Theory: Question 1)

AI Assignment 2

Theory

Page:
Date:

Question 1) Let say

G_t → Traffic light is green at time t.
Y_t → Traffic light is yellow at time t.
R_t → Traffic light is red at time t.

To represent that the traffic light remains only in one state at a time,

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

To represent the switching of the light,

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

To represent that the light can't stay in the same state for more than 3 consecutive cycles. and have to transition to the next state.

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

Can't be three consecutive greens.

$$G_t \wedge G_{t+1} \wedge G_{t+2} \rightarrow Y_{t+3}$$

- Can't be three consecutive yellows.

$$Y_t \wedge Y_{t+1} \wedge Y_{t+2} \rightarrow R_{t+3}.$$

- Can't be three consecutive reds.

$$R_t \wedge R_{t+1} \wedge R_{t+2} \rightarrow G_{t+3}.$$

Question 2)

Page :
Date :

Question 2) Defining some predicates.

Colour (x, c) → node x is given a colour c.
Edge (x, y) → edge exists between x and y.
x to y ⇒ directed edge.
Mindist (x, y) → minimum distance between x and y.
Reachable (x, y) → y node is reachable from node x.
Clique (x, c) → Node x belongs to clique of colour c.

Assumption ⇒ Assuming that all the nodes are coloured and we are dealing with non coloured nodes. Defining some axioms.

1) ∀x ∀y Reachable (x, y) → edge(x, y)

2) ∀x ∀y (Edge (x, y) → Mindist (x, y, 1))
↳ min distance will be 1 between two nodes connected by a edge.

3) ∀x ∀y ∀z ∀n. (Edge (x, z) ∧ Mindist (z, y, n) → Mindist (x, y, n+1))
↳ ie; if there is an edge from x to z and the min distance from z to another node y is n, then the min distance from x to y is n+1.

4) ∀x ∃c ∈ C Colour (x, c)
↳ ie; every node is assigned a colour from the non empty set C.

Page :
Date :

5) ~~$\forall x \exists c \in C$~~

$\forall x \forall c_1 \forall c_2 \; (\text{Colour}(x, c_1) \wedge \text{Colour}(x, c_2) \rightarrow c_1 = c_2)$

↳ A node can be given atmost one colour.

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

↳ A node can belong to only one clique atmost.

Rules for first order logic.

1) connected nodes dont have same colour.

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

2) Exactly two nodes can have colour yellow.

$\exists x \exists y \; (x \neq y \wedge \text{colour}(x, \text{yellow}) \wedge \text{colour}(y, \text{yellow}) \wedge \forall z \; (\text{colour}(z, \text{yellow}) \rightarrow ((z = x) \vee (z = y))))$

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

If we define reachability for two nodes x and y as →

Reachable $(x, y) \rightarrow$ ~~$\forall x \forall y$~~ $(\text{colour}(x, \text{Red}) \wedge \text{colour}(y, \text{Green}) \wedge$

$\exists z_1, z_2, z_3 \; (\text{edge}(x, z_1) \wedge \text{edge}(z_1, z_2) \wedge \text{Edge}(z_2, z_3) \wedge \text{edge}(z_3, y)))$

Also, $\forall x \; (\text{colour}(x, \text{Red}) \rightarrow \exists y \; (\text{colour}(y, \text{green}) \wedge \text{reachable}(x, y)))$

4) For every colour in the palette, there is atleast one node in every colour.

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

5) The nodes are divided in (C) Cliques and none of them are empty, also one for each colour. and disjoint cliques.

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

↳ For each colour there exists atleast one node x belonging to its clique.

→ $\neg \forall x \forall y \forall c_1 \forall c_2 \; (\text{colour}(x, c_1) \wedge \text{colour}(y, c_2) \wedge c_1 \neq c_2 \rightarrow (\neg \text{Clique}(x, c_2)) \wedge (\neg \text{Clique}(y, c_1)))$

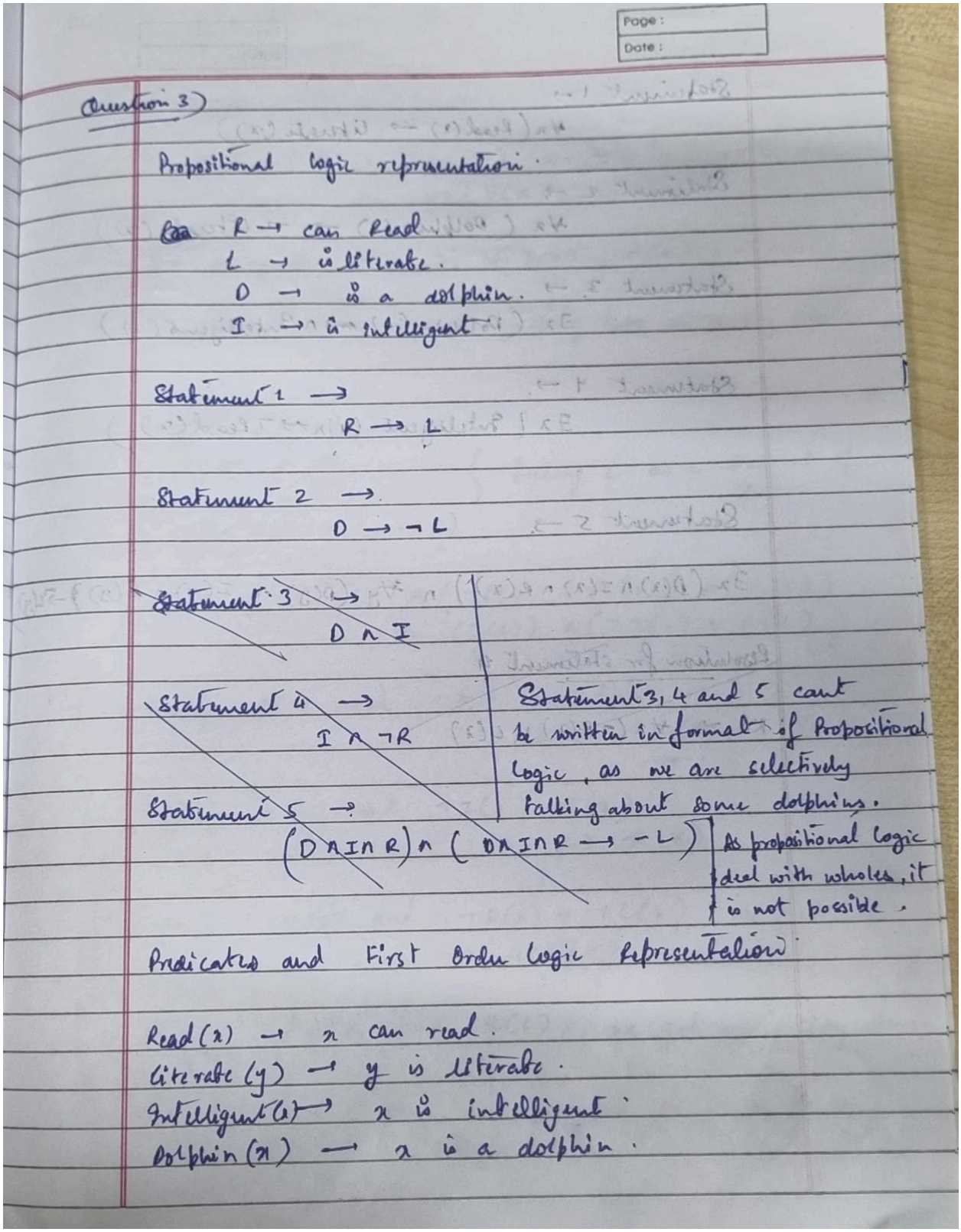
So if two nodes x and y have different colours then x cant belong to clique with the colour of y, indicating the cliques are disjoint.

Another way of writing:

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

If two nodes belong to the same clique they have the same colour.

Question 3)



Question 3)

Propositional logic representation:

R → can Read
L → is literate.
D → is a dolphin.
I → is intelligent

Statement 1 →

$R \rightarrow L$

Statement 2 →

$D \rightarrow \neg L$

Statement 3 →

$D \wedge I$

Statement 4 →

$I \wedge \neg R$

Statement 5 →

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

Statement 3, 4 and 5 cant be written informal of Propositional Logic, as we are selectively talking about some dolphins. As propositional logic deal with wholes, it is not possible.

Predicates and First Order Logic Representation:

Read(x) → x can read.
Literate(y) → y is literate.
Intelligent(x) → x is intelligent.
Dolphin(x) → x is a dolphin.

Page :
Date :

Statement 1 →

$\forall x (Read(x) \rightarrow Literate(x))$

Statement 2 →

$\forall x (Dolphin(x) \rightarrow \neg Literate(x))$

Statement 3 →

$\exists x (Dolphin(x) \wedge Intelligent(x))$

Statement 4 →

$\exists x (Intelligent(x) \wedge \neg Read(x))$

Statement 5 →

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

Resolution for Statement 4.

Page :
Date :

Our KB includes the statement 1, 2, and 3.

$KB \Rightarrow \forall x (\neg R(x) \vee L(x)) \wedge \forall x (\neg D(x) \vee \neg L(x)) \wedge \exists x (D(x) \wedge I(x))$

Our Query is $\Rightarrow \exists x (I(x) \wedge \neg R(x))$

Proving that KB $\wedge \neg$ Query results into an empty clause,

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

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

$D(c)$

$I(c)$ { taking c as a subset of x.

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

$\therefore (\neg R(x) \vee \neg L(x)) \wedge (\neg D(x) \vee \neg L(x)) \wedge (D(c)) \wedge (I(c)) \wedge (\neg I(x) \vee R(x))$

Taking $D(c)$ and $\neg D(x) \vee \neg L(x)$

$\hookrightarrow \neg L(z)$

Taking $I(c)$ and $\neg I(x) \vee R(x)$

$\hookrightarrow R(z)$.

Taking $\neg L(z)$ and $\neg R(x) \vee \neg L(x)$ we get

$\Rightarrow \neg R(z)$

Taking $\neg R(c)$ and $R(c)$, we get an empty clause.

Thus by contradiction, the query is supposed to be satisfiable.

Resolution for Statement 5

Page :
Date :

The knowledge base includes.

$\forall x \ (R(x) \rightarrow L(x))$

$\forall x \ (O(x) \rightarrow \neg L(x))$

$\exists x \ (O(x) \rightarrow I(x)) \Rightarrow O(c)$ and $I(c)$

$\exists x \ (I(x) \rightarrow \neg R(x)) \Rightarrow I(c)$ and $\neg R(c)$

The Query to be proved.

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

$\Rightarrow \exists x \ (O(x) \wedge I(x) \wedge R(x)) \wedge \forall y \ (\neg(O(x) \wedge I(y) \wedge R(y)) \vee \neg L(y))$

$\Rightarrow$ Taking some 'c' which satisfies this statement.

For some c, the statement (query simplifies to $\neg L(c)$)

$\forall c \ (O(x) \wedge I(x) \wedge R(x))$

Proving that KB $\wedge$ ~Query results in an empty clause.

∴

From $O(c)$ and $\neg O(x) \vee \neg L(x)$

$\hookrightarrow \neg L(c)$

From $\neg L(c)$ and $\neg R(x) \vee L(x)$

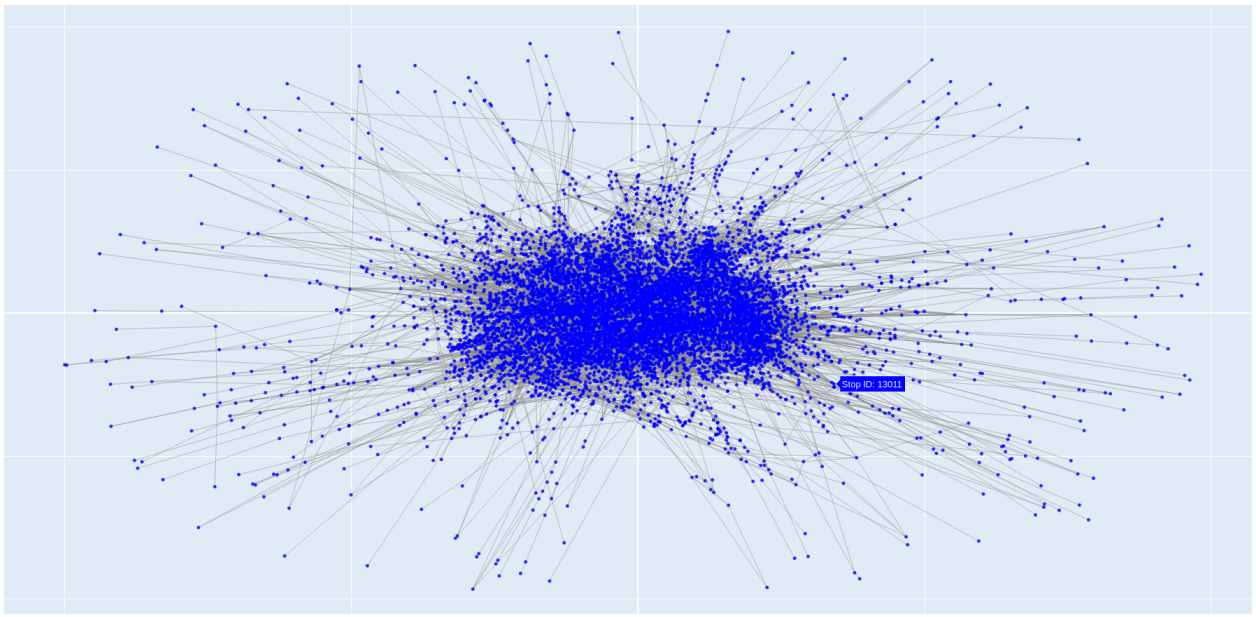
$\hookrightarrow \neg R(c)$

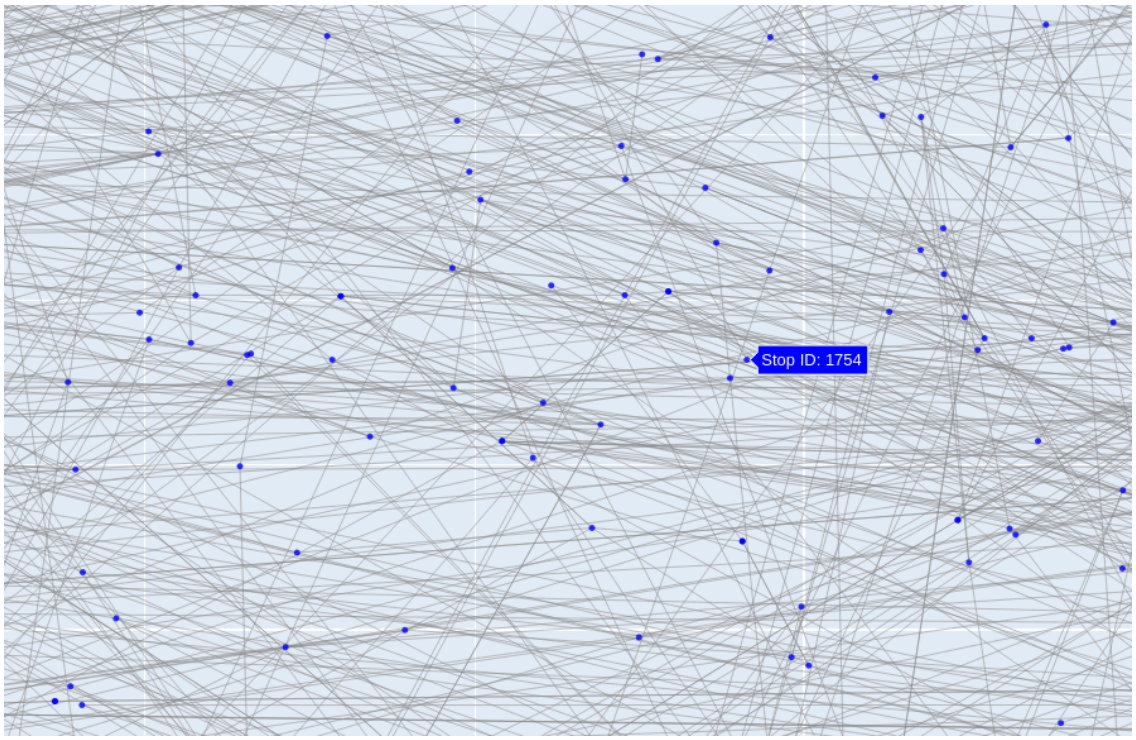
Now for all the clauses we are left with terms which can't be resolved any further. Thus this means that the query is unsatisfiable.

Computational:

Question 1) Data Loading and Knowledge Base Creation:

2D Graph made using route_to_stops, where nodes correspond to the stops and the edges signify the routes passing through different stops.





Question 2) Reasoning:

a) Evaluation of time and space complexities of FOL query_direct_route approach and the direct_route_brute_force approach:

```
Total time for query_direct_routes: 2.102134 seconds
Total memory for query_direct_routes: 2.236057 MB
Total time for direct_route_brute_force: 0.002547 seconds
Total memory for direct_route_brute_force: 0.004257 MB
```

(Iterated through 0 to 30 stops for start and end stops respectively, leading to about 900 queries and calculated the time and space complexities.)

- Time Complexity:
  - Time for FOL query_direct_route > Time for Brute_force_direct_route
  - FOL-Based Approach: Requires extra time for setting up a knowledge base and processing inference rules for each query, making it slower per query than direct checks.
  - Brute-Force Approach: Directly iterates through routes and stops, avoiding setup and inference, resulting in faster query times.
- Space Complexity:
  - Space for FOL query_direct_route > Space for Brute_force_direct_route
  - FOL-Based Approach: Stores each stop-route relationship and additional rule-based structures, increasing memory usage.
  - Brute-Force Approach: Uses only the original data dictionary without extra storage, making it more space-efficient.

b) Intermediate steps involved in the Brute Force solution of finding the direct routes between two stops.

- **Initialize an Empty List**: Create an empty list, route_ids_list, to store all valid route IDs that directly connect the specified stops.
- I**terate Over Routes**: Loop through each route_id and its corresponding list of stops in the route_to_stops dictionary.
- **Check for Both Stops in Each Route**: For each route, check if both start_stop and end_stop are present in the list of stops associated with the current route_id.
- **Add Valid Route IDs**: If both stops are found in the route's stop list, add the route_id to route_ids_list.
- **Return the Result:** After iterating through all routes, return route_ids_list, which now contains all route IDs directly connecting the specified stops.

Intermediate steps involved in the FOL Library-Based Reasoning approach of finding the direct routes between two stops.

- **Initialize and Clear Terms**: Clear any previously defined terms and initialize the terms and predicates needed for reasoning (RouteHasStop, DirectRoute, OptimalRoute, etc.).
- **Define the rule for a direct route:** DirectRoute(X, Y, R) <= (RouteHasStop(R, X) & RouteHasStop(R, Y)), which states that a direct route exists if both stops are associated with the same route.
- **Populate the Knowledge Base:** For each route in route_to_stops, add facts to the knowledge base by associating each stop with its route using the RouteHasStop predicate. This sets up the foundational data for querying.
- **Query for Direct Routes:** Use the DirectRoute rule to query the system for routes that connect the specified start and end stops. The query will automatically infer connections based on the defined rule and stored facts.
- **Return the Sorted Result:** Collect the route IDs returned by the query and return them in a sorted order.

c) Brute-Force Solution Complexity

- Time Complexity: **O(m×n)**, where m is the number of route IDs, and n is the maximum number of stops in a route.
- For each query with specific start and end stops, the brute-force approach iterates through all routes, checking for the presence of both stops in each route. This results in a complexity of O(m×n) for each query.

FOL Library-Based Approach Complexity

- Time Complexity for Knowledge Base Creation: The initial setup involves creating and populating the knowledge base with facts (RouteHasStop). This setup only occurs once and is then cached, making subsequent queries significantly faster.
- Therefore, the query complexity is proportional to **O(Complexity of RouteHasStop × Complexity of RouteHasStop).** PyDatalog optimizes this process, leading to reduced complexity and faster execution compared to brute-force, especially for multiple queries.

Question 3) Planning:

a) Time and Space complexity check for Forward and Backward Chaining

```
Total time for Forward Chaining: 5.916114 seconds
Total memory for Forward Chaining: 83.935605 MB
Total time for Backward Chaining: 5.817608 seconds
Total memory for Backward Chaining: 85.281628 MB
```

(Iterating through 0 to 21 stops ids for start_stop, end_stop and stop_to_include. Total iteration for query = 8000)

Complexity = **O(Complexity of RouteHasStop X Complexity of RouteHasStop X Complexity of DirectRoute)**

Since we are finding all direct routes between the start and end stop for both forward and backward chaining, the time and space complexities for these approaches are comparable.

b)

- Initialize and Clear Terms: Clear any previously defined terms and initialize the necessary terms and predicates needed for reasoning (e.g., RouteHasStop, DirectRoute, OptimalRoute, etc.)
- Define the Rule for an Optimal Route:
    - OptimalRoute(X, Y, Z, R1, R2) <= (DirectRoute(X, Z, R1) & DirectRoute(Z, Y, R2) & (R1 != R2)). This rule states that an optimal route exists from X to Y via Z, using two different routes R1 and R2.
- Populate the Knowledge Base as before.
- Query the Knowledge Base for Optimal Routes:
    - For forward chaining, query the knowledge base for optimal routes starting from start_stop_id and ending at end_stop_id, passing through stop_id_to_include.
    - For backward chaining, query the knowledge base for optimal routes starting from end_stop_id and going back to start_stop_id, also passing through stop_id_to_include.
    - This query will infer all valid route combinations that satisfy the transfer condition, ensuring that two different routes are used for the transfer.

- Return the Result: After the query executes, return the list of valid route combinations that include the intermediate stop stop_id_to_include and satisfy the transfer condition (using two different routes, R1 and R2).

c) Forward chaining infers routes starting from the start_stop_id towards end_stop_id, while backward chaining starts from end_stop_id and works backward to start_stop_id.
Despite this directional difference, each approach ultimately evaluates the same relationships and constraints, so their total step count remains effectively equal.

Bonus part)

a) Time and space complexity for pddl planning:

```
Total time for PDDL Planning: 5.886281 seconds
Total memory for PDDL Planning: 84.596599 MB
```

(Iterating through 0 to 21 stops ids for start_stop, end_stop and stop_to_include. Total iteration for query = 8000)

b) After initialization of the datalog that includes creating the knowledge base and defining the predicates like BoardRoute, TransferRoute and PDDL, we try to run the query for PDDL. The steps mainly include:

Step 1: Check if start_stop_id is served by a route (BoardRoute predicate).
Step 2: Identify if a transfer is possible from start_stop_id to stop_to_include using TransferRoute.
Step 3: Verify if the destination (end_stop_id) is served by a different route after the transfer.

The PDDL predicate is defined as follows:

```
PDDL(X, Y, Z, R1, R2) <= (
        BoardRoute(X, R1) &
        TransferRoute(Y, R1, R2) &
        BoardRoute(Z, R2)
)
```

c) The similarity in time and space complexity across forward chaining, backward chaining, and PDDL planning is because of:
   - Rule and Constraint Similarity: All three methods—forward chaining, backward chaining, and PDDL—are querying the same underlying rules (DirectRoute, OptimalRoute, etc.). These rules have similar levels of complexity and entail evaluating similar conditions (e.g., checking route-stop mappings, ensuring transfer points are distinct, etc.). This results in each approach performing similar underlying operations.

- Shared Intermediate Data Structures: Each method leverages the same knowledge base populated with route-stop relationships. Since they rely on similar or identical intermediate computations (like accessing RouteHasStop and DirectRoute), memory usage is comparable. Each approach stores a similar amount of intermediate states and paths, leading to nearly identical memory footprints.

The optimal routes found through forward chaining, backward chaining and PDDL are similar