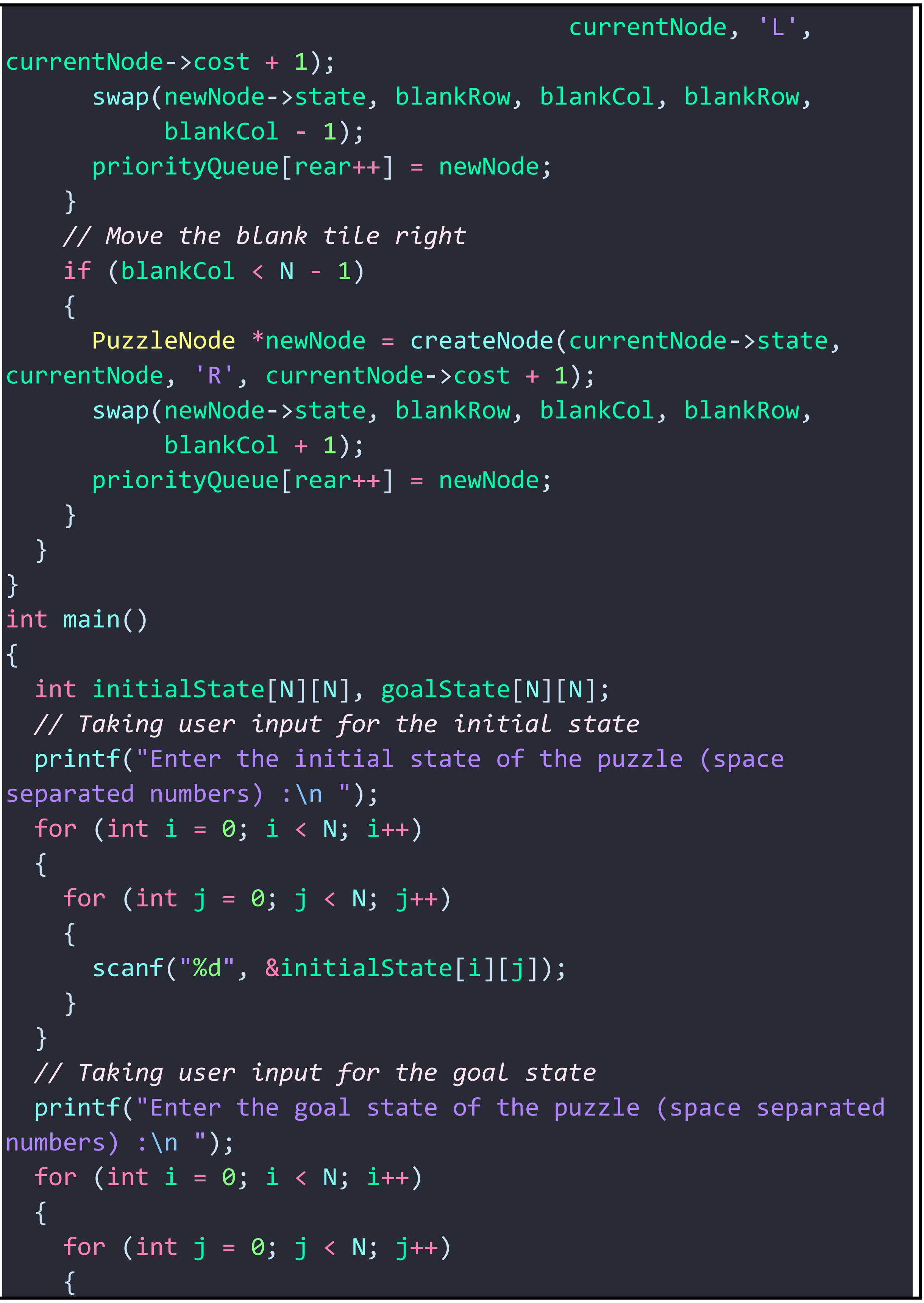
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define N 4
typedef struct PuzzleNode
{
  int state[N][N];
  struct PuzzleNode *parent;
  char action;
  int cost;
} PuzzleNode;
// Define a stack structure for storing states
typedef struct Stack
{
  PuzzleNode *items[10000];
  int top;
} Stack;
// Function to initialize the stack
void initializeStack(Stack *stack)
{
  stack->top = -1;
}
// Function to push an item onto the stack
void push(Stack *stack, PuzzleNode *item)
{
  stack->items[++stack->top] = item;
}
// Function to pop an item from the stack
```

```
PuzzleNode *pop(Stack *stack)
{
  return stack->items[stack->top--];
}
// Function to create a new PuzzleNode
PuzzleNode *createNode(int state[N][N], PuzzleNode *parent,
char action, int cost)
{
  PuzzleNode *newNode = (PuzzleNode
*)malloc(sizeof(PuzzleNode));
  if (newNode == NULL)
  {
    printf("Memory allocation failed.\n");
    exit(1);
  }
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      newNode->state[i][j] = state[i][j];
    }
  }
  newNode->parent = parent;
  newNode->action = action;
  newNode->cost = cost;
  return newNode;
}
// Function to check if the current state is the goal state
bool isGoalState(int state[N][N], int goalState[N][N])
{
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      if (state[i][j] != goalState[i][j])
      {
        return false;
      }
    }
  }
  return true;
}
// Function to print the state of the puzzle
void printState(int state[N][N])
{
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      printf("%d ", state[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}

// Function to swap two tiles in the state matrix
void swap(int state[N][N], int i1, int j1, int i2, int j2)
{
  int temp = state[i1][j1];
  state[i1][j1] = state[i2][j2];
  state[i2][j2] = temp;
}

// Function to find the position of the blank tile in the
state matrix
void findBlankPosition(int state[N][N], int *blankRow, int
*blankCol)
{
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      if (state[i][j] == 0)
```

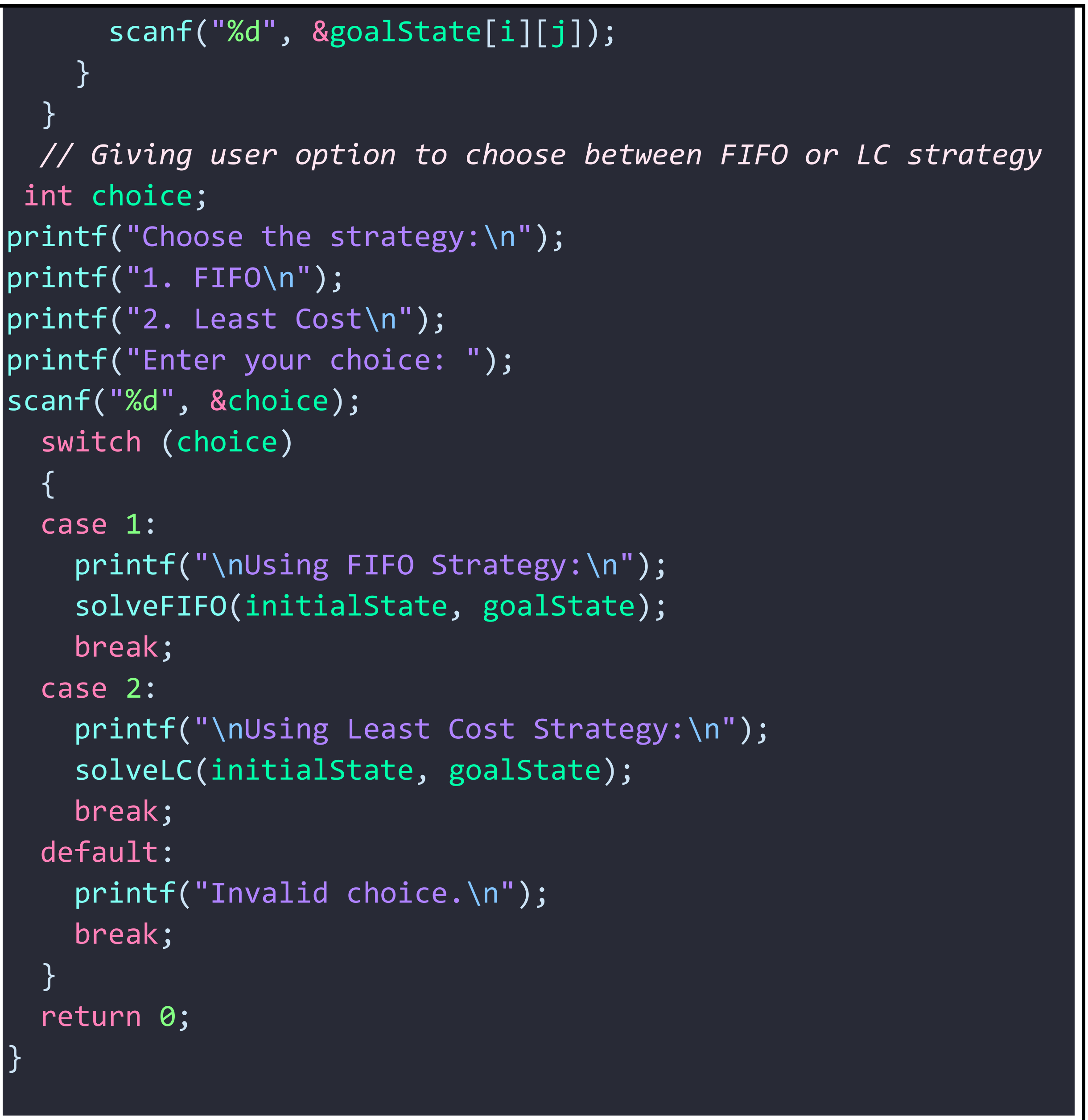


```
      {
        *blankRow = i;
        *blankCol = j;
        return;
      }
    }
  }
}

// Function to perform the Branch and Bound algorithm using
FIFO strategy
void solveFIFO(int initialState[N][N], int goalState[N][N])
{
  // Queue to store the PuzzleNodes
  PuzzleNode *queue[10000];
  int front = 0, rear = 0;
  queue[rear++] = createNode(initialState, NULL, '\0', 0);
  // Initialize a stack to store states
  Stack stack;
  initializeStack(&stack);
  while (front < rear)
  {
    PuzzleNode *currentNode = queue[front++];
    int blankRow, blankCol;
    findBlankPosition(currentNode->state, &blankRow,
&blankCol);
    // Check if the current state is the goal state
    if (isGoalState(currentNode->state, goalState))
    {
      // Push the solution path onto the stack
      while (currentNode != NULL)
      {
        push(&stack, currentNode);
        currentNode = currentNode->parent;
      }
      // Pop and print the states from the stack to reverse
the order
```

```
      while (stack.top != -1)
      {
        currentNode = pop(&stack);
        printState(currentNode->state);
      }
      return;
    }
    // Move the blank tile up
    if (blankRow > 0)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
                                       currentNode, 'U',
currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow - 1,
           blankCol);
      queue[rear++] = newNode;
    }
    // Move the blank tile down
    if (blankRow < N - 1)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
                                       currentNode, 'D',
currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow + 1,
           blankCol);
      queue[rear++] = newNode;
    }
    // Move the blank tile left
    if (blankCol > 0)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
                                       currentNode, 'L',
currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow,
           blankCol - 1);
      queue[rear++] = newNode;
    }
    // Move the blank tile right
    if (blankCol < N - 1)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
currentNode, 'R', currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow,
           blankCol + 1);
      queue[rear++] = newNode;
    }
  }
}
// Function to perform the Branch and Bound algorithm using
Least Cost strategy

void solveLC(int initialState[N][N], int goalState[N][N])
{
  // Priority queue to store the PuzzleNodes based on cost
  PuzzleNode *priorityQueue[10000];
  int front = 0, rear = 0;
  priorityQueue[rear++] = createNode(initialState, NULL, '\0',
0);

  // Initialize a stack to store states
  Stack stack;
  initializeStack(&stack);

  while (front < rear)
  {
    PuzzleNode *currentNode = priorityQueue[front++];
    int blankRow, blankCol;
    findBlankPosition(currentNode->state, &blankRow,
&blankCol);
    // Check if the current state is the goal state
    if (isGoalState(currentNode->state, goalState))
    {
      // Push the solution path onto the stack
      while (currentNode != NULL)
      {
        push(&stack, currentNode);
        currentNode = currentNode->parent;
      }

      // Pop and print the states from the stack to reverse
the order
      while (stack.top != -1)
      {
        currentNode = pop(&stack);
        printState(currentNode->state);
      }
      return;
    }
    // Move the blank tile up
    if (blankRow > 0)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
currentNode, 'U', currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow - 1,
           blankCol);
      priorityQueue[rear++] = newNode;
    }
    // Move the blank tile down
    if (blankRow < N - 1)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
currentNode, 'D', currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow + 1,
           blankCol);
      priorityQueue[rear++] = newNode;
    }
    // Move the blank tile left
    if (blankCol > 0)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
```



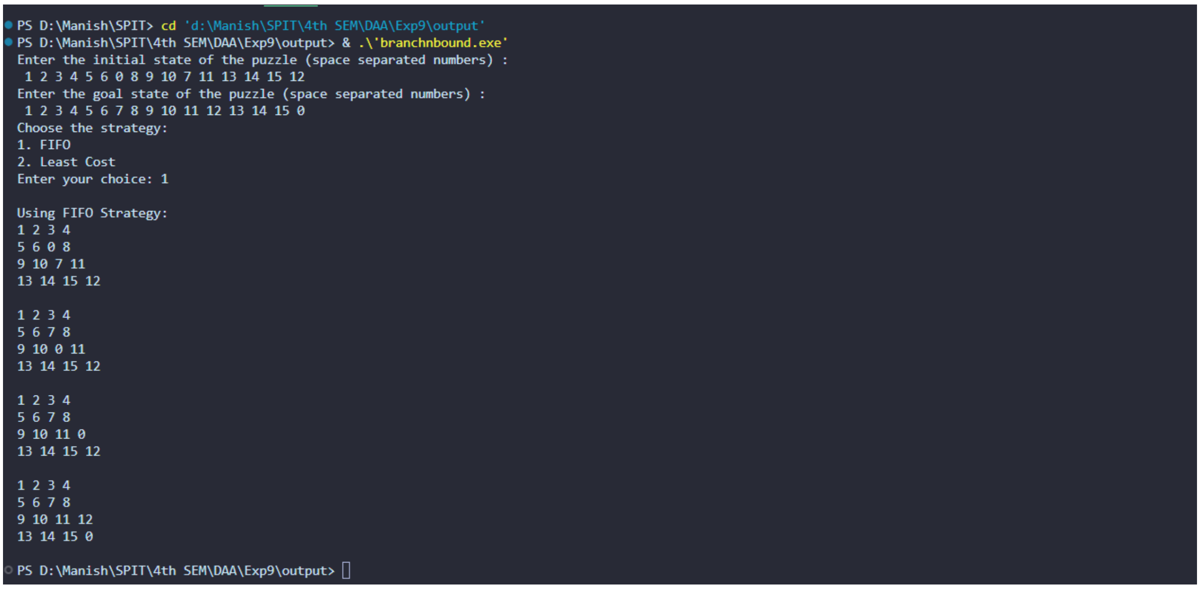
```
                                        currentNode, 'L',
currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow,
           blankCol - 1);
      priorityQueue[rear++] = newNode;
    }
    // Move the blank tile right
    if (blankCol < N - 1)
    {
      PuzzleNode *newNode = createNode(currentNode->state,
currentNode, 'R', currentNode->cost + 1);
      swap(newNode->state, blankRow, blankCol, blankRow,
           blankCol + 1);
      priorityQueue[rear++] = newNode;
    }
  }
}
int main()
{
  int initialState[N][N], goalState[N][N];
  // Taking user input for the initial state
  printf("Enter the initial state of the puzzle (space
separated numbers) :\n ");
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      scanf("%d", &initialState[i][j]);
    }
  }
  // Taking user input for the goal state
  printf("Enter the goal state of the puzzle (space separated
numbers) :\n ");
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
```



```
      scanf("%d", &goalState[i][j]);
    }
  }
  // Giving user option to choose between FIFO or LC strategy
 int choice;
printf("Choose the strategy:\n");
printf("1. FIFO\n");
printf("2. Least Cost\n");
printf("Enter your choice: ");
scanf("%d", &choice);
  switch (choice)
  {
  case 1:
    printf("\nUsing FIFO Strategy:\n");
    solveFIFO(initialState, goalState);
    break;
  case 2:
    printf("\nUsing Least Cost Strategy:\n");
    solveLC(initialState, goalState);
    break;
  default:
    printf("Invalid choice.\n");
    break;
  }
  return 0;
}
```

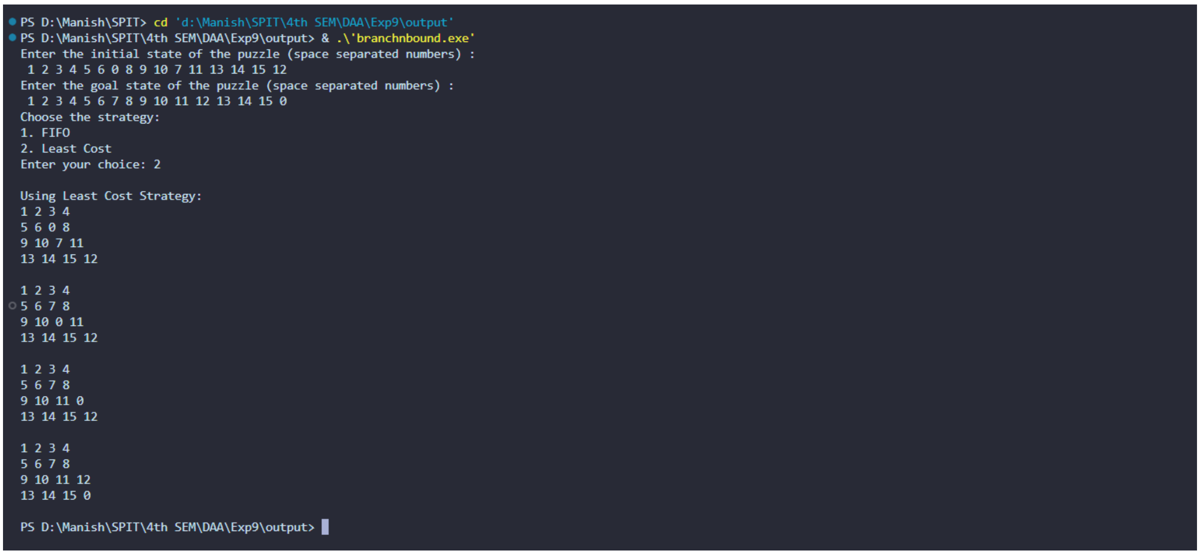
Output

1. FIFO



```
PS D:\Manish\SPIT> cd 'd:\Manish\SPIT\4th SEM\DAA\Exp9\output'
PS D:\Manish\SPIT\4th SEM\DAA\Exp9\output> & .\'branchnbound.exe'
Enter the initial state of the puzzle (space separated numbers) :
 1 2 3 4 5 6 0 8 9 10 7 11 13 14 15 12
Enter the goal state of the puzzle (space separated numbers) :
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
Choose the strategy:
1. FIFO
2. Least Cost
Enter your choice: 1

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

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

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

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

PS D:\Manish\SPIT\4th SEM\DAA\Exp9\output> 
```

2. LC



```
PS D:\Manish\SPIT> cd 'd:\Manish\SPIT\4th SEM\DAA\Exp9\output'
PS D:\Manish\SPIT\4th SEM\DAA\Exp9\output> & .\'branchnbound.exe'
Enter the initial state of the puzzle (space separated numbers) :
 1 2 3 4 5 6 0 8 9 10 7 11 13 14 15 12
Enter the goal state of the puzzle (space separated numbers) :
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
Choose the strategy:
1. FIFO
2. Least Cost
Enter your choice: 2

Using Least Cost Strategy:
1 2 3 4
5 6 0 8
9 10 7 11
13 14 15 12

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

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

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

PS D:\Manish\SPIT\4th SEM\DAA\Exp9\output> 
```

Conclusion

Hence, by completing this experiment I came to know about implementation of FIFO and LC using Branch and Bound.