

AI ASSIGNMENT - 2

SHOBHIT RAJ
2022482

Theory

a1. I have introduced Propositional Logic (PL), these propositions:-

$G_t \rightarrow$ The light is green.

$Y_t \rightarrow$ The light is yellow.

$R_t \rightarrow$ The light is red.

a) Traffic only in one state \rightarrow i.e. atleast one & not 2 simultaneously.

\therefore atleast one $\Rightarrow (G_t \vee Y_t \vee R_t)$

not 2 simultaneously $\Rightarrow \neg(G_t \wedge Y_t) \wedge \neg(Y_t \wedge R_t) \wedge \neg(R_t \wedge G_t)$

$\therefore \underline{(G_t \vee Y_t \vee R_t) \wedge \neg(G_t \wedge Y_t) \wedge \neg(Y_t \wedge R_t) \wedge \neg(R_t \wedge G_t)}$.

b) I have used this notation :-

Colour_t \rightarrow The light is 'colour' at time 't'

$\therefore \begin{array}{l} G_t \rightarrow Y_{t+1} \\ Y_t \rightarrow R_{t+1} \\ R_t \rightarrow G_{t+1} \end{array} \quad \left. \begin{array}{c} \\ \\ \end{array} \right\} \text{Transitions}$

c) Light can't stay same for 3 consecutive states :-

i.e.

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

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

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

Q2. I have introduced these predicates :- $(x \in N)$.

$\text{Yellow}(x) \rightarrow$ Node x is yellow

$\text{Red}(x) \rightarrow$ Node x is red

$\text{Green}(x) \rightarrow$ Node x is green

$\text{Colour}(x, c) \rightarrow$ Node x has colour c , where $c \in C$.

$\text{Edge}(x, y) \rightarrow$ Directed edge from node x to node y .

$\text{Clique}(x, c) \rightarrow$ Node x belongs to clique of colour c , where $c \in C$.

$\text{steps}(x, y, n) \rightarrow$ Exists path from x to y of atmost n steps. $c \in C$.

a) End points of edge have different colour.

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

$(x, y \in N) \quad (c \in C)$

b) Only 2 nodes that can have yellow. (exactly 2 nodes)

$$\exists x_1 \exists x_2 (\text{Yellow}(x_1) \wedge \text{Yellow}(x_2) \wedge (x_1 \neq x_2)) \wedge \forall x_3 (\text{Yellow}(x_3) \rightarrow (x_3 = x_1 \vee x_3 = x_2))$$

c) From any red node, can reach a green node within 4 steps.

$$\forall x (\text{Red}(x) \rightarrow \exists y (\text{Green}(y) \wedge \text{steps}(x, y, 4)))$$

d) Every colour has one node atleast.

$$\forall c \exists x (\text{Colour}(x, c))$$

e) Each colour has its own clique & these cliques are disjoint & non empty.

$$\forall x \forall y \forall c (\text{Clique}(x, c) \wedge \text{Clique}(y, c) \rightarrow \text{colour}(x, c) \wedge \text{colour}(y, c))$$

$$\wedge \forall c \exists x \text{Clique}(x, c) \wedge \forall x \forall c_1 \forall c_2 (\text{Clique}(x, c_1) \wedge \text{Clique}(x, c_2) \rightarrow c_1 = c_2)$$

$$\wedge \forall x \forall y \forall c (\text{Clique}(x, c) \wedge \text{Clique}(y, c) \rightarrow (\text{Edge}(x, y) \wedge \text{Edge}(y, c)))$$

Q3. Proposition Logic (PL) :-

I introduced following propositional variables/predicates :-

$L(x)$: x is literate

$R(x)$: x can read

$I(x)$: x is intelligent

$D(x)$: x is a dolphin

PL - a) $R(x) \rightarrow L(x)$

b) $D(x) \rightarrow \neg L(x)$

c) $\exists x (D(x) \wedge I(x))$

d) $\exists x (I(x) \wedge \neg R(x))$

e) $\exists x (D(x) \wedge I(x) \wedge R(x)) \wedge \forall y (D(y) \wedge I(y) \wedge R(y) \rightarrow \neg L(y))$

POL - a) $\forall x (R(x) \rightarrow L(x))$

b) $\forall x (D(x) \rightarrow \neg L(x))$

c) $\forall x (D(x) \wedge I(x))$

d) $\exists x (I(x) \wedge \neg R(x))$

e) $\exists x (D(x) \wedge I(x) \wedge R(x)) \wedge \forall y (D(y) \wedge I(y) \wedge R(y) \rightarrow \neg L(y))$

Satisfiability of 4th :-

$$1 \rightarrow \neg R(x) \vee L(x)$$

$$2 \rightarrow \neg D(x) \vee \neg L(x)$$

$$3 \rightarrow \exists x (D(x) \wedge I(x)) \rightsquigarrow D(x), I(x)$$

negate 4 & add to KB $\Rightarrow \neg(\exists x (I(x) \wedge \neg R(x)))$

$$\Rightarrow \forall x (\neg I(x) \vee R(x)) \Rightarrow \forall x (I(x) \rightarrow R(x))$$

$$\Rightarrow \neg I(x) \vee R(x) \quad -(4)$$

$$\text{Using } 3 \& 4 \Rightarrow I(x) \vee \neg I(x) \vee R(x) \equiv R(x) \quad -(5)$$

$$\text{Using } 1 \& 5 \Rightarrow \neg R(x) \vee L(x) \vee R(x) \equiv L(x) \quad -(6)$$

$$\text{Using } 3 \& 2 \Rightarrow D(x) \vee \neg D(x) \vee \neg L(x) \equiv \neg L(x) \quad -(7)$$

$$\text{Using } 6 \& 7 \Rightarrow L(x) \vee \neg(L(x)) \equiv \text{empty clause}$$

proving 4th statement. \leftarrow contradiction.

Satisfiability of 5th -

(e) $\rightarrow D(x), I(x), R(x) \rightsquigarrow$ original 5th added to KB

(e) $\rightarrow D(x) \wedge I(x) \rightarrow \neg L(x)$

$\perp \rightarrow R(x) \rightarrow \underline{L(x)}$ / (resolve $R(x) \wedge \neg R(x)$)

$L(x) \wedge \neg L(x) \equiv$ empty clause

↓
contradiction.

Statement 5 is not satisfiable.

→ X →

Computational (Coding) Questions

1. Data Loading and Knowledge Base Creation

- (a) After loading the OTD static data, the IDs are stored as strings & Time fields (in stop_times) are converted to datetime objects.
- (b) To create the knowledge base, we establish several mappings and dictionaries, capturing relationships between routes, trips, and stops:
1. **Route to Stops Mapping** (route_to_stops): A dictionary mapping each route ID to an ordered list of stops. This represents the sequence of stops in each route, essential for reasoning about direct routes.
 2. **Trip to Route Mapping** (trip_to_route): A dictionary mapping each trip ID to its respective route ID, which helps in retrieving routes associated with any trip.
 3. **Stop Trip Count** (stop_trip_count): A dictionary counting the number of trips that pass through each stop, providing insight into stop popularity and frequency.
 4. **Fare Rules Mapping** (fare_rules): A dictionary mapping each route ID to its associated fare information, combining data from fare_rules and fare_attributes.
 5. **Merged fare dataset** (merged_fare_df): fare_rules is merged with fare_attributes using dictionary-based joins to avoid redundant iterations.

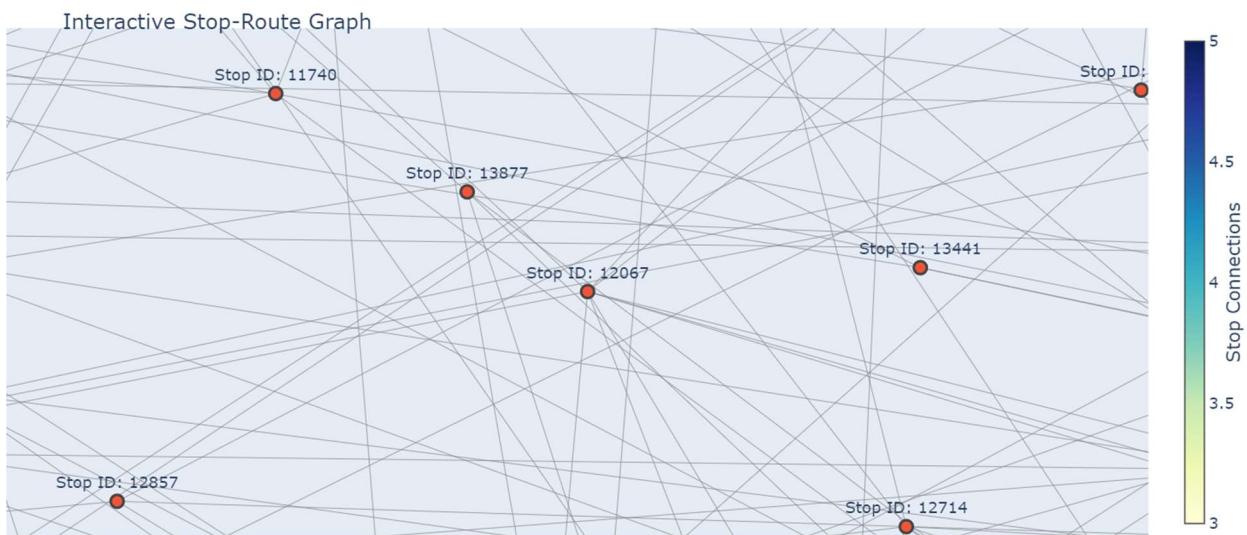
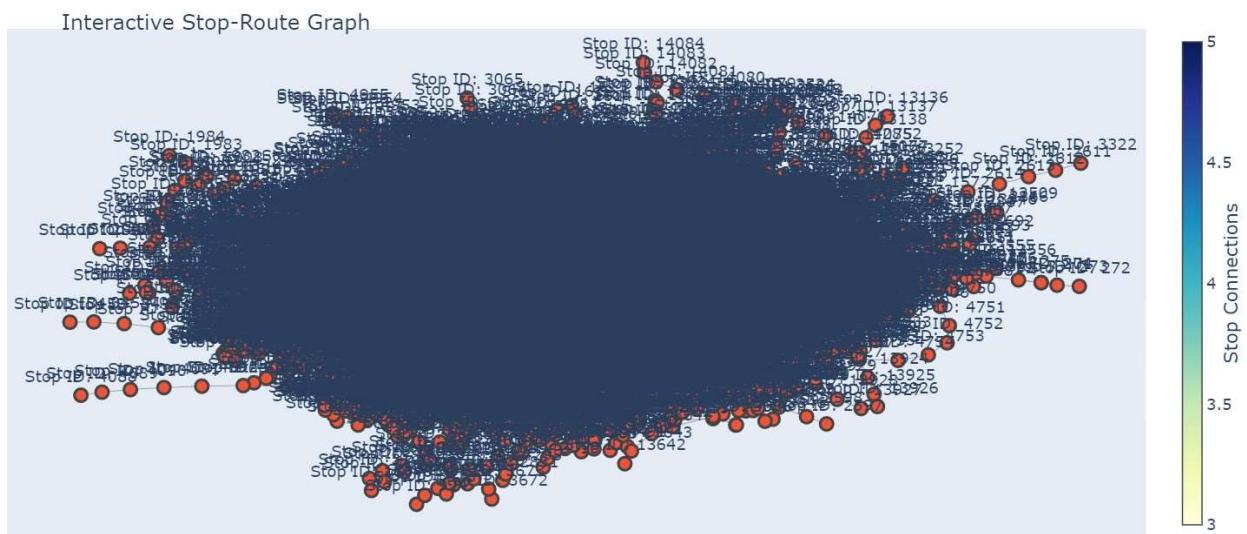
(c) Verification of the Knowledge Base:

1. **Top 5 Busiest Routes**: Based on the total number of trips associated with each route.
2. **Top 5 Stops with Most Frequent Trips**: Ranked by the count of trips passing through each stop.
3. **Top 5 Busiest Stops by Route Count**: Sorted by the number of different routes that pass through each stop.
4. **Top 5 Pairs of Consecutive Stops on a Single Direct Route**: Using the route_to_stops mapping, we identify unique pairs of stops connected by only one direct route and rank them by trip frequency.

(d) Interactive Graph Visualization:

To better understand and visually explore the transit network structure, we developed an interactive graph using Plotly, representing the relationships between stops and routes. This visualization allows us to intuitively analyze how stops are connected within each route, providing a clear picture of route structure and stop adjacency.

Some images of the graph created are attached below :-



2. Reasoning

- (i) Brute-Force Approach: I implemented a brute-force search algorithm to determine direct routes between two specified stops, leveraging the route_to_stops data structure. The methodology involved iterating over each route and checking if both the start and end stops were present within the list of stops for that route.

Example – start_stop = 100, end_stop = 101 ; Output =>

Direct Routes: [10409, 10426, 2122, 427, 10493, 1067, 1572, 1790, 1521, 1663, 247, 10595, 927, 10627, 10637, 10646]

Execution Time (s): 0.006003141403198242

Memory Usage (MB): 0.0004119873046875

- (ii) FOL Library-Based Reasoning: I used the PyDatalog library to define relationships between stops and routes declaratively. I created terms and predicates, adding facts about stops and routes to a knowledge base. Using logical inferences, PyDatalog efficiently identifies direct routes without manually iterating through each route. This declarative approach, which defines rules and lets the library handle inference, is particularly advantageous for large datasets, as it optimizes reasoning and reduces computational steps.

Example – start_stop = 100, end_stop = 101 ; Output =>

Direct Routes: [247, 427, 927, 1067, 1521, 1572, 1663, 1790, 2122, 10409, 10426, 10493, 10595, 10627, 10637, 10646]

Execution Time (s): 0.0059814453125

Memory Usage (MB): 0.028474807739257812

- (a) Time & Space Complexity analysis: Based on some test cases I ran like one above, I found that PyDatalog approach is more efficient in terms of execution time as it leverages logical inference while Brute force approach takes more time because of explicit route checks. Although, in terms of memory usage, PyDatalog approach uses more memory than brute force because it stores additional relational facts and rules in a knowledge base, enabling logical inferences.

(b) Intermediate Reasoning Steps:

- i. Brute Force approach – Procedural i.e. checks routes individually
 - **Route_to_stops preprocessing:** Making the mapping.
 - **Route Traversal:** The algorithm iterates over each route.
 - **Direct Match Check:** For each route, the algorithm checks if start_stop and end_stop are in the same stops list. This check occurs for every route, regardless of whether it has both stops or not.
 - **Building Output:** If a route is found to contain both stops, it is added to the direct_routes list.
- ii. FOL Library-Based approach – Declarative logic i.e. infers relationships
 - **Predicate Initialization:** DirectRoute(X, Y, R) is defined based on RouteHasStop to establish that there is a direct route R between X and Y if both stops are on the same route.
 - **Data Insertion:** Facts for each stop and route are added using RouteHasStop(route_id, stop).
 - **Query Execution:** The query_direct_routes function then leverages these facts and applies the DirectRoute predicate to find all routes connecting start_stop and end_stop directly.
 - **Inference:** The Datalog engine infers direct routes based on the defined relationships without needing to iterate through the entire data explicitly.

(c) Comparison of Overall Steps Involved: The number of intermediate steps (mentioned above) are same (4 each) in both but number of steps is the computational steps the algorithm performs, such as the number of checks in brute force or the number of rule applications in PyDatalog.

- **PyDatalog Approach:** In this approach, defining relationships between stops and routes at the outset allows the engine to make efficient inferences with a logical framework. When querying, it only retrieves data relevant to start_stop and end_stop directly, saving computational steps compared to scanning through all data.
- **Brute-Force Approach:** In contrast, the brute-force approach would involve iterating through each route, checking for both start_stop and end_stop within each route, and manually adding routes that meet this criterion. This approach could result in $O(n * m)$ complexity for n routes and m stops per route, which is less efficient as the data size grows.

3. Planning

- (i) Forward Chaining: In forward chaining, we aim to find an optimal route from the start stop to the end stop, ensuring the path includes an intermediate stop (via stop) and respects the maximum number of transfers. The OptimalRoute predicate is used to check if routes connect the start stop to the intermediate stop and the intermediate stop to the destination. Valid paths are returned, which include the intermediate stop and follow the transfer constraints.
- (ii) Backward Chaining: In backward chaining, the search starts from the destination stop and works backward to find a valid route that connects the end stop to the intermediate stop and then to the start stop. This ensures the journey includes a single transfer at the intermediate stop. The OptimalRoute predicate checks the validity of the paths, returning routes that meet the transfer constraints.

Example –

start_stop_id = 340
end_stop_id = 951
stop_id_to_include = 300
max_transfers = 1

Output =>

Forward Chaining Result: [(712, 300, 37), (712, 300, 10453), (712, 300, 1038), (712, 300, 387), (712, 300, 1211), (712, 300, 10433), (712, 300, 49), (712, 300, 121), (712, 300, 1571)]

Forward Chaining - Time: 0.0199 seconds, Memory: 0.0895 MB

Backward Chaining Result: [(1211, 300, 712), (49, 300, 712), (10433, 300, 712), (387, 300, 712), (1571, 300, 712), (1038, 300, 712), (10453, 300, 712), (37, 300, 712), (121, 300, 712)]

Backward Chaining - Time: 0.0485 seconds, Memory: 0.1167 MB

(a) Time & Space Complexity analysis: Based on some test cases I ran like one above, I found that backward chaining is using little more time and space than forward chaining because forward chaining operates sequentially and backward chaining involve more steps due to the reversed reasoning process. It might be using more memory due to the reverse exploration, requiring storing more intermediate states or additional steps in the reasoning process.

(b) Intermediate Reasoning Steps: Both Forward Chaining and Backward Chaining follow the same steps to find the optimal route, but with a key difference in directionality:

- **Predicate Evaluation:** Both methods start by evaluating whether the start stop is connected to an intermediate stop and whether that intermediate stop is connected to the destination stop, based on the predefined relationships in the knowledge base. This involves checking for valid routes that meet the given constraints (1 in this case).
- **Path Construction:** After finding a valid intermediate stop, the path is constructed from the start stop to the intermediate stop, and then to the destination stop, ensuring the transfer constraint is respected.
- **Query Execution:** A query is executed to check for routes that meet the criteria, and the valid paths are collected and formatted for output.

While both approaches use these same steps, Backward Chaining simply follows the reverse order of Forward Chaining. Instead of starting from the start stop and moving forward, it begins at the destination stop and works backward towards the start stop, identifying routes that pass through the intermediate stop while ensuring that only one transfer occurs. This reverse approach leads to the same final result but in a different search direction.

(c) Comparison of Overall Steps Involved: In Forward Chaining, the process starts by querying the knowledge base from the starting stop to the intermediate stop and then to the end stop, ensuring only one transfer occurs. The search checks connections between the start, intermediate, and end stops, evaluating predicates for each possible path. In Backward Chaining, the search begins at the destination and works backward, narrowing the possibilities through the intermediate stop to the start stop. This reverse reasoning may require more steps, as it checks additional conditions to find valid paths. As a result, backward chaining typically involves a more complex and step-intensive process than forward chaining.