

INDEX

Name Divesh Sah

Standard 5 Section B Roll No. 092

Subject Artificial Intelligence Lab.

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	01-Oct-2024	Vacuum cleaner; Tic-Tac-Toe.	01-07	81/100
02.	8-Oct-2024	8-puzzle problem - DFS & BFS	8-12	S.Prabu 8/10/24 (10/10)
03.	15-Oct-2024	A* search for 8-puzzle with Misplaced Tiles & Manhattan Distance	13-16	S.Prabu 15/10/24 (10/10)
04.	22-Oct-2024	Hill Climbing Search Algorithm.	17-19	S.Prabu 22/10/24 (10/10)
05.	29-Oct-2024	Stimulated Annealing to solve N-Queens Problem.	20-25	(10/10) S.Prabu
06.	12-Nov-2024	Propositional Logic.	21-22	S.Prabu 12/11/24 (10/10)
07.	19-Nov-2024	Unification in First Order logic	23-24	(10/10) S.Prabu
08.	26-Nov-2024	Forward Reasoning in FOL	27-28	(10/10) S.Prabu 26/11/24 (10/10)

Lab - 1

1) Vacuum Cleaner.

Algorithm :

1. Initialize goal state:

- Set goal-state = { 'A': '0'; 'B': '0' }
(0 = clean, 1 = dirty)
- Initialize cost = 0.

2. Input the vacuum location:

- Get vacuum's location:
location_input = 'A' or 'B'.
- Get status of the current room:
status_input = '0' or '1'.
- Get status of the other room:
status_input_complement = '0' or '1'.

3. Vacuum in Room A:

- If location_input == 'A':
 - If status_input == '1' (A is dirty):
 - Clean A, set goal-state['A'] = '0', increment cost.
 - If status_input_complement == '1'
(B is dirty):
 - Move to B, clean it.
 - update goal-state['B'] = '0'
 - increment cost.

4. Vacuum in Room B:

- If location_input == 'B':
 - If status_input == '1' (B is dirty):

- Clean B, set goal-state [‘B’] = ‘0’,
• increment cost.
- If status-input-complement == ‘1’
(A is dirty):
 - Move to A, clean it
 - update goal-state [‘A’] = ‘0’.
 - increment cost.

5. Output :

- print goal-state and cost as the performance measure.

Output: Initial condition: {‘A’: ‘1’, ‘B’: ‘1’}

Vacuum is placed in location A.
Location A is dirty.

Cost for cleaning A: 1

Location A has been cleaned.

Location B is Dirty.

Moving right to location B.

Cost for moving right: 2

Cost for cleaning: 3

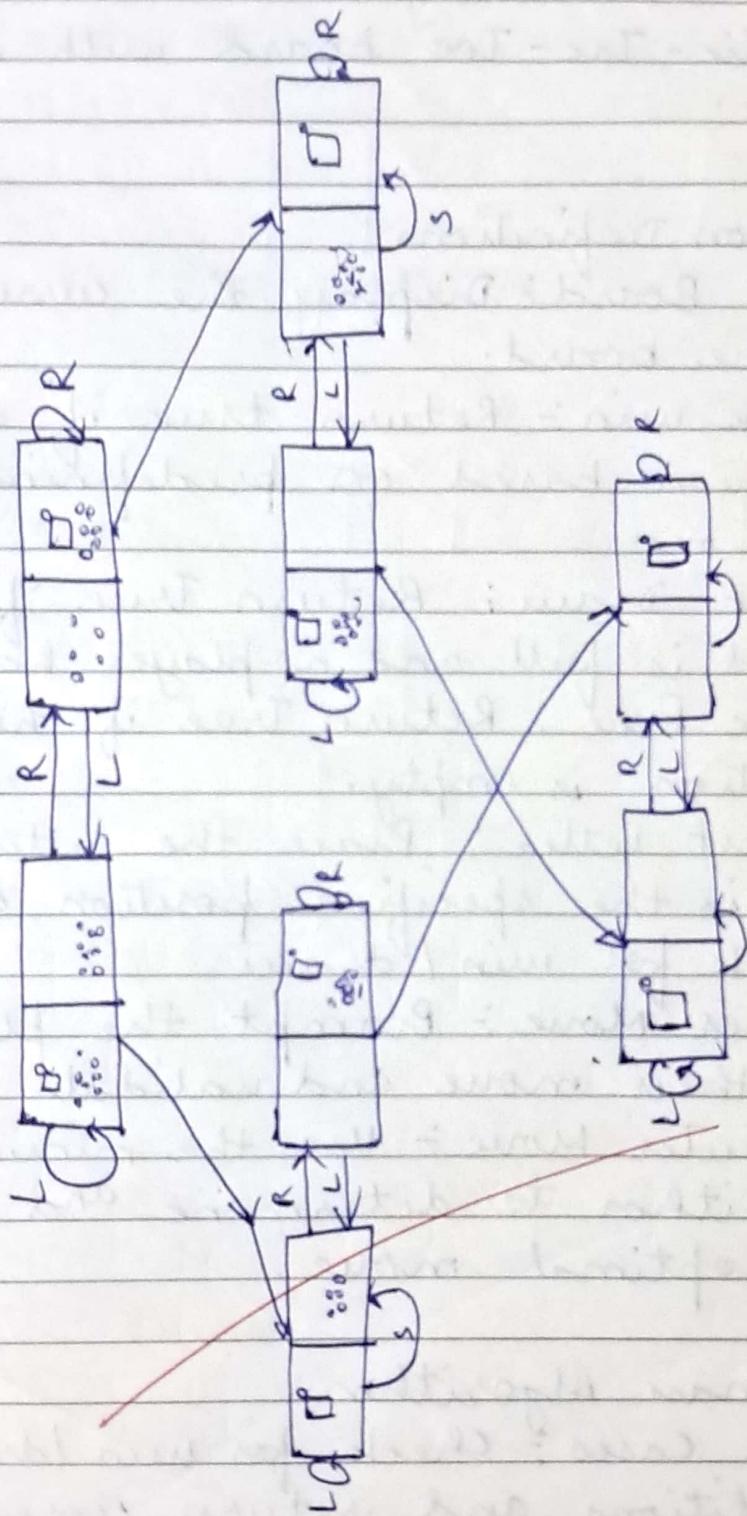
Location B has been cleaned.

GOAL State:

{‘A’: ‘0’, ‘B’: ‘0’}

Performance Measurement: 3

State space tree.



2) Tic-Tac-Toe:

Algorithm:

1. Initialize the board.

Create a dictionary (or list) to represent the Tic-Tac-Toe board with empty spaces.

2. Function Definitions:

- Print Board: Display the current state of the board.
- check win: Return true if any player has won based on predefining winning conditions.
- check draw: Return True if the board is full and no player has won.
- space free: Return True if the selected position is empty.
- Insert letter: Place the letter ('X' or 'O') in the specified position and check for win / draw.
- Player Move: Prompt the player for their move and validate input.
- Computer Move: Use the minimax algorithm to determine and make the optimal move.

3. Minimax Algorithm:

- Base Cases: Check for win / draw conditions and return scores based on the outcome.
- Recursion: For maximizing (computer) and minimizing (player), evaluate

possible moves and return the best score.

4. Main Game Loop.

- While the game is ongoing:
 - If no win/draw, call computerMove.
 - Check for win/draw conditions.
 - If no win/draw, call playerMove.
 - Check for win/draw conditions.

5. End Game:

- Display the final board and announce the result (win/draw).

Pseudocode :-

Initialize board

While true:

If not checkWin() & not checkDraw();
compMove();

If checkWin() or checkDraw();
break;

If not checkWin() & not checkDraw();
playerMove();

If checkWin() or checkDraw();
break

Display final board.

Announce result.

Output: X

Enter position for O (1-9) : 3

X	O	X	O
		X	

Enter position for O (1-9) : 7.

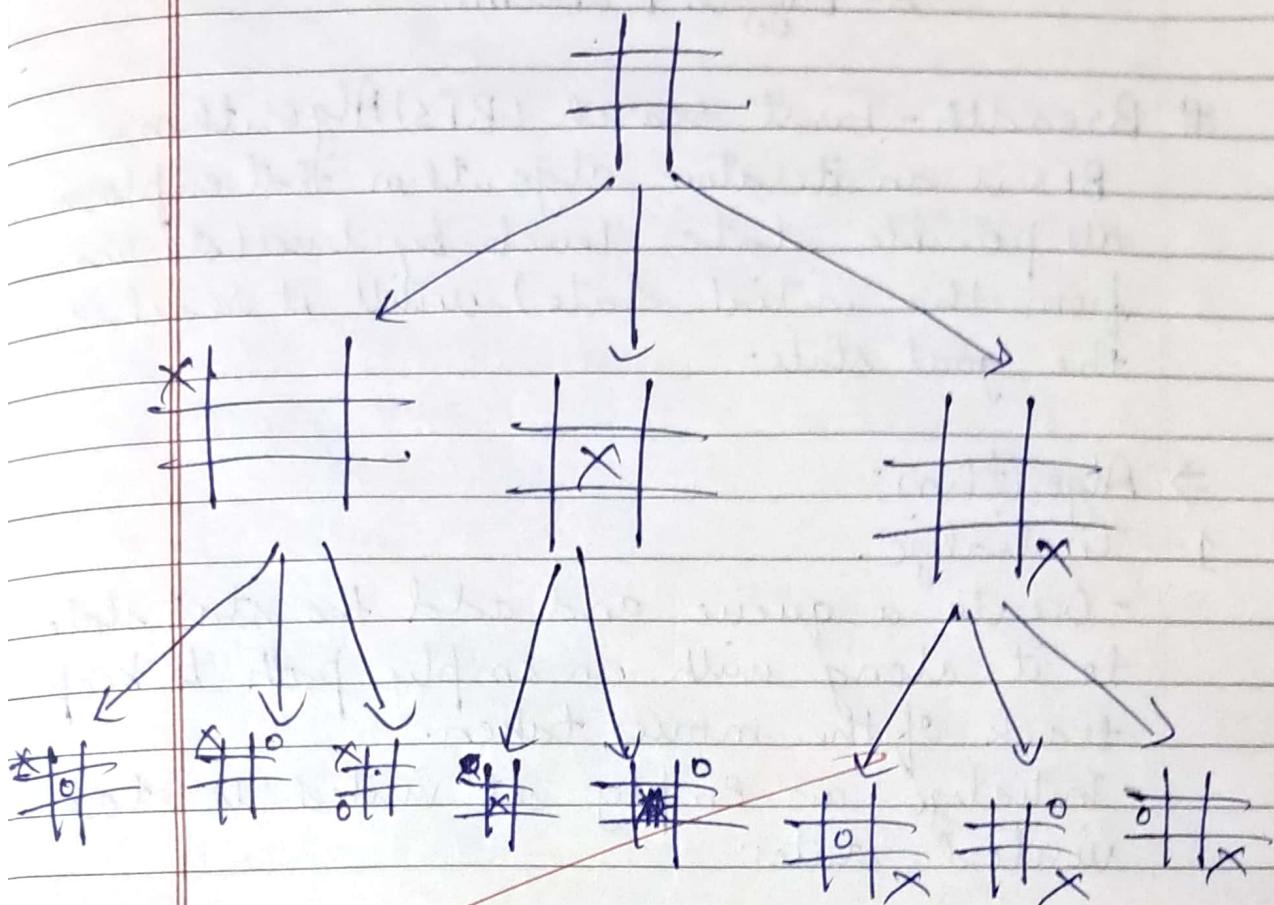
X	O	X	O
X		X	X
O		O	

Enter position for O (1-9) : 6

X	O	X	O
X	X	X	O
O		O	X

Bot wins!

- State Space Tree:



1/10/24

Lab - 2

8 - Puzzle Problem.

Breadth - First search (BFS) Algorithm.

BFS is an iterative algorithm that explores all possible states level by level (starting from the initial state) until it reaches the goal state.

→ Algorithm:-

1. Initialize.

- Create a queue and add the start state to it, along with an empty path to keep track of the moves taken.
- Initialize an empty set 'visited' to store visited states.

2. While queue is not empty:

- dequeue the front element from the queue (current state and the path).

3. Check goal.

- If the current state is the goal state, return the path (solution).

4. Mark current state as visited.

- add the current state to 'visited set'.

5. Generate successors:

- For each valid move from the current state:
 - apply the move to get a new state.
 - If the new state has not been visited:

- add the new state and the updated path to the queue.

6. Return Failure:

- If the queue is empty and no solution has been found, return failure (no solution).

→ Output:

Initial state:

$$\begin{bmatrix} 1, & 2, & 3 \\ 4, & 0, & 6 \\ 7, & 5, & 8 \end{bmatrix}$$

Solving using BFS:

Exploring state in BFS:

$$[1, 2, 3]$$

$$[4, 0, 6]$$

$$[7, 5, 8],$$

Exploring state in BFS:

$$[1, 0, 3]$$

$$[4, 2, 6]$$

$$[7, 5, 8]$$

↑ ↑ ↑
↓ ↓ ↓

• BFS solution: [‘down’, ‘right’]

Exploring state in DFS:

$$[1, 2, 3]$$

$$[4, 8, 6]$$

$$[7, 8, 0]$$

8-puzzle problem using Depth-First Search Algorithm
DFS explores a branch as far as possible before backtracking when it reaches a dead end (or goal).

⇒ Algorithm:

1. Initialize:

- Create a stack and add the start state to it, along with an empty path to keep track of the moves taken.
- Initialize an empty set 'visited' to store visited states.

2. While stack is not empty:

- pop the top element from the stack (the current state and the path).

3. Check goal:

- If the current state is the goal, return the path (solution).

4. Mark current state as visited:

- add the current state to the 'visited set'.

5. Generate successors:

- For each valid move from the current state:
 - apply the move to get a new state.
 - If the new state has not been visited:
 - push the new state and the updated path to the stack.

6. Return failure:

- If the stack is empty and no solution has been found, return failure (no solution exists)

⇒ Output:

Solving using DFS:

Exploring state in DFS:

[1, 0, 3]

[4, 2, 6]

[7, 5, 8]

⋮ ⋮ ⋮

Exploring state in DFS:

[2, 3, 6]

[8, 0, 5]

[4, 1, 7]

⋮ ⋮ ⋮

⋮ ⋮ ⋮

⋮ ⋮ ⋮

Exploring state in DFS:

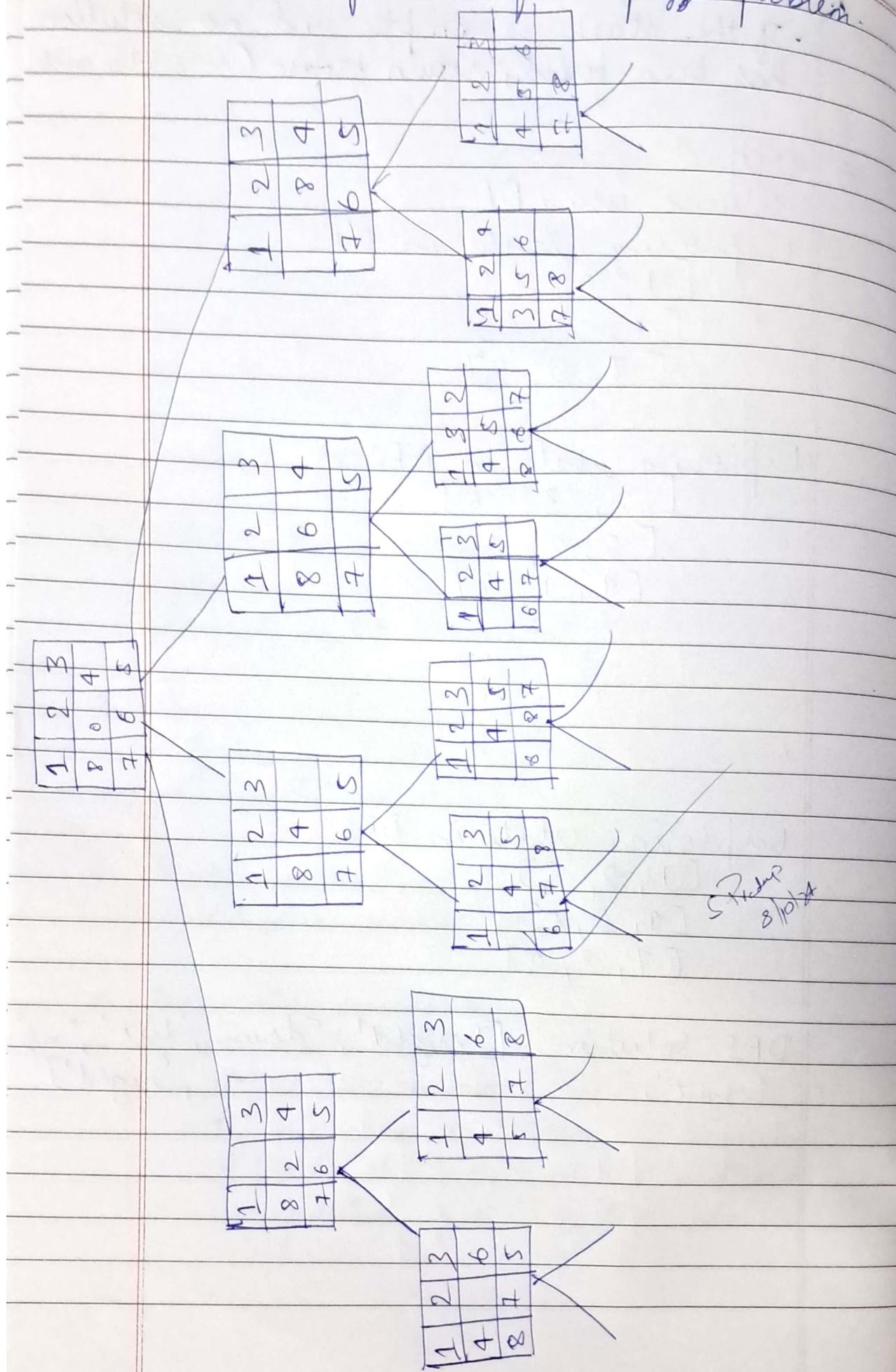
[1, 2, 3]

[7, 8, 6]

[7, 8, 0]

DFS Solution: ['right', 'down', 'left', 'up',
'right', ..., ..., ..., ..., 'right']

State Space Tree for 8-puzzle problem.



Lab-3

Algorithm 1: A* search for 8-puzzle with Misplaced Tiles Heuristic.

1. Initialize

Open list $\leftarrow \{\text{start node}\}$

Closed list $\leftarrow \emptyset$

$g(\text{start}) \leftarrow 0$

$h(\text{start}) \leftarrow \text{number of misplaced tiles.}$

$f(\text{start}) \leftarrow g(\text{start}) + h(\text{start})$

2. Loop until Open list is empty

- Select the node n from the Open list with the lowest $f(n)$.
- Remove n from the Open list and add it to the Closed list.

3. For each neighbour of node n .

- If the neighbour is in the Closed list, skip it.
- Calculate $g(\text{neighbour}) \leftarrow g(n) + 1$ (depth of node)
- Calculate $h(\text{neighbour}) \leftarrow \text{number of misplaced tiles}$
- Calculate $f(\text{neighbour}) \leftarrow g(\text{neighbour}) + h(\text{neighbour})$
- If the neighbour is not in the Open list, add it.
- If the neighbour is not in the Open list, and this path is better (lower $f(n)$), update the path.

4. Repeat steps 2-3 until the goal is reached or the Open list is empty.

State Space Tree.

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

Initial

Goal state.

$g=10$
 $h_1=8$
 $f=g$

5	4	
6	1	8
7	3	2

5	4	8	$g=1$
6	1		$h_1=8$
7	3	2	$f=9$

$g=2, h_1=7$
 $f=9$

5	1	4	
6		8	
7	3	2	

5	4	
6	1	8
7	3	2

5	4	8	$g=2$
6	1		$h_1=8$
7	3	2	$f=10$

5	4	8	$g=2$
6	1		$h_1=8$
7	3	2	$f=10$

$g=3$
 $h_1=7$
 $f=10$

5	1	4	$g=3$
6		8	$h_1=7$
7	3	2	$f=10$

5	1	4	$g=3$
6		8	$h_1=7$
7	3	2	$f=10$

5	1	4	$g=3$
6	3	8	$h_1=7$
7	2		$f=10$

5	1	4	$g=4$
6	3	8	$h_1=7$
7	2		$f=11$

5	1	4	$g=5$
6	3	8	$h_1=5$
7	2		$f=10$

Algorithm 2: A* Search for 8-puzzle with Manhattan Distance Heuristic.

1. Initialize

$\text{open list} \leftarrow \{\text{start node}\}$

$\text{closed list} \leftarrow \emptyset$

$g(\text{start}) \leftarrow 0$

$h(\text{start}) \leftarrow \text{Manhattan distance}$

$f(\text{start}) \leftarrow g(\text{start}) + h(\text{start})$

2. Loop until open list is empty.

- Select the node n from the open list with the lowest $f(n)$
- If n is the goal state/node, terminate and return the path.
- Remove n from the open list and add it to the closed list.

3. For each neighbour of node n ,

- If the neighbour is in the closed list, skip it.
- Calculate $g(\text{neighbour}) \leftarrow g(n) + 1$ (depth of node)
- Calculate $h(\text{neighbour}) \leftarrow \text{Manhattan Distance}$.
- Calculate $f(\text{neighbour}) \leftarrow g(\text{neighbour}) + h(\text{neighbour})$
- If the neighbour is not in the open list, add it.
- If the neighbour is in the open list and this path is better(lower($f(n)$)), update the path.

4. Repeat steps 2-3 until the goal is reached or the open list is empty.

State Space Tree

Initial

$g=0$	2	8	3		1	2	3
$h_2=4$	1		4		8		4
$f=4+0=4$	7	6	5		7	6	5

$g=1$	2	8	3	$g=1$	2	3	$g(n)=1$	2	8	3
$h(n)=1+1+2+1$ =5	1	4		$h(n)=3$	1	8	$h(n)=5$	2	4	
$f=6$	7	6	5	$f(n)=4$	7	6	$f(n)=6$	7	6	5

$g(n)=2$		2	3	$g(n)=2$	2	3	
$h(n)=2+1=3$	1	8	4	$h(n)=2+1+1+1=5$	1	8	4
$f(n)=4$	7	6	5	$f(n)=2+4=6$	7	6	5

1	2	3	$g(n)=3$
8		4	$h(n)=1$
7	6	5	$f(n)=3+1=4$

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

*S. Rizvi
15/10/2024*

Final state
goal reached.

Lab - 4

Algorithm for : Implement Hill Climbing Search algorithm to solve N-Queens problem.

1. Generate a random initial configuration of N queens on the board with one queen per column.
2. Evaluate the cost (heuristic cost) of the current configuration. This cost is the number of pairs of queens that are attacking each other.
3. Generate neighbours by moving each queen within its column to every other row and calculate the cost for each neighbour.
4. Select the neighbour with the lowest cost. If no neighbour has a lower cost than the current state, terminate.
(reached a local or global optimum).
5. If a neighbour has a lower cost, move to that neighbour and repeat the process.
6. If stuck at a local optimum, optionally restart with a new random configuration (random restart hill climbing).
7. Stop when the goal (zero conflict) is reached.

Pseudocode:

function hill-climbing-n-queens(initial state)
current-state = initial-state

while true:

 current_heuristic = calculate_heuristic(current)

 if current_heuristic == 0:

 return current_state

 neighbours = generate_neighbours(current_state)

 best_neighbour = None

 best_heuristic = infinity

 for each neighbour in neighbours:

 heuristic = calculate_heuristic(neighbour)

 if heuristic < best_heuristic

 best_heuristic = heuristic

 best_neighbour = neighbour

 if best_heuristic >= current_heuristic:

 break // No better neighbour found.

 current_state = best_neighbour

return None // No solutions found.

State Space Tree

Step. 0:

				Θ
		Θ	Θ	
initial state	Θ			$n_0 = 3, n_1 = 1, n_2 = 2, n_3 = 0$ cost = 2

Step. 1:- Move to neighbours

				Θ
	Θ			
			Θ	
	Θ			cost = 1

Step. 2: Move to better neighbours.

				Θ
	Θ			
			Θ	
	Θ			cost = 0

→ Final state (Optimal)

Step. 3: Reached local minimum already,
so no better solution can be found.

				Θ
	Θ			
			Θ	
	Θ			Final cost = 0

S.Pradeep
22/10/24

Lab-5 (continued)

Stimulated Annealing for N-Queens.

Algorithm:

1. Initialization

- Start with a random configuration of queens on board.
- Calculate initial energy.
- Set the initial temperature T_0 and final temperature T_f .

2. Annealing Loop:

- Repeat until $T < T_f$:
 - Generate neighbouring configuration.
 - Calculate the energy of new config.
 - If new config has lower energy, accept it.
 - If new config has higher energy,
 ΔE is change in energy.
 - Reduce temp T using cooling rate α ; $T = T \times d$.

3. Termination :

- Stop when T reaches T_f , or a solution with zero conflicts is found.

4. Output :

- If a valid solution (zero conflicts) is found, then return solution.
- Otherwise, return the best config found.

State - Space Tree

	0		0
0		0	
	0		

cost = 1

no. of queens = 4
 initial set = {2, 1, 3, 0}
 $T = 100$
 $\alpha = 0.95$

- Generate a neighbour :

Move queen 2 from row 1 to row 0.

	0	0	
0			
	0		

new state = {2, 0, 3, 0}
 cost = 1
 $A \text{ cost} = 1 - 1 = 0$

$$n = e^{-T \text{ cost}} = e^{-0} = e$$

$$T = T \times 0.95 = 95$$

- Generate neighbours

	0	0	
0		0	
	0		

new state = {2, 0, 3, 1}
~~cost = 0~~
 since cost = 0, stop.

∴ accept solution, thus final state
 S_{Final}

Lab - 6 Propositional Logic.

1. Initialize Truth Value Combinations.
 - Generate all possible True/False combinations for the propositional variables.
2. Define Operator Priority.
 - Set priorities for ' \sim ' highest, ' \wedge ' lowest.
3. User Input.
 - Prompt the user for the knowledge Base (KB) and Query (Q) expressions.
4. Convert Expressions to Postfix.
 - Convert both KB and Q from infix to postfix notation.
 - Add operands directly to the postfix expressions.
 - Use a stack to handle parenthesis and operator precedence.
5. Evaluate Postfix Expressions.

For each truth value combination :

 - Evaluate KB and evaluate Q using a stack-based postfix evaluation.
 - Push operand values from the current combination onto the stack.
 - Apply operators using stack operations (e.g. ' \sim ' negates the top, ' \wedge ' and ' \vee ' operate on the two values).

\wedge AND
 $\vee \text{OR}$

Rule $p \wedge q$
Query - q

6. Check Entailment.

- If there's any combination where KB is True and Q is false, return False.
- If no such combination exists, return True.

7. Output Result.

- Print "The Knowledge Base entails the query" if entailment holds; otherwise, print "The knowledge Base does not entail the query".

#

State #	p	q	r	KB evaluation	Query	$KB \Rightarrow Query$
1.	True	True	True	True	True	True
2.	True	True	False	True	False	False
3.	True	False	True	False	True	True
4.	True	False	False	False	False	True
5.	False	True	True	False	True	True
6.	False	True	False	False	False	True
7.	False	False	True	False	True	True
8.	False	False	False	False	False	True

Result: The query doesn't entail the query as there is a True-False relation which entails to False.

S. Radha
21/12/2021

Lab - 7

Unification in First Order Logic.

Algorithm.

1. If Ψ_1 or Ψ_2 is a variable or constant:
 - a) If Ψ_1 or Ψ_2 are identical :
then return NIL.
 - b) Else if Ψ_1 is a variable,
i) then if Ψ_1 occurs in Ψ_2 , then
return Failure.
ii) else return $\{(\Psi_2/\Psi_1)\}$.
 - c) Else if Ψ_2 is a variable,
i) If Ψ_2 occurs in Ψ_1 , then
return Failure.
ii) Else return $\{(\Psi_1/\Psi_2)\}$
 - d) Else return $\{(\Psi_1/\Psi_2)\}$ Failure.
2. If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.
3. If Ψ_1 and Ψ_2 have a different number of arguments, then return Failure.
4. Set Substitution set(SUBS) to NIL.
5. For $i = 1$ to the number of elements in Ψ_1 :
 - a) Call Unify function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2 , and put the result into S .
 - b) If $S = \text{failure}$ then return Failure.
 - c) If $S \neq \text{NIL}$, then do .
 - i) Apply S to the remainder of both L_1 & L_2 .
 - ii) $SUBST = APPEND(S, SUBST)$.
6. Return $SUBST$.

Output:

Case 1: Basic Unification.

Enter the first expression (Ψ_1): $[\text{`p', `x', } [\text{`f', `y'}]]$

Enter the second expression (Ψ_2): $[\text{`p', } [\text{`f', `a'}], \text{`z'}]$

Output:

Unification Result: $[[[\text{`f', `a'}], \text{`x'}], [\text{`z', `y'}]]$

Case 2: Failure Due to Mismatched Predicates.

Enter the first expression (Ψ_1): $[\text{`p', `x', `y'}]$

Enter the second expression (Ψ_2): $[\text{`q', `x', `y'}]$

Output:

Unification Result: FAILURE.

Case 3: Substitution Chains.

Enter the first expression (Ψ_1): $[\text{`p', `x', `y'}]$

Enter the second expression (Ψ_2): $[\text{`p', `a', `b'}]$.

Output:

Unification Results: $[[\text{`a', `x'}], [\text{`b', `y'}]]$

S. Pradeep
15/11/24

Lab- 2

Forward Reasoning in FOL

Algorithm:

1. Initialization:

- Begin with a given knowledge base KB containing a set of first-order definite clauses.
- Define the query q , an atomic sentence to be proven.
- Initialize new as an empty set to track newly derived facts.

2. Iteration:

- Loop until no new facts can be inferred.
- For each rule r in the knowledge base:
 - i.) Standardize Variables:
 - Replace all variables in r with unique variables to avoid clashes with existing variables in the KB.
 - ii.) Check Premises:
 - Find substitutions θ such that all premises P_1, P_2, \dots, P_n of the rule hold in the current KB.
 - iii.) Derive Conclusions:
 - Apply θ to the conclusion O to infer a new fact ' O' .
 - iv.) Avoid Redundancy:
 - Ensure O' is not already in KB or new before adding it to new.

v) Unify with Query :

- Check if O' unifies with q. If so, return the substitution O.

3. Update knowledge base:

- Add all new facts from new to the KB

4. Termination :

- If no new facts can be inferred and the query is not resolved, return False.

Query) As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen.

Prove that Robert is a criminal.

Output) Robert is a criminal.

State Space Tree

Start
26/11/22

Criminal (Robert)

American (Robert) \wedge Sold (Robert, Missiles, A) \wedge Hostile (A)

American (Robert)

Sold (Robert, Missiles, A)

Hostile (A)