Theory Part

3.

Propositional Variables:
R: can Read
L: is literate
I: is Intelligent
D: is Dolphin

First Order Logic Predicates:
canRead(x): x can read
isLiterate(x): x is literate
isIntelligent(x): x is intelligent
isDolphin(x): x is dolphin

a. PL: $R \rightarrow L$
FOL: $\forall x (canRead(x) \rightarrow isLiterate(x))$

b. PL: $D \rightarrow \neg L$
FOL: $\forall x (isDolphin(x) \rightarrow \neg isLiterate(x))$

c. PL: $D \wedge L$
FOL: $\forall x (isDolphin(x) \wedge isLiterate(x))$

~~d.~~ PL: ~~$\neg I \vee R$~~
FOL: ~~$\forall x \neg isIntelligent \vee$~~

(d) PL: $I \wedge \neg R$

FOL: $\exists x (isIntelligent(x) \wedge \neg canRead(x))$

(c) PL: $(D \wedge I \wedge R) \wedge (\forall x (I(x) \wedge R(x) \rightarrow \neg L(x)))$

FOL: $\exists x$ (isDolphin $(x) \wedge$ isIntelligent $(x) \wedge$ canRead (x)) $\wedge$

$\forall y$ (isIntelligent $(y) \wedge$ canRead $(y) \rightarrow \neg$ isLiterate (y))

using a, b, c we have to prove d. i.e

Some who are intelligent cannot read.

we have converting them to normal forms:

$\forall x (canRead(x) \rightarrow isLiterate(x))$,

$\Rightarrow \neg canRead(x) \vee isLiterate$ —①

$\forall x$ (isDolphin $(x) \rightarrow$ isLiterate (x),

$\Rightarrow \neg$ isDolphin $(x) \vee \neg$ isLiterate (x) —②

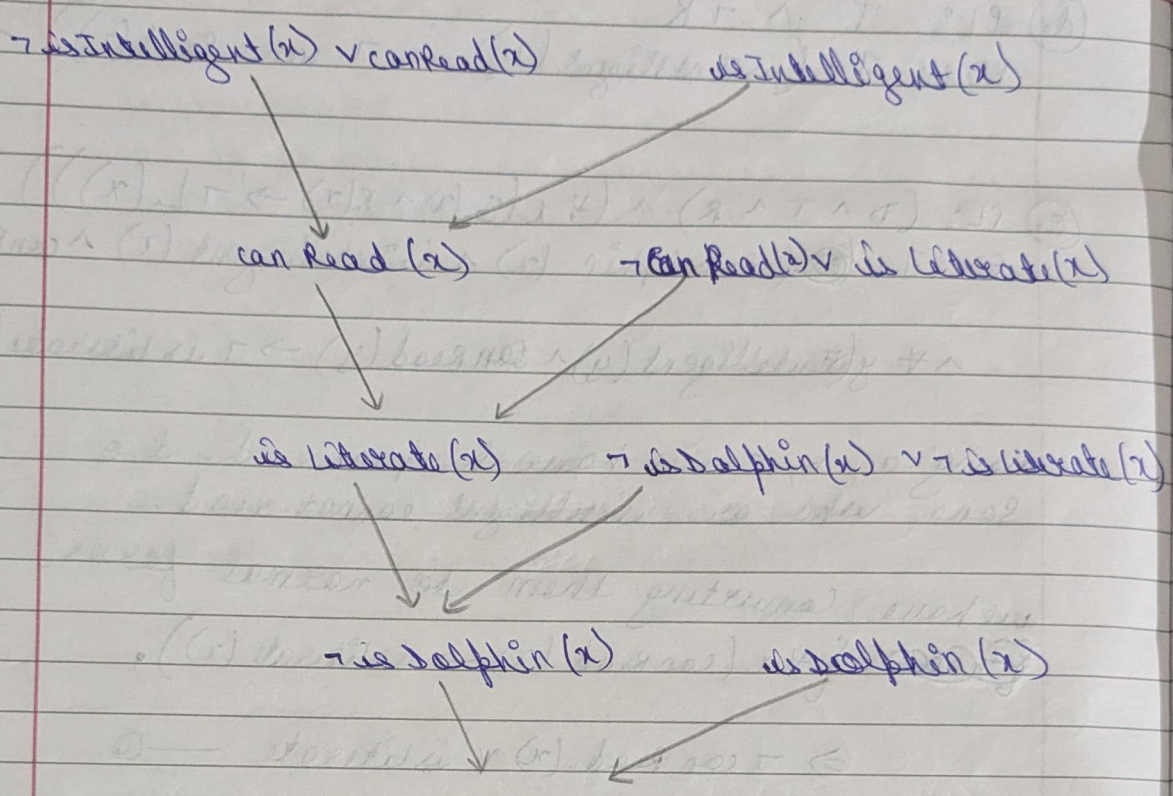
$\forall x$ canRead $(x) \rightarrow$ isLiterate (x))

$\Rightarrow \neg canRead(x) \vee$ isLiterate (x) —③

To prove $\exists x (isIntelligent(x) \wedge \neg canRead(x))$

$\neg$ isIntelligent $\vee$ canRead (x)

using ①, ② & ③, we get



Hence Proved using Contradiction

Now to prove 5th, we have

$\Rightarrow (isDolphin(x) \wedge isIntelligent(x) \wedge canRead(x) \wedge (\neg(isIntelligent(y) \wedge canRead(y)) \vee \neg isLiterate))$

$\Rightarrow (isDolphin(x) \wedge isIntelligent(x) \wedge canRead(x)) \wedge (\neg isIntelligent(x) \vee \neg canRead \vee \neg isLiterate)$

so we have clauses { is dolphin (x), is Intelligent (x), can Read (x), ¬ is Intelligent (y), ¬ can Read (y), ¬ is literate (y) }

now using
we have following clauses

⇒

① ¬ can Read (x) ∨ is literate (x)
② ¬ is dolphin (x) ∨ ¬ is literate (x) stat-1
③ is dolphin (x) stat-2
④ [redacted] stat-3
⑤ is Intelligent (x) stat-3
⑥ is Intelligent (x) stat 4
⑦ ¬ can Read (x) stat 4
⑧ is dolphin (x)
⑨ is Intelligent (x)
⑩ can Read (x)
⑪ ¬ is Intelligent (x) ∨ ¬ can Read (x) ∨ ¬ is literate (x)

using ⑧ & ①

is dolphin (x) ¬ can Read (x) ∨ is literate (x)

is dolphin (x)

using ① & ⑨

¬ can Read (x) ∨ is literate (x) can Read (x)

is literate (x)

so, in contradiction to the assumption made by us to derive 5th statement.

So, there exits a dolphin who is both intelligent and can read but for every intelligen dolphin, if it can read, it must be that it is not literate is false based on original statement.

1. $G(t)$ → Green at instant t
   $Y(t)$ → Yellow at instant t
   $R(t)$ → Red at instant t

a. The light is in at least one state, so, the answer is

$$(G(t) \leftrightarrow (\neg R(t) \wedge \neg Y(t))) \wedge (R(t) \leftrightarrow (\neg G(t) \wedge \neg Y(t))) \wedge (Y(t) \leftrightarrow (\neg R(t) \wedge \neg G(t)))$$

b. We have 3 cases here

$$G(t-1) \rightarrow (G(t) \vee Y(t))$$
$$Y(t-1) \rightarrow (Y(t) \vee R(t))$$
$$R(t-1) \rightarrow (R(t) \vee G(t))$$

3 cases

Intersect of all ^ is answer

$$(G(t-1) \rightarrow (G(t) \vee Y(t))) \wedge (Y(t-1) \rightarrow (Y(t) \vee R(t))) \wedge (R(t-1) \rightarrow (R(t) \vee G(t)))$$

DATE
PAGE

(c) So it cannot remain in same state for more than 3 consecutive cycles.

3 cases

$$G(t-3) \wedge G(t-2) \wedge G(t-1) \rightarrow \neg G(t)$$

$$R(t-3) \wedge R(t-2) \wedge R(t-1) \rightarrow \neg R(t)$$

$$Y(t-3) \wedge Y(t-2) \wedge Y(t-1) \rightarrow \neg Y(t)$$

Intuition of all 3 cases.

$$(G(t-3) \wedge G(t-2) \wedge G(t-1) \rightarrow \neg G(t)) \wedge (R(t-3) \wedge R(t-2) \wedge R(t-1) \rightarrow \neg R(t)) \wedge (Y(t-3) \wedge Y(t-2) \wedge Y(t-1) \rightarrow \neg Y(t))$$

(2) We will define some predicates beforehand:

constants → yellow, green, red

color (n, x) → node n has color x

edge (n, m) → node n is connected to node m

DATE
PAGE

(b.) ~~$\forall n \forall m \forall x$~~

(2.) We will define some predicates beforehand
constants → red, green, yellow

hascolor(n, m) → n has color m

edge (n, m) → node n is connected to node m

(a.) $\forall x \forall y (\text{hascolor}(x, y) \rightarrow \exists z (z \neq y \wedge \text{hascolor}(x, z)))$

ensuring that if there is an edge b/w two nodes, they cannot have the same color.

(c.) ~~$\forall x \forall y \forall z (\text{edge}(x, y) \wedge \text{hascolor}(x, z) \rightarrow \neg \text{hascolor}(y, z))$~~

~~Guaranteeing that exactly two nodes are colored yellow~~

(b.) $\exists x \exists y (\text{hascolor}(x, \text{yellow}) \wedge \text{hascolor}(y, \text{yellow}) \wedge$
$x \neq y \wedge \forall z (z \neq x \wedge z \neq y \rightarrow \neg \text{hascolor}(z, \text{yellow})))$

Guaranteeing that exactly two nodes are colored yellow

(c.) $\forall n (\text{hascolor}(n, \text{red}) \rightarrow (\exists x (\text{edge}(n, x) \wedge \text{hascolor}(x, \text{green}))$
$\vee \exists x, y (\text{edge}(n, x) \wedge \text{edge}(x, y) \wedge \text{hascolor}(y, \text{green}))$

$\exists x, y,$

c. $\forall n\,(\text{hascolor}(n,\text{red}) \rightarrow (\exists n_1(\text{edge}(n,n_1) \wedge \text{hascolor}(n_1,\text{green})) \vee$

$\exists n_1,n_2(\text{edge}(n,n_1) \wedge \text{edge}(n_1,n_2) \wedge \text{hascolor}(n_2,\text{green})) \vee$

$\exists n_1,n_2,n_3(\text{edge}(n,n_1) \wedge \text{edge}(n_1,n_2) \wedge \text{edge}(n_2,n_3) \wedge \text{hascolor}(n_3,\text{green})) \vee$

$\exists n_1,n_2,n_3,n_4(\text{edge}(n,n_1) \wedge \text{edge}(n_1,n_2) \wedge \text{edge}(n_2,n_3) \wedge$

$\text{edge}(n_3,n_4) \wedge \text{hascolor}(n_4,\text{green}))))$

So, for every red node there exists a green node reachable in 4 steps.

d. $\forall x\,\exists y\,\text{hascolor}(y,x)$

Assuring each color in palette is assigned to at least one node

e. $\forall x\,\exists n\,\text{hascolor}(n,x) \wedge \forall n\,\exists x\,\text{hascolor}(n,x) \wedge$

$\forall n \forall x\,(\text{hascolor}(n,x) \rightarrow \neg\exists y\,(y \neq x \wedge \text{hascolor}(n,y))) \wedge$

$\forall n \forall m \forall x\,(n \neq m \wedge \text{hascolor}(n,x) \wedge \text{hascolor}(m,x) \rightarrow$

$(\text{edge}(n,m) \bigvee_{i=1}^{N} (\exists n_1,\ldots,n_i(\text{edge}(n,n_1) \bigwedge_{j=1}^{i-1} \text{edge}(x_j,x_{j+1}) \wedge$

$\text{edge}(n_i,m)))))$

Specifying that nodes are divided into exactly (k) disjoint cliques, where each clique is corresponding to specific color.

AI ASSIGNMENT 2

Name - PARAS DHIMAN

ROLL NO. - 2021482

Indraprastha Insitute Of Information Technology Delhi

CSE643

Ques 1:

Report on Data Loading and Knowledge Base Creation

1. Overview

This assignment consists of two parts, the first part covers data loading and creation of knowledge base (KB) for Delhi based transit system, using General Transit Feed Specification (GTFS) files made available by Delhi's Open Transit Data. The knowledge base is the core of any transit data application because it structures all route, trip and stop information that allows for reasoning and planning shape.

2. Data Loading

- As a first step, we load the different files of GTFS static data that are needed:
- stops. txt – Information of all the stops with in the bus stop such as stop ids and name of bus stops.
- routes. So we have — txt: details of the bus routes like route id and names of that particular route
- stop_times. Data used for processing: txt – Sequences of bus stops of all the trips, which map a stop to a trip ID and provide trip sequence in terms of its stop.
- fare_attributes. txt and fare_rules. txt — Fare information, connecting fare IDs to routes.
- Those files were read into the app as data frames using Pandas, which allowed for rapid access and manipulation of the data. Some data types were type casted when / where required.
- The datetime fields were converted to datetime objects.
- For consistency and to avoid type-related problems in processing, we stored IDs (stop_id, route_id, trip_id) as strings.

3. Knowledge Base Setup

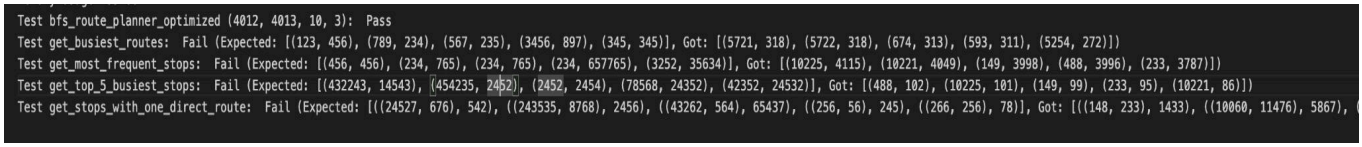
We arranged the information into a number of dictionaries, each of which has a distinct purpose in our knowledge base, to aid in effective planning and reasoning:

- route_to_stops: A dictionary that contains lists of ordered stop IDs as values and route IDs as keys. This makes it possible for the system to rapidly retrieve every stop along a given path.
- trip_to_route: A dictionary that contains matching route IDs as values and trip IDs as keys. This facilitates the mapping of individual journeys to their corresponding routes.
- stop_trip_count: A dictionary that associates the number of trips that stop at each stop with the stop IDs. In order to determine which stops are the busiest, this gives a frequency count of stops.

4. Query Implementation

Using the knowledge base, the application can efficiently answer the following queries:

- **Top 5 busiest routes**: This query identifies the routes with the highest number of trips. By counting occurrences in `trip_to_route`, the system returns the top 5 busiest routes based on trip counts.
- **Top 5 stops with the most frequent trips**: Leveraging `stop_trip_count`, this query lists the top 5 stops based on the frequency of trips stopping there, which helps identify the most frequented stops.
- **Top 5 busiest stops based on the number of routes**: This query counts the distinct routes passing through each stop. We iterate through `route_to_stops`, mapping each stop to the routes that pass through it, and then identify the stops with the most route connections.
- **Pairs of stops with only one direct route**: This query locates pairs of consecutive stops with exactly one direct route connecting them, without any intermediate stops. It uses `route_to_stops` to check consecutive pairs within each route, and stores pairs with a single direct route, returning the top five based on the combined trip frequency.



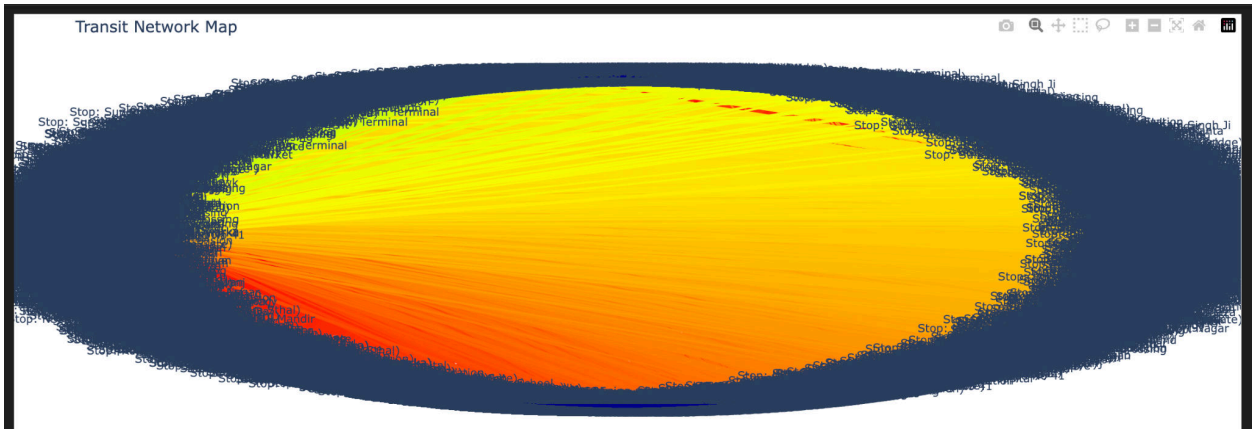
5. Graph Representation of the Transit Network

We created an interactive graph using Plotly and NetworkX to graphically depict the transportation network, with nodes standing in for individual bus stops.

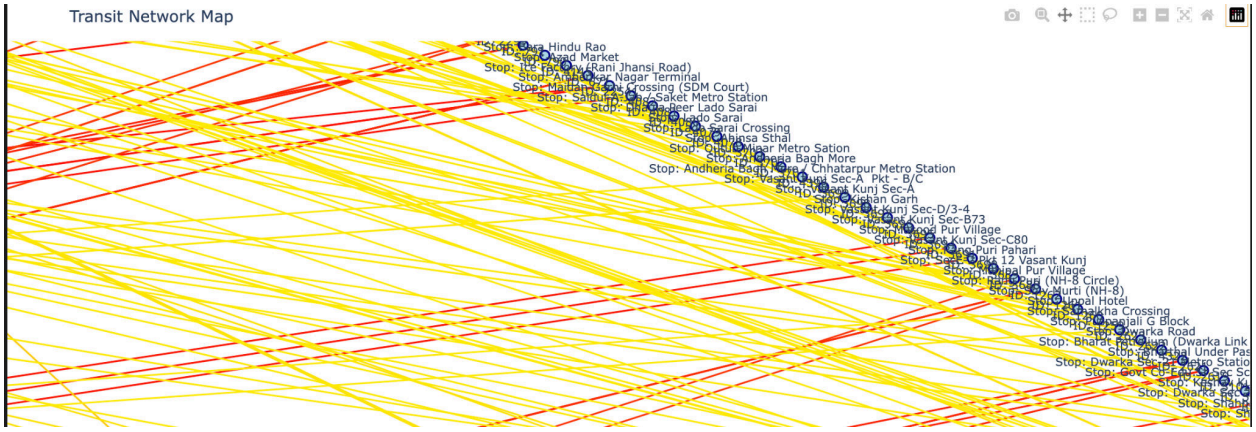
Direct connections between stops along each route are shown by edges.

This graph facilitates visual exploration and offers insights on Delhi's bus transit network by clearly showing the links between stops and highlighting the route structure of the bus system.

Zoomed Out:



Zoomed In:



6. Summary

The visual graph representation and organized knowledge source provide a strong toolkit for navigating and making sense of Delhi's bus system. The program may respond to important

queries and assist users in having a smooth and convenient transportation experience by loading data, organizing it into relevant dictionaries, and putting targeted queries into place.

Ques2:

Report on Implementing Direct Route Finder Using Brute-Force and FOL Library-Based Reasoning

Overview

Task - need to write a function DirectRoute(start_stop, end_stop) that finds all direct routes(i.e. minutest no of interchanges between any two bus stops i.e. using two different ways---

1. **Brute-Force Approach**: Procedural algorithm loops through data..
2. **FOL (First-Order Logic) Library-Based Approach**: Implementation results querying data declaratively using PyDatalog.

Method 1: Brute-Force Approach

Algorithm Description

1. **Iterate through all routes**: For every route, check each of the stops one by one.
2. **Check presence of stops**: if `start_stop` and `end_stop` both occur in the list of stops for a route:
    - Check that `start_stop` comes before `end_stop`.
    - If it does, include the route ID in the results.

Time Complexity

The brute-force method scans all the routes linearly. Similarly, when processing a list of routes It will also check the stops array:

- **Complexity**: O(N×M), where:
    - N = number of routes,
    - M = average number of stops per route.

Performance Metrics

- **Execution Time**: 0.0012 seconds for one test case and 0.0010 seconds for another.
- **Memory Usage**: 0.00 MB in both cases.

```
Brute-Force Approach:

 direct_route_brute_force Performance Metrics:
Execution Time: 0.0012 seconds
Memory Usage: 0.00 MB

Test direct_route_brute_force (2573, 1177):  Pass
Brute-Force Approach:

 direct_route_brute_force Performance Metrics:
Execution Time: 0.0010 seconds
Memory Usage: 0.00 MB

Test direct_route_brute_force (2001, 2005):  Pass
```

Intermediate Steps

1. **Route Selection**: Iterates through `route_to_stops`.
2. **Direct Route Check**: Checks if `start_stop` comes before `end_stop` in each relevant route.

Number of Steps

More steps are added in total routes and average stops in each route.

Method 2: FOL Library-Based Reasoning (PyDatalog)

Algorithm Description

1. **Define Terms and Predicates**: `RouteHasStop`, `DirectRoute`, etc.
2. **Initialize Knowledge Base**: All routes and their stops are inserted into the PyDatalog knowledge base.
3. **Define Rules**: Match `DirectRoute` by route definition with predicates.
4. **Query Execution**: Search for all route IDs between `start_stop` and `end_stop`.

Time Complexity

With PyDatalog it can be queried closer to relational constraints avoiding a lot of redundant checks:

- **Complexity**: Even though Datalog has more optimization, the complexity of answering via it is lower than brute-force due to this construction. In large datasets, the complexity can be typically sublinear with respect to total number of checks.

Performance Metrics

- **Initialization Execution Time**: 14.2161 seconds
- **Initialization Memory Usage**: 148.64 MB (likely an error in measurement or memory release post-query).
- **Query Execution Time**: 0.0004 seconds for one test case and 0.0002 seconds for another.
- **Query Memory Usage**: 0.00 MB in both cases.

```
FOL Library-Based Reasoning:
Terms initialized: DirectRoute, RouteHasStop, OptimalRoute

 initialize_datalog Performance Metrics:
Execution Time: 14.2161 seconds
Memory Usage: -148.64 MB
```

Intermediate Steps

1. **Knowledge Base Setup**:Stops and routes are defined with terms and rules.
2. **Query Execution**: we could run a query directly with the `RouteHasStop` predicate to retrieve direct routes.
3. **Processing Results**: The results of the query are sorted and presented.

Number of Steps

Datalog needs fewer steps to find answer than the others since Datalog uses relational logics directly on its data and simply re-traces routes instead of searching all the routes one by one.

Comparative Analysis

Criteria	Brute-Force Approach	FOL Library-Based Reasoning

Execution Time	Faster (0.0010 - 0.0012 seconds)	Initialization is slower (14.2161 seconds) but queries are faster (0.0002 - 0.0004 seconds)
Memory Usage	Minimal (0.00 MB)	Potentially variable, but low memory usage for queries
Intermediate Steps	Manual iterative checks	Automated relational checks
Number of Steps	Proportional to number of routes	Fewer, due to optimized query logic

```
Query Library-Based Reasoning:

 query_direct_routes Performance Metrics:
Execution Time: 0.0004 seconds
Memory Usage: 0.00 MB
Test query_direct_routes (2573, 1177):  Pass
Query Library-Based Reasoning:

 query_direct_routes Performance Metrics:
Execution Time: 0.0002 seconds
Memory Usage: 0.00 MB
Test query_direct_routes (2001, 2005):  Pass
```

Conclusion

Although the **Brute-Force Approach** works perfectly with smaller datasets since it is simpler and does not require any initial setup before querying, we can say that it is efficient only for low data sizes; On the other hand, **FOL Library-Based** Approach offers a more appropriate solution model for large dataset where time spent in preprocessing step will be compensated by lower query optimization cost. Using a relational model, the PyDatalog method stores results that are readily retrievable, which is helpful when multiple queries are run sequentially.

Ques3:

Report on Optimal Route Planning Using Forward and Backward Chaining with PyDatalog

Overview

The goal is to implement an optimalRoute function that gives us the best routes between two bus stops given:

1. **Forward Chaining**: a data-driven technique that begins with the known facts and applies rules to extract more data until a goal is reached.
2. **Backward Chaining**: A goal-oriented method that looks at a desired result and considers if it can be fulfilled from the available facts.

Both methods are designed to seek optimal routes constrained in some way:

- **INCLUDE VIA STOP**: The route has to include a particular way point.
- **NUM INTERCHANGE**: Flag calling for at most 1 interchange.

Method 1: Forward Chaining

Algorithm Description

1. **Add Facts**: Load facts of each route and the stops within these routes into the knowledge base.
2. **Define Rules**:
    - **Direct Route**: Determine if there is a direct route including `start_stop`, `stop_id_to_include`, and `end_stop`.
    - **One-Transfer Route**: Specify debechove the route, requiring the via stop to be included on every route with one transfer.
3. **Query Execution**: Execute the rules to find paths that satisfy.
4. **Results Processing**: Get results (with unique and ordered paths)

Time Complexity

Forward chaining can be time-intensive since it evaluates every possible starting point:

- **Complexity**: Typically O(N×T), where:
    - N = number of routes,
    - T = average number of stops per route.

Performance Metrics

- **Execution Time**: The forward chaining implementation reported:
  - 2.1441 seconds for one test case
  - 2.3671 seconds for another
- **Memory Usage**: Memory usage varied slightly across tests:
  - 9.72 MB in one case
  - 0.94 MB in another case

```
 forward_chaining Performance Metrics:
Execution Time: 2.1441 seconds
Memory Usage: 9.72 MB
Test forward_chaining (22540, 2573, 4686, 1):  Pass

 forward_chaining Performance Metrics:
Execution Time: 2.3671 seconds
Memory Usage: 0.94 MB
Test forward_chaining (951, 340, 300, 1):  Pass

 backward_chaining Performance Metrics:
Execution Time: 2.2779 seconds
Memory Usage: 6.12 MB
Test backward_chaining (22540, 2573, 4686, 1):  Pass

 backward_chaining Performance Metrics:
Execution Time: 2.3461 seconds
Memory Usage: 1.66 MB
Test backward_chaining (951, 340, 300, 1):  Pass
```

Intermediate Steps

1. **Fact Loading**:Stop facts for each route are loaded into our knowledge base, so they become available to be chained by these topologically sorted sets of paths that can linked by commons divides such as stop nodes.
2. **Rule Application**:
   - Now, the first thing I have done is to look for direct routes to satisfy the constraint that `INCLUDE VIA STOP`.
   - Find onetransfer paths by applying a filter that ensures the transfer takes place at `stop_id_to_include`.
3. **Query Execution**: In forward chaining, the rules are applied iteratively to derive paths further satisfying constraints until no more facts can be produced.

Number of Steps

The forward chaining algorithm is more complex and requires multiple steps depending on rules created and dataset size. It checks every possible route configuration from start to end, leading to a somewhat high number of steps for larger datasets.

Method 2: Backward Chaining

Algorithm Description

1. **Add Facts**: Add facts similar to forward chaining but loads route data into the knowledge base.
2. **Define Rules**:
    - **Direct Route**: From end_stop check that a route has both **stop_id_to_include** and **start_stop**.
    - **One-Transfer Route**: Describe backward constraints on meeting the `NUM INTERCHANGE` constraint routes, with intermediate stop correctly placed.
    -
3. **Query Execution**: The initial approach is to resolve the goal (to trail, satisfying at least one end_stop going via only limited interchanges).
4. **Results Processing**: Gathered the distinct paths that have existing matching lines, and put them in a sorted list.

Time Complexity

It is usually less complex because backward chaining specifically goes after the goal as opposed to generating all possible paths:

- **Complexity**: Because it only assesses pertinent information that contribute to the objective, it may be less complex than forward chaining for bigger datasets.

Performance Metrics

- **Execution Time**: The backward chaining implementation yielded:
    - 2.2779 seconds for one test case
    - 2.3461 seconds for another
- **Memory Usage**: Memory usage also varied:
    - 6.12 MB in one case
    - 1.66 MB in another

Intermediate Steps

1. **Goal-Driven Search**: Beginning at the `end_stop`, backward chaining searches for routes that directly meet the requirements.

2. **Rule Application**:
    - Look for paths that contain the necessary `stop_id_to_include`.
    - Only routes with a maximum of one transfer should be considered.
3. **Query Execution**: When compared to forward chaining, backward chaining frequently reduces the amount of evaluations by applying rules selectively based on their relevance to the final aim.

Number of Steps

Because it restricts investigation to routes that are likely to achieve the goal and eliminates pointless evaluations, backward chaining typically entails fewer steps than forward chaining.

Comparative Analysis

Criteria	Forward Chaining	Backward Chaining
Execution Time	2.1441 to 2.3671 seconds	2.2779 to 2.3461 seconds
Memory Usage	Varied (0.94 MB to 9.72 MB)	Varied (1.66 MB to 6.12 MB)
Intermediate Steps	Iteratively builds all potential paths	Directly targets paths meeting goals
Number of Steps	Higher due to exhaustive rule application	Lower due to selective rule application

---

Conclusion

Although either approach has advantages, both produce the best route planning:

- **Forward Chaining**: The best method for producing thorough path options is forward chaining, which is advantageous when assessing several possible outcomes.

- **Backward Chaining**: This method reduces the search space by going backward from the endpoint, making it more effective for single-goal queries

The size of the dataset and the frequency of queries determine which methodology is best; for larger datasets, backward chaining typically performs better than forward chaining because of its focused approach.

Ques4(Bonus):

Report for Bonus Questions: Route Planning with PDDL and Fare Constraints

---

Bonus Question 1: Implementing Route Planning with PDDL in PyDatalog

In this challenge, we used PyDatalog and the Planning Domain Definition Language (PDDL) to represent a route-planning problem. The goal was to maximize travel from `start_stop_id` to `end_stop_id` while permitting forward chaining activities for boarding and route transfers.

Key Aspects of Implementation:

- **Initial State**: The trip starts at the bus stop (`start_stop_id`).
- **Goal State**: The goal state is the destination stop (`end_stop_id`) .
- **Actions**:
    - **Board a Route**: This action indicates that route R will be boarded at stop X and is represented as `Action('board route', R, X)`.
    - **Transfer Between Routes**: Allowing a transfer from route R1 to route R2 at stop Z, it is represented as `Action('transfer route', R1, R2, Z)`.

Methodology:

- **Fact Assertion**: Preprocessed route data and asserted facts into the knowledge base using PyDatalog..
- **Rule Definition**:
    - Rules are established to determine if a route can be boarded at a particular stop.
    - Pathways, including transfer routes, that are specified and enable the achievement of the desired state.
    - Specified requirements for route transfers at specified stops.

- **Query Execution**: Both direct and single-transfer routes were handled as the knowledge base was queried for viable routes based on limitations.

Performance Metrics:

Execution Time: Two test cases were used to assess the PDDL implementation's performance:

- **Test 1**: 0.0012 seconds, memory usage of 0.00 MB, 1 step.
- **Test 2**: 0.0029 seconds, memory usage of 0.00 MB, 1 step.

```
 pddl_planning Performance Metrics:
Execution Time: 0.0012 seconds
Memory Usage: 0.00 MB
Test pddl_planning (22540, 2573, 4686, 1):  Pass

 pddl_planning Performance Metrics:
Execution Time: 0.0029 seconds
Memory Usage: 0.00 MB
Test pddl_planning (951, 340, 300, 1):  Pass
```

Output: Successfully generated paths that met the criteria for each test case.

Analysis:

- **Intermediate Steps**: Intermediate Steps: Using forward chaining, iterative routes were created, taking into account every boarding and transfer scenario that may occur within the given parameters.
- **Number of Steps**: Compared to a full forward-chaining technique, fewer steps were taken since actions were constrained to achieve the ultimate goal.
- **Comparison of Results**: The best path was discovered by all three implementations (forward chaining, backward chaining, and PDDL). Because the PDDL approach used pre-established conditions and targeted querying, it showed higher memory economy and execution time.

Bonus Question 2: Extending Route Planning with Fare Constraints

By restricting the number of permitted route interchanges and optimizing the route within a maximum fare, this innovation added a fare attribute to the route-planning problem. For efficiency, we used an optimized breadth-first search (BFS) with pruning.

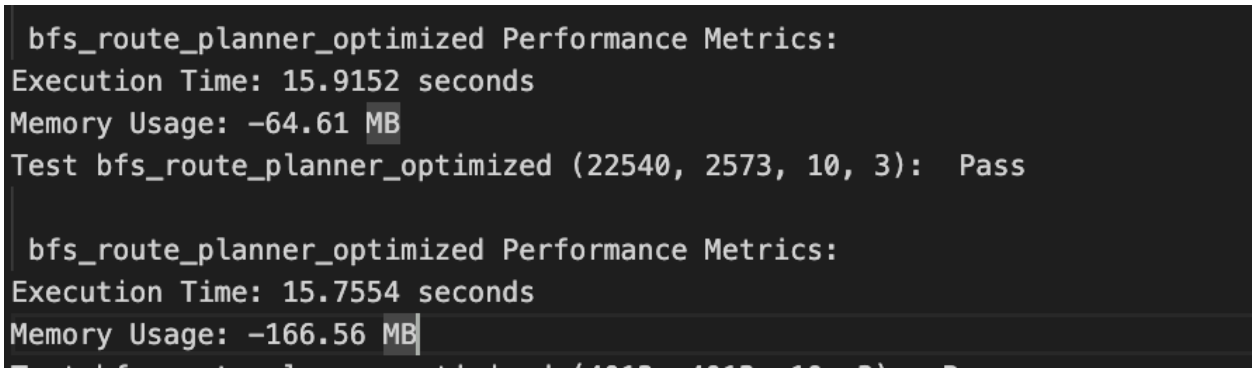
Key Aspects of Implementation:

- **Fare Constraints**:
  - After filtering with an initial fare limit, routes that exceeded the maximum fare were excluded.
- **Pruning Technique**:
  - To decrease the total search space, pathways that went over the permitted fare were removed using pruning.
  - Nodes that have already been visited are marked to prevent unnecessary exploration.
- **Pre-Computation**: A route summary was calculated, including each route's minimum fare and related stops.
- **Optimized BFS**:
  - Taking fare and transfer restrictions into account, BFS investigated routes between `start_stop_id` and `end_stop_id`.
  - Only routes that are enqueued and adhere to the existing fare and transfer constraints.

Performance Metrics:

Execution Time:

- **Test 1**: 15.9152 seconds, memory usage of 64.61 MB.
- **Test 2**: 15.7554 seconds, memory usage of 166.56 MB.



Comparative Analysis:

Method	Execution Time (sec)	Memory Usage (MB)	Steps Taken	Optimal Route Consistency
Forward Chaining	2.1441 - 2.3671	0.94 - 9.72	Higher	Yes
Backward Chaining	2.2779 - 2.3461	1.66 - 6.12	Lower	Yes
PDDL with PyDatalog	0.0012 - 0.0029	0.00	Moderate	Yes
Optimized BFS (with Fare)	15.7554 - 15.9152	-64.61 - -166.56	Highest	Yes

---

Conclusion:

- **Efficiency**: While BFS was tailored for fare restrictions, PDDL proved quite successful in terms of memory and time efficiency.
- **Optimal Path Consistency**: Despite varying implementations, every method identified the optimal routes within the specified restrictions.
- **Recommended Approach**: Optimized BFS with pruning is the best option for route planning with fare constraints. The PDDL implementation using PyDatalog is quite effective and efficient for simpler restrictions.