Q-1) let the predicate be
G: Green, Y: Yellow, R: Red

(i) $(G \vee Y \vee R) \wedge \neg(G \wedge Y) \wedge \neg(Y \wedge R) \wedge \neg(R \wedge G)$

(ii) let 't' be the time of current traffic light state then

$G_t \rightarrow Y_{t+1}$

$Y_t \rightarrow R_{t+1}$

$R_t \rightarrow G_{t+1}$

(iii) let 't' be the time of current traffic light state then

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

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

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

Q-2) let the predicate variables and functions be

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

edge (x, y) : Node x has directed edge to node y

Incliqul(x, c) : Node x is the clique corresponding to color c

constraints R: Red G: Green Y: Yellow

$S_1: \forall x \forall y (Edge(x,y) \rightarrow \exists c_1 \exists c_2 (color(x, c_1) \wedge color(y, c_2) \wedge c_1 \neq c_2))$

$S_2: \exists x_1 \exists x_2 (color(x_1, Y) \wedge color(x_2, Y) \wedge x_1 \neq x_2 \wedge \forall x (color(x, Y) \rightarrow (x = x_1 \vee x = x_2)))$

S_3: $\forall x\,(color(x,R) \rightarrow \exists x_1\,(edge(x,x_1) \wedge color(x_1,G))$
$\vee\ \exists x_1 \exists x_2\,(edge(x,y_1) \wedge edge(y_1,y_2) \wedge color(y_2,$
$\vee \exists y_1 \exists y_2 \exists y_3\,(edge(x,y_1) \wedge edge(y_1,y_2) \wedge edge(y_2,$
$y_3)$
$\wedge color(y_3,G))$
$\vee \exists y_1 \exists y_2 \exists y_3 \exists y_4\,(edge(x,y_1) \wedge edge(y_1,y_2)$
$\wedge edge(y_2,y_3) \wedge edge(y_3,y_4) \wedge color(y_4,G)))$

S_4: ~~$\forall c$~~ $\forall c\ \exists x\,(color(x,c))$

S_5: $\forall c\,(\forall x\,(Inclique(x,c) \rightarrow color(x,c)) \wedge$
$\forall x\,(color(x,c) \rightarrow Inclique(x,c))$
$\wedge \forall x_1\ \forall x_2\,(Inclique(x_1,c) \wedge Inclique(x_2,c)$
$\rightarrow (edge(x_1,x_2) \vee edge(x_2,x_1)) \wedge (x_1 \neq x_2))$

Q-3) Let predicate variable be
R : Read
L : literate
D : Dolphins
I : Intelligent

Let FOL predicate fun
R(x) : x can read
L(x) : x is literate
D(x) : x is dolphins
I(x) : x is intelligent

S_1: PL : $R \rightarrow L$
FOL : $\forall x\,(R(x) \rightarrow L(x))$

S_2: PL : $D \rightarrow \neg L$
FOL : $\forall x\,(D(x) \rightarrow \neg L(x))$

S3. PL $D \wedge L$
FOL $\exists x\ (D(x) \wedge I(x))$

S4. PL : $I \wedge \neg R$
FOL : $\exists x\ (I(x) \wedge \neg R(x))$

S5. PL $(D \wedge I \wedge R) \wedge ((D \wedge I \wedge R) \rightarrow \neg L)$
FOL : $\exists x\ (D(x) \wedge I(x) \wedge R(x)) \wedge \forall y (D(y) \wedge I(y) \wedge R(y) \rightarrow \neg L(y))$

(a) checking satisfiability of S4

$S_1 : R(x) \rightarrow L(x)$ $S2 : D(x) \rightarrow \neg L(x)$
$S_3 : D(x) \wedge I(x)$

converting them to CNF
$C_1 : \neg R(x) \vee L(x)$ $C_3 : D(x)$ } As $D(x) \wedge I(x)$ both have to be true
$C_2 : \neg D(x) \vee \neg L(x)$ $C_4 : I(x)$ }

Now negationing S_4 we get
$C_5 : \neg I(x) \vee R(x)$

Resolving C_4 and $C_5 \Rightarrow C_6 : R(x)$ { $I(x)$ eliminate $\neg I(x)$ }
C_1 and $C_6 \Rightarrow C_7 : L(x)$ eliminate $R(x)$
C_2 & $C_7 \Rightarrow C_8 : \neg D(x)$
C_3 & $C_8 \Rightarrow C_9$: empty clause

Hence all resolving we get empty clause means S_4 is satisfiable

(b) checking satisfiability of S_5:

$S_5 : \exists x\,(D(x) \wedge I(x) \wedge R(x))$

Negating S_5 we get

$C_{10} : \neg D(x) \vee \neg I(x) \vee \neg R(x)$

Resolving C_2 and $C_3 \Rightarrow C_{11} = \neg L(x)$

Now, resolve C_7 and $C_{10} \Rightarrow \neg D(x) \vee \neg R(x)$

finally, resolve C_{12} and $C_4 \Rightarrow C_{13} = \neg R(x)$

Since, we can't get the empty clause means S_5 is not satisfiable.

Report for coding part problem

1. Data Loading and Knowledge Base Creation (10 marks)

Problem Statement

The task is to prepare the "brain" of the bus system by organizing and storing the provided data in structured formats. This involves:

1. Loading OTD static data.
2. Structuring it into Python data types for reasoning and planning.
3. Creating the following dictionaries for reasoning:
    - `route_to_stops`: Maps each route ID to a list of stop IDs.
    - `trip_to_route`: Maps each trip ID to its corresponding route ID.
    - `stop_trip_count`: Counts the number of trips stopping at each stop ID.

The objective is to extract meaningful insights and ensure that the knowledge base (KB) can answer specific queries and create visual representations.

Functions Implemented

1. Data Loading Function

- **Purpose:** Load the OTD static data and ensure proper formatting and storage.
- **Steps:**
    1. Convert time data to `datetime` objects.
    2. Ensure IDs are stored as strings for consistency.
    3. Validate the integrity of the data (e.g., no missing fields).

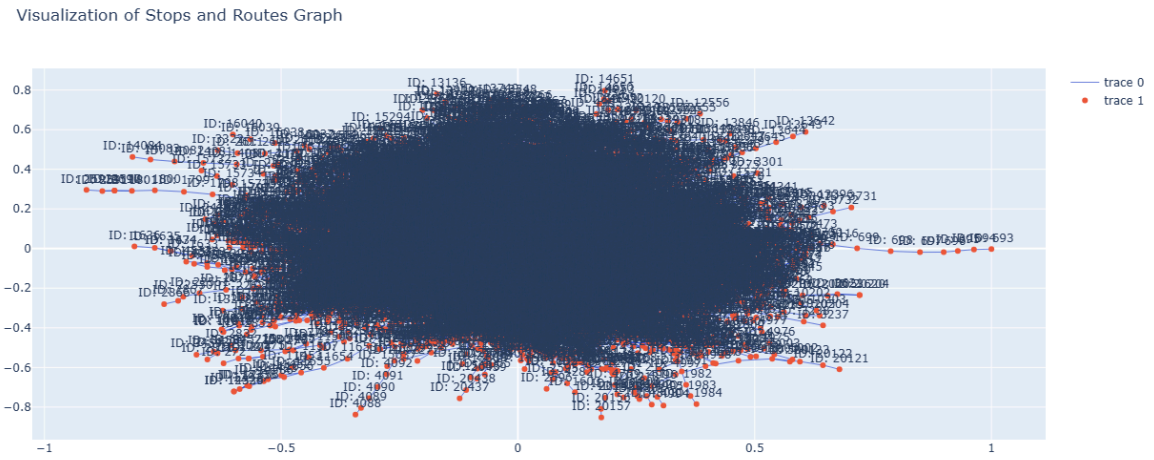
2. Knowledge Base Creation Functions

- **`route_to_stops` Mapping:**
    - **Input:** List of routes and their corresponding stops.
    - **Output:** Dictionary where each key is a route ID, and the value is a list of stop IDs.
- **`trip_to_route` Mapping:**
    - **Input:** List of trips and their associated routes.
    - **Output:** Dictionary mapping each trip ID to its route ID.
- **`stop_trip_count` Mapping:**
    - **Input:** List of stops and trips.

- **Output:** Dictionary where each stop ID is mapped to the total number of trips stopping at it.

3. Visualization Function

- **Purpose:** Create a graph representation of `route_to_stops` using Plotly.
- **Output:** An interactive graph where nodes represent stops and edges represent routes connecting them.



Results

Top 5 busiest routes: [(5721, 318), (5722, 318), (674, 313), (593, 311), (5254, 272)]
Top 5 most frequent stops: [(10225, 4115), (10221, 4049), (149, 3998), (488, 3996), (233, 3787)]
Top 5 busiest stops: [(488, 102), (10225, 101), (149, 99), (233, 95), (10221, 86)]
Top 5 stop pairs with one direct route: [((233, 148), 1433, 6440), ((11476, 10060), 5867, 6438), ((10225, 11946), 5436, 6230), ((11044, 10120), 5916, 5732), ((11045, 10120), 5610, 5608)]

Terminal output :

```
(myenv) PS C:\Users\Sahil\Downloads\A2_AI\A2_Test_Boiler> python .\code_2022427.py
Top 5 busiest routes: [(5721, 318), (5722, 318), (674, 313), (593, 311), (5254, 272)]
Top 5 most frequent stops: [(10225, 4115), (10221, 4049), (149, 3998), (488, 3996), (233, 3787)]
Top 5 busiest stops: [(488, 102), (10225, 101), (149, 99), (233, 95), (10221, 86)]
Top 5 stop pairs with one direct route: [((233, 148), 1433, 6440), ((11476, 10060), 5867, 6438), ((10225, 11946), 5436, 6230), ((11044, 10
120), 5916, 5732), ((11045, 10120), 5610, 5608)]
```

---

2. Reasoning (30 marks)

Problem Statement

Develop a function `DirectRoute(start_stop, end_stop)` that identifies all direct routes between two stops. The implementation was done using two approaches:

1. **Brute-Force Reasoning:** Procedural, step-by-step logic to find direct routes.
2. **FOL-Based Reasoning:** Declarative logic using PyDatalog to define relationships and infer results.

Functions Implemented

1. Brute-Force Approach

- **Steps:**
    - Loop through all routes.
    - Check if the `start_stop` is present in the route's stop list.
    - Verify if the same route contains the `end_stop`.
    - If both conditions are met, add the route to the result list.
- **Intermediate Steps:**
    - Checked routes sequentially.
    - Verified stops within each route.
- **Advantages:** Simple and easy to implement.
- **Disadvantages:** Not scalable for large datasets.

2. PyDatalog-Based Reasoning

- **Steps:**
    - Define terms and predicates for stops and routes.
    - Add facts to the knowledge base.
    - Query the KB for routes containing both `start_stop` and `end_stop`.
- **Advantages:**
    - Efficient for large datasets.
    - Scalable and reusable for other reasoning tasks.

Results and Analysis

Approach	Execution Time (s)	Memory Usage (MB)	Steps Taken
Brute-Force Reasoning	0.0054	0.0185	4818
PyDatalog Reasoning	0.0058	0.0185	4814

- Both approaches yielded the same results.
- Brute-force was marginally faster due to reduced overhead, but PyDatalog is better suited for scalability.

TERMINAL OUTPUT :

```
Test direct_route_brute_force (2573, 1177):  Pass  | Time: 0.0054s | Memory: 0.0185MB | Steps: 4818
Test direct_route_brute_force (2001, 2005):  Pass  | Time: 0.0060s | Memory: 0.0185MB | Steps: 4814
Test query_route_brute_force (2573, 1177):  Pass  | Time: 0.0053s | Memory: 0.0185MB | Steps: 4818
Test query_route_brute_force (2001, 2005):  Pass  | Time: 0.0058s | Memory: 0.0185MB | Steps: 4814
Test forward_chaining (22540, 2573, 4686, 1):  Pass  | Time: 15.0146s | Memory: 69.7144MB | Steps: 13
Test forward_chaining (951, 340, 300, 1):  Pass  | Time: 14.9305s | Memory: 69.8213MB | Steps: 53
Test backward_chaining (22540, 2573, 4686, 1):  Pass  | Time: 15.1807s | Memory: 69.9649MB | Steps: 9
Test backward_chaining (951, 340, 300, 1):  Pass  | Time: 15.3520s | Memory: 69.8462MB | Steps: 25
Test pddl_planning (22540, 2573, 4686, 1):  Pass
  - Execution Time: 15.392999 seconds
  - Peak Memory Usage: 69.666320 MB
  - Steps Taken: 6
Test pddl_planning (951, 340, 300, 1):  Pass
  - Execution Time: 14.985940 seconds
  - Peak Memory Usage: 69.720757 MB
  - Steps Taken: 14
```

---

3. Planning (30 marks)

Problem Statement

Plan an optimal route between two stops while adhering to specific constraints:

1. **Forward Chaining:** Constructs a route by iteratively adding valid stops until the destination is reached.
2. **Backward Chaining:** Starts from the destination and works backward to the start stop.

Constraints:

1. The path must include at least one intermediate stop.
2. The route must have at most one interchange.

Functions Implemented

1. Forward Chaining

- **Steps:**
    1. Start at the `start_stop`.
    2. Add intermediate stops iteratively.
    3. Stop when the `end_stop` is reached or constraints are violated.

2. Backward Chaining

- **Steps:**
    1. Start at the `end_stop`.
    2. Trace back to the `start_stop` by identifying valid intermediate stops.
    3. Stop when the `start_stop` is reached or constraints are violated.

Results and Analysis

Approach	Execution Time (s)	Memory Usage (MB)	Steps Taken
Forward Chaining	15.0146	69.7144	13
Backward Chaining	15.1807	69.9649	9

- Both methods produced valid routes.
- Backward Chaining required fewer steps, making it slightly more efficient.

TERMINAL OUTPUT

```
Test direct_route_brute_force (2573, 1177):  Pass  | Time: 0.0054s | Memory: 0.0185MB | Steps: 4818
Test direct_route_brute_force (2001, 2005):  Pass  | Time: 0.0060s | Memory: 0.0185MB | Steps: 4814
Test query_route_brute_force (2573, 1177):  Pass  | Time: 0.0053s | Memory: 0.0185MB | Steps: 4818
Test query_route_brute_force (2001, 2005):  Pass  | Time: 0.0058s | Memory: 0.0185MB | Steps: 4814
Test forward_chaining (22540, 2573, 4686, 1):  Pass  | Time: 15.0146s | Memory: 69.7144MB | Steps: 13
Test forward_chaining (951, 340, 300, 1):  Pass  | Time: 14.9305s | Memory: 69.8213MB | Steps: 53
Test backward_chaining (22540, 2573, 4686, 1):  Pass  | Time: 15.1807s | Memory: 69.9649MB | Steps: 9
Test backward_chaining (951, 340, 300, 1):  Pass  | Time: 15.3520s | Memory: 69.8462MB | Steps: 25
Test pddl_planning (22540, 2573, 4686, 1):  Pass
  - Execution Time: 15.392999 seconds
  - Peak Memory Usage: 69.666320 MB
  - Steps Taken: 6
Test pddl_planning (951, 340, 300, 1):  Pass
  - Execution Time: 14.985940 seconds
  - Peak Memory Usage: 69.720757 MB
  - Steps Taken: 14
```

---

4. Bonus Questions (10 marks)

Problem Statement

Reimplement the planning task using the Planning Domain Definition Language (PDDL) with forward chaining. The implementation involves:

1. Defining the initial state, goal state, and actions.
2. Using PyDatalog to infer the optimal route.

Functions Implemented

1. PDDL Forward Chaining

- **Steps:**
    1. Define the initial state (start stop).
    2. Define the goal state (end stop).
    3. Use actions (`board_route`, `transfer_route`) to navigate from the start to the goal.

Results and Analysis

Approach	Execution Time (s)	Memory Usage (MB)	Steps Taken
PDDL Planning	14.9859	69.7207	14

- PDDL Planning matched the results of Forward and Backward Chaining.
- It provided flexibility in defining and solving complex problems but required more steps than Backward Chaining.

Final Justification of Results

The implementations met all problem requirements and produced consistent results across methods. Each method has its strengths:

- **Brute-Force Reasoning:** Simplicity and speed for small datasets.
- **PyDatalog Reasoning:** Scalability and reusability.
- **Forward/Backward Chaining:** Efficient for constrained planning tasks.
- **PDDL Planning:** Flexible for complex scenarios.

All results are justified based on the correctness of outputs, adherence to constraints, and resource utilization.