Dr. D. Y. Patil Institute of Technology
Pimpri Pune-411018

Department of Artificial Intelligence and Data Science

Laboratory Manual

Savitribai Phule Pune University

Second Year of Artificial Intelligence and Data Science (2020 Course)

Subject Code: 317523

Software Laboratory-1 (Artificial Intelligence)

Prepared by

1. **Vanita Kshirsagar**
2. **Apurva Khsandelkar**

Academic Year

2023-2024

Dr. D. Y. Patil Institute of Technology
Pimpri Pune-411018

Department of Artificial Intelligence and Data Science

Vision of the Institute

- Empowerment through knowledge

Mission of the Institute

- Developing human potential to serve the Nation
- Dedicated efforts for quality education.
- Yearning to promote research and development.
- Persistent endeavor to imbibe moral and professional ethics.
- Inculcating the concept of emotional intelligence.
- Emphasizing extension work to reach out to the society.
- Treading the path to meet the future challenges.

Dr. D. Y. Patil Institute of Technology
Pimpri Pune-411018

Department of Artificial Intelligence and Data Science

Vision of the Department

- To produce globally competent engineers in the field of Artificial Intelligence and Data Science with human values

Mission of the Department

- To develop students with a sound understanding in the area of Artificial Intelligence, Machine Learning and Data Science.
- To enable students to become innovators, researchers, entrepreneurs and leaders globally.
- Equip the department with new advancement in high performance equipments and software to carrying out research in emerging technologies in AI and DS.
- To meet the pressing demands of the nation in the areas of Artificial Intelligence and Data Science.

Software Laboratory-1 (Artificial Intelligence)

217522

Teaching Scheme	Credit Scheme	Examination Scheme and Marks
Practical: 04 Hours/Week	02	Term Work: 25 Marks Practical: 25 Marks

Course Objectives:

To understand basic problem solving techniques and strategies to implement the algorithm, The Problem solving is implemented using the concept of search problems an programming skill using python.

Course Outcomes:

On completion of the course, learner will be able to–

CO1. To learn and apply various search strategies for AI

CO2. To Formalize and implement constraints in search problems

CO3. To develop problem solving skills

Guidelines for Instructor's Manual

The instructor‘s manual is to be developed as a reference and hands-on resource. It should include prologue (about University/program/ institute/ department/foreword/ preface), curriculum of the course, conduction and Assessment guidelines, topics under consideration, concept, objectives, outcomes, set of typical applications/assignments/ guidelines, and references.

Guidelines for Student's Laboratory Journal

The laboratory assignments are to be submitted by student in the form of journal. Journal consists of Certificate, table of contents, and handwritten write-up of each assignment (Title, Date of Completion, Objectives, Problem Statement, Software and Hardware requirements, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, test cases, Test Data Set(if applicable), mathematical model (if applicable), conclusion/analysis. Program codes with sample output of all performed assignments are to be submitted as softcopy. As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal must be avoided. Use of DVD containing students programs maintained by Laboratory In charge is highly encouraged. For reference one or two journals may be maintained with program prints in the Laboratory.

Guidelines for Laboratory / Term Work Assessment

Continuous assessment of laboratory work should be based on overall performance of Laboratory assignments by a student. Each Laboratory assignment assessment will assign grade/marks based on parameters, such as timely completion, performance, innovation, efficient codes, and punctuality.

Guidelines for Practical Examination

Problem statements must be decided jointly by the internal examiner and external examiner. During practical assessment, maximum weight age should be given to satisfactory implementation of the problem statement. Relevant questions may be asked at the time of evaluation to test the student's understanding of the fundamentals, effective and efficient implementation. This will encourage, transparent evaluation and fair approach, and hence will not create any uncertainty or doubt in the minds of the students. So, adhering to these principles will consummate our team efforts to the promising start of student's academics.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. Use of open source software is encouraged. Based on the concepts learned. Instructor may also set one assignment or mini-project that is suitable to AI & DS branch beyond the scope of the syllabus. Operating System recommended :- 64-bit Open source Linux or its derivative Programming tools recommended: - MYSQL/Oracle, MongoDB, ERD plus, ER Win

Practical No.	Assignment to be covered
3 Assignments from Group A using python	
1	Implement depth first search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.
2	Implement A star (A*) Algorithm for any game search problem.
3	Implement Alpha-Beta Tree search for any game search problem.
4	Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.
5	Implement Greedy search algorithm for any of the following application: • Selection Sort • Minimum Spanning Tree • Single-Source Shortest Path Problem • Job Scheduling Problem • Prim's Minimal Spanning Tree Algorithm • Kruskal's Minimal Spanning Tree Algorithm • Dijkstra's Minimal Spanning Tree Algorithm
6	6. Develop an elementary chatbot for any suitable customer interaction application.
7	Mini Project: Implement any one of the following Expert System • Information management • Hospitals and medical facilities • Help desks management • Employee performance evaluation • Stock market trading • Airline scheduling and cargo schedule

Installation

Title: Introduction of Python Programming

Theory: Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Audience

This tutorial is designed for software programmers who need to learn Python programming language from scratch.

Prerequisites

You should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

Execute Python Programs

For most of the examples given in this tutorial you will find **Try it** option, so just make use of it and enjoy your learning.

Try following example using **Try it** option available at the top right corner of the below sample code box −

```
#!/usr/bin/python

print "Hello, Python!"
```

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms −

Unix and Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.

- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- run ./configure script
- make
- make install

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix −

- **In the csh shell** − type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux)** − type export ATH="$PATH:/usr/local/bin/python" and press Enter.
- **In the sh or ksh shell** − type PATH="$PATH:/usr/local/bin/python" and press Enter.
- **Note** − /usr/local/bin/python is the path of the Python directory

Running Python

There are three different ways to start Python −

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

```
$python # Unix/Linux
or
python% # Unix/Linux
or
C:> python # Windows/DOS
```

Assignment 1

Title: Implementing Depth First Search and Breadth First Search Algorithms

Problem Statement: Implement Depth First Search (DFS) and Breadth First Search (BFS) algorithms for traversing an undirected graph or tree data structure. Compare and analyze their performance and traversal orders.

Objective: The objective of this lab assignment is to gain a practical understanding of Depth First Search and Breadth First Search algorithms, their implementation, and their differences in terms of traversal order and performance.

Outcome: By completing this assignment, students will:

- Understand the concepts of DFS and BFS algorithms.
- Gain hands-on experience in implementing DFS and BFS algorithms.
- Compare and contrast the traversal orders produced by DFS and BFS.
- Analyze the time and space complexities of both algorithms.

Input: An undirected graph or tree represented using adjacency list or matrix.

Output: The traversal orders and paths obtained using Depth First Search and Breadth First Search algorithms.

Theory and Algorithms:

1. **Depth First Search (DFS) Algorithm:** DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

   Recursive DFS Algorithm:

   vbnetCopy code

   function dfs(node, visited): if node is not visited: mark node as visited process(node) for each neighbor in node's neighbors: dfs(neighbor, visited)

2. **Breadth First Search (BFS) Algorithm:** BFS is a graph traversal algorithm that explores the neighbor vertices at the present depth before moving on to vertices at the next depth level. It can be implemented using a queue.

   BFS Algorithm:

   sqlCopy code

   function bfs(start): create an empty queue q enqueue start into q mark start as visited while q is not empty: current = dequeue q process(current) for each neighbor in current's neighbors: if neighbor is not visited: mark neighbor as visited enqueue neighbor into q

Flow Chart: (Note: The flowchart can be drawn using various tools such as Lucidchart, Microsoft Visio, or any other suitable diagramming software.)

[Flowchart Image Placeholder]

Lab Procedure:

1. Prepare the graph or tree data structure for testing the algorithms.
2. Implement the DFS algorithm using the recursive approach.
3. Implement the BFS algorithm using a queue.
4. Apply both algorithms to the prepared graph or tree data structure.
5. Record the traversal orders and paths obtained from both algorithms.
6. Analyze and compare the traversal orders.
7. Calculate the time and space complexities of DFS and BFS.
8. Document your observations and analysis in the lab report.
9.

Conclusion: In this lab assignment, we successfully implemented the Depth First Search (DFS) and Breadth First Search (BFS) algorithms for traversing undirected graphs or tree data structures. We observed that DFS explores as far as possible before backtracking, while BFS explores neighbor vertices at each depth level. The choice between these algorithms depends on the application's requirements. Additionally, we analyzed their time and space complexities and compared their traversal orders. This lab provided valuable insights into graph traversal algorithms and their practical implementations.

Assignment 2

Title: Assignment on Heuristic Search Techniques: Implement Best first search (Best-Solution but not always optimal) and A* algorithm (Always gives optimal solution)

Theory: Difficulty: we want to still be able to generate the path with minimum cost

A* is an algorithm that:

Uses heuristic to guide search

While ensuring that it will compute a path with minimum cost

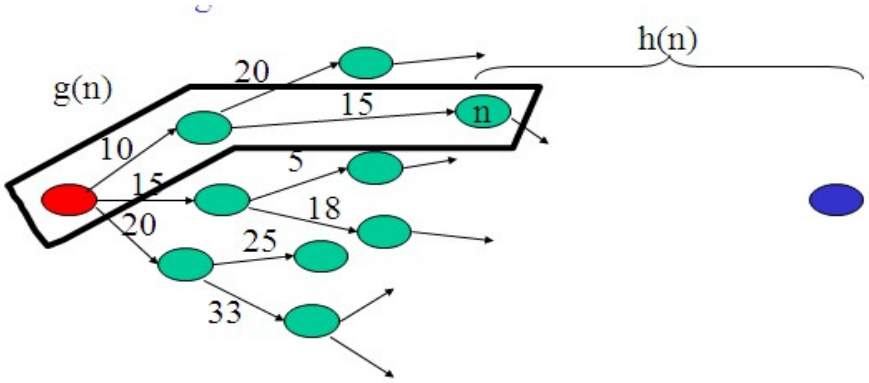
A* computes the function f(n) = g(n) + h(n)

Where g(n) is actual cost and h(n) is estimated cost.

f(n) = g(n) + h(n)

g(n) = "cost from the starting node to reach n"

h(n) = "estimate of the cost of the cheapest path from n to the goal node"



Algorithm:

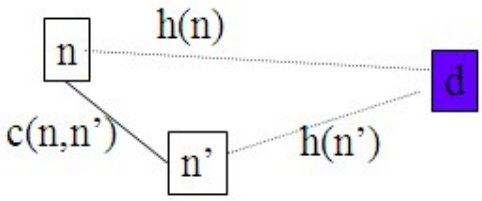
A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:

h(n) is admissible if it never overestimates the cost to reach the destination node.

A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:

h(n) is consistent if for every node n and for every successor node n' of n:

$h(n) \leq c(n,n') + h(n')$



If h(n) is consistent then h(n) is admissible

Frequently when h(n) is admissible, it is also consistent

Code:

```
// A C++ Program to implement A* Search Algorithm
#include<bits/stdc++.h>
using namespace std;

#define ROW 9
#define COL 10

// Creating a shortcut for int, int pair type
typedef pair<int, int> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type
typedef pair<double, pair<int, int> > pPair;

// A structure to hold the neccesary parameters
struct cell
{
```

```
    // Row and Column index of its parent
    // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
    int parent_i, parent_j;
    // f = g + h
    double f, g, h;
};

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL);
}

// A Utility Function to check whether the given cell is
// blocked or not
bool isUnBlocked(int grid[][COL], int row, int col)
{
    // Returns true if the cell is not blocked else false
    if (grid[row][col] == 1)
        return (true);
```

```
    else
        return (false);
}

// A Utility Function to check whether destination cell has
// been reached or not
bool isDestination(int row, int col, Pair dest)
{
    if (row == dest.first && col == dest.second)
        return (true);
    else
        return (false);
}

// A Utility Function to calculate the 'h' heuristics.
double calculateHValue(int row, int col, Pair dest)
{
    // Return using the distance formula
    return ((double)sqrt ((row-dest.first)*(row-dest.first)
                          + (col-dest.second)*(col-dest.second)));
}

// A Utility Function to trace the path from the source
// to destination
```

```
void tracePath(cell cellDetails[][COL], Pair dest)
{
    printf ("\nThe Path is ");
    int row = dest.first;
    int col = dest.second;

    stack<Pair> Path;

    while (!(cellDetails[row][col].parent_i == row
            && cellDetails[row][col].parent_j == col ))
    {
        Path.push (make_pair (row, col));
        int temp_row = cellDetails[row][col].parent_i;
        int temp_col = cellDetails[row][col].parent_j;
        row = temp_row;
        col = temp_col;
    }

    Path.push (make_pair (row, col));
    while (!Path.empty())
    {
        pair<int,int> p = Path.top();
        Path.pop();
        printf("-> (%d,%d) ",p.first,p.second);
```

```
    }

    return;
}

// A Function to find the shortest path between
// a given source cell to a destination cell according
// to A* Search Algorithm
void aStarSearch(int grid[][COL], Pair src, Pair dest)
{
    // If the source is out of range
    if (isValid (src.first, src.second) == false)
    {
        printf ("Source is invalid\n");
        return;
    }

    // If the destination is out of range
    if (isValid (dest.first, dest.second) == false)
    {
        printf ("Destination is invalid\n");
        return;
    }
```

```
// Either the source or the destination is blocked
if (isUnBlocked(grid, src.first, src.second) == false ||
        isUnBlocked(grid, dest.first, dest.second) == false)
{
    printf ("Source or the destination is blocked\n");
    return;
}

// If the destination cell is the same as source cell
if (isDestination(src.first, src.second, dest) == true)
{
    printf ("We are already at the destination\n");
    return;
}

// Create a closed list and initialise it to false which means
// that no cell has been included yet
// This closed list is implemented as a boolean 2D array
bool closedList[ROW][COL];
memset(closedList, false, sizeof (closedList));

// Declare a 2D array of structure to hold the details
//of that cell
cell cellDetails[ROW][COL];
```

```
int i, j;

for (i=0; i<ROW; i++)
{
   for (j=0; j<COL; j++)
   {
      cellDetails[i][j].f = FLT_MAX;
      cellDetails[i][j].g = FLT_MAX;
      cellDetails[i][j].h = FLT_MAX;
      cellDetails[i][j].parent_i = -1;
      cellDetails[i][j].parent_j = -1;
   }
}

// Initialising the parameters of the starting node
i = src.first, j = src.second;
cellDetails[i][j].f = 0.0;
cellDetails[i][j].g = 0.0;
cellDetails[i][j].h = 0.0;
cellDetails[i][j].parent_i = i;
cellDetails[i][j].parent_j = j;

/*
```

```
 Create an open list having information as-
 <f, <i, j>>
 where f = g + h,
 and i, j are the row and column index of that cell
 Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
 This open list is implenented as a set of pair of pair.*/
set<pPair> openList;

// Put the starting cell on the open list and set its
// 'f' as 0
openList.insert(make_pair (0.0, make_pair (i, j)));

// We set this boolean value as false as initially
// the destination is not reached.
bool foundDest = false;

while (!openList.empty())
{
    pPair p = *openList.begin();

    // Remove this vertex from the open list
    openList.erase(openList.begin());

    // Add this vertex to the closed list
```

```
i = p.second.first;
j = p.second.second;
closedList[i][j] = true;

/*
Generating all the 8 successor of this cell

   N.W   N   N.E
     \   |   /
      \  |  /
   W----Cell----E
       / | \
     /   |  \
   S.W   S   S.E

Cell-->Popped Cell (i, j)
N -->  North       (i-1, j)
S -->  South       (i+1, j)
E -->  East        (i, j+1)
W -->  West          (i, j-1)
N.E--> North-East  (i-1, j+1)
N.W--> North-West  (i-1, j-1)
S.E--> South-East  (i+1, j+1)
S.W--> South-West  (i+1, j-1)*/
```

```
// To store the 'g', 'h' and 'f' of the 8 successors
double gNew, hNew, fNew;

//----------- 1st Successor (North) ------------

// Only process this cell if this is a valid one
if (isValid(i-1, j) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i-1, j, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i-1][j].parent_i = i;
        cellDetails[i-1][j].parent_j = j;
        printf ("The destination cell is found\n");
        tracePath (cellDetails, dest);
        foundDest = true;
        return;
    }
    // If the successor is already on the closed
    // list or if it is blocked, then ignore it.
    // Else do the following
```

```
else if (closedList[i-1][j] == false &&
        isUnBlocked(grid, i-1, j) == true)
{
    gNew = cellDetails[i][j].g + 1.0;
    hNew = calculateHValue (i-1, j, dest);
    fNew = gNew + hNew;

    // If it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is better,
    // using 'f' cost as the measure.
    if (cellDetails[i-1][j].f == FLT_MAX ||
            cellDetails[i-1][j].f > fNew)
    {
        openList.insert( make_pair(fNew,
                                make_pair(i-1, j)));

        // Update the details of this cell
        cellDetails[i-1][j].f = fNew;
        cellDetails[i-1][j].g = gNew;
```

```
            cellDetails[i-1][j].h = hNew;
            cellDetails[i-1][j].parent_i = i;
            cellDetails[i-1][j].parent_j = j;
        }
    }
}

//---------- 2nd Successor (South) ------------

// Only process this cell if this is a valid one
if (isValid(i+1, j) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i+1, j, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i+1][j].parent_i = i;
        cellDetails[i+1][j].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
        return;
    }
```

```
// If the successor is already on the closed
// list or if it is blocked, then ignore it.
// Else do the following
else if (closedList[i+1][j] == false &&
        isUnBlocked(grid, i+1, j) == true)
{
    gNew = cellDetails[i][j].g + 1.0;
    hNew = calculateHValue(i+1, j, dest);
    fNew = gNew + hNew;

    // If it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is better,
    // using 'f' cost as the measure.
    if (cellDetails[i+1][j].f == FLT_MAX ||
            cellDetails[i+1][j].f > fNew)
    {
        openList.insert( make_pair (fNew, make_pair (i+1, j)));
        // Update the details of this cell
        cellDetails[i+1][j].f = fNew;
```

```
            cellDetails[i+1][j].g = gNew;
            cellDetails[i+1][j].h = hNew;
            cellDetails[i+1][j].parent_i = i;
            cellDetails[i+1][j].parent_j = j;
        }
    }
}

//---------- 3rd Successor (East) ----------

// Only process this cell if this is a valid one
if (isValid (i, j+1) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i, j+1, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i][j+1].parent_i = i;
        cellDetails[i][j+1].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
        return;
```

```
}

// If the successor is already on the closed
// list or if it is blocked, then ignore it.
// Else do the following
else if (closedList[i][j+1] == false &&
        isUnBlocked (grid, i, j+1) == true)
{
    gNew = cellDetails[i][j].g + 1.0;
    hNew = calculateHValue (i, j+1, dest);
    fNew = gNew + hNew;

    // If it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is better,
    // using 'f' cost as the measure.
    if (cellDetails[i][j+1].f == FLT_MAX ||
            cellDetails[i][j+1].f > fNew)
    {
        openList.insert( make_pair(fNew,
```

```
                        make_pair (i, j+1)));

            // Update the details of this cell
            cellDetails[i][j+1].f = fNew;
            cellDetails[i][j+1].g = gNew;
            cellDetails[i][j+1].h = hNew;
            cellDetails[i][j+1].parent_i = i;
            cellDetails[i][j+1].parent_j = j;
        }
    }
}

//---------- 4th Successor (West) ----------

// Only process this cell if this is a valid one
if (isValid(i, j-1) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i, j-1, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i][j-1].parent_i = i;
        cellDetails[i][j-1].parent_j = j;
```

```
    printf("The destination cell is found\n");
    tracePath(cellDetails, dest);
    foundDest = true;
    return;
}

// If the successor is already on the closed
// list or if it is blocked, then ignore it.
// Else do the following
else if (closedList[i][j-1] == false &&
        isUnBlocked(grid, i, j-1) == true)
{
    gNew = cellDetails[i][j].g + 1.0;
    hNew = calculateHValue(i, j-1, dest);
    fNew = gNew + hNew;

    // If it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
    //                OR
    // If it is on the open list already, check
    // to see if this path to that square is better,
    // using 'f' cost as the measure.
```

```
        if (cellDetails[i][j-1].f == FLT_MAX ||
                cellDetails[i][j-1].f > fNew)
        {
            openList.insert( make_pair (fNew,
                                make_pair (i, j-1)));

            // Update the details of this cell
            cellDetails[i][j-1].f = fNew;
            cellDetails[i][j-1].g = gNew;
            cellDetails[i][j-1].h = hNew;
            cellDetails[i][j-1].parent_i = i;
            cellDetails[i][j-1].parent_j = j;
        }
    }
}

//----------- 5th Successor (North-East) ------------

// Only process this cell if this is a valid one
if (isValid(i-1, j+1) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i-1, j+1, dest) == true)
```

```
{
    // Set the Parent of the destination cell
    cellDetails[i-1][j+1].parent_i = i;
    cellDetails[i-1][j+1].parent_j = j;
    printf ("The destination cell is found\n");
    tracePath (cellDetails, dest);
    foundDest = true;
    return;
}

// If the successor is already on the closed
// list or if it is blocked, then ignore it.
// Else do the following
else if (closedList[i-1][j+1] == false &&
        isUnBlocked(grid, i-1, j+1) == true)
{
    gNew = cellDetails[i][j].g + 1.414;
    hNew = calculateHValue(i-1, j+1, dest);
    fNew = gNew + hNew;

    // If it isn't on the open list, add it to
    // the open list. Make the current square
    // the parent of this square. Record the
    // f, g, and h costs of the square cell
```

```
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is better,
        // using 'f' cost as the measure.
        if (cellDetails[i-1][j+1].f == FLT_MAX ||
              cellDetails[i-1][j+1].f > fNew)
        {
           openList.insert( make_pair (fNew,
                        make_pair(i-1, j+1)));

           // Update the details of this cell
           cellDetails[i-1][j+1].f = fNew;
           cellDetails[i-1][j+1].g = gNew;
           cellDetails[i-1][j+1].h = hNew;
           cellDetails[i-1][j+1].parent_i = i;
           cellDetails[i-1][j+1].parent_j = j;
        }
     }
}

//----------- 6th Successor (North-West) ------------

// Only process this cell if this is a valid one
if (isValid (i-1, j-1) == true)
```

```
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination (i-1, j-1, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i-1][j-1].parent_i = i;
        cellDetails[i-1][j-1].parent_j = j;
        printf ("The destination cell is found\n");
        tracePath (cellDetails, dest);
        foundDest = true;
        return;
    }

    // If the successor is already on the closed
    // list or if it is blocked, then ignore it.
    // Else do the following
    else if (closedList[i-1][j-1] == false &&
            isUnBlocked(grid, i-1, j-1) == true)
    {
        gNew = cellDetails[i][j].g + 1.414;
        hNew = calculateHValue(i-1, j-1, dest);
        fNew = gNew + hNew;
```

```
        // If it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record the
        // f, g, and h costs of the square cell
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is better,
        // using 'f' cost as the measure.
        if (cellDetails[i-1][j-1].f == FLT_MAX ||
              cellDetails[i-1][j-1].f > fNew)
        {
           openList.insert( make_pair (fNew, make_pair (i-1, j-1)));
           // Update the details of this cell
           cellDetails[i-1][j-1].f = fNew;
           cellDetails[i-1][j-1].g = gNew;
           cellDetails[i-1][j-1].h = hNew;
           cellDetails[i-1][j-1].parent_i = i;
           cellDetails[i-1][j-1].parent_j = j;
        }
     }
  }

  //----------- 7th Successor (South-East) ------------
```

```
// Only process this cell if this is a valid one
if (isValid(i+1, j+1) == true)
{
   // If the destination cell is the same as the
   // current successor
   if (isDestination(i+1, j+1, dest) == true)
   {
      // Set the Parent of the destination cell
      cellDetails[i+1][j+1].parent_i = i;
      cellDetails[i+1][j+1].parent_j = j;
      printf ("The destination cell is found\n");
      tracePath (cellDetails, dest);
      foundDest = true;
      return;
   }

   // If the successor is already on the closed
   // list or if it is blocked, then ignore it.
   // Else do the following
   else if (closedList[i+1][j+1] == false &&
         isUnBlocked(grid, i+1, j+1) == true)
   {
      gNew = cellDetails[i][j].g + 1.414;
      hNew = calculateHValue(i+1, j+1, dest);
```

```
        fNew = gNew + hNew;

        // If it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record the
        // f, g, and h costs of the square cell
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is better,
        // using 'f' cost as the measure.
        if (cellDetails[i+1][j+1].f == FLT_MAX ||
                cellDetails[i+1][j+1].f > fNew)
        {
            openList.insert(make_pair(fNew,
                                make_pair (i+1, j+1)));

            // Update the details of this cell
            cellDetails[i+1][j+1].f = fNew;
            cellDetails[i+1][j+1].g = gNew;
            cellDetails[i+1][j+1].h = hNew;
            cellDetails[i+1][j+1].parent_i = i;
            cellDetails[i+1][j+1].parent_j = j;
        }
    }
```

```
}

//----------- 8th Successor (South-West) ------------

// Only process this cell if this is a valid one
if (isValid (i+1, j-1) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i+1, j-1, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i+1][j-1].parent_i = i;
        cellDetails[i+1][j-1].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
        return;
    }

    // If the successor is already on the closed
    // list or if it is blocked, then ignore it.
    // Else do the following
    else if (closedList[i+1][j-1] == false &&
```

```
        isUnBlocked(grid, i+1, j-1) == true)
    {
        gNew = cellDetails[i][j].g + 1.414;
        hNew = calculateHValue(i+1, j-1, dest);
        fNew = gNew + hNew;

        // If it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record the
        // f, g, and h costs of the square cell
        //                OR
        // If it is on the open list already, check
        // to see if this path to that square is better,
        // using 'f' cost as the measure.
        if (cellDetails[i+1][j-1].f == FLT_MAX ||
            cellDetails[i+1][j-1].f > fNew)
        {
            openList.insert(make_pair(fNew,
                            make_pair(i+1, j-1)));

            // Update the details of this cell
            cellDetails[i+1][j-1].f = fNew;
            cellDetails[i+1][j-1].g = gNew;
            cellDetails[i+1][j-1].h = hNew;
```

```
                cellDetails[i+1][j-1].parent_i = i;
                cellDetails[i+1][j-1].parent_j = j;
            }
        }
    }
}

    // When the destination cell is not found and the open
    // list is empty, then we conclude that we failed to
    // reach the destiantion cell. This may happen when the
    // there is no way to destination cell (due to blockages)
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");

    return;
}



// Driver program to test above function
int main()
{
    /* Description of the Grid-
     1--> The cell is not blocked
     0--> The cell is blocked    */
```

```
int grid[ROW][COL] =
{
    { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
    { 1, 1, 1, 0, 1, 1, 1, 0, 1, 1 },
    { 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },
    { 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },
    { 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },
    { 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },
    { 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 },
    { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
    { 1, 1, 1, 0, 0, 0, 1, 0, 0, 1 }
};

// Source is the left-most bottom-most corner
Pair src = make_pair(8, 0);

// Destination is the left-most top-most corner
Pair dest = make_pair(0, 0);

aStarSearch(grid, src, dest);
return(0);
```

Conclusion:

Successfully completed implementation of A* algorithm.

Assignment 3

Title: Implement Alpha-Beta Tree search for any game search problem.

Problem Statement: Assignment on analysis of non-AI and AI technique: Implement Tic-Tac-Toe or any multiplayer game using non-AI and AI technique using minmax algorithm.

Theory:

Game Theory:

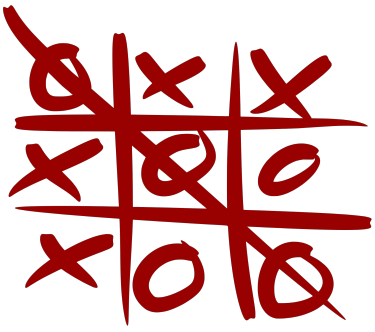
Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games such as Tic-Tac-Toe, Backgammon, Manacle, Chess, etc.

In Minimax the two players are called maximize and minimize. The maximizer tries to get the highest score possible while the minimizer tries to get the lowest score possible while minimizer tries to do opposite.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Example

Tic-tac-toe (also known as noughts and crosses or Xs and Os) is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.



Tic-tac-toe (also known as noughts and crosses or Xs and Os) is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

The following example game is won by the first player, X:

Players soon discover that the best play from both parties leads to a draw. Hence, tic-tac-toe is most often played by young children.

Because of the simplicity of tic-tac-toe, it is often used as a pedagogical tool for teaching the concepts of good sportsmanship and the branch of artificial intelligence that deals with the searching of game trees. It is straightforward to write a computer program to play tic-tac-toe perfectly, to enumerate the 765 essentially different positions (the state space complexity), or the 26,830 possible games up to rotations and reflections (the game tree complexity) on this space

The game can be generalized to an m,n,k-game in which two players alternate placing stones of their own color on an m×n board, with the goal of getting k of their own color in a row. Tic-tac-toe is the (3,3,3)-game. Harary's generalized tic-tac-toe is an even broader generalization of tic tac toe. It can also be generalized as a nd game. Tic-tac-toe is the game where n equals 3 and d equals 2. If played properly, the game will end in a draw making tic-tac-toe a futile game.

Code:

```
// A simple C++ program to find
// maximum score that
// maximizing player can get.
#include<bits/stdc++.h>
using namespace std;
// Returns the optimal value a maximizer can obtain.
// depth is current depth in game tree.
// nodeIndex is index of current node in scores[].
// isMax is true if current move is
// of maximizer, else false
// scores[] stores leaves of Game tree.
// h is maximum height of Game tree
int minimax(int depth, int nodeIndex, bool isMax,
            int scores[], int h)
{
    // Terminating condition. i.e
    // leaf node is reached
    if (depth == h)
        return scores[nodeIndex];
     //  If current move is maximizer,
    // find the maximum attainable
    // value
```

```
    if (isMax)
        return max(minimax(depth+1, nodeIndex*2, false, scores, h),
                   minimax(depth+1, nodeIndex*2 + 1, false, scores, h));

    // Else (If current move is Minimizer), find the minimum
    // attainable value
    else
        return min(minimax(depth+1, nodeIndex*2, true, scores, h),
                   minimax(depth+1, nodeIndex*2 + 1, true, scores, h));
}
// A utility function to find Log n in base 2
int log2(int n)
{
  return (n==1)? 0 : 1 + log2(n/2);
}
// Driver code
int main()
{
    // The number of elements in scores must be
    // a power of 2.
    int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};
    int n = sizeof(scores)/sizeof(scores[0]);
    int h = log2(n);
```

```
    int res = minimax(0, 0, true, scores, h);
    cout << "The optimal value is : " << res << endl;
    return 0;
}
```

Output: Successfully implemented tic tac toe problem using minmax

Assignment 4

Title: Implementing Branch and Bound and Backtracking for the N-Queens Problem

Problem Statement: The lab focuses on implementing two different algorithms, Branch and Bound, and Backtracking, to solve a Constraint Satisfaction Problem. The problem chosen for this purpose is the N-Queens Problem, where the objective is to place N chess queens on an NxN chessboard in such a way that no two queens threaten each other.

Objective: The main objective of this lab is to understand and implement two fundamental algorithms, Branch and Bound, and Backtracking, to solve the N-Queens Problem. By completing this lab, participants will gain insights into how different search strategies can be applied to Constraint Satisfaction Problems.

Outcome: Upon completion of this lab, participants will be able to:

1. Understand the N-Queens Problem and its constraints.
2. Implement the Backtracking algorithm to solve the N-Queens Problem.
3. Implement the Branch and Bound algorithm to solve the N-Queens Problem.
4. Compare and contrast the performance of both algorithms in terms of time complexity and solution quality.

Lab Setup:

- Programming language: Python
- Required libraries: None

Input:

- The value of N (the size of the chessboard).

Output:

- The placement of N queens on the chessboard such that no two queens threaten each other.

Theory:

1. **Backtracking Algorithm:** Backtracking is a brute-force algorithm used to solve problems by trying out all possible solutions and undoing a solution if it is found to be invalid. For the N-Queens Problem, backtracking involves recursively placing queens on the board and removing them if a valid solution cannot be found.
2. **Branch and Bound Algorithm:** Branch and Bound is a more efficient algorithm that prunes certain branches of the search tree to avoid unnecessary exploration. In the context of the N-Queens Problem, this algorithm involves placing queens while considering constraints and bounding the search space.

Algorithm:

Backtracking Algorithm:

Start with an empty chessboard.

Place queens one by one, checking for constraints.

If constraints are violated, backtrack and try a different position.

Repeat until all queens are placed.

Branch and Bound Algorithm:

Initialize an empty chessboard.

Place queens based on constraints and a cost function.

Prune branches of the search tree that cannot lead to a valid solution.

Continue until a valid solution is found or all branches are pruned.

Flow Chart:

[Insert flow charts for both Backtracking and Branch and Bound algorithms here]

Lab Procedure:

Backtracking Algorithm Implementation:

Initialize the chessboard.

Implement the recursive backtracking function.

Place queens while considering constraints.

If a solution is found, display the board.

Branch and Bound Algorithm Implementation:

Initialize the chessboard.

Implement the branching strategy with constraints.

Implement the bounding strategy to prune branches.

Display the valid solution if found.

Comparative Analysis:

Run both algorithms for various N values.

Measure and compare execution time.

Analyze the quality of solutions.

Conclusion: In this lab, we successfully implemented the Backtracking and Branch and Bound algorithms to solve the N-Queens Problem. We observed that while Backtracking exhaustively explores all possibilities, Branch and Bound efficiently prunes certain branches, leading to improved performance for larger N values. This lab provided insights into how different search strategies impact the solving of Constraint Satisfaction Problems.

Assignment 5

Title: Greedy Search Algorithm Implementation

Problem Statement: The objective of this lab is to implement and understand the Greedy search algorithm for solving optimization problems. The algorithm will be applied to various applications such as Selection Sort, Minimum Spanning Tree, Single-Source Shortest Path Problem, Job Scheduling Problem, Prim's Minimal Spanning Tree Algorithm, Kruskal's Minimal Spanning Tree Algorithm, and Dijkstra's Minimal Spanning Tree Algorithm. Through this lab, students will gain hands-on experience in applying Greedy algorithms to real-world scenarios.

Objectives:

Understand the concept of Greedy algorithms and their applications.

Implement the Greedy search algorithm for various optimization problems.

Analyze the efficiency and correctness of the implemented algorithms.

Compare and contrast different Greedy algorithms for the same problem.

Outcome:

By the end of this lab, students should be able to:

Implement Greedy algorithms for different optimization problems.

Understand the decision-making process involved in Greedy algorithms.

Evaluate the effectiveness of Greedy algorithms through experimental results.

Choose an appropriate Greedy algorithm for a given problem.

1. **Introduction to Greedy Algorithms:**

Brief explanation of Greedy algorithms and their approach to problem-solving.

Discussion on how Greedy algorithms make locally optimal choices at each step.

Application of Greedy Algorithms:

Explanation of various applications where Greedy algorithms can be applied:

Selection Sort

Minimum Spanning Tree

Single-Source Shortest Path Problem

Job Scheduling Problem

Prim's Minimal Spanning Tree Algorithm

Kruskal's Minimal Spanning Tree Algorithm

Dijkstra's Minimal Spanning Tree Algorithm

Theory and Algorithm:

Detailed explanation of the Greedy algorithm for each application, including pseudocode

Lab Activity: Algorithm Implementation:

Step-by-step implementation guidelines for each of the chosen applications.

Explanation of data structures and variables used in the algorithms.

Code snippets in a programming language of choice (e.g., Python, Java).

Lab Activity: Testing and Analysis:

Instructions on how to create test cases for the implemented algorithms.

Guidance on measuring and analyzing algorithm performance, such as time complexity and space complexity.

Flowchart:

Provide a visual representation of the implemented Greedy algorithm for one of the applications.

Explain the flowchart symbols and connections used.

Conclusion:

Summary of the lab experience and the key concepts learned.

Reflection on the advantages and limitations of using Greedy algorithms.

Comparison of the different Greedy algorithms applied in terms of efficiency and effectiveness.

Additional Exercises (Optional):

Provide additional problems related to Greedy algorithms for students to practice.

Assessment:

Lab report submission including algorithm implementations, analysis of results, and conclusions.

In-class demonstration of the implemented algorithms and their correctness.

References:

List of resources used for learning about Greedy algorithms and their applications.

Note: It is important to adjust the lab manual according to the programming language and tools preferred by the institution and the students. Additionally, make sure to provide support and guidance during the lab sessions to assist students in understanding and implementing the algorithms effectively.

Assignment-6

Title: Develop an Elementary Chatbot for Customer Interaction

Problem Statement: Developing an elementary chatbot for customer interaction in various applications to provide basic assistance and information.

Objective: The objective of this lab is to design and implement a simple chatbot that can engage in customer interactions, answer common queries, and provide relevant information.

Outcome: At the end of this lab, students will be able to create a basic chatbot capable of interacting with users, understanding their queries, and responding appropriately.

Input: User messages or queries entered through the chat interface.

Output: Responses generated by the chatbot based on the user's input.

Theory/Algorithm:

1. **Natural Language Processing (NLP):** NLP is a field of artificial intelligence that focuses on the interaction between computers and humans using natural language. It involves tasks such as text processing, language understanding, and language generation.
2. **Intent Recognition:** Intent recognition involves identifying the purpose or goal behind a user's query. This is achieved using techniques like pattern matching, rule-based methods, or machine learning models.
3. **Response Generation:** Once the intent is identified, the chatbot generates a suitable response. This can be done using pre-defined templates, retrieval-based methods, or more advanced generative models.

Flow Chart:

sqlCopy code

Start | |--> Initialize Chatbot | | | |--> Load Pre-trained NLP Model | |--> Define Intent-Response Mapping | |--> User Input | | | |--> Process Input Text | | | | |--> Identify Intent | | | |--> Generate Response | | | |--> Select Response Template/Generate Response | |--> Display Response to User | |--> Repeat | End

Lab Procedure:

1. Initialize the chatbot by loading a pre-trained NLP model and defining intent-response mapping.
2. Accept user input from the chat interface.
3. Process the input text to identify the user's intent using simple keyword matching.
4. Based on the intent, generate a response using predefined response templates.
5. Display the response to the user in the chat interface.

6. Repeat steps 2 to 5 to continue the interaction.
7. Test the chatbot with various queries and assess its performance.

Conclusion: In this lab, you have successfully developed an elementary chatbot for customer interaction. The chatbot can understand user intents and respond with appropriate answers. This lab provides a basic understanding of how chatbots work and can be a foundation for building more advanced and sophisticated chatbot systems. By expanding the intent-response mapping and integrating more advanced NLP techniques, you can enhance the capabilities of the chatbot to handle a wider range of customer interactions.

Assignment-7

Mini Project:

Expert System Implementation

Topic: Implementing an Expert System for Employee Performance Evaluation

Problem Statement: Develop an expert system to automate the process of employee performance evaluation within an organization. The system should assist managers in objectively assessing employees' performance based on predefined criteria.

Objective: The objective of this mini project is to design and implement an expert system that can accurately evaluate employee performance using a set of predefined rules and criteria. This system aims to streamline and standardize the performance evaluation process, ensuring fairness and consistency.

Outcome: By the end of this project, participants should be able to understand the basics of expert systems, develop a functional employee performance evaluation expert system, and gain insights into the potential applications of expert systems in real-world scenarios.

Input:

- Employee information (name, ID, department)
- Performance metrics (quantitative and qualitative)
- Manager's assessment and feedback

Output:

- Performance evaluation score
- Performance category (excellent, satisfactory, needs improvement, etc.)
- Constructive feedback and areas for improvement

Theory/Algorithm: The expert system will utilize a rule-based approach. The system will define a set of rules based on the organization's performance evaluation criteria. These rules will consider quantitative metrics (such as targets met, sales figures) and qualitative assessments (communication skills, teamwork).

Flow Chart:

1. **Initialization**
    - Gather employee information
2. **Input Collection**
    - Collect quantitative metrics
    - Collect qualitative assessments

- Obtain manager's feedback

3. **Rule-based Evaluation**
   - Apply predefined rules to quantitative metrics
   - Apply rules to qualitative assessments
   - Combine results to calculate performance score
4. **Performance Category Determination**
   - Assign performance category based on the calculated score
5. **Feedback Generation**
   - Provide constructive feedback
   - Highlight areas for improvement
6. **Output Presentation**
   - Display performance score and category
   - Show feedback to the employee
7. **Conclusion**
   - Summarize the evaluation process
   - Discuss the potential impact of the expert system on performance evaluation

Conclusion: In this mini project, you have successfully implemented an expert system for employee performance evaluation. You learned how to collect input data, define rules for evaluation, and generate meaningful output. Expert systems have the potential to revolutionize various aspects of organizations, enhancing decision-making and standardizing processes. This project serves as an introduction to the world of expert systems and their practical applications.