Silver Oak College of Engineering& Technology

GUJARAT TECHNOLOGICAL UNIVERSITYBACHELOR OF ENGINEERING

ARTIFICIAL INTELLIGENCE(2180703)

8th SEMESTER

COMPUTER ENGINEERING DEPARTMENT

Laboratory Manual

Prepared By: Prof. Jigar Dalvadi



# DEPARTMENT OF COMPUTER ENGINEERING

## VISION

To be recognized for the quality education and research in the field of Computer Engineering known for its accomplished graduates.

## MISSION

1. To produce technically competent and ethically sound Computer Engineering professionals by imparting quality education, training, hands on experience and value based education.
2. To inculcate ethical attitude, sense of responsibility towards society and leadership ability required for a responsible professional computer engineer.
3. To pursue creative research, adapt to rapidly changing technologies and promote self-learning approach in Computer Engineering and across disciplines to serve the dynamic needs of industry, government and society.

## Program Educational Objectives (PEO):

PEO1: To provide fundamental knowledge of science and engineering for an IT professional and to equip them with proficiency of mathematical foundations and algorithmic principles and inculcate competent problem- solving ability.

PEO2: To implant ability in creativity &amp; design of IT systems and transmit knowledge and skills to analyze, design, test and implement various software applications.

PEO3: To exhibit leadership capability, triggering social and economic commitment and inculcate community services.

PEO4: To inculcate professional-social ethics, teamwork in students and acquaint them with requisite technical and managerial skills to attain a successful career.



## PROGRAM OUTCOMES (POs)

**Engineering Graduates will be able to:**

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage



Projects and in multidisciplinary environments.

1. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.



ARTIFICIAL INTELLIGENCE PRACTICAL BOOK

## DEPARTMENT OF INFORMATION & TECHNOLOGY

**PREFACE**

It gives us immense pleasure to present the first edition of Artificial Intelligence Practical Book for the B.E. 4th year students of Silver Oak Group of Institutes.

The theory and laboratory course of Artificial Intelligence, at Silver Oak Group of Institutes, Ahmedabad, is designed in such a manner that students can develop the basic understanding of the subject during theory classes and gain the hands-on practical experience during their laboratory sessions.

The Laboratory Manual presented here to you help you in understanding various search methods, which explores the possible moves that one can make in a space of 'states', called the search space. It also take you in learning various Game Playing techniques. It will help you in learning prolog (programming in logic) which will be very useful programming language in Artificial Intelligence programming.

Lab Manual Revised by: Prof. Nirav Shah, Silver Oak College of Engineering and Technology Prof. Jalpa Shah, Aditya Silver Oak Institute of Technology

Prof. Jigar Dalvadi, Silver Oak College of Engineering and Technology Manual Revision No.: SOGI\_2180714\_LM\_2020-21\_2



## INSTRUCTIONS TO STUDENTS

* 1. Be prompt in arriving to the laboratory and always come well prepared for the experiment.
  2. Students need to maintain a proper decorum in the computer lab. Students mustuse the equipment with care. Any damage is caused is punishable.
  3. Students are supposed to occupy the systems allotted to them and are not supposedto talk or make noise in the lab.
  4. Students are required to carry their observation book and lab records withcompleted exercises while entering the lab.
  5. Lab records need to be submitted every week.
  6. Students are not supposed to use pen drives in the lab.
  7. The grades for the Artificial Intelligence Practical course work will be awarded basedon your performance in the laboratory, regularity, recording of experiments in the Artificial Intelligence practical Final Notebook, lab quiz, regular viva-voce and end- term examination.
  8. Find the answers of all the questions mentioned under the section ‘Post Practical Questions' at the end of each experiment in the Artificial Intelligence Practical Book.



**CERTIFICATE**

*This is to certify that Mr./~~Ms.~~ PARMAR DHAVAL BHARAT KUMAR with enrollment number 170770107100 from Semester 8TH Div C has successfully completed his/~~her~~ laboratory experiments in the* ***Artificial Intelligence (2180703)*** *from the department of Computer Engineering during the academic year 2020-21.*

Date of Submission: ......................... Staff In charge: ...........................

Head of Department: ...........................................

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TABLE OF CONTENT** | | | | | | | | |
|  | **Sr. No** | **Experiment Title** | **Page No** | | **Date of Start** | **Date of Completion** | **Sign** | **Marks (out of 10)** |
| **To** | **From** |
| 1 | Study of Prolog | 1 | 3 |  |  |  |  |
| 2 | Write a program to implement Tic-Tac-Toe game problem | 4 | 15 |  |  |  |  |
| 3 | Write a program to implement BFS (for 8 puzzle problem or  Water Jug problem or any AI search problem) | 16 | 22 |  |  |  |  |
| 4 | Write a program to implement DFS (for 8 puzzle problem or  Water Jug problem or any AI search problem) | 23 | 31 |  |  |  |  |
| 5 | Write a program to implement Single Player Game (Using Heuristic Function) | 32 | 40 |  |  |  |  |
| 6 | Write a program to Implement A\* Algorithm. | 41 | 53 |  |  |  |  |
| 7 | Write a program to solve N- Queens problem using Prolog. | 54 | 57 |  |  |  |  |
| 8 | Write a program to solve 8 puzzle problem using Prolog. | 58 | 60 |  |  |  |  |
|  | | | | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | 9 | Write a program to solve travelling salesman problem using Prolog. | 61 | 66 |  |  |  |  |
| 10 | Convert following Prolog predicates into Semantic Net cat(tom).  cat(cat1). mat(mat1). sat\_on(cat1,mat1). bird(bird1). caught(tom,bird1).  like(X,cream) :– cat(X).  mammal(X) :– cat(X).  has(X,fur) :– mammal(X).  animal(X) :– mammal(X).  animal(X) :– bird(X). owns(john,tom). is\_coloured(tom,ginger). | 67 | 69 |  |  |  |  |
|  | | | | | | | | |



## Theory

**PRACTICAL SET – 1**

Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Prolog = Programmation en Logique (Programming in Logic).

Prolog is a declarative programming language unlike most common programming languages. In a declarative language-

* The programmer specifies a goal to be achieved
* The Prolog system works out how to achieve it

Traditional programming languages are said to be procedural, procedural programmer must specify in detail how to solve a problem but in purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system.

## Relations

Prolog programs specify relationships among objects and properties of objects.

For example, "John owns the book", we are declaring the ownership relationship between two objects: John and the book.

When we ask, "Does John own the book?" we are trying to find out about a relationship.

Relationships can also rules .A rule allows us to find out about a relationship even if the relationship isn't explicitly stated as a fact.

Programming in Prolog



It declare facts describing explicit relationships between objects and properties objects might have (e.g. Mary likes pizza, grass has\_colour green)

It define rules defining implicit relationships between objects and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:

asking questions above relationships between objects, and/or about object properties (e.g. does Mary like pizza? is Joe a parent?)

A simple clause in prolog-

likes(mary,food). likes(mary,wine). likes(john,wine). likes(john,mary). The following queries yield the specified answers.

| ?- likes(mary,food). yes.

| ?- likes(john,wine). yes.

## Aim: Study of prolog.

**Give Applications of Prolog:**

Some applications of Prolog are:

1. Intelligent data base retrieval
2. Natural language understanding
3. Expert systems
4. Specification language
5. Machine learning
6. Robot planning
7. Automated reasoning
8. Problem solving

## Explain the meaning of following rule:



1. Bird(X):- animal(X), has feathers(X).

``X is a bird if X is an animal and X has feathers''

1. Grandparent(X, Y):- parent(X, Z), parent (Z, Y).

Y is X's grandparent if Z is X's parent and Y is Z's parent''.

Post Practical Questions:

|  |  |
| --- | --- |
| **1.** | What is Artificial intelligence?   1. Putting your intelligence into Computer 2. Programming with your own intelligence 3. **Making a Machine intelligent** 4. Playing a Game |
| **2.** | Which is not the commonly used programming language for AI?  (a) PROLOG (b) Java (c) LISP **(d) Perl** |
| **3.** | What is state space?   1. The whole problem 2. Your Definition to a problem 3. Problem you design 4. **Representing your problem with variable and parameter** |
| **4.** | A production rule consists of  (a) A set of Rule (b) A sequence of steps  **(c) Both (a) and (b)** (d) Arbitrary representation to problem |
| **5.**  **Ans**. | Mention What Is The Difference Between Prolog And Normal Programming Language?  **Prolog:** It is a “declarative programming” language which means that you have to specify the goals, but not the strategy to reach the goal. It figures it out on its own. **Normal Programming language:** Languages like C/C++/Java/Python are imperative (IP) languages that does specify instructions of how to reach some goal,  but leaving the real goal implicit |



|  |  |
| --- | --- |
| **6.**  **Ans**. | Explain Why Matching Variable Is Important In Prolog? What Is The Method To Match Variables?  In Prolog, in order to process the elements of a list, we have to match them to  variables. |
| **7.**  **Ans.** | Explain What Is Backtracking In Prolog?  In Prolog, backtracking is a process adopted by Prolog. When a sub-goal fails in Prolog, the Prolog system moves its steps backwards to the previous goal and tries to re-satisfy it. Backtracking enables Prolog to find all alternative solutions to a given  query. |
| **8.**  **Ans.** | Explain What Is Recursion In Prolog?  In any language, a function that can call itself until the goal has been succeed is  referred as recursion. In Prolog, recursion happens when a predicate contains a goal that refers to itself. |
| **9.**  **Ans.** | Explain What Is Swi-prolog?  SWI-Prolog is a free prolog compiler licensed under GPL, targeting primarily the research and education. |
| **10.**  **Ans.** | What Are The Features Of Prolog Language? Following are the features of prolog language:   * Intelligent Systems * Expert Systems * Natural Language Systems * Relational Database Systems |

## References:

1. <http://www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html>
2. <https://en.wikipedia.org/wiki/Prolog>



# PRACTICAL SET – 2

Tic-tac-toe, also spelled tick tack toe, or noughts and crosses/Xs and Os as it is known in the UK, Australia and New Zealand, is a pencil-and-paper game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The X player usually goes first. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game.

## Strategy

1. Win: If the player has two in a row, play the third to get three in a row.
2. Block: If the opponent has two in a row, play the third to block them.
3. Fork: Create an opportunity where you can win in two ways.
4. Block opponent's fork:
   * Option 1: Create two in a row to force the opponent into defending, as long as it doesn't result in them creating a fork or winning. For example, if "X" has a corner, "O" has the center, and "X" has the opposite corner as well, "O" must not play a corner in order to win. (Playing a corner in this scenario creates a fork for "X" to win.)
   * Option 2: If there is a configuration where the opponent can fork, block that fork.
5. Center: Play the center.
6. Opposite corner: If the opponent is in the corner, play the opposite corner.
7. Empty corner: Play in a corner square.
8. Empty side: Play in a middle square on any of the 4 sides

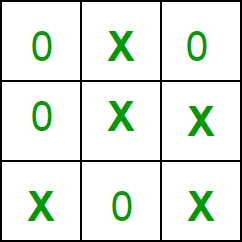
The first player, whom we shall designate "X", has 3 possible positions to mark during the first turn. Superficially, it might seem that there are 9 possible positions, corresponding to the 9 squares in the grid. However, by rotating the board, we will find that in the first turn, every corner mark is strategically equivalent to every other corner mark. The same is true of every edge mark. For strategy purposes, there are therefore only three possible first marks: corner, edge, or center. Player X can win or force a draw from any of these starting marks; however, Playing the corner gives the opponent the smallest choice of squares which must be played to avoid losing. The second player, whom we shall designate "O", must respond to X's opening mark in such a way as to avoid the forced win. Player O



must always respond to a corner opening with a center mark, and to a center opening with a corner mark. An edge opening must be answered either with a center mark, a corner mark next to the X, or an edge mark opposite the X. Any other responses will allow X to force the win. Once the opening is completed, O's task is to follow the above list of priorities in order to force the draw, or else to gain a win if X makes a weak play.

## Rules of the Game

* The game is to be played between two people.
* One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
* The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
* If no one wins, then the game is said to be draw.



## Winning Strategy – An Interesting Fact

If both the players play optimally then it is destined that you will never lose (“although the match can still be drawn”). It doesn’t matter whether you play first or second. In another ways – “Two expert players will always draw”.



## Aim : Write a program to implement Tic-Tac-Toe game problem Code :

// A C++ Program to play tic-tac-toe #include<bits/stdc++.h>

using namespace std; #define COMPUTER 1

#define HUMAN 2

#define SIDE 3 // Length of the board

// Computer will move with 'O'

// and human with 'X' #define COMPUTERMOVE 'O' #define HUMANMOVE 'X'

// A function to show the current board status void showBoard(char board[][SIDE])

{

printf("\n\n");

printf("\t\t\t%c | %c| %c\n", board[0][0],board[0][1], board[0][2]); printf("\t\t\t\n");

printf("\t\t\t%c | %c| %c\n", board[1][0],board[1][1], board[1][2]); printf("\t\t\t\n");

printf("\t\t\t%c | %c| %c\n\n", board[2][0],board[2][1], board[2][2]); return;

}

// A function to show the instructions void showInstructions()



{

printf("\t\t\tTic-Tac-Toe\n\n");

printf("Choose a cell numbered from 1 to 9 as below" " and play\n\n"); printf("\t\t\t1 | 2| 3\n");

printf("\t\t\t \n"); printf("\t\t\t 4 | 5 | 6 \n"); printf("\t\t\t \n"); printf("\t\t\t7 | 8| 9\n\n");

printf("-\t-\t-\t-\t-\t-\t-\t-\t-\t-\n\n"); return;

}

// A function to initialise the game

void initialise(char board[][SIDE], int moves[])

{

// Initiate the random number generator so that

// the same configuration doesn't arises srand(time(NULL));

// Initially the board is empty for (int i=0; i<SIDE; i++)

{

for (int j=0; j<SIDE; j++) board[i][j] = ' ';



}

// Fill the moves with numbers for (int i=0; i<SIDE\*SIDE; i++) moves[i] = i;

// randomise the moves random\_shuffle(moves, moves + SIDE\*SIDE); return;

}

// A function to declare the winner of the game void declareWinner(int whoseTurn)

{

if (whoseTurn == COMPUTER) printf("COMPUTER has won\n"); else

printf("HUMAN has won\n"); return;

}

// A function that returns true if any of the row

// is crossed with the same player's move bool rowCrossed(char board[][SIDE])

{

for (int i=0; i<SIDE; i++)



{

if (board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] != ' ') return (true);

}

return(false);

}

// A function that returns true if any of the column

// is crossed with the same player's move bool columnCrossed(char board[][SIDE])

{

for (int i=0; i<SIDE; i++)

{

if (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] != ' ') return (true);

}

return(false);

}

// A function that returns true if any of the diagonal

// is crossed with the same player's move bool diagonalCrossed(char board[][SIDE])

{

if (board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] != ' ')



return(true);

if (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] != ' ') return(true);

return(false);

}

// A function that returns true if the game is over

// else it returns a false

bool gameOver(char board[][SIDE])

{

return(rowCrossed(board) || columnCrossed(board) || diagonalCrossed(board) );

}

// A function to play Tic-Tac-Toe void playTicTacToe(int whoseTurn)

{

// A 3\*3 Tic-Tac-Toe board for playing char board[SIDE][SIDE];

int moves[SIDE\*SIDE];

// Initialise the game initialise(board,moves);

// Show the instructions before playing showInstructions();

int moveIndex = 0, x, y;



// Keep playing till the game is over or it is a draw

while (gameOver(board) == false && moveIndex != SIDE\*SIDE)

{

if (whoseTurn == COMPUTER)

{

x = moves[moveIndex] / SIDE;

y = moves[moveIndex] % SIDE; board[x][y] = COMPUTERMOVE;

printf("COMPUTER has put a %c in cell %d\n", COMPUTERMOVE, moves[moveIndex]+1);

showBoard(board);

moveIndex ++; whoseTurn = HUMAN;

}

else if (whoseTurn == HUMAN)

{

x = moves[moveIndex] / SIDE;

y = moves[moveIndex] % SIDE; board[x][y] = HUMANMOVE;

printf ("HUMAN has put a %c in cell %d\n", HUMANMOVE, moves[moveIndex]+1); showBoard(board); moveIndex ++; whoseTurn = COMPUTER;

}

}

// If the game has drawn

if (gameOver(board) == false && moveIndex == SIDE \* SIDE)



printf("It's a draw\n"); else

{

// Toggling the user to declare the actual

// winner

if (whoseTurn == COMPUTER) whoseTurn = HUMAN;

else if (whoseTurn == HUMAN) whoseTurn = COMPUTER;

// Declare the winner declareWinner(whoseTurn);

}

return;

}

// Driver program int main()

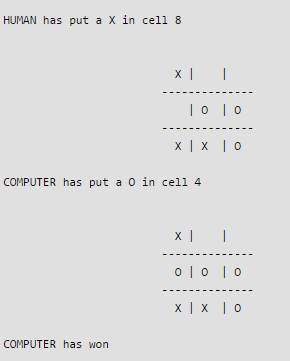
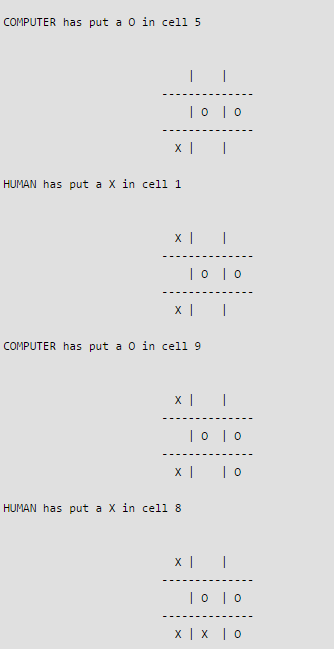
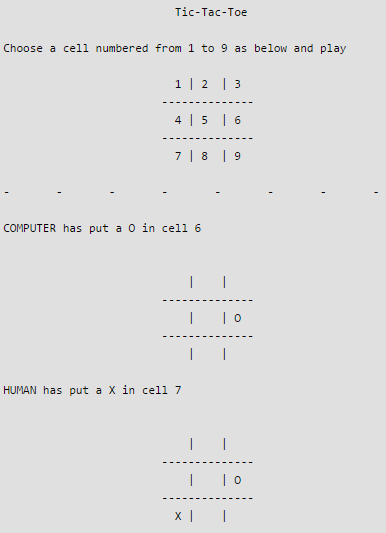
{

// Let us play the game with COMPUTER starting first playTicTacToe(COMPUTER); return (0);

}



## Output:





**References:**

|  |  |
| --- | --- |
| [1] | [http://anna.fi.muni.cz/~x139877/prezentace/2011\_03\_24\_prednaska\_mafye\_teorie\_her/piskvorky.](http://anna.fi.muni.cz/~x139877/prezentace/2011_03_24_prednaska_mafye_teorie_her/piskvorky.pdf)  [pdf](http://anna.fi.muni.cz/~x139877/prezentace/2011_03_24_prednaska_mafye_teorie_her/piskvorky.pdf) |
| [2] | <https://www.geeksforgeeks.org/implementation-of-tic-tac-toe-game/> |



**PRACTICAL SET – 3**

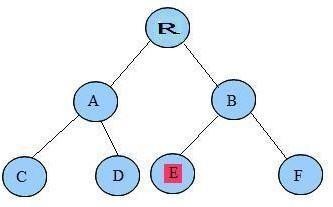
Breadth First Search (BFS): searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may be a potential candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

## Algorithm:

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
   1. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
   2. For each way that each rule can match the state described in E do:
3. Apply the rule to generate a new state.
4. If the new state is the goal state, quit and return this state.
5. Otherwise add this state to the end of NODE-LIST

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, breadth-first search always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d. Since the number of nodes at level d is bd, the total number of nodes generated in the worst case is b + b2 + b3 +… + bd i.e. O(bd) , the asymptotic time complexity of breadth first search.





Look at the above tree with nodes starting from root node, R at the first level, A and B at the second level and C, D, E and F at the third level. If we want to search for node E then BFS will search level by level. First it will check if E exists at the root. Then it will check nodes at the second level. Finally it will find E a third level.

## Advantages of BFS:

* 1. Breadth first search will never get trapped exploring the useless path forever.
  2. If there is a solution, BFS will definitely find it out.
  3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

## Disadvantages of BFS:

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(bd). As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breath first search will consume lot of time.



**Aim:** Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

**Code:** #include<iostream.h> #include<conio.h> #include<stdio.h> class Queue

{

public:

Queue() : head(NULL), tail(NULL)

{

}

void enqueue(int i)

{

if (head == NULL)

head = tail = new Node(i, NULL); else tail = tail->next = new Node(i, NULL);

}

int dequeue()

{

Node\* old = head; head = head->next; int i = old->i; delete old; return i;

}

int isEmpty()

{

return (head == NULL);

}

~Queue()

{



while (!isEmpty()) dequeue();

}

private:

struct Node

{

int i;

Node\* next;

} iQueue;

Node(int iP, Node\* nextP) : i(iP), next(nextP)

{

}

} \*head, \*tail;

} iQueue;

const int MAX = 100;

const int MAX\_I = (MAX + 1) \* (MAX + 1); int N, M, k, n, m; int distance[MAX\_I]; int prev[MAX\_I];

int nmToI(int n, int m)

{

return n \* (M + 1) + m;

}

int iToN(int i)

{

return i / (M + 1);

}

int iToM(int i)

{

return i % (M + 1);

}

void trace(int i)

{



if (i > 0) trace(prev[i]);

cout <<" "<<iToN(i) << " | " <<iToM(i) << "\n";

}

void test(int n, int m, int n1, int m1)

{

if (n1 < 0 || n1 > N || m1 < 0 || m1 > M) return;

int i1 = nmToI(n1, m1); if (distance[i1] != 0) return;

int i = nmToI(n, m); distance[i1] = distance[i] + 1;

prev[i1] = i; iQueue.enqueue(i1);

}

int solve()

{

n = m = 0; distance[0] = 1; iQueue.enqueue(0); while (!iQueue.isEmpty())

{

int i = iQueue.dequeue(); int n = iToN(i); int m = iToM(i);

if (n == k || m == k || n + m == k) return i;

// empty out a jug test(n, m, 0, m);

test(n, m, n, 0);

// fill a jug test(n, m, N, m);

test(n, m, n, M);

// pour one to another until source is empty test(n, m, 0, n + m);



test(n, m, n + m, 0);

// pour one to another until destination is full

test(n, m, n - M + m, M);

test(n, m, N, m - N + n);

}

void main()

{

}

return -1;



clrscr();

cout<<"Please enter the number of gallons in first jug: "; cin>>N; cout<<"Please enter the number of gallons in second jug: "; cin>>M; cout<<"Please enter the vol. of water to be left finally: "; cin>>k;

int i = solve();

cout<<" JUG 1 "<<" JUG 2 \n"; cout<<" \n"; if (i == -1)

cout << 0 << "\n"; else

{

cout << distance[i] << "\n"; trace(i);

}

cout << -1 << "Press the SPACE bar"; getch();

}

## Output:

Initially state(0,0) Rule => (4,0) Rule => (4,3)

Rule => (0,3)

Rule => (3,0)

Rule => (3,3)

Rule => (4,2)

Rule => (0,2)

Rule => (2,0)

Press the SPACE bar

## References:

[1] <http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html#.XAEJmmgzbIU>



# PRACTICAL SET – 4

Depth First Search (DFS): searches deeper into the problem space. Breadth-first search always generates successor of the deepest unexpanded node. It uses last-in first-out stack for keeping the unexpanded nodes. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

## Algorithm:

1. If the initial state is a goal state, quit and return success.
2. Otherwise, loop until success or failure is signalled.
   1. Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.
   2. Call Depth-First Search with E as the initial state.
   3. If success is returned, signal success. Otherwise continue in this loop.

## Advantages:

* The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
* The time complexity of a depth-first Search to depth d is O(b^d) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
* If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

## Disadvantages:

* + The disadvantage of Depth-First Search is that there is a possibility that it may go down the left- most path forever. Even a finite graph can generate an infinite tree. One solution to this problem



is to impose a cut-off depth on the search. Although the ideal cut-off is the solution

depth d and this value is rarely known in advance of actually solving the problem. If the chosen cut-off depth is less than d, the algorithm will fail to find a solution, whereas if the cut-off depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

* Depth-First Search is not guaranteed to find the solution.
* And there is no guarantee to find a minimal solution, if more than one solution exists.



**Aim:** Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

## Code:

#include <stack> #include <map> #include <algorithm> using namespace std;

// Representation of a state (x, y)

// x and y are the amounts of water in litres in the two jugs respectively struct state

{

int x, y;

// Used by map to efficiently implement lookup of seen states bool operator < (const state& that) const

{

if (x != that.x) return x < that.x; return y < that.y;

}

};

// Capacities of the two jugs respectively and the target amount int capacity\_x, capacity\_y, target;

void dfs(state start, stack <pair <state, int> >& path)

{

stack <state> s;

state goal = (state) {-1, -1};

// Stores seen states so that they are not revisited and

// maintains their parent states for finding a path through

// the state space

// Mapping from a state to its parent state and rule no. that



// led to this state

map <state, pair <state, int> > parentOf; s.push(start);

parentOf[start] = make\_pair(start, 0); while (!s.empty())

{

// Get the state at the front of the stack state top = s.top();

s.pop();

// If the target state has been found, break if (top.x == target || top.y == target)

{

goal = top; break;

}

// Find the successors of this state

// This step uses production rules to produce successors of the current state

// while pruning away branches which have been seen before

// Rule 1: (x, y) -> (capacity\_x, y) if x < capacity\_x

// Fill the first jug

if (top.x < capacity\_x)

{

state child = (state) {capacity\_x, top.y};

// Consider this state for visiting only if it has not been visited before if (parentOf.find(child) == parentOf.end())

{

s.push(child);

parentOf[child] = make\_pair(top, 1);

}

}

// Rule 2: (x, y) -> (x, capacity\_y) if y < capacity\_y



// Fill the second jug if (top.y < capacity\_y)

{

state child = (state) {top.x, capacity\_y};

if (parentOf.find(child) == parentOf.end())

{

s.push(child);

parentOf[child] = make\_pair(top, 2);

}

}

// Rule 3: (x, y) -> (0, y) if x > 0

// Empty the first jug if (top.x > 0)

{

state child = (state) {0, top.y};

if (parentOf.find(child) == parentOf.end())

{

s.push(child);

parentOf[child] = make\_pair(top, 3);

}

}

// Rule 4: (x, y) -> (x, 0) if y > 0

// Empty the second jug if (top.y > 0)

{

state child = (state) {top.x, 0};

if (parentOf.find(child) == parentOf.end())

{

s.push(child);

parentOf[child] = make\_pair(top, 4);

}



}

// Rule 5: (x, y) -> (min(x + y, capacity\_x), max(0, x + y - capacity\_x)) if y > 0

// Pour water from the second jug into the first jug until the first jug is full

// or the second jug is empty if (top.y > 0)

{

state child = (state) {min(top.x + top.y, capacity\_x), max(0, top.x + top.y - capacity\_x)

};

if (parentOf.find(child) == parentOf.end()) { s.push(child);

parentOf[child] = make\_pair(top, 5);

}

}

// Rule 6: (x, y) -> (max(0, x + y - capacity\_y), min(x + y, capacity\_y)) if x > 0

// Pour water from the first jug into the second jug until the second jug is full

// or the first jug is empty if (top.x > 0)

{

state child = (state) {max(0, top.x + top.y - capacity\_y), min(top.x + top.y, capacity\_y)};

if (parentOf.find(child) == parentOf.end())

{

s.push(child);

parentOf[child] = make\_pair(top, 6);

}

}

}

// Target state was not found

if (goal.x == -1 || goal.y == -1) return;



}

int main()

{

// backtrack to generate the path through the state space path.push(make\_pair(goal, 0));

// remember parentOf[start] = (start, 0)

while (parentOf[path.top().first].second != 0) path.push(parentOf[path.top().first]);



stack <pair <state, int> > path;

printf("Enter the capacities of the two jugs : "); scanf("%d %d", &capacity\_x, &capacity\_y); printf("Enter the target amount : "); scanf("%d", &target);

dfs((state) {0, 0}, path); if (path.empty())

printf("\nTarget cannot be reached.\n"); else

{

printf("\nNumber of moves to reach the target : %d\nOne path to the target is as follows :\n", path.size() - 1);

while (!path.empty())

{

state top = path.top().first; int rule = path.top().second; path.pop();

switch (rule)

{

case 0: printf("State : (%d, %d)\n#\n", top.x, top.y); break;

case 1: printf("State : (%d, %d)\nAction : Fill the first jug\n", top.x,top.y); break;

case 2: printf("State : (%d, %d)\nAction : Fill the second jug\n", top.x,top.y); break;

case 3: printf("State : (%d, %d)\nAction : Empty the first jug\n", top.x,top.y);

break;

case 4: printf("State : (%d, %d)\nAction : Empty the second jug\n", top.x,top.y); break;

case 5: printf("State : (%d, %d)\nAction : Pour from second jug into first



jug\n", top.x, top.y); break;

case 6: printf("State : (%d, %d)\nAction : Pour from first jug into second jug\n", top.x, top.y);

break;

}

}

}

return 0;

}

## Output:

Enter the capacities of the two jugs: 5 4 Enter the target amount: 3

Number of moves to reach the target: 4

One path to the target is as follows: State: (0, 0) Action: Fill the second jug State: (0, 4)

Action: Pour from second jug into first jug State: (4, 0) Action: Fill the second jug State: (4, 4)

Action: Pour from second jug into first jug State: (5, 3)

## References:

[1] <http://intelligence.worldofcomputing.net/ai-search/depth-first-search.html>



# PRACTICAL SET – 5

## Informed (Heuristic) Search Strategies

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

## Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these number of moves for all tiles.

## Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.



**Aim:** Write a program to implement Single Player Game (Using Heuristic Function)

## Code:

import java.util.List; import java.util.Random;

import kkukreja.algorithms.LinearCombinationHeuristic; import org.ggp.base.util.statemachine.MachineState; import org.ggp.base.util.statemachine.Move;

import org.ggp.base.util.statemachine.Role;

import org.ggp.base.util.statemachine.StateMachine;

Import org.ggp.base.util.statemachine.exceptions.GoalDefinitionException; import org.ggp.base.util.statemachine.exceptions.MoveDefinitionException; import org.ggp.base.util.statemachine.exceptions.TransitionDefinitionException; public final class Util

{

/\*\*

* @return (av. branching factor) ^ (av. game tree depth) for 20 randomly played games
* @throws TransitionDefinitionException
* @throws MoveDefinitionException

\*/

public static double complexityOfGameTree (StateMachine theMachine) throws MoveDefinitionException, TransitionDefinitionException



{

MachineState initialState = theMachine.getInitialState(); Random theRandom = new Random();

double avBranchingFactor = 0.0, avDepth = 0.0; int numNodes = 0;

final int iterations = 20;

for (int i = 0; i < iterations; i++)

{

MachineState state = initialState; while (!theMachine.isTerminal(state))

{

List<List<Move>> moves = theMachine.getLegalJointMoves(state); avBranchingFactor += moves.size(); numNodes++;

state = theMachine.getNextState(state, moves.get(theRandom.nextInt(moves.size())));

}

}

avBranchingFactor /= numNodes; avDepth = numNodes / (double) iterations;

return Math.pow(avBranchingFactor, avDepth);



}

/\*\*

* plays the game with several randomly chosen heuristic combinations to find the best one
* @param theMachine
* @return the best LinearCombinationHeuristic seen during simulated playoffs against random player
* @throws TransitionDefinitionException
* @throws MoveDefinitionException
* @throws GoalDefinitionException

\*/

public static LinearCombinationHeuristic findBestHeuristic (StateMachine theMachine, Role role, long timeout)

throws MoveDefinitionException, TransitionDefinitionException, GoalDefinitionException

{

long start = System.currentTimeMillis(); Random theRandom = new Random();

int roleIndex = theMachine.getRoleIndices().get(role); MachineState initialState = theMachine.getInitialState(); LinearCombinationHeuristic bestHeuristic = new LinearCombinationHeuristic(theMachine, role);

double bestScore = playoff (theMachine, role, initialState, roleIndex,



bestHeuristic, theRandom) - 10;

long timeTakenForFirstPlayOff = System.currentTimeMillis() - start;

int totalMatchesPossible = (int) (((timeout-start) / (timeTakenForFirstPlayOff)) - 2); int matches, rounds;

if (totalMatchesPossible > 10000)

{

matches = 1000; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 1000)

{

matches = 100; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 100)

{

matches = 10; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 50)

{

matches = 7; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 30)

{



matches = 5; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 10)

{

matches = 3; rounds = totalMatchesPossible / matches;

}

else if (totalMatchesPossible > 5)

{

matches = 2; rounds = totalMatchesPossible / matches;

}

else

{

matches = 1; rounds = totalMatchesPossible;

}

System.out.println("Total matches possible: " + totalMatchesPossible + " Rounds: " + rounds

+ " Matches per round: " + matches + " Time taken by first playoff: " + timeTakenForFirstPlayOff/1000.0 + " Total time: " + (timeout-start)/1000.0);

for (int round = 0; (round < rounds || (timeout-System.currentTimeMillis()-100)

> timeTakenForFirstPlayOff\*matches) && bestScore < 100.0; round++)

{

LinearCombinationHeuristic newHeuristic = new LinearCombinationHeuristic(theMachine, role); double score = 0.0;



for (int i = 0; i < matches; i++)

{

if ((System.currentTimeMillis()+200) > timeout) break;

score += playoff (theMachine, role, initialState, roleIndex, newHeuristic, theRandom);

}

score /= matches;

if (score > bestScore)

{

bestHeuristic = newHeuristic; bestScore = score;

}

}

long stop = System.currentTimeMillis();

System.out.println("Heuristic selected: " + bestHeuristic + " Best score: " + bestScore + " Time taken: " + (stop-start)/1000.0);

return bestHeuristic;

}

public static int playoff (StateMachine theMachine, Role role, MachineState initialState, int roleIndex, LinearCombinationHeuristic heuristic, Random theRandom) throws MoveDefinitionException, TransitionDefinitionException, GoalDefinitionException

{



MachineState state = initialState; while (!theMachine.isTerminal(state))

{

// find the best move against a 3-random move looking goal-directed player looking at 3 random moves according to the given heuristic List<Move> moves = theMachine.getLegalMoves(state, role); double bestScore = -1.0;

MachineState bestNextState = null; for (int j = 0; j < 3; j++)

{

Move m = moves.get(theRandom.nextInt(moves.size())); MachineState nextState = null;

int minscore = 101;

for (int i = 0; i < 3; i++)

{

MachineState randomstate = theMachine.getNextStateDestructively(state, theMachine.getRandomJointMove(state, role, m));

int score = theMachine.getGoal(randomstate, role); if (score < minscore)

{

minscore = score; nextState = randomstate;



}

}

double score = heuristic.evalState(nextState); if (score > bestScore)

{

bestScore = score; bestNextState = nextState;

}

}

state = bestNextState;

}

return theMachine.getGoal(state, role);

}

}

## Output:

References:

[1] <http://intelligence.worldofcomputing.net/ai-search/heuristic-search.html>



# PRACTICAL SET – 6

## What is A\* Search Algorithm?

A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

## Why A\* Search Algorithm?

Informally speaking, A\* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

## Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.

What A\* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

Algorithm We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A\* Search Algorithm



1. Initialize the open list
2. Initialize the closed list

put the starting node on the open list (you can leave its f at zero)

1. while the open list is not empty
   1. find the node with the least f on the open list, call it "q"
   2. pop q off the open list
   3. generate q's 8 successors and set their parents to q
   4. for each successor
      1. if successor is the goal, stop search successor.g = q.g + distance between successor and q successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h

* + 1. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
    2. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list end (for loop)
  1. push q on the closed list end (while loop)



**Aim :** Write a program to Implement A\* Algorithm.

## Code:

import java.util.\*; public class AStar

{

public static final int DIAGONAL\_COST = 14; public static final int V\_H\_COST = 10;

static class Cell

{

int heuristicCost = 0; //Heuristic cost int finalCost = 0; //G+H

int i, j;

Cell parent;

Cell(int i, int j)

{

this.i = i; this.j = j;

}

@Override

public String toString()

{



return "["+this.i+", "+this.j+"]";

}

}

//Blocked cells are just null Cell values in grid static Cell [][] grid = new Cell[5][5];

static PriorityQueue<Cell> open; static boolean closed[][];

static int startI, startJ; static int endI, endJ;

public static void setBlocked(int i, int j)

{

grid[i][j] = null;

}

public static void setStartCell(int i, int j)

{

startI = i; startJ = j;

}

static void checkAndUpdateCost(Cell current, Cell t, int cost)

{

if(t == null || closed[t.i][t.j]) return;



int t\_final\_cost = t.heuristicCost+cost; boolean inOpen = open.contains(t); if(!inOpen || t\_final\_cost<t.finalCost)

{

t.finalCost = t\_final\_cost; t.parent = current; if(!inOpen)

open.add(t);

}

}

public static void AStar()

{

//add the start location to open list. open.add(grid[startI][startJ]);

Cell current; while(true)

{

current = open.poll(); if(current==null) break;

closed[current.i][current.j]=true; if(current.equals(grid[endI][endJ]))



{

return;

}

Cell t;

if(current.i-1>=0)

{

t = grid[current.i-1][current.j];

checkAndUpdateCost(current, t, current.finalCost+V\_H\_COST); if(current.j-1>=0)

{

t = grid[current.i-1][current.j-1];

checkAndUpdateCost(current,t,current.finalCost+DIAGONAL\_COST

);

}

if(current.j+1<grid[0].length)

{

t = grid[current.i-1][current.j+1];

checkAndUpdateCost(current,t,current.finalCost+DIAGONAL\_COST

);

}

}

if(current.j-1>=0)

{



t = grid[current.i][current.j-1];

checkAndUpdateCost(current, t, current.finalCost+V\_H\_COST);

}

if(current.j+1<grid[0].length)

{

t = grid[current.i][current.j+1];

checkAndUpdateCost(current, t, current.finalCost+V\_H\_COST);

}

if(current.i+1<grid.length)

{

t = grid[current.i+1][current.j];

checkAndUpdateCost(current, t, current.finalCost+V\_H\_COST); if(current.j-1>=0)

{

t = grid[current.i+1][current.j-1];

checkAndUpdateCost(current,t,current.finalCost+DIAGONAL\_COST

);

}

if(current.j+1<grid[0].length)

{

t = grid[current.i+1][current.j+1];

checkAndUpdateCost(current, t, current.finalCost+DIAGONAL\_COST);



}

}

}

}

/\* Params :

tCase = test case No.

x, y = Board's dimensions

si, sj = start location's x and y coordinates ei, ej = end location's x and y coordinates int[][] blocked = array containing inaccessible cell coordinates

\*/

public static void test(int tCase, int x, int y, int si, int sj, int ei, int ej, int[][] blocked)

{

System.out.println("\n\nTest Case #"+tCase);

//Reset

grid = new Cell[x][y]; closed = new boolean[x][y];

open = new PriorityQueue<>((Object o1, Object o2) -> { Cell c1 = (Cell)o1; Cell c2 = (Cell)o2;

return c1.finalCost<c2.finalCost?-1: c1.finalCost>c2.finalCost?1:0;

});

//Set start position

setStartCell(si, sj); //Setting to 0,0 by default. Will be useful for the UI part setEndCell(ei, ej); for(int i=0;i<x;++i){



for(int j=0;j<y;++j){ grid[i][j] = new Cell(i, j); grid[i][j].heuristicCost = Math.abs(i-endI)+Math.abs(j-endJ);

// System.out.print(grid[i][j].heuristicCost+" ");

}

// System.out.println();

}

grid[si][sj].finalCost = 0;

for(int i=0;i<blocked.length;++i){ setBlocked(blocked[i][0], blocked[i][1]);

}

System.out.println("Grid: "); for(int i=0;i<x;++i){

for(int j=0;j<y;++j){ if(i==si&&j==sj)System.out.print("SO "); //Source

else if(i==ei && j==ej)System.out.print("DE "); //Destination else if(grid[i][j]!=null)System.out.printf("%-3d ", 0);

else System.out.print("BL ");

}

System.out.println();

}

System.out.println(); AStar();



System.out.println("\nScores for cells: "); for(int i=0;i<x;++i){

for(int j=0;j<x;++j){

if(grid[i][j]!=null)System.out.printf("%-3d ", grid[i][j].finalCost); else System.out.print("BL ");

}

System.out.println();

}

System.out.println();

if(closed[endI][endJ])

{

//Trace back the path System.out.println("Path: "); Cell current = grid[endI][endJ]; System.out.print(current);

while(current.parent!=null){ System.out.print(" -> "+current.parent); current = current.parent;

}

System.out.println();

}

else System.out.println("No possible path");

}



public static void main(String[] args) throws Exception{ test(1, 5, 5, 0, 0, 3, 2, new int[][]{{0,4},{2,2},{3,1},{3,3}});

test(2, 5, 5, 0, 0, 4, 4, new int[][]{{0,4},{2,2},{3,1},{3,3}});

test(3, 7, 7, 2, 1, 5, 4, new int[][]{{4,1},{4,3},{5,3},{2,3}});

test(1, 5, 5, 0, 0, 4, 4, new int[][]{{3,4},{3,3},{4,3}});

}

}

## Output :

Test Case #1 Grid:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SO | 0 | 0 | 0 | BL |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | BL | 0 | 0 |
| 0 | BL | DE | BL | 0 |
| 0 | 0 | 0 | 0 | 0 |

Scores for cells:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 14 | 27 | 41 | BL |
| 14 | 17 | 29 | 42 | 56 |
| 27 | 29 | BL | 45 | 59 |
| 39 | BL | 43 | BL | 0 |
| 52 | 55 | 0 | 0 | 0 |

Path:

[3, 2] -> [2, 1] -> [1, 1] -> [0, 0]

Test Case #2 Grid:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SO | 0 | 0 | 0 | BL |
| 0 | 0 | 0 | 0 | 0 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | BL | 0 | 0 |
| 0 | BL | 0 | BL | 0 |
| 0 | 0 | 0 | 0 | DE |

Scores for cells:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 17 | 33 | 48 | BL |
| 17 | 20 | 35 | 49 | 62 |
| 33 | 35 | BL | 52 | 64 |
| 48 | BL | 52 | BL | 67 |
| 62 | 65 | 64 | 67 | 77 |

Path:

[4, 4] -> [3, 4] -> [2, 3] -> [1, 2] -> [1, 1] -> [0, 0]

Test Case #3 Grid:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | SO | 0 | BL | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | BL | 0 | BL | 0 | 0 | 0 |
| 0 | 0 | 0 | BL | DE | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Scores for cells:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 40 | 35 | 37 | 40 | 53 | 68 | 0 |
| 22 | 17 | 20 | 34 | 48 | 63 | 0 |
| 17 | 0 | 15 | BL | 48 | 61 | 75 |
| 20 | 15 | 18 | 31 | 43 | 56 | 70 |
| 34 | BL | 31 | BL | 46 | 58 | 7 |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 48 | 48 | 43 | BL | 56 | 61 | 0 |

63 61 56 59 0 0 0

Path:

[5, 4] -> [4, 4] -> [3, 3] -> [3, 2] -> [2, 1]

Test Case #1 Grid:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SO | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | BL | BL |
| 0 | 0 | 0 | BL | DE |

Scores for cells:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 17 | 33 | 48 | 62 |
| 17 | 20 | 35 | 49 | 62 |
| 33 | 35 | 38 | 51 | 63 |
| 48 | 49 | 51 | BL | BL |
| 62 | 62 | 63 | BL | 0 |

No possible path

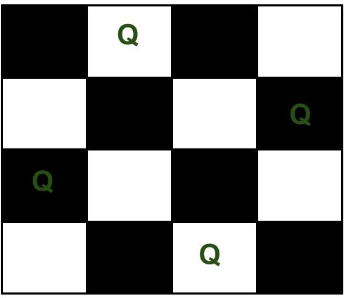
## References:

[1] <https://www.geeksforgeeks.org/a-search-algorithm/>



# PRACTICAL SET 7

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

## Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

while there are untried configurations

{

generate the next configuration

if queens don't attack in this configuration then

{

print this configuration;

}

}

## Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column,



if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1. Start in the leftmost column
2. If all queens are placed return true
3. Try all rows in the current column. Do following for every tried row.
   1. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   2. If placing the queen in [row, column] leads to a solution then return true.
   3. If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, return false to trigger backtracking.



**Aim :** Write a program to solve N – Queens problems using prolog.

## Code :

DOMAINS

cell=c(integer,integer) list=cell\* int\_ list=integer\*

PREDICATES

solution(list) member(integer,int\_list) nonattack(cell,list)

CLAUSES

solution([]).

solution([c(X,Y)|Others]):- solution(Others), member(Y,[1,2,3,4,5,6,7,8]), nonattack(c(X,Y),Others). nonattack(\_,[]).

nonattack(c(X,Y),[c(X1,Y1)|Others]):- Y<>Y1, Y1-Y<>X1-X, Y1-Y<>X-X1,

nonattack(c(X,Y),Others). member(X,[X|\_]). member(X,[\_|Z]):- member(X,Z).

GOAL solution([c(1,A),c(2,B),c(3,C),c(4,D),c(5,E),c(6,F),c(7,G),c(8,H)]).

## Output:



|  |  |
| --- | --- |
| A=4, B=2, C=7, D=3, E=6, F=8, G=5, H=1 A=5, B=2, C=4, D=7, E=3, F=8, G=6, H=1 A=3, B=5, C=2, D=8, E=6, F=4, G=7, H=1 A=3, B=6, C=4, D=2, E=8, F=5, G=7, H=1 A=5, B=7, C=1, D=3, E=8, F=6, G=4, H=2 A=4, B=6, C=8, D=3, E=1, F=7, G=5, H=2 A=3, B=6, C=8, D=1, E=4, F=7, G=5, H=2 A=5, B=3, C=8, D=4, E=7, F=1, G=6, H=2 A=5, B=7, C=4, D=1, E=3, F=8, G=6, H=2 A=4, B=1, C=5, D=8, E=6, F=3, G=7, H=2 A=3, B=6, C=4, D=1, E=8, F=5, G=7, H=2 A=4, B=7, C=5, D=3, E=1, F=6, G=8, H=2 A=6, B=4, C=2, D=8, E=5, F=7, G=1, H=3 A=6, B=4, C=7, D=1, E=8, F=2, G=5, H=3 A=1, B=7, C=4, D=6, E=8, F=2, G=5, H=3 A=6, B=8, C=2, D=4, E=1, F=7, G=5, H=3 A=6, B=2, C=7, D=1, E=4, F=8, G=5, H=3 A=4, B=7, C=1, D=8, E=5, F=2, G=6, H=3 A=5, B=8, C=4, D=1, E=7, F=2, G=6, H=3 A=4, B=8, C=1, D=5, E=7, F=2, G=6, H=3 A=2, B=7, C=5, D=8, E=1, F=4, G=6, H=3 A=1, B=7, C=5, D=8, E=2, F=4, G=6, H=3 A=2, B=5, C=7, D=4, E=1, F=8, G=6, H=3 A=4, B=2, C=7, D=5, E=1, F=8, G=6, H=3 A=5, B=7, C=1, D=4, E=2, F=8, G=6, H=3 A=6, B=4, C=1, D=5, E=8, F=2, G=7, H=3 A=5, B=1, C=4, D=6, E=8, F=2, G=7, H=3 A=5, B=2, C=6, D=1, E=7, F=4, G=8, H=3 A=6, B=3, C=7, D=2, E=8, F=5, G=1, H=4 A=2, B=7, C=3, D=6, E=8, F=5, G=1, H=4 A=7, B=3, C=1, D=6, E=8, F=5, G=2, H=4 A=5, B=1, C=8, D=6, E=3, F=7, G=2, H=4 A=1, B=5, C=8, D=6, E=3, F=7, G=2, H=4 A=3, B=6, C=8, D=1, E=5, F=7, G=2, H=4 A=6, B=3, C=1, D=7, E=5, F=8, G=2, H=4 A=7, B=5, C=3, D=1, E=6, F=8, G=2, H=4 A=7, B=3, C=8, D=2, E=5, F=1, G=6, H=4 A=5, B=3, C=1, D=7, E=2, F=8, G=6, H=4 A=2, B=5, C=7, D=1, E=3, F=8, G=6, H=4 A=3, B=6, C=2, D=5, E=8, F=1, G=7, H=4 A=6, B=1, C=5, D=2, E=8, F=3, G=7, H=4 A=8, B=3, C=1, D=6, E=2, F=5, G=7, H=4 | A=2, B=8, C=6, D=1, E=3, F=5, G=7, H=4 A=5, B=7, C=2, D=6, E=3, F=1, G=8, H=4 A=3, B=6, C=2, D=7, E=5, F=1, G=8, H=4 A=6, B=2, C=7, D=1, E=3, F=5, G=8, H=4 A=3, B=7, C=2, D=8, E=6, F=4, G=1, H=5 A=6, B=3, C=7, D=2, E=4, F=8, G=1, H=5 A=4, B=2, C=7, D=3, E=6, F=8, G=1, H=5 A=7, B=1, C=3, D=8, E=6, F=4, G=2, H=5 A=1, B=6, C=8, D=3, E=7, F=4, G=2, H=5 A=3, B=8, C=4, D=7, E=1, F=6, G=2, H=5 A=6, B=3, C=7, D=4, E=1, F=8, G=2, H=5 A=7, B=4, C=2, D=8, E=6, F=1, G=3, H=5 A=4, B=6, C=8, D=2, E=7, F=1, G=3, H=5 A=2, B=6, C=1, D=7, E=4, F=8, G=3, H=5 A=2, B=4, C=6, D=8, E=3, F=1, G=7, H=5 A=3, B=6, C=8, D=2, E=4, F=1, G=7, H=5 A=6, B=3, C=1, D=8, E=4, F=2, G=7, H=5 A=8, B=4, C=1, D=3, E=6, F=2, G=7, H=5 A=4, B=8, C=1, D=3, E=6, F=2, G=7, H=5 A=2, B=6, C=8, D=3, E=1, F=4, G=7, H=5 A=7, B=2, C=6, D=3, E=1, F=4, G=8, H=5 A=3, B=6, C=2, D=7, E=1, F=4, G=8, H=5 A=4, B=7, C=3, D=8, E=2, F=5, G=1, H=6 A=4, B=8, C=5, D=3, E=1, F=7, G=2, H=6 A=3, B=5, C=8, D=4, E=1, F=7, G=2, H=6 A=4, B=2, C=8, D=5, E=7, F=1, G=3, H=6 A=5, B=7, C=2, D=4, E=8, F=1, G=3, H=6 A=7, B=4, C=2, D=5, E=8, F=1, G=3, H=6 A=8, B=2, C=4, D=1, E=7, F=5, G=3, H=6 A=7, B=2, C=4, D=1, E=8, F=5, G=3, H=6 A=5, B=1, C=8, D=4, E=2, F=7, G=3, H=6 A=4, B=1, C=5, D=8, E=2, F=7, G=3, H=6 A=5, B=2, C=8, D=1, E=4, F=7, G=3, H=6 A=3, B=7, C=2, D=8, E=5, F=1, G=4, H=6 A=3, B=1, C=7, D=5, E=8, F=2, G=4, H=6 A=8, B=2, C=5, D=3, E=1, F=7, G=4, H=6 A=3, B=5, C=2, D=8, E=1, F=7, G=4, H=6 A=3, B=5, C=7, D=1, E=4, F=2, G=8, H=6 A=5, B=2, C=4, D=6, E=8, F=3, G=1, H=7 A=6, B=3, C=5, D=8, E=1, F=4, G=2, H=7 A=5, B=8, C=4, D=1, E=3, F=6, G=2, H=7 A=4, B=2, C=5, D=8, E=6, F=1, G=3, H=7 |

**References:**

[1] <https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/>



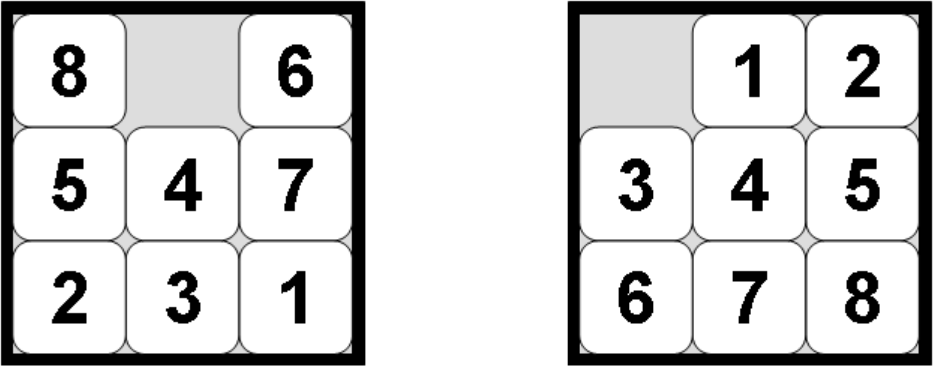
# PRACTICAL SET 8

The 8-puzzle is a smaller version of the slightly better known 15-puzzle. The puzzle consists of an area divided into a grid, 3 by 3 for the 8-puzzle, 4 by 4 for the 15-puzzle. On each grid square is a tile, expect for one square which remains empty. Thus, there are eight tiles in the 8-puzzle and 15 tiles in the 15-puzzle. A tile that is next to the empty grid square can be moved into the empty space, leaving its previous position empty in turn. Tiles are numbered, 1 thru 8 for the 8- puzzle, so that each tile can be uniquely identified.

The aim of the puzzle is to achieve a given configuration of tiles from a given (different) configuration by sliding the individual tiles around the grid as described above.

Searching for a Solution

This problem can be solved by searching for a solution, which is a sequence of actions (tile moves) that leads from the initial state to the goal state. Two possible states of the 8-puzzle are shown in figure. The state on the right is a typical goal state. The state on the left is a configuration that represents a worst case: transforming this state into the goal state requires at least 31 actions, which is the diameter of the search space. For search algorithms the problem is often to find the shortest solution, that is, one which consists of the least number of tile moves.





**Aim :** Write a program to solve 8 puzzle problem using Prolog. goal(1/2/3/8/0/4/7/6/5).

left( A/0/C/D/E/F/H/I/J , 0/A/C/D/E/F/H/I/J ). left( A/B/C/D/0/F/H/I/J , A/B/C/0/D/F/H/I/J ). left( A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/0/H/J ). left( A/B/0/D/E/F/H/I/J , A/0/B/D/E/F/H/I/J ). left( A/B/C/D/E/0/H/I/J , A/B/C/D/0/E/H/I/J ). left( A/B/C/D/E/F/H/I/0 , A/B/C/D/E/F/H/0/I ).

up( A/B/C/0/E/F/H/I/J , 0/B/C/A/E/F/H/I/J ). up( A/B/C/D/0/F/H/I/J , A/0/C/D/B/F/H/I/J ). up( A/B/C/D/E/0/H/I/J , A/B/0/D/E/C/H/I/J ). up( A/B/C/D/E/F/0/I/J , A/B/C/0/E/F/D/I/J ). up( A/B/C/D/E/F/H/0/J , A/B/C/D/0/F/H/E/J ). up( A/B/C/D/E/F/H/I/0 , A/B/C/D/E/0/H/I/F ).

right( A/0/C/D/E/F/H/I/J , A/C/0/D/E/F/H/I/J ). right( A/B/C/D/0/F/H/I/J , A/B/C/D/F/0/H/I/J ). right( A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/H/J/0 ). right( 0/B/C/D/E/F/H/I/J , B/0/C/D/E/F/H/I/J ). right( A/B/C/0/E/F/H/I/J , A/B/C/E/0/F/H/I/J ). right( A/B/C/D/E/F/0/I/J , A/B/C/D/E/F/I/0/J ).

down( A/B/C/0/E/F/H/I/J , A/B/C/H/E/F/0/I/J ). down( A/B/C/D/0/F/H/I/J , A/B/C/D/I/F/H/0/J ). down( A/B/C/D/E/0/H/I/J , A/B/C/D/E/J/H/I/0 ). down( 0/B/C/D/E/F/H/I/J , D/B/C/0/E/F/H/I/J ). down( A/0/C/D/E/F/H/I/J , A/E/C/D/0/F/H/I/J ). down( A/B/0/D/E/F/H/I/J , A/B/F/D/E/0/H/I/J ).



s\_fcn(A/B/C/D/E/F/G/H/I, S) :- s\_aux(A,B,S1), s\_aux(B,C,S2), s\_aux(C,F,S3),

s\_aux(F,I,S4), s\_aux(I,H,S5), s\_aux(H,G,S6),

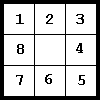
s\_aux(G,D,S7), s\_aux(D,A,S8), s\_aux(E,S9), S is S1+S2+S3+S4+S5+S6+S7+S8+S9.

s\_aux(0,0) :- !. s\_aux(\_,1).

s\_aux(X,Y,0) :- Y is X+1, !.

s\_aux(8,1,0) :- !. s\_aux(\_,\_,2).

## Output:



References:

[1] <http://www.aiai.ed.ac.uk/~gwickler/eightpuzzle-inf.html>



**PRACTICAL SET – 9**

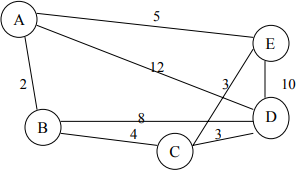
The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

The traveling salesman problem can be described as follows:

TSP = {(G, f, t): G = (V, E) a complete graph, f is a function V×V → Z, t ∈ Z,

G is a graph that contains a traveling salesman tour with cost that does not exceed t} Example:

Consider the following set of cities:



The problem lies in finding a minimal path passing from all vertices once. For example, the path Path1

{A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.



**Aim** : Write a program to solve travelling salesman problem using Prolog.

## Code :

class Convert\_pract1\_1{ public static

domains

/\* will allow us cooperate with better names, for me this is like #typedef in C++ \*/ town = symbol

distance = unsigned

rib = r(town,town,distance) tlist = town\* rlist = rib\*

predicates

nondeterm way(town,town,rlist,distance) nondeterm route(town,town,rlist,tlist,distance) nondeterm route1(town,tlist,rlist,tlist,distance) nondeterm ribsmember(rib,rlist)

nondeterm townsmember(town,tlist)

nondeterm tsp(town,town,tlist,rlist,tlist,distance) nondeterm ham(town,town,tlist,rlist,tlist,distance)

nondeterm shorterRouteExists(town,town,tlist,rlist,distance) nondeterm alltown(tlist,tlist)

nondeterm write\_list(tlist) clauses

/\*

Nothing special with write\_list. If list is empty we do nothing, and if something there we write head and call ourselves for tail.

\*/ write\_list([]).

write\_list([H|T]):-

write(H,‘ ‘),



write\_list(T).

/\* Is true if town X is in list of towns… \*/ townsmember(X,[X|\_]). townsmember(X,[\_|L]):- townsmember(X,L).

/\* Is true if rib X is in list of ribs… \*/ ribsmember(r(X,Y,D),[r(X,Y,D)|\_]). ribsmember(X,[\_|L]):- ribsmember(X,L).

/\* Is true if Route consists of all Towns presented in second argument \*/ alltown(\_,[]).

alltown(Route,[H|T]):- townsmember(H,Route), alltown(Route,T).

/\* Is true if there is a way from Town1 to Town2, and also return distance between them \*/ way(Town1,Town2,Ways,OutWayDistance):-

ribsmember(r(Town1,Town2,D),Ways), OutWayDistance = D.

%/\*

/\* If next is uncommented then we are using non-oriented graph\*/ way(Town1,Town2,Ways,OutWayDistance):- ribsmember(r(Town2,Town1,D),Ways), /\*switching direction here…\*/ OutWayDistance = D.

%\*/

/\* Is true if we could build route from Town1 to Town2 \*/ route(Town1,Town2,Ways,OutRoute,OutDistance):- route1(Town1,[Town2],Ways,OutRoute,T1T2Distance),

%SWITCH HERE



way(Town2,Town1,Ways,LasDist),

/\* If you want find shortest way comment this line\*/

OutDistance = T1T2Distance + LasDist. /\* And make this: OutDistance = T1T2Distance.\*/

route1(Town1,[Town1|Route1],\_,[Town1|Route1],OutDistance):- OutDistance = 0.

/\* Does the actual finding of route. We take new TownX town and if it is not member of PassedRoute, we continue searching with including TownX in the list of passed towns.\*/ route1(Town1,[Town2|PassedRoute],Ways,OutRoute,OutDistance):- way(TownX,Town2,Ways,WayDistance),

not(townsmember(TownX,PassedRoute)), route1(Town1,[TownX,Town2|PassedRoute],Ways,OutRoute,CompletingRoadDistance), OutDistance = CompletingRoadDistance + WayDistance. shorterRouteExists(Town1,Town2,Towns,Ways,Distance):- ham(Town1,Town2,Towns,Ways,\_,Other),

Other < Distance.

/\* calling tsp(a,a,…. picks any one connected to a town and calls another tsp\*/ tsp(Town1,Town1,Towns,Ways,BestRoute,MinDistance):- way(OtherTown,Town1,Ways,\_), tsp(Town1,OtherTown,Towns,Ways,BestRoute,MinDistance).

/\*Travelling Salesman Problem is Hammilton way which is the shortes of other ones.\*/ tsp(Town1,Town2,Towns,Ways,BestRoute,MinDistance):- ham(Town1,Town2,Towns,Ways,Route,Distance), not(shorterRouteExists(Town1,Town2,Towns,Ways,Distance)),

BestRoute = Route, MinDistance = Distance.

/\*Hammilton route from Town1 to Town2 assuming that Town2->Town1 way exists.\*/ ham(Town1,Town2,Towns,Ways,Route,Distance):- route(Town1,Town2,Ways,Route,Distance),

%SWITCH HERE

alltown(Route,Towns), % if you want simple road without including all towns you could uncomment



this line write\_list(Route),

write(” tD = “,Distance,“n“).

% fail.

goal

/\* EXAMPLE 1

AllTowns = [a,b,c,d],

AllWays = [r(a,b,1),r(a,c,10),r(c,b,2),r(b,c,2),r(b,d,5),r(c,d,3),r(d,a,4)],

\*/

/\* EXAMPLE 2 \*/ AllTowns = [a,b,c,d,e],

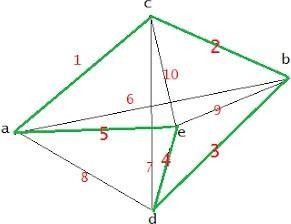
AllWays =[r(a,c,1),r(a,b,6),r(a,e,5),r(a,d,8),r(c,b,2),r(c,d,7),r(c,e,10),r(b,d,3),r(b,e,9),r(d,e,4)], tsp(a,a,AllTowns,AllWays,Route,Distance),

%SWITCH HERE

% tsp(a,b,AllTowns,AllWays,Route,Distance), write(“Finally:n“), write\_list(Route),

write(” tMIN\_D = “,Distance,“n“).

## Output :





**References:**

1. <https://www.csd.uoc.gr/~hy583/papers/ch11.pdf>
2. <https://brainly.in/question/2438847>
3. [https://www.google.com/search?q=%5B3%5D+https%3A%2F%2Fpeople.eecs.berkeley.edu%2F~vazirani](https://www.google.com/search?q=%5B3%5D%2Bhttps%3A%2F%2Fpeople.eecs.berkeley.edu%2F~vazirani%2Falgorithms%2Fchap6.pdf)

[%2Falgorithms%2Fchap6.pdf](https://www.google.com/search?q=%5B3%5D%2Bhttps%3A%2F%2Fpeople.eecs.berkeley.edu%2F~vazirani%2Falgorithms%2Fchap6.pdf)



# PRACTICAL SET – 10

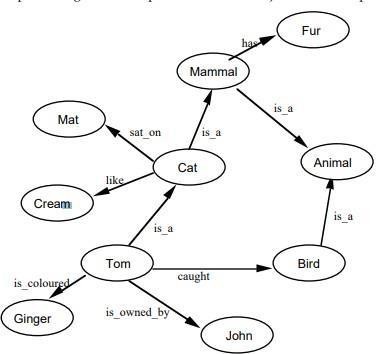
Semantic networks are an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects

For example, the following is intended to represent the data: Tom is a cat. Tom caught a bird. Tom is owned by John. Tom is ginger in colour. Cats like cream. The cat sat on the mat.

A cat is a mammal.

A bird is an animal.

All mammals are animals. Mammals have fur.





**Aim** : Convert following Prolog predicates into Semantic Net cat(tom).

cat(cat1). mat(mat1). sat\_on(cat1,mat1). bird(bird1). caught(tom,bird1).

like(X,cream) :– cat(X).

mammal(X) :– cat(X).

has(X,fur) :– mammal(X).

animal(X) :– mammal(X).

animal(X) :– bird(X). owns(john,tom). is\_coloured(tom,ginger).

## Output:-



**References:**

1. <http://www.eecs.qmul.ac.uk/~mmh/AINotes/AINotes4.pdf>
2. <https://www.sanfoundry.com/artificial-intelligence-mcqs-semantic-net-1/>