AI
Assignment - 2

Name: Himanshu kumar
Roll no. : 2022215

Theory (30 Marks)

Q.1 Let

G_t : The light is green at time t
Y_t : The light is yellow at time t
R_t : The light is Red at time t

Representing Rule 1 (At any given moment, the traffic light is either green, yellow or Red.) It's never in more than one state at a time

We Can represent Rule 1 as

$$(G_t \vee Y_t \vee R_t) \wedge \neg(G_t \wedge Y_t) \wedge \neg(G_t \wedge R_t) \wedge \neg(Y_t \wedge R_t)$$

Representing Rule 2 (The traffic light switches from green to yellow, yellow to red and red to green) There is no Creative jumping around in the sequence - it stick to its routine

We Can represent Rule 2 as

$$(G_t \rightarrow Y_{t+1}) \wedge (Y_t \rightarrow R_{t+1}) \wedge (R_t \rightarrow G_{t+1})$$

Rule 3 (The traffic light Cannot remain in the same state for more than 3 Consecutive cycles)

This Can be represented as

$$(G_t \wedge G_{t+1} \wedge G_{t+2}) \rightarrow \neg G_{t+3}$$
$$(Y_t \wedge Y_{t+1} \wedge Y_{t+2}) \rightarrow \neg Y_{t+3}$$
$$(R_t \wedge R_{t+1} \wedge R_{t+2}) \rightarrow \neg R_{t+3}$$

$$\therefore ((G_t \wedge G_{t+1} \wedge G_{t+2}) \rightarrow \neg G_{t+3}) \wedge ((Y_t \wedge Y_{t+1} \wedge Y_{t+2}) \rightarrow \neg Y_{t+3})$$
$$\wedge ((R_t \wedge R_{t+1} \wedge R_{t+2}) \rightarrow \neg R_{t+3})$$

Q.2 Given

Node N

Edges $R \subseteq N \times N$

Color $C = \{c_1, \dots c_k\}$: A finite, non-empty set of Colours where each Color is represented by a Constant

Predicates

Color (x, c) : Node x has Color c

Edge (x, y) : There is a directed edge from x to y

Reachable (x, y, n) : Node y is reachable from x within n steps

Axioms for the Rules

Rule 1 Connected node don't have the same Color

It means No two nodes Connected by an edge should share the same Color.

$$\forall x \forall y (\text{Edge}(x,y) \rightarrow \forall c (\text{Color}(x,c) \rightarrow \neg \text{Color}(y,c)))$$

This axiom ensure that if there is an edge from x to y, then x and y Cannot have the same Color C

Rule 2 Exactly two nodes are allowed to wear yellow

It means yellow Color is limited to only two nodes

$$\exists x \exists y (x \neq y \wedge \text{Yellow}(x) \wedge \text{Yellow}(y) \wedge \forall z (\text{Yellow}(z) \rightarrow (z = x \vee z = y))$$

This formula ensure that there are exactly two distinct nodes x ~~andy~~ y with the Color yellow, and no other node z is yellow

Rule 3 Starting from any red node, you can reach a green node in no more than 4 steps

$\forall x \, (Red(x) \rightarrow \exists y \, (Green(y) \wedge Reachable(x, y, 4)))$

This axiom state that if x is a Red node, there exist some green node y such that y is reachable from x within 4 steps.

Rule 4 For every Color in the palette, there is at least one node with this Color.

Each Color in the palette should be represented by atleast one node

$\forall c \in C \; \exists x \; Color(x, c)$

This formula ensure that each Color c in C is assigned to at least one node x.

Rule 5 The nodes are divided into exactly |C| disjoint non-empty cliques one for each Color.

1) each Color form a clique:

$\forall c \in C, \forall x, \forall y \, (Color(x, c) \wedge Color(y, c) \wedge x \neq y \rightarrow Edge(x, y) \wedge Edge(y, x))$

This formula state that if two nodes x and y share the same color c, there is a bidirectional edge between them.

2) Disjointness of cliques:

$\forall x, \forall y, \forall c_1, \forall c_2 (\text{color}(x,c_1) \wedge \text{color}(y,c_2) \wedge c_1 \neq c_2 \rightarrow \neg \text{edge}(x,y))$

This ensure that if two node x and y have different color, there is no edge between them

3) Non-emptiness of each clique:

$$\forall c \in C \; \exists x \; \text{color}(x,c)$$

This ensure that each color clique has atleast one node.

Q.3 Proposition Variable:

R : A dolphin can Read
L : A dolphin is literate.
I : A dolphin is intelligent
D : A dolphin is dolphin

Represent Statement in PL.

1) Whoever can read is literate.
It can be represent in PL as

$$R \rightarrow L$$

2) Dolphin are not literate.
It can be represented in PL as

$$D \rightarrow \neg L$$

3) Some dolphin are intelligent
It can be represent as

$$D \land I$$

This is not a direct PL statement but implies atleast one dolphin is intelligent.

4) Some who are intelligent cannot read.

$$I \land \neg R$$

again same as 3, implies the existance of some intelligent dolphin that cannot read.

5) a) There exist a dolphin who is both intelligent and Can read.

$D \wedge I \wedge R$

b) For every intelligent dolphin, if it can read, it must be that it is not literate.

~~$\forall x$ (I~~ $(D \wedge I \wedge R \rightarrow \neg L)$

from a and b

$(D \wedge I \wedge R) \wedge (D \wedge I \wedge R \rightarrow \neg L)$

Now for first order let's

Predicates:

Dolphin(x) : x is a dolphin
CanRead(x) : x Can Read.
IsLiterate(x) : x is literate
Isintelligent(x) : x is intelligent.

Now Represent Statement in FoL:

1) $\forall x\ (\text{CanRead}(x) \rightarrow \text{IsLiterate}(x))$

2) $\forall x\ (\text{Dolphin}(x) \rightarrow \neg \text{IsLiterate}(x))$

3) $\exists x\ (\text{Dolphin}(x) \wedge \text{IsIntelligent}(x))$

4) $\exists x\ (\text{Isintelligent}(x) \wedge \neg \text{CanRead}(x))$

5) a)

$\exists x\,(\text{Dolphin}(x) \wedge \text{IsIntelligent}(x) \wedge \neg\text{CanRead}(x))$

b) $\forall x\,(\text{IsIntelligent}(x) \wedge \text{CanRead}(x) \rightarrow \neg\text{IsLiterate}(x))$

final anener is

$a \wedge b$

* Resolution Refutation for satisfiability

i) # checking the Fourth Statement: "Some who are intelligent cannot read"

We can use first three statement:

$\forall x\,(\text{CanRead}(x) \rightarrow \text{IsLiterate}(x))$ ... ①

$\forall x\,(\text{Dolphin}(x) \rightarrow \neg\text{IsLiterate}(x))$ ... ②

$\exists x\,(\text{Dolphin}(x) \wedge \text{IsIntelligent}(x))$ ... ③

To express the fourth statement, we need to show that there exist some dolphin that is intelligent and cannot Read.

$\exists x\,(\text{IsIntelligent}(x) \wedge \neg(\text{CanRead}(x)))$ ... ④

<u>Resolution Steps:</u>

a) assume statement 4 is true. let's denote this dolphin as d.

b) From statement 3 we have

$\text{Dolphin}(d) \wedge \text{IsIntelligent}(d)$

c) From statement 2

$\text{Dolphin}(d) \rightarrow \neg \text{IsLitterate}(d)$ implies $\neg \text{IsLitterate}(d)$

4) If d is intelligent and we also assume CanRead(d), then from Statement 1 we would have
$\text{CanRead}(d) \rightarrow \text{IsLitterate}(d)$,
which Contradict $\neg \text{IsLitterate}(d)$.

5) Hence, it is consistent to say that there exist a dolphin this is intelligent and cannot Read, thus the fourth Statement is satisfiable.

ii) # Checking the fifth Statement : "There exist a dolphin who is both intelligent and Can Read."

We will use first four Statement :

$\forall x(\text{CanRead}(x) \rightarrow \text{IsLiterate}(x))$ ...①
$\forall x(\text{Dolphin}(x) \rightarrow \neg \text{IsLiterate}(x))$ ...②
$\exists x(\text{Dolphin}(x) \wedge \text{IsIntelligent}(x))$ ...③
$\exists x(\text{IsIntelligent}(x) \wedge \neg \text{CanRead}(x))$ ...④

To express the fifth Statement : we need to show that there exist a dolphin that is both intelligent and Can Read.

$\exists x(\text{Dolphin}(x) \wedge \text{IsIntelligent}(x) \wedge \text{CanRead}(x))$ ...⑤

Resolution Steps :

1) Assume Statement 5 is true. Denote this dolphin as d.

2) From Statement 3, We have

$$Dolphin(d) \wedge IsIntelligent(d) \wedge CanRead(d)$$

3) If CanRead(d) is true, then from statement 1 We get

$$IsIlliterate(d)$$

4) However, from statement 2, Since d is a dolphin We have

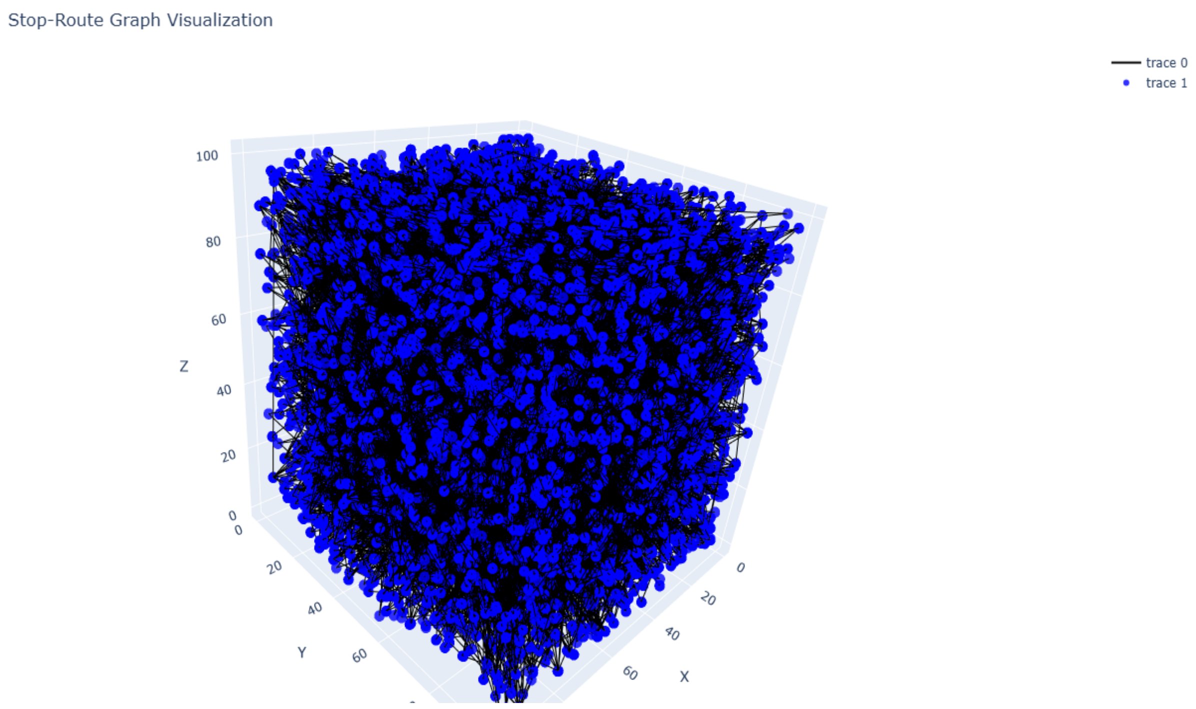
$$\neg IsIlliterate(d)$$

5) This leads to a Contradiction because IsIlliterate(d) and ¬ IsIlliterate(d) Cannot both be true

Thus, we conclude that it is not possible for there to exist a dolphin who is both intelligent and can read while satisfying all previous condition, indicating that the fifth Statement is not satisfiable .

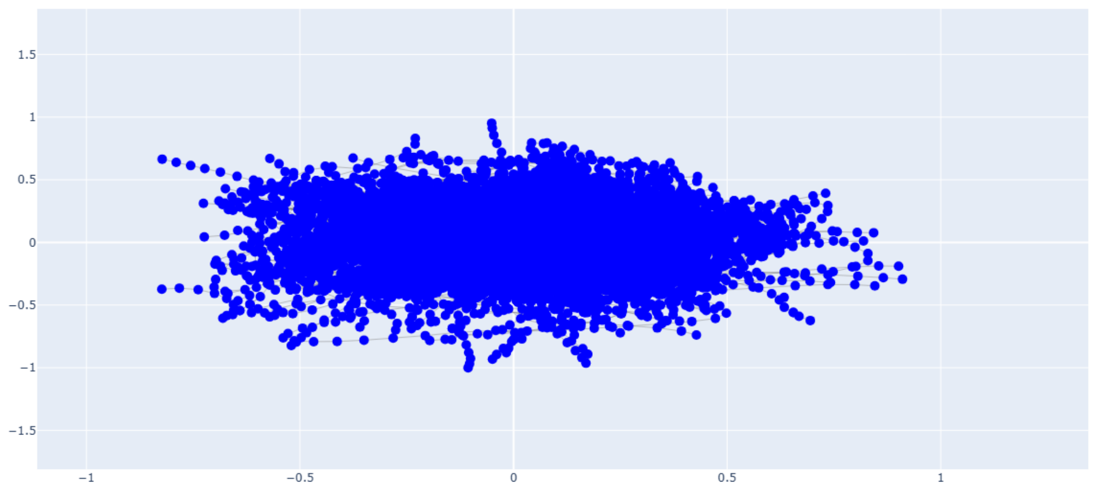
Computational Part

1)



Random 3D Coordinates: For demonstration, random 3D coordinates are assigned to each stop. Replace this part with real data if available.

Nodes and Edges: Stops are represented as nodes, and connections (edges) between consecutive stops on the same route are displayed as lines.



2)

```
 measure performance for test case : 2001, 2005
Brute force Performance:
Execution Time: 0.001001119613647461 seconds
Peak Memory Usage: 0.0001068115234375 MB

query optimisation Performance:
Execution Time: 0.003998756408691406 seconds
Peak Memory Usage: 0.0219268798828125 MB
measure performance for test case :2573, 1177
Brute force Performance:
Execution Time: 0.0010013580322265625 seconds
Peak Memory Usage: 0.0001068115234375 MB

query optimisation Performance:
Execution Time: 0.003306150436401367 seconds
```

Peak Memory Usage: 0.02225494384765625 MB

Comparison of Steps in Both Approaches:

- **Brute-Force Approach**:
  - **Steps**: The brute-force approach checks every pair of stops, directly or indirectly (via transfers), which can involve a large number of checks (O(N^3) in the worst case).
  - **Efficiency**: The brute-force approach is highly inefficient as it does not take advantage of any pre-existing knowledge or logic, thus requiring more iterations and condition checks.
- **PyDatalog Approach**:
  - **Steps**: PyDatalog handles the reasoning declaratively. It applies the facts and rules to infer valid routes without needing to explicitly check every possible combination.
  - **Efficiency**: PyDatalog uses a logic engine that can optimize reasoning, significantly reducing the number of steps compared to brute-force. The system infers valid routes by applying rules to facts in a more structured and efficient way.

Conclusion:

- **Brute-Force Approach**: The brute-force approach involves many manual steps, iterating over every possible connection (direct or indirect) between stops, which leads to a large number of checks and steps. It can be highly inefficient, especially as the problem size grows (with O(N^3) steps in the worst case).
- **PyDatalog Approach**: The PyDatalog approach abstracts away the details of the reasoning process, focusing on relationships (facts) and rules. The system handles reasoning much more efficiently by leveraging logic and inference, reducing the number of steps to a manageable level, typically around O(F + R) steps.

3) a)

```
measure performance for :(951, 340, 300, 1)
Forward Chaining Performance:
Execution Time: 0.07586860656738281 seconds
Peak Memory Usage: 0.09922409057617188 MB

Backward Chaining Performance:
Execution Time: 0.008977174758911133 seconds
Peak Memory Usage: 0.06063652038574219 MB
------------------------------------------------------------
measure performance for :(22540, 2573, 4686, 1)
Forward Chaining Performance:
Execution Time: 0.004985332489013672 seconds
Peak Memory Usage: 0.022439002990722656 MB

Backward Chaining Performance:
Execution Time: 0.002103090286254883 seconds
Peak Memory Usage: 0.027588844299316406 MB
```

b) After executing both methods with step counting and logging, you could summarize the findings like this:

Execution Path:

In Forward Chaining, we iterate over each route sequentially, which results in more steps as it evaluates the possible via stops and interchanges at each level.

Backward Chaining may take fewer steps because it can eliminate paths earlier if they don't reach the required stops, especially when there are fewer interchanges.

Overall Comparison:

Forward Chaining typically involves more steps due to the depth-first exploration of each potential path.

Backward Chaining can be more optimal in certain cases due to its ability to work backward from the goal and reject paths sooner.

c) Forward Chaining

> Process: In Forward Chaining, the search starts from the starting stop and explores all possible routes forward to reach the destination. This approach evaluates each route from the beginning and continues until it finds a valid path that meets the criteria (such as including a via stop and a maximum of one interchange).
>
> Steps: This can lead to more overall steps because it needs to evaluate each forward path sequentially from the start, potentially evaluating many intermediate stops before finding a valid route that meets all conditions.
>
> Best Case: If a valid via stop is close to the start stop, Forward Chaining can potentially find a solution more quickly by exploring forward.
>
> Worst Case: Forward Chaining can take longer if valid paths are deep in the search space, as it may need to evaluate all possible paths forward, even those that don't meet the criteria.

Backward Chaining

> Process: In Backward Chaining, the search begins at the destination and works backward to find possible routes leading to the starting stop, checking if a route passes through the via stop and meets the interchange constraint.
>
> Steps: This can result in fewer steps overall because paths can be eliminated earlier if they do not lead back to the starting stop or meet the via stop requirement. Backward Chaining often stops searching sooner because invalid paths are discarded early.
>
> Best Case: If valid routes are closer to the end stop, Backward Chaining will reach them quickly and avoid unnecessary paths.

Worst Case: In cases with many possible routes leading to dead ends or failing constraints, Backward Chaining might still need to explore a large number of paths but will generally perform fewer steps than Forward Chaining.

Comparison

In most cases, Backward Chaining will tend to take fewer steps than Forward Chaining, particularly when:

There are many routes, but only a few meet the via stop and interchange constraints.

The solution space contains many paths that can be quickly ruled out from the destination side.

4) a) pddl- planning :

```
Execution Time: 0.09207820892333984 seconds

Peak Memory Usage: 0.09199047088623047 MB


[(37, 300, 712),
 (49, 300, 712),
 (121, 300, 712),
 (387, 300, 712),
 (1038, 300, 712),
 (1211, 300, 712),
 (1571, 300, 712),
 (10433, 300, 712),
 (10453, 300, 712)]
```

b,c) Overall Comparison of Steps Involved in Each Approach:

PDDL:

Number of steps: Moderate to Low depending on the planner and its ability to handle constraints efficiently.

Reason: PDDL planners apply constraints and actions in a structured manner, reducing the search space by enforcing the constraints at the beginning of the search. However, it might require more steps than Backward Chaining if the planner doesn't prune paths efficiently.

Efficiency: The efficiency of PDDL depends on the planner used. Some planners may require fewer steps by avoiding unnecessary explorations, while others might require more depending on the complexity of the search space.

Conclusion:

Forward Chaining typically involves the most steps, as it exhaustively explores all possibilities before considering constraints, leading to higher computational cost.

Backward Chaining usually involves fewer steps, as it focuses on paths that lead directly to the goal, pruning invalid paths early.

PDDL generally involves moderate steps, but its efficiency depends heavily on the specific planner and the way it handles the search space and constraints.

Thus, Backward Chaining tends to be the most efficient in terms of the number of steps involved, followed by PDDL, with Forward Chaining usually requiring the highest number of steps.

d) The three approaches (Forward Chaining, Backward Chaining, and Planning Domain Definition Language - PDDL) will not always produce the same optimal route for the following reasons:

1. Forward Chaining

    How it works: Forward Chaining starts from the initial state (start stop) and works its way forward to the goal state (destination stop). It explores all possible paths from the start, considering the available actions (boarding routes, transferring between routes) at each stop.

    Constraints application: Forward Chaining will encounter all possible intermediate steps, but because it explores each path forward, it can encounter and apply the constraints (such as including a via stop and limiting interchanges) later in the process. If a path violates the constraints, it may backtrack and try another route. Forward Chaining's search might involve revisiting or recalculating multiple paths, leading to more computational effort and potentially suboptimal routes if the path exploration is too extensive.

    Risk of suboptimal routes: In some cases, if Forward Chaining does not prioritize paths that immediately meet constraints (e.g., via stop or interchange limits), it might explore more paths than necessary, which could lead to suboptimal solutions, especially in a large search space.

2. Backward Chaining

    How it works: Backward Chaining starts from the goal (end stop) and works backwards, trying to find paths that lead back to the starting stop, considering the transfer points and route connections.

    Constraints application: Backward Chaining can more efficiently discard invalid paths because it works backwards and quickly eliminates paths that don't lead to the goal. By checking the conditions from the end, it can sometimes avoid exploring paths that are already invalid, which makes it faster and more efficient in some cases.

    Risk of suboptimal routes: Since Backward Chaining only explores routes that are viable from the destination backwards, it might ignore some potential solutions that Forward Chaining might catch. This could lead to missing an optimal path, especially in cases where a path from the start to the destination has a more efficient way forward but isn't included due to a narrow backward search scope.

3. Planning Domain Definition Language (PDDL)

How it works: PDDL allows you to define a formalized problem in terms of actions, states, and goals. For route planning, you define actions such as "board route" and "transfer between routes," the initial state (starting stop), and the goal state (destination stop). PDDL uses search algorithms like STRIPS to find the best sequence of actions to achieve the goal state.

Constraints application: PDDL allows you to formally encode constraints like having an intermediate stop (via stop) and limiting interchanges. Depending on the search strategy and the PDDL planner used, the constraints will be enforced at each step of the planning process, potentially pruning non-optimal paths. PDDL planners tend to apply constraints upfront during the planning process, helping to avoid exploring invalid routes early.

Risk of suboptimal routes: The efficiency of the PDDL planner depends on the complexity of the planner and how it handles the constraints. Some PDDL planners might focus on finding the quickest solution, which can sometimes lead to ignoring paths that meet the constraints but are slightly longer in terms of the number of steps or interchanges.

Comparison and Potential Suboptimality

Forward Chaining: Can lead to suboptimal solutions if it doesn't prioritize the constraint application early enough. It might explore many paths before applying the constraints, resulting in inefficiency.

Backward Chaining: Tends to perform better in terms of eliminating paths earlier. However, it could still miss optimal solutions if a viable path from the start to the goal is not explored in the backward search.

PDDL: Tends to be the most structured approach, as constraints and goals are formally defined. However, depending on the planner's search algorithm, it might either over-prioritize efficiency (leading to suboptimal routes) or under-prioritize constraint satisfaction, depending on the planner's configuration and approach.

Conclusion

Do all algorithms produce the same optimal route? No, because each algorithm has different ways of exploring the search space and applying constraints.

Forward Chaining might explore more paths than necessary and could produce suboptimal routes.

Backward Chaining might discard paths earlier but may also miss solutions that Forward Chaining could find.

PDDL provides a formalized approach but can be limited by the specific planner's performance in enforcing constraints and finding the optimal path.