SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

ARTIFICIAL INTELLIGENCE LABORATORY

Course Code: INT 408

Semester: VII

Lab Manual

2024

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

Course Objective:

This course will help the learner to gain practical knowledge in various problem-solving approaches for artificially intelligent systems. It will also help the learner to practice knowledge representation and retrieval techniques.

Course Learning Outcomes:

Upon successful completion of this course, the learner will be able to

- Choose an appropriate Informed and uninformed search strategy for problem solving
- Demonstrate Local Search Algorithms for solving real world problems
- Implement Adversarial Search algorithms for games
- Choose appropriate ontology and logic for knowledge representation and inference

List of Experiments:

1. Solve path planning problem using Breadth First Search
2. Apply Depth First Search for searching attribute subset space
3. Use Iterative Deepening Depth limited search to solve 8-puzzle problem
4. Implement greedy search for searching attribute subset space
5. Implement A* algorithm for shortest path problem
6. Implement Genetic Algorithm for solving 8-queens' problem
7. Implement Alpha-Beta Pruning for solving Tic-Tac-Toe game
8. Solve Map Colouring Problem using Backtracking, by formulating the problem as Constraint Satisfaction Problem
9. Solve crypt arithmetic problem using ALLDIFF Constraints
10. Write simple programs for creating terms variables and atoms
11. Write simple FOL statements for Knowledge Representation
12. Implement Unification for sample terms

Exercise No. 1 Solve path planning problem using Breadth First Search

Objectives

To learn the basic concepts of BFS and learn how to do path finding for a solution starting from the initial state to the goal state, using BFS

Theory or Concept

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- Breadth-First Search algorithm is a graph traversing technique, where you select a random initial node (source or root node) and start traversing the graph layer-wise in such a way that all the nodes and their respective children nodes are visited and explored.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- This is achieved very simply by using a FIFO queue for the frontier.

Procedure / Algorithm

function BREADTH-FIRST-SEARCH (GRAPH) returns a solution, or failure

NODE = a node in GRAPH with STATE marked as INITIAL-STATE,

Initialize all NODES EXPLORED to true

NODE's PARENT = NULL

Path-Cost = 0

if GOAL-TEST (NODE's STATE) then return PRINT-SOLUTION (NODE, GRAPH)

FRONTIERQ ←a FIFO queue with node as the only element

loop do

if (FRONTIERQ is empty) then return failure

NODE = DEQUEUE (FRONTIERQ)

Add NODE's STATE to EXPLORED

for each adjacent node of NODE's STATE do

CHILD ←CHILD-NODE (graph, NODE)

if CHILD of STATE is not in EXPLORED or FRONTIERQ then

CHILD'S PARENT = NODE

if (CHILD's STATE is goal state) then

return PRINT-SOLUTION (CHILD, GRAPH)

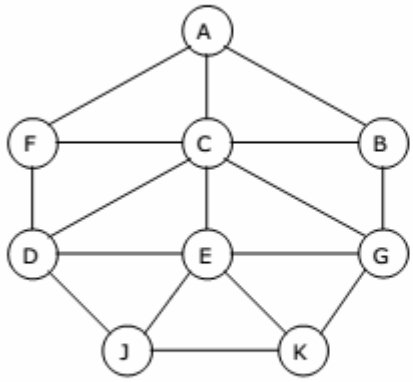
ENQUEUE (CHILD, FRONTIERQ)

Mark CHILD's EXPLORED as true

```
function PRINT-SOLUTION (NODE, GRAPH)
    if (NODE's PARENT is not NULL)
        PRINT-SOLUTION (NODE's PARENT, GRAPH)
        Print NODE's NAME "- - >"
```

Sample Input

Find a min path from A to K



A Graph G

Node	*Adjacency List*
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Output / Performance Measures

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2

Exercise No. 2 Apply Depth First Search for searching attribute subset space

Objectives

To learn the basic concepts of DFS and how to do path finding for a solution starting from the initial state to the goal state, using DFS

Theory or Concept

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.
- Depth-first search uses a LIFO queue.
- A LIFO queue means that the most recently generated node is chosen for expansion.
- This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

Procedure / Algorithm

function DEPTH-FIRST-SEARCH (GRAPH) returns a solution, or failure

return RECURSIVE-DFS(NODE with GRAPH's INITIAL-STATE), GRAPH)

function RECURSIVE-DFS (NODE, GRAPH) returns a solution, or failure

Add NODE to EXPLORED

if GOAL-TEST(NODE's STATE) then

PRINT-SOLUTION (NODE, GRAPH)

return success

else

for each adjacent CHILD (action) of the NODE do

if CHILD is not in EXPLORED then

CHILD's PARENT← NODE

CHILD'S ACTION ← action

RESULT ←RECURSIVE-DLS(CHILD, GRAPH)

if RESULT ≠ failure then return RESULT

return failure

function PRINT-SOLUTION (NODE, GRAPH)

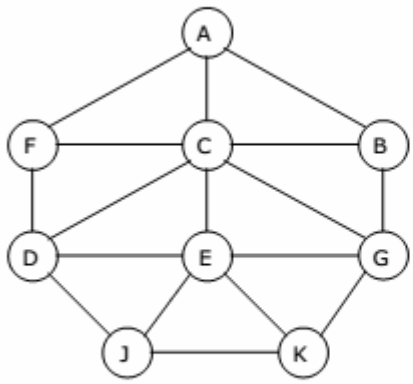
if (NODE's PARENT is not NULL)

PRINT-SOLUTION (NODE's PARENT, GRAPH)

Print NODE's NAME "- - >"

Sample Input

Find a min path from A to K



A Graph G

Node	*Adjacency List*
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Output / Performance Measures

Depth-first search and traversal:

The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3

Exercise No. 3 Solve 8-Puzzle problem using Iterative Deepening Depth Limited Search (IDDLS)

Objectives

To learn the basic concepts of IDDLS algorithm and learn how to find solution for 8-puzzle problem, starting from the initial state to the goal state, using IDDLS

Theory or Concept

- Iterative Deepening Depth Limited Search (ID-DFS or ID-DLS) is a search strategy that combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS).
- It is particularly useful in scenarios where the depth of the solution is unknown and memory is a concern.
- The 8-Queens problem involves placing eight queens on an 8x8 chessboard such that no two queens can attack each other.
- This means no two queens can be in the same row, column, or diagonal.
- Iterative Deepening Depth-Limited Search (ID-DLS) can be used to solve this problem by exploring the state space with increasing depth limits until a solution is found.

Procedure / Algorithm

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure

for DEPTH = 0 to ∞ do

RESULT ←DEPTH-LIMITED-SEARCH (problem, DEPTH)

if RESULT ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, LIMIT) returns a solution, or failure / cutoff

return RECURSIVE-DLS(NODE with problem's INITIAL-STATE), problem, LIMIT)

function RECURSIVE-DLS(NODE, problem, LIMIT) returns a solution, or failure / cutoff

if (NODE is of a GOAL.STATE) then

PRINT-SOLUTION (NODE, Graph)

Return success

else if LIMIT = 0 then return cutoff

else

CUTOFF-OCCURRED←false

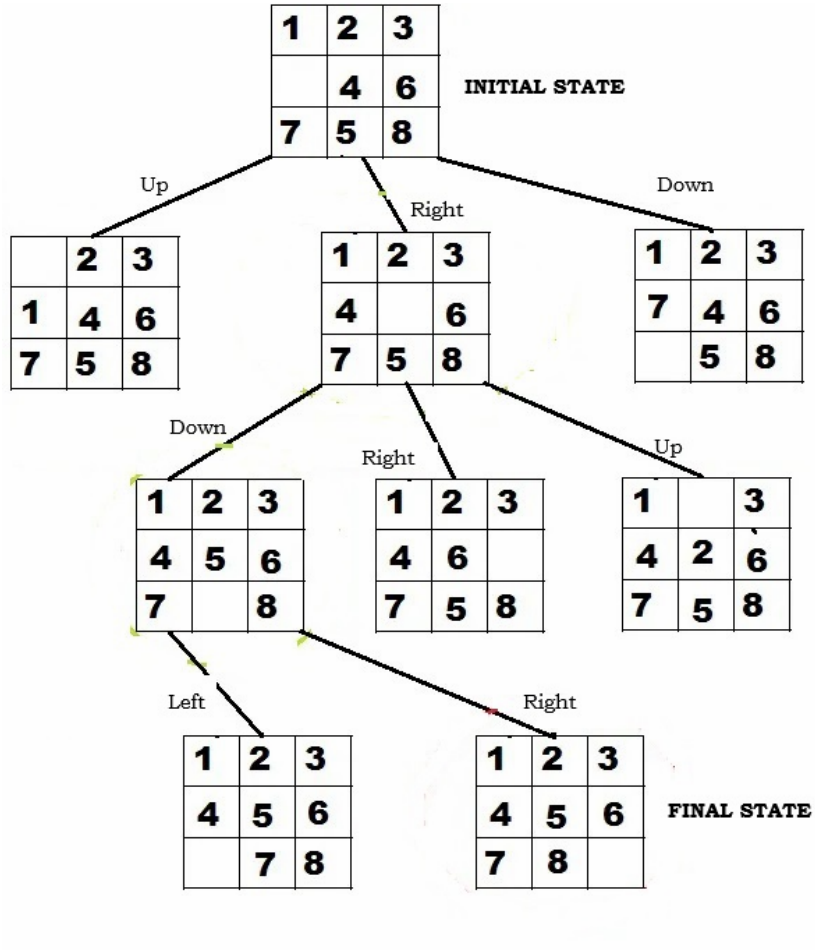
for each ACTION of NODE's STATE do

Create a CHILD node

```
        CHILD.STATE ←CONSTRUCT-STATE (NODE.STATE, ACTION)
        If (CHILD.STATE ≠ failure )
            RESULT ←RECURSIVE-DLS (CHILD , problem, LIMIT − 1)
             if RESULT = cutoff then CUTOFF OCCURRED←true
            else if result ≠ failure then return result
            if CUTOFF OCCURRED then return cutoff else return failure


function PRINT-SOLUTION (NODE, GRAPH)
    if (NODE's PARENT is not NULL)
         PRINT-SOLUTION (NODE's PARENT, GRAPH)
         Print NODE's STATE ( NODE's ACTION) "- - >"
function CONSTRUCT-STATE( CURR-STATE, ACTION) returns new STATE / failure
      [I, J] = FIND-EMPTY-TILE(CURR-STATE)
      If ( I == 1 and ACTION == UP ) OR (I == 3 and ACTION == DOWN) or (J == 1 and
ACTION == LEFT ) or (J = 3 and ACTION == RIGHT)
        Return failure
      else if ( ACTION == LEFT)
        EXCHANGE( CURR-STATE[I, J-1], CURR-STATE[I, J] )
      else if ( ACTION == RIGHT)
        EXCHANGE( CURR-STATE[I, J+1, CURR-STATE[I, J] )
      else if ( ACTION == UP)
        EXCHANGE( CURR-STATE[I-1, J], CURR-STATE[I, J] )
      else if ( ACTION == DOWN)
        EXCHANGE( CURR-STATE[I+1, J], CURR-STATE[I, J] )
      Return CURR-STATE
```

Sample Input and output



Exercise No. 4 Implement greedy search for searching attribute subset space

Objectives

To learn the basic concepts of greedy algorithm and about heuristics. Also, to learn how to find solution for path finding problem, starting from the initial state to the goal state, using greedy algorithm.

Theory or Concept

Heuristic Function (h(n)): GBFS uses a heuristic function

h(n) to estimate the cost from the current node n to the goal. The heuristic is problem-specific and provides a measure of the desirability of each node.

Priority Queue: Nodes are stored in a priority queue (often implemented as a min-heap), where the priority of each node is determined by its heuristic value

h(n). The node with the smallest heuristic value is expanded first.

Key Concepts

g(n): The cost of the path from the start node to node n

h(n): A heuristic estimate of the cost from node n to the goal node.

Evaluation function : f(n) = h(n)

Procedure / Algorithm

function A-STAR-SEARCH(GRAPH) returns a solution, or failure

```
NODE ← a node with STATE = GRAPH.INITIAL-STATE
NODE.PATH-COST = H(NODE)
FRONTIER ←a priority queue (ordered by F (NODE) =  H(NODE) ),
                with NODE as the only element
EXPLORED ←an empty set
loop do
    if EMPTY? ( FRONTIER) then return failure
    NODE ← POP (FRONTIER) /* chooses the lowest-F-VALUE node in frontier */
    if GRAPH.GOAL-TEST (NODE.STATE)
    then return PRINT-SOLUTION (NODE, GRAPH)
    Add NODE.STATE to EXPLORED
    for each adjacent node of NODE.STATE do
       CHILD ←CHILD-NODE(GRAPH, NODE.STATE)
       CHILD.F-VALUE ← H(CHILD)
```

if CHILD.STATE is not in EXPLORED or FRONTIER then

FRONTIER ←INSERT(CHILD , FRONTIER)

else if CHILD.STATE is in FRONTIER with higher F-VALUE then

replace that node in the FRONTIER with CHILD

function PRINT-SOLUTION (NODE, GRAPH)

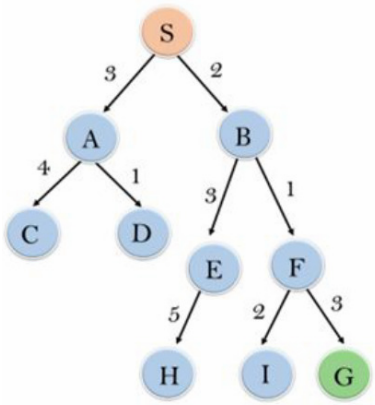
if (NODE's PARENT is not NULL)

PRINT-SOLUTION (NODE's PARENT, GRAPH)

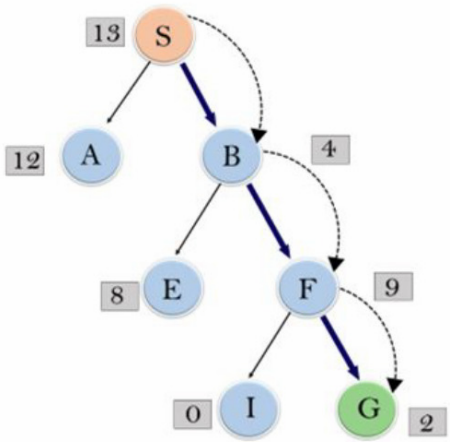
Print NODE's STATE (NODE's ACTION) "- - >"

Sample Input

Diagram: Greedy Best First Search



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



Pathcost = 2+1+3 = 6

Exercise No. 5 Solve Path finding problem using A* Search

Objectives

To learn the basic concepts of A* algorithm and about heuristics. Also, to learn how to find solution for path finding problem, starting from the initial state to the goal state, using A*

Theory or Concept

- The A* (A-star) algorithm is a popular pathfinding and graph traversal algorithm, often used in many fields of computer science due to its completeness, optimality, and efficiency.
- It finds the shortest path from a start node to a goal node, using a heuristic to prioritize nodes.
- A* combines the benefits of Dijkstra's algorithm (which considers the cost from the start to a given node) and Greedy Best-First-Search (which considers the estimated cost from a given node to the goal).

Key Concepts

g(n): The cost of the path from the start node to node n

h(n): A heuristic estimate of the cost from node n to the goal node.

f(n) = g(n) + h(n): The estimated total cost of a path going through node n

Procedure / Algorithm

function A-STAR-SEARCH(GRAPH) returns a solution, or failure

NODE ← a node with STATE = GRAPH.INITIAL-STATE

NODE.PATH-COST = H(NODE)

FRONTIER ←a priority queue (ordered by F (NODE) = NODE.PATHCOST + H(NODE)),

with NODE as the only element

EXPLORED ←an empty set

loop do

if EMPTY? (FRONTIER) then return failure

NODE ← POP (FRONTIER) /* chooses the lowest-F-VALUE node in frontier */

if GRAPH.GOAL-TEST (NODE.STATE)

then return PRINT-SOLUTION (NODE, GRAPH)

Add NODE.STATE to EXPLORED

for each adjacent node of NODE.STATE do

CHILD ←CHILD-NODE(GRAPH, NODE.STATE)

CHILD.F-VALUE ← NODE.PATHCOST + EDGECOST(NODE, CHILD)

+ H(CHILD)

if CHILD.STATE is not in EXPLORED or FRONTIER then

FRONTIER ←INSERT(CHILD , FRONTIER)

else if CHILD.STATE is in FRONTIER with higher F-VALUE then

replace that node in the FRONTIER with CHILD

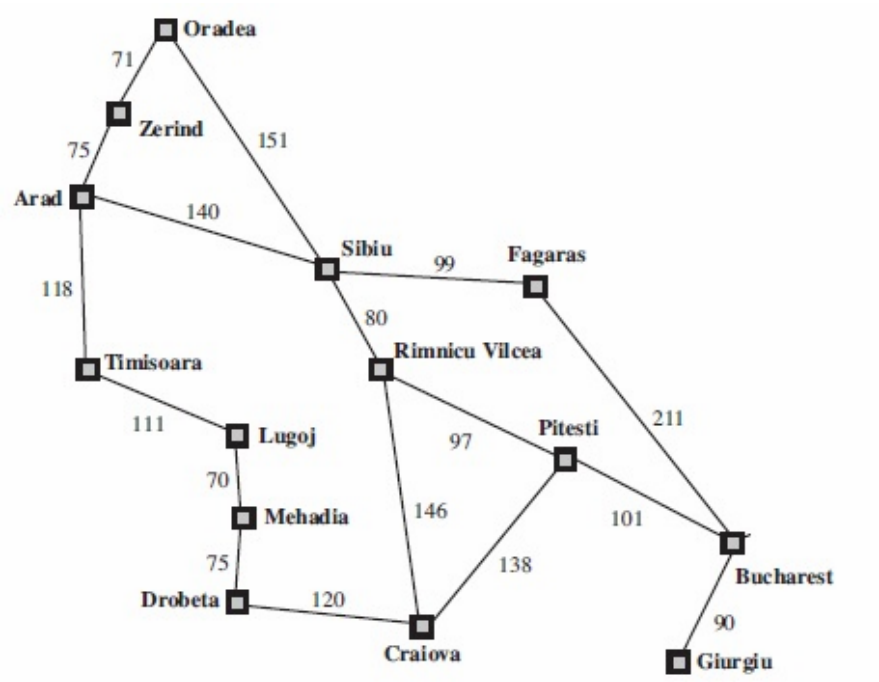
function PRINT-SOLUTION (NODE, GRAPH)

if (NODE's PARENT is not NULL)

PRINT-SOLUTION (NODE's PARENT, GRAPH)

Print NODE's STATE (NODE's ACTION) "- - >"

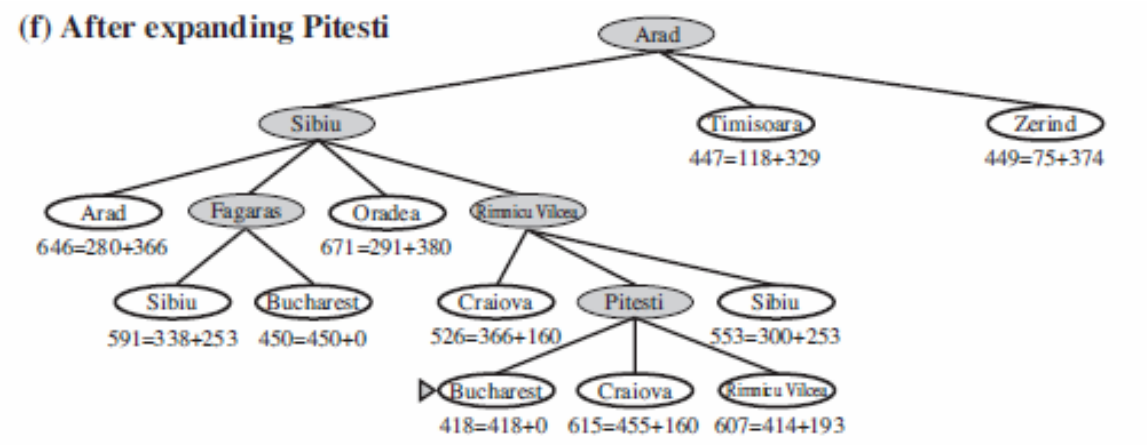
Sample Input



- **Take only necessary h(n) values from the table below:**

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Output / Performance Measures



Exercise No. 6 Implement Genetic Algorithm for solving 8-queens' problem

Objectives

To learn the basic concepts of Genetic algorithm with operators' mutation, selection and cross over. final solution that no queens should share common row and column

Theory or Concept

- A genetic algorithm (GA) is a search heuristic inspired by natural selection.
- It generates high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover, and selection.
- The GA starts with a population of randomly generated solutions and evolves this population over successive generations to improve the quality of solutions.

Algorithms:

function GENETIC-ALGORITHM (POPULATION) returns an individual

inputs: POPULATION, a set of individuals given as chromosomes along with their fitness function values to be calculated

for i = 1 to POPULATION.size do

POPULATION[i].Fitness_Value = FITNESS-FUNCTION (POPULATION[I].Chromosome)

repeat

NEW-POPULATION ←empty set

for i = 1 to (SIZE(population) / 2) do

x ←RANDOM-SELECTION (POPULATION)

y ←RANDOM-SELECTION(POPULATION)

CHILD1 , CHILD2←REPRODUCE(x , y)

if (small random probability) then CHILD1 ←MUTATE(child)

if (small random probability) then CHILD2 ←MUTATE(child)

add CHILD1 & CHILD2 to NEW-POPULATION

POPULATION ← NEW-POPULATION

until some individual is fit enough, or enough time has elapsed

return the best individual in population, according to FITNESS-FN and also the number of iterations

function FITNESS-FUNCTION(x) returns a fitness function value

input: x is a chromosome in the POPULATION

FITNESSVALUE = 0

for i = 1 to 7

for j = i+1 to 8

if (x[i] ≠ x[j] and (| x[j] – x[i] | ≠ j – I)

FITNESSVALUE = FITNESSVALUE + 1

return FITNESSVALUE

function RANDOM-SELECTION (POPULATION) returns an individual

input: POPULATION contains possible candidates for the reproduction

Select POPULATION[i] with probability proportional to POPULATION[i].Fitness_Value

function REPRODUCE(x , y) returns an individual

inputs: x , y, parent individuals

n←LENGTH (x); c ← random number between 1 and n

C1 = APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))

C2 = APPEND(SUBSTRING(y, 1, c), SUBSTRING(x, c + 1, n))

return C1 and C2

function MUTATE(x) **returns** an individual

n←LENGTH (x); c ← random number between 1 and n

m ← a random number between 1 and 8 different from x[c]

x[c] ← m

return x

Sample Input

Randomly generated four (n=4) chromosomes each representing the queens positions as described above

Sample Output

Optimal arrangements of non-attacking queens obtained or the chromosome with best fitness value after 20 iterations

Exercise No. 7 Implement Alpha-Beta Pruning for solving Tic-Tac-Toe game

Objectives

To learn the basic concepts of Alpha-Beta Pruning and its steps. Also, to learn how to represent complete arrangement of Tic-Tac-Toe instance for a game tree

Theory or Concept

- Alpha-Beta Pruning is an optimization technique for the Minimax algorithm. Minimax is a decision rule used to minimize the possible loss in a worst-case scenario.
- When making a sequence of decisions, the algorithm can evaluate all possible moves and choose the best one.
- However, the number of game states like Tic-Tac-Toe can be large. Alpha-Beta Pruning reduces the number of nodes evaluated by the Minimax algorithm in its search tree.

Algorithm

function ALPHA-BETA-FOR-TICTACTOE( ) returns an action

GRAPH = CONSTRUCT-GRAPH ( )

v ←MAX-VALUE(INIT-STATE, GRAPH, SYS-MIN-VALUE, SYS-MAX-VALUE)

return the action in ACTIONS(state) with value v

function MAX-VALUE(STATE,GRAPH, α, β) returns a utility value

if TERMINAL-TEST(state, 1) then return UTILITY(state)

v ← - ∞

for each a in ACTIONS(state) do

v ←MAX(v, MIN-VALUE(RESULT(s,a),α, β))

if v ≥ β then return v

α←MAX(α, v)

return v

function MIN-VALUE(state,α, β) returns a utility value

if TERMINAL-TEST(state, 2) then return UTILITY(state)

v ←+∞

for each a in ACTIONS(state) do

v ←MIN(v, MAX-VALUE(RESULT(s,a) ,α, β))

if v ≤ α then return v

β←MIN(β, v)

return v

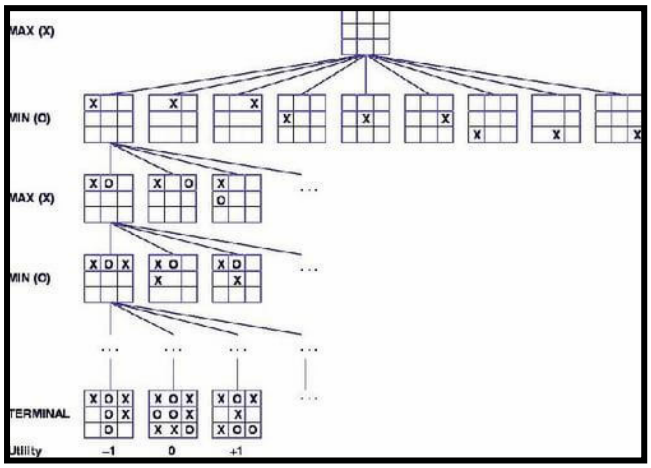
function TERMINAL-TEST(STATE, PLAYER) returns true or false

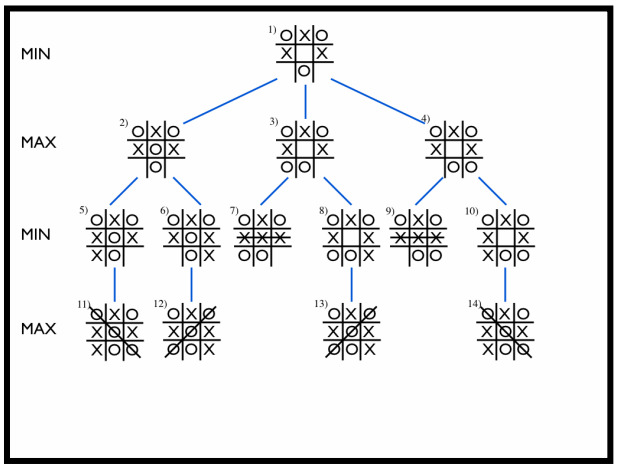
I/P: PLAYER = 1 indicates MAX-PLAYER AND PLAYER = 2 indicates MIN playe

```
if  (WINNING(STATE, PLAYER) = True)
    return true
else if ( DRAW (STATE) )
        return true
else return false
```

Sample Input

A graph with an empty Tic Tac Toe board described above, as root. an incremental construction of nodes in each level is advisable





Sample Output :

Sequence of optimal Move for the Max Player and the corresponding move for the MIN player.

Exercise No. 8 Solve Map Colouring Problem using Backtracking, by formulating the problem as Constraint Satisfaction Problem

Objectives

To learn the basic concepts of Backtracking. Also, to learn how to represent an instance of map colouring as a graph and how to represent constraints in an efficient way

Theory or Concept

- The Map Coloring Problem is a classic example of a Constraint Satisfaction Problem (CSP).
- In this problem, the goal is to color a map in such a way that no two adjacent regions (e.g., countries, states) share the same color, using a limited number of colors. This problem can be formulated as a CSP, where:

Variables: Regions on the map.

Domains: Set of colors available for coloring.

Constraints: No two adjacent regions can have the same color.

- Backtracking is a recursive algorithm used to solve CSPs. It tries to build a solution incrementally, one variable at a time, and backtracks as soon as it determines that the current partial assignment cannot lead to a valid solution.

ALGORITHM

function BACKTRACKING-SEARCH (CSP) returns a solution, or failure

```
ASSIGNMENT = INIT-ASSIGN(CSP.VARIABLES)
return BACKTRACK (ASSIGNMENT, CSP)
```

function BACKTRACK(ASSIGNMENT, CSP) returns a solution, or failure

```
if IS-COMPLETE (ASSIGNMENT) then return ASSIGNMENT
VAR ←SELECT-UNASSIGNED-VARIABLE(CSP)
for each value in ORDER-DOMAIN-VALUES (VAR, ASSIGNMENT, CSP) do
        if IS-CONSISTENT(VALUE, ASSIGNMENT) then
                add {VAR = VALUE} to ASSIGNMENT
                INFERENCES ←INFERENCE(CSP, VAR , VALUE)
                 if INFERENCES  ≠   failure then
                        RESULT ←BACKTRACK (ASSIGNMENT, CSP)
                        if RESULT ≠  failure then
                            return result
        remove {VAR = VALUE} and INFERENCES from ASSIGNMENT
return failure
```

function INIT-ASSIGN(VARIABLES) returns assignment

create a list ASSIGNMENT with VARS, VALS
for each V in VARAIBLES
ASSIGNMENT.VARIABLE = V
ASSIGNMENT.VALUE = NIL
Return ASSIGNMENT

function IS-COMPLETE (ASSIGNMENT) returns True or False

for all **ASSIGNMENT.VALUEs** do
if any VAUE is NULL, return False
return True

function SELECT-UNASSIGNED-VARIABLE(CSP) retuns VAR

MIN_DOMAIN_SIZE = ∞
for each variable X in CSP with ASSIGNMENT VALUE = NIL do
if (X.DOMAINSIZE < MIN_DOMAIN_SIZE)
MIN-DOMAIN-VAR = X
return MIN-DOMAIN-VAR

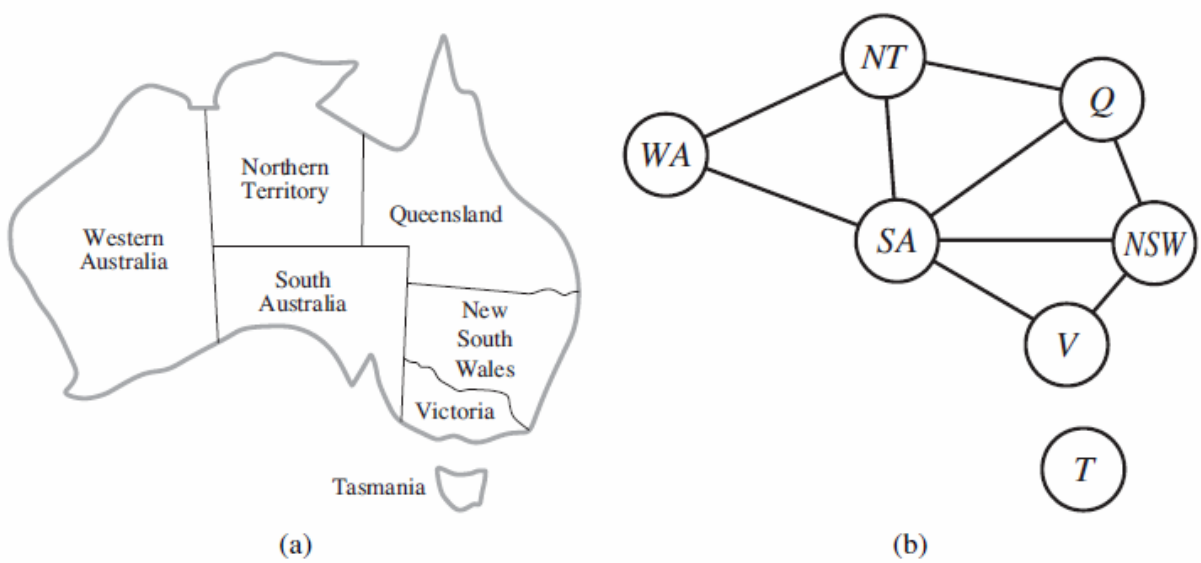
function IS_CONSISTENT(ASSIGNMENT, CSP)

for each CONSTRAINT in CSP
if any pair of vars assignments in ASIGNMENT are not satisfying the CONSTRAINT
return **False**
return **True**

function INFERENCE(CSP, VAR , VALUE) returns failure or success

for each variable v of CSP with VALUE = NIL and is in a constraint along with VAR do
Make the DOMAIN of V consistent with the VALUE with respect to the constraint
if (SIZEOF(DOMAIN of V) = 0)
return **failure**
return **success**

Sample Input



Sample Output

Any Solution with each variable getting a colour in the form:

{ {Var1 = Val1}, (Var2 = Val2}, ….}

Exercise No. 9 Solve crypt arithmetic problem using ALLDIFF Constraints

Objective

To learn the basic concepts of cryptarithmetic puzzle solver in python.

Theory or Concept

- Crypt arithmetic is a type of mathematical puzzle in which the digits are replaced by letters of the alphabet. The challenge is to find the unique digits that correspond to each letter to satisfy the given mathematical equation.
- The ALLDIFF constraint ensures that all variables (letters) must take different values. This is particularly useful in crypt arithmetic problems where each letter must represent a unique digit from 0 to 9.

Algorithm

```
function solveCryptarithmetic(problem):
    letters = extractUniqueLetters(problem)
    usedDigits = array of 10 booleans initialized to False
    mapping = dictionary from letters to digits initialized to None

    if solve(letters, usedDigits, mapping, 0, problem):
        printSolution(mapping)
    else:
        print("No solution found")

function solve(letters, usedDigits, mapping, index, problem):
    if index == length(letters):
        return isValid(mapping, problem)
    for digit from 0 to 9:
        if not usedDigits[digit]:
            usedDigits[digit] = True
            mapping[letters[index]] = digit
            if solve(letters, usedDigits, mapping, index + 1, problem):
                return True
            usedDigits[digit] = False
            mapping[letters[index]] = None
```

return False

function isValid(mapping, problem):

evaluate the problem using the current mapping

return True if the problem is satisfied, otherwise False

function extractUniqueLetters(problem):

extract and return the unique letters from the problem

function printSolution(mapping):

print the solution

Sample Input

```
  T W O
+ T W O
 -------
F O U R
```

Sample Output

Any Solution with each alphabet getting a distinct digit:

{ {T = dig1}, (W = dig2}, ….}

Exercise No. 10 Write simple programs for creating terms variables and atoms

Objective

To learn the symbols, terms variables from normal English sentence to FOL

Theory or basics concepts

Constants KingJohn; 2; UCB; : : :

Predicates Brother; >; : : :

Functions Sqrt; LeftLegOf; : : :

Variables x; y; a; b; : : :

Connectives ^ _ :) ,

Equality =

Quantifiers Universal , Existential

Sample Input

- Constant symbols to objects in D

  I(*John*) = 

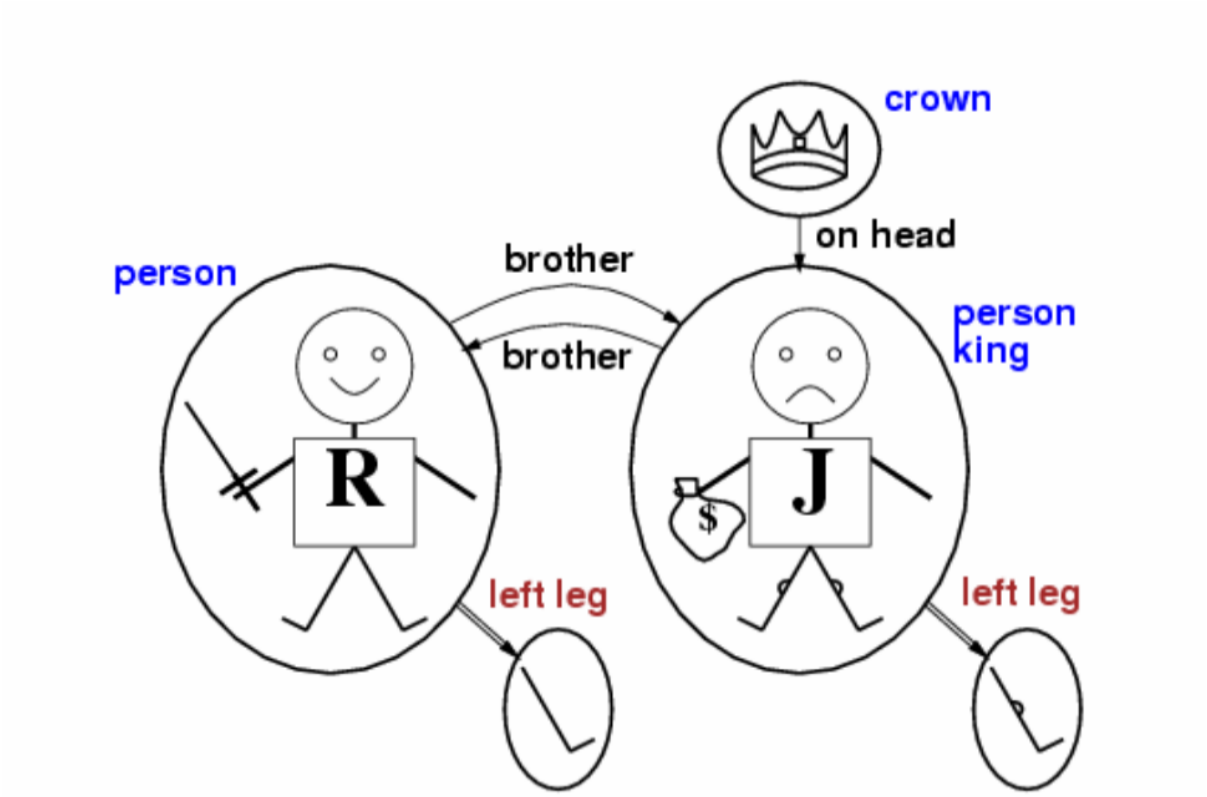
- Predicate symbols to relations, properties on D

  I(*brother*) = 

- Function symbols to functional relations on D

  I(*father-of*) = 

Sample Output



Exercise No. 11 Write simple FOL statements for Knowledge Representation

Objective

To learn the basic concepts of Knowledge Base creation in python, using TELL function

Theory or Concept

- Objects - people
- Two unary predicates - Male and Female
- Kinship relations—parenthood, brotherhood, marriage, and so on—represented by binary predicates:
  - Parent, Sibling, Brother , Sister , Child , Daughter, Son, Spouse, Wife, Husband, Grandparent , Grandchild , Cousin, Aunt, and Uncle
- Functions - Mother and Father

Sample Input :

A grandparent is a parent of one's parent
A sibling is another child of one's parents
One's mother is one's female parent
One's husband is one's male spouse
Male and female are disjoint categories
Parent and child are inverse relations

Sample Output

$\forall$ g, c Grandparent (g, c) $\Leftrightarrow \exists$p Parent(g, p) $\wedge$ Parent(p, c)

$\forall$ x, y Sibling(x, y) $\Leftrightarrow$ x = y $\wedge \exists$p Parent(p, x) $\wedge$ Parent(p, y)

$\forall$ m, c Mother (c)=m $\Leftrightarrow$ Female(m) $\wedge$ Parent(m, c)

$\forall$ w, h Husband(h,w) $\Leftrightarrow$ Male(h) $\wedge$ Spouse(h,w)

$\forall$ x Male(x) $\Leftrightarrow \neg$Female(x)

$\forall$ p, c Parent(p, c) $\Leftrightarrow$ Child (c, p)

$\forall$ x, y Sibling(x, y) $\Leftrightarrow$ Sibling(y, x)

Exercise No. 12 Use Unification algorithm to unify the given pair of Logical expressions

Objective

To learn the implementation of unification in the given Knowledge Base

Procedure

function UNIFY(x , y, θ) returns a substitution to make x and y identical

inputs: x, a string storing a variable, constant, list, or compound expression
y, a string storing a a variable, constant, list, or compound expression
θ, the substitution built up so far (optional, defaults to empty)

if θ = failure **then return** failure
else if x = y **then return** θ
else if VARIABLE(x) **then**
return UNIFY-VAR(x , y, θ)
else if VARIABLE(y) **then**
return UNIFY-VAR(y, x , θ)
else if COMPOUND(x) **and** COMPOUND(y) **then**
return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
else if LIST(x) **and** LIST(y) **then**
return UNIFY(x .REST, y.REST, UNIFY(x .FIRST, y.FIRST, θ))
else return failure

function UNIFY-VAR(var, x , θ) returns a substitution

if {var/val} ∈ θ **then**
return UNIFY(val , x , θ)
else if {x/val} ∈ θ **then**
return UNIFY(var, val , θ)
else if OCCUR-CHECK(var, x) **then**
return failure
else return add {var/x } to θ

Sample Input

Unify the following Statements

Prime(13) and Prime(y)
Knows(John, x) and Knows(y, Mother(y))
pro(b, X, f(g(Z))) and pro(Z, f(Y), f(Y))
Quick(a, g(x, a), f(y)) and Quick(a, g(f(b), a), x)

Sample Output

Θ = {y/13}

Θ = { {y/John, x/Mother(John) }
Θ = { Z/b, X/f(Y), Y/g(b) }
Θ = { a/a, x/f(b), y/b}