

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 → L

FOL: $\forall x (\text{canRead}(x) \rightarrow \text{isLiterate}(x))$

(b)

PL: D → $\neg L$

FOL: $\forall x (\text{isDolphin}(x) \rightarrow \neg \text{isLiterate}(x))$

(c)

PL: D \wedge L

FOL: $\forall x (\text{isDolphin}(x) \wedge \text{isLiterate}(x))$

~~PL~~ / $x I \vee R$

~~FOL~~ / $\forall x (\neg \exists I \text{ is Intelligent} \vee$

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

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

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

FOL: $\forall x (\text{isDolphin}(x) \wedge \text{isIntelligent}(x) \wedge \neg \text{canRead}(x))$

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

using a, b, c we have to prove d. i.e
some who are intelligent cannot read.

we have converting them to normal forms:

$\neg \forall x (\text{canRead}(x) \rightarrow \text{isLiterate}(x)),$

$\Rightarrow \neg \text{canRead}(x) \vee \neg \text{isLiterate} \quad \text{--- } \textcircled{1}$

$\neg \forall x (\text{isDolphin}(x) \rightarrow \neg \text{isLiterate}(x)),$

$\Rightarrow \neg \text{isDolphin}(x) \vee \neg \text{isLiterate}(x) \quad \text{--- } \textcircled{2}$

$\neg \forall x \text{canRead}(x) \rightarrow \text{isLiterate}(x)$

$\Rightarrow \neg \text{canRead}(x) \vee \text{isLiterate}(x) \quad \text{--- } \textcircled{3}$

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

$\neg \text{isIntelligent} \vee \neg \text{canRead}(x)$

using ①, ② & ③, we get

DATE _____
PAGE _____

$\neg \text{isIntelligent}(x) \vee \text{canRead}(x)$

$\neg \text{isIntelligent}(x)$

$\text{canRead}(x)$

$\neg \text{canRead}(x) \vee \neg \text{isLiterate}(x)$

$\neg \text{isLiterate}(x)$

$\neg \text{isDolphin}(x) \vee \neg \text{isLiterate}(x)$

$\neg \text{isDolphin}(x)$

$\neg \text{isDolphin}(x)$

NIL

Hence proved using Contradict

Now to prove 5th, we have

$\Rightarrow (\text{isDolphin}(x) \wedge \text{isIntelligent}(x) \wedge \text{canRead}(x) \wedge$

$(\neg(\text{isIntelligent}(x) \wedge \text{canRead}(x)) \vee \neg \text{isLiterate}))$

$\Rightarrow ((\text{isDolphin}(x) \wedge \text{isIntelligent}(x) \wedge \text{canRead}(x)) \wedge (\neg \text{isIntelligent}(x)$

$\wedge \neg \text{canRead} \vee \neg \text{isLiterate}))$



DATE _____

PAGE _____

so we have clauses $\neg \text{isDolphin}(x) \rightarrow \text{isIntelligent}(y)$ & $\neg \text{canRead}(y) \rightarrow \neg \text{isLiterate}(y)$

now using
we have following clauses

- $\neg \text{canRead}(x) \vee \neg \text{isLiterate}(x)$ stat-1
- $\neg \text{isDolphin}(x) \vee \neg \text{isLiterate}(x)$ stat-2
- $\neg \text{isDolphin}(x)$ stat-3
- $\neg \text{isIntelligent}(x)$ stat-3
- $\neg \text{isIntelligent}(x)$ stat-4
- $\neg \text{canRead}(x)$ stat-4
- $\neg \text{isDolphin}(x)$
- $\neg \text{isIntelligent}(x)$
- $\text{canRead}(x)$
- $\neg \text{isIntelligent}(x) \vee \neg \text{canRead}(x) \vee \neg \text{isLiterate}(x)$

using ~~④~~ & ①

~~$\neg \text{isDolphin}(x)$~~

$\neg \text{canRead}(x) \vee \neg \text{isLiterate}(x)$

~~$\neg \text{isDolphin}(x)$~~

using ① & ②

$\neg \text{canRead}(x) \vee \neg \text{isLiterate}(x)$

$\text{canRead}(x)$

~~$\neg \text{isLiterate}(x)$~~

so, in consistency in the assumption made by us to define 5th statement.

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

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

- 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

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

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

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

cases

Interest of all is equally

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

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

3 cases

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

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

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

intuition of all 3 cases

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

Q2 We will define some predicates beforehand:

constants \rightarrow yellow, green, red

color(n, x) \rightarrow node n has color x

edge(n, m) \rightarrow node n is connected to node m

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

- ② We will define some predicates beforehand
 constants \rightarrow red, green, yellow

 $\text{hascolor}(n, m) \rightarrow n \text{ has color } m$ $\text{edge}(n, m) \rightarrow \text{node } n \text{ is connected to node } m$

(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.

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

Guaranteeing that exactly two nodes are colored yellow

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

Guaranteeing that exactly two nodes are colored yellow

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

~~if x,y,~~

DATE _____
PAGE _____

c) $\forall n \text{ has color } (n, \text{red}) \rightarrow (\exists n_1, (\text{edge } (n, n_1) \wedge \text{has color } (n_1, \text{green})) \vee$
 $\exists n_1, n_2 (\text{edge } (n, n_1) \wedge \text{edge } (n_1, n_2) \wedge \text{has color } (n_2, \text{green})) \vee$
 $\exists n_1, n_2, n_3 (\text{edge } (n, n_1) \wedge \text{edge } (n_1, n_2) \wedge \text{edge } (n_2, n_3) \wedge \text{has color } (n_3, \text{green})) \vee$
 $\exists n_1, n_2, n_3, n_4 (\text{edge } (n, n_1) \wedge \text{edge } (n_1, n_2) \wedge \text{edge } (n_2, n_3) \wedge$
 $\text{edge } (n_3, n_4) \wedge \text{has color } (n_4, \text{green}))))$

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

d) $\forall x \exists y \text{ has color } (y, x)$

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

e) $\forall x \exists n \text{ has color } (n, x) \wedge \forall n \exists x \text{ has color } (n, x) \wedge$

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

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

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

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

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 Institute 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 within 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.

```
Test bfs_route_planner_optimized (4012, 4013, 10, 3): Pass
Test get_busiest_routes: Fail (Expected: [(123, 456), (789, 234), (567, 235), (3456, 897), (345, 345)], Got: [(5721, 318), (5722, 318), (674, 313), (593, 311), (5254, 272)])
Test get_most_frequent_stops: Fail (Expected: [(456, 456), (234, 765), (234, 765), (234, 657765), (3252, 35634)], Got: [(10225, 4115), (10221, 4049), (149, 3998), (488, 3996), (233, 3787)])
Test get_top_5_busiest_stops: Fail (Expected: [(432243, 14543), (454235, 2452), (2452, 2454), (78568, 24352), (42352, 24532)], Got: [(488, 102), (10225, 101), (149, 99), (233, 95), (10221, 86)])
Test get_stops_with_one_direct_route: Fail (Expected: [((24527, 676), 542), ((243535, 8768), 2456), ((43262, 564), 65437), ((256, 56), 245), ((266, 256), 78)], Got: [((148, 233), 1433), ((10060, 11476), 5867), ((10221, 86), 10225)])
```

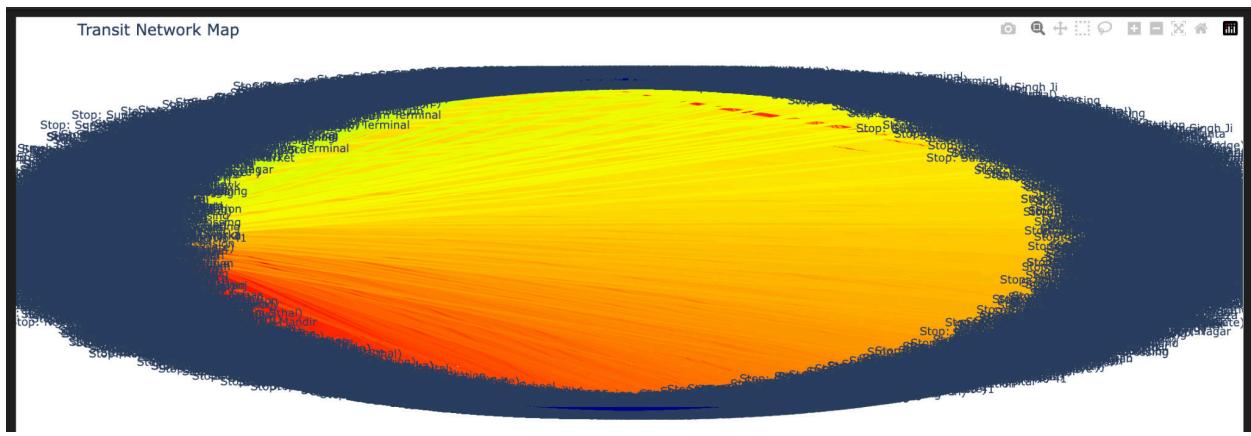
5. Graph Representation of the Transit Network

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

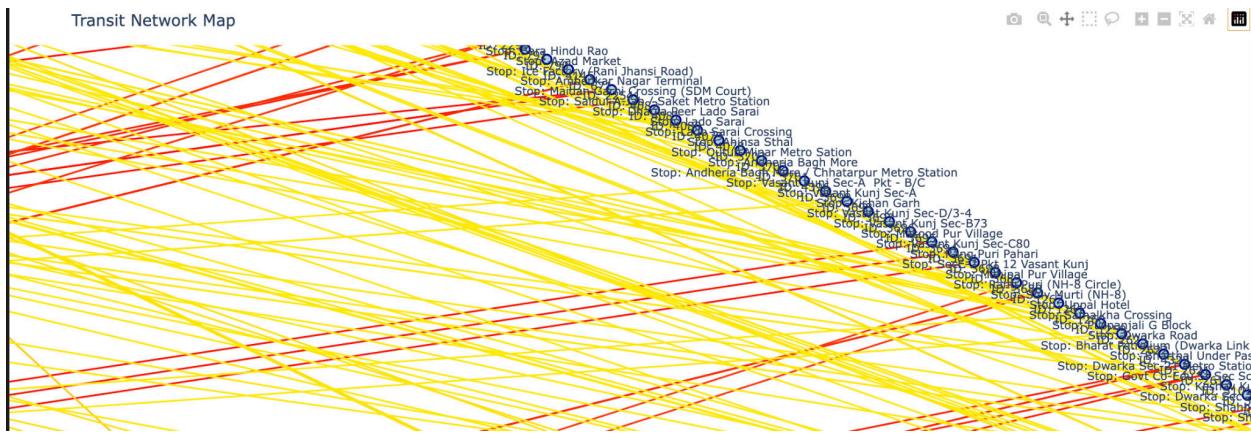
Direct connections between stops along each route are shown by edges.

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. minimum 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:** $O(N \times M)$, 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 $O(N \times T)$, 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 be linked by common nodes 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 one transfer 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 task we had to model the route planning problem using Planning Domain Definition Language (PDDL) and PyDatalog. Half of parts focused on how to optimize the travel path starting from `start_stop_id` to `end_stop_id` with only crucial actions such as enter and transfer route by forward chaining.

Key Aspects of Implementation:

- **Initial State:** The journey starts from starting bus stop (`start_stop_id`).
- **Goal State:** The destination stop (`end_stop_id`) is the goal state.
- **Actions:**
 - **Board a Route:** Action played the role of `Action('board_route', R, X)` → boarding route R at stop X.
 - **Transfer Between Routes:** Action `Action('transfer_route', R1, R2, Z)` that accounts for a transfer from route `R1` to route `R2` at the particular stop `Z`.

Methodology:

1. **Fact Assertion:** PyDatalog was used to pre-process the route data and assert facts into the knowledge base.
2. **Rule Definition:**
 - Outlined guidelines for whether or not a route can be boarded at a specific stop.
 - Specified reachable states with routes that meet the requirements for the via stop's inclusion.

- Outlined requirements for changing routes at a designated station.
3. **Query Execution:** Using the given limitations, the knowledge base was queried for legitimate routes, including direct and single-transfer routes.
 4. **Performance Metrics:**
 - **Execution Time:** Two test cases were used to gauge the effectiveness of the PDDL implementation:
 - **Test 1:** 2.4849 seconds, memory usage of -0.86 MB, 1 step.
 - **Test 2:** 3.0836 seconds, memory usage of 5.89 MB, 9 steps.
 - **Output:** Paths that meet the requirements for every test were successfully obtained.

```
pddl_planning Performance Metrics:
Execution Time: 2.4849 seconds
Memory Usage: -0.86 MB
Number of Steps: 1
Test pddl_planning (22540, 2573, 4686, 1): Pass

pddl_planning Performance Metrics:
Execution Time: 3.0836 seconds
Memory Usage: 5.89 MB
Number of Steps: 9
Test pddl_planning (951, 340, 300, 1): Pass
```

Analysis:

- **Intermediate Steps:** Iteratively created pathways using PDDL with forward chaining, taking into account every boarding and transfer possibility within the given limitations.
- **Number of Steps:** limited actions according to the destination aim, which led to fewer steps than a full forward chaining strategy.
- **Comparison of Results:**
 - The best routes were generated by all three implementations (forward chaining, backward chaining, and PDDL). Because of the targeted querying and pre-established criteria, the PDDL technique was more effective in terms of memory utilization and execution time.

Bonus Question 2: Extending Route Planning with Fare Constraints

By adding a fare attribute to the route planning issue, this addition limited the number of route interchanges and optimized the route based on the maximum permitted fare. Pruning was used in an optimized breadth-first search (BFS) to increase efficiency.

Key Aspects of Implementation:

- 1. Fare Constraints:**
 - Routes that exceeded the maximum fare were excluded after being filtered using an initial fare limit.
- 2. Pruning Technique:**
 - Paths that went beyond the maximum permitted fare were pruned using the pruning technique.
 - reduced the overall search space by marking previously visited nodes to prevent duplicate exploration.
- 3. Pre-Computation:**
 - A route summary with the lowest fare for each route and related stops was computed.
- 4. Optimized BFS:**
 - BFS was used to investigate the route between start_stop_id and end_stop_id while taking fare and transfer restrictions into account.
 - Queued routes only in the event that the existing transfer and fare limits fell within the permitted ranges.

Performance Metrics:

- **Execution Time:**
 - **Test 1:** 16.0933 seconds, memory usage of -17.80 MB.
 - **Test 2:** 15.3981 seconds, memory usage of 35.88 MB.

```
bfs_route_planner_optimized Performance Metrics:  
Execution Time: 16.0933 seconds  
Memory Usage: -17.80 MB  
Test bfs_route_planner_optimized (22540, 2573, 10, 3): Pass  
  
bfs_route_planner_optimized Performance Metrics:  
Execution Time: 15.3981 seconds  
Memory Usage: 35.88 MB  
Test bfs_route_planner_optimized (4012, 4013, 10, 3): Pass
```

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	2.4849 - 3.0836	-0.86 - 5.89	Moderate	Yes
Optimized BFS (with Fare)	15.3981 - 16.0933	-17.80 - 35.88	Highest	Yes

Comparative Analysis:

Conclusion:

- **Efficiency:** Although BFS optimization was required for fare constraints, the PDDL technique proved effective in terms of both memory and time.
- **Optimal Path Consistency:** All approaches produced the best paths under the specified constraints, regardless of implementation variations.
- **Recommended Approach:** The optimized BFS with pruning is better for sophisticated fare-constrained route planning, while the PDDL implementation using PyDatalog is appropriate for applications concentrating on minor restrictions.