AISC

Class: B.E COMP

Experiment 1a: Specify problem formulation

Learning Objective:

Basic Experiments

Specify problem formulation for an AI problem and Implement the same.

Tools: Python

Theory:

Problem Formulation

• Goal formulation World states with certain properties

• Definition of the state space (important: only the relevant aspects abstraction

• Definition of the actions that can change the world state

• Definition of the problem type, which is dependent on the knowledge of the world states and actions states in the search space

• Determination of the search cost (search costs, offline costs) and the execution costs (path costs, online costs)

Note: The type of problem formulation can have a big influence on the difficulty of finding a solution.

Example search problem: 8-puzzle Formulate goal• • Pieces to end up in order as shown… Formulate search problem• • States: configurations of the puzzle (9! configurations) • Actions: Move one of the movable pieces (≤4 possible) • Performance measure: minimize total moves Find solution• • Sequence of pieces moved: 3,1,6,3,1,…

Holiday in Romania II On holiday in Romania; currently in Arad• • Flight leaves tomorrow from Bucharest Formulate goal• • Be in Bucharest Formulate search problem• • States: various cities • Actions: drive between cities • Performance measure: minimize distance Find solution• • Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest,

More formally, a problem is defined by:

1. A set of states S S∈

2. An initial state si

3. A set of actions A s Actions(s) = the set of actions that can be executed in s,∀— that are applicable in s. sr→Actions(s) Result(s, a) ∊ a∀ s∀

4. Transition Model: —sr is called a successor of s —{si Successors(si∪})* = state space

5. Goal test Goal(s) — Can be implicit, e.g. checkmate(x) — s is a goal state if Goal(s) is true

6. Path cost (additive) —e.g. sum of distances, number of actions executed, … —c(x,a,y) is the step cost, assumed ≥ 0 – (where action a goes from state x to state y)

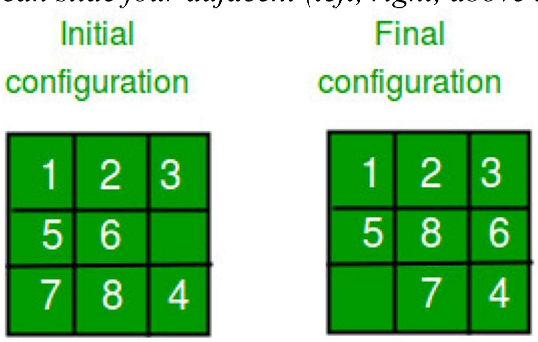
Solution A solution is a sequence of actions from the initial state to a goal state. Optimal Solution: A solution is optimal if no solution has a lower path cost.

8 puzzle Problem using Branch And Bound

We have introduced Branch and Bound and discussed 0/1 Knapsack problem in below posts.

- Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)
- Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)

In this puzzle solution of 8 puzzle problem is discussed.
Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.



Complete Algorithm:

```
/* Algorithm LCSearch uses c(x) to find an answer node
 *LCSearch uses Least() and Add() to maintain the list
   of live nodes
 *Least() finds a live node with least c(x), deletes
   it from the list and returns it
 * Add(x) adds x to the list of live nodes
 * Implement list of live nodes as a min-heap */
```

```
struct list_node
{
  list_node *next;

  // Helps in tracing path when answer is found
  list_node *parent;
  float cost;
}

algorithm LCSearch(list_node *t)
{
  // Search t for an answer node
  // Input: Root node of tree t
  // Output: Path from answer node to root
  if (*t is an answer node)
  {
     print(*t);
     return;
  }

  E = t; // E-node

  Initialize the list of live nodes to be empty;
  while (true)
  {
    for each child x of E
    {
       if x is an answer node
       {
```

```
            print the path from x to t;
            return;
          }
          Add (x); // Add x to list of live nodes;
          x->parent = E; // Pointer for path to root
      }

      if there are no more live nodes
      {
        print ("No answer node");
        return;
      }

      // Find a live node with least estimated cost
      E = Least();

      // The found node is deleted from the list of
      // live nodes
   }
}
// Program to print path from root node to destination node
// for N*N -1 puzzle algorithm using Branch and Bound
// The solution assumes that instance of puzzle is solvable
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
   // stores the parent node of the current node
   // helps in tracing path when the answer is found
   Node* parent;
```

```
    // stores matrix
    int mat[N][N];

    // stores blank tile coordinates
    int x, y;

    // stores the number of misplaced tiles
    int cost;

    // stores the number of moves so far
    int level;
};

// Function to print N x N matrix
int printMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

// Function to allocate a new node
Node* newNode(int mat[N][N], int x, int y, int newX,
              int newY, int level, Node* parent)
{
    Node* node = new Node;

    // set pointer for path to root
    node->parent = parent;

    // copy data from parent node to current node
    memcpy(node->mat, mat, sizeof node->mat);

    // move tile by 1 position
    swap(node->mat[x][y], node->mat[newX][newY]);

    // set number of misplaced tiles
    node->cost = INT_MAX;

    // set number of moves so far
```

```
    node->level = level;

    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;

    return node;
}

// botton, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };

// Function to calculate the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])
{
    int count = 0;
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        if (initial[i][j] && initial[i][j] != final[i][j])
           count++;
    return count;
}

// Function to check if (x, y) is a valid matrix cordinate
int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// print path from root node to destination node
void printPath(Node* root)
{
    if (root == NULL)
        return;
    printPath(root->parent);
    printMatrix(root->mat);

    printf("\n");
}

// Comparison object to be used to order the heap
struct comp
```

```
{
   bool operator()(const Node* lhs, const Node* rhs) const
   {
      return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
   }
};

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
        int final[N][N])
{
   // Create a priority queue to store live nodes of
   // search tree;
   priority_queue<Node*, std::vector<Node*>, comp> pq;

   // create a root node and calculate its cost
   Node* root = newNode(initial, x, y, x, y, 0, NULL);
   root->cost = calculateCost(initial, final);

   // Add root to list of live nodes;
   pq.push(root);

   // Finds a live node with least cost,
   // add its childrens to list of live nodes and
   // finally deletes it from the list.
   while (!pq.empty())
   {
      // Find a live node with least estimated cost
      Node* min = pq.top();

      // The found node is deleted from the list of
      // live nodes
      pq.pop();

      // if min is an answer node
      if (min->cost == 0)
      {
         // print the path from root to destination;
         printPath(min);
         return;
      }
```

```
        // do for each child of min
        // max 4 children for a node
        for (int i = 0; i < 4; i++)
        {
            if (isSafe(min->x + row[i], min->y + col[i]))
            {
                // create a child node and calculate
                // its cost
                Node* child = newNode(min->mat, min->x,
                              min->y, min->x + row[i],
                              min->y + col[i],
                              min->level + 1, min);
                child->cost = calculateCost(child->mat, final);

                // Add child to list of live nodes
                pq.push(child);
            }
        }
    }
}

// Driver code
int main()
{
    // Initial configuration
    // Value 0 is used for empty space
    int initial[N][N] =
    {
        {1, 2, 3},
        {5, 6, 0},
        {7, 8, 4}
    };

    // Solvable Final configuration
    // Value 0 is used for empty space
    int final[N][N] =
    {
        {1, 2, 3},
        {5, 8, 6},
        {0, 7, 4}
    };

    // Blank tile coordinates in initial
    // configuration
```

```
    int x = 1, y = 2;

    solve(initial, x, y, final);

    return 0;
}
```

Output :

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4
```

Learning Outcomes: Students should have the ability to

LO1: Understand the problem formulation

Course Outcomes: Upon completion of the course students will be able to understand problem formulation in IIS.

Conclusion: Thus, we have successfully understood the concept of problem formulation Viva Questions:

Ques.1 What do you understand by Problem Formulation?

Ques. 2. Explain the basic steps in problem formulation?

Ques. 3. Write types of problem discussed in detail.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Source Code:

```
#Dijkstra's Algorithm in Python
import sys

# Providing the graph
vertices = [[0, 0, 1, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 1, 0],
            [1, 1, 0, 1, 1, 0, 0],
            [1, 0, 1, 0, 0, 0, 1],
            [0, 0, 1, 0, 0, 1, 0],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],
         [0, 0, 2, 0, 0, 3, 0],
         [1, 2, 0, 1, 3, 0, 0],
         [2, 0, 1, 0, 0, 0, 1],
         [0, 0, 3, 0, 0, 2, 0],
         [0, 3, 0, 0, 2, 0, 1],
         [0, 0, 0, 1, 0, 1, 0]]

# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v = -10
    for index in range(num_of_vertices):
        if visited_and_distance[index][0] == 0 \
            and (v < 0 or visited_and_distance[index][1] <=
                 visited_and_distance[v][1]):
            v = index
    return v


num_of_vertices = len(vertices[0])

visited_and_distance = [[0, 0]]
for i in range(num_of_vertices-1):
    visited_and_distance.append([0, sys.maxsize])

for vertex in range(num_of_vertices):

    # Find next vertex to be visited
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):

        # Updating new distances
        if vertices[to_visit][neighbor_index] == 1 and \
                visited_and_distance[neighbor_index][0] == 0:
            new_distance = visited_and_distance[to_visit][1] \
                + edges[to_visit][neighbor_index]
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance
```

```
        visited_and_distance[to_visit][0] = 1

i = 0

# Printing the distance
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i),
          " from source vertex: ", distance[1])
    i = i + 1
```

OUTPUT:

```
Distance of  a  from source vertex:  0
Distance of  b  from source vertex:  3
Distance of  c  from source vertex:  1
Distance of  d  from source vertex:  2
Distance of  e  from source vertex:  4
Distance of  f  from source vertex:  4
Distance of  g  from source vertex:  3
```