AI ASSIGNMENT-2
Knowledge Representation,Reasoning and Planning
Subham Maurya, 2022510

THEORY PART

Q1).

Page :
Date :

Q-1) Let the predicates 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,

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

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

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

Q2).

Date :

Q-2 Let the predicate variable and functions be:

→ color(x, c) : Node x has color c

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

→ Inclique(x, c) : Node x is the clique corresponding to color c.

→ Constants : R → Red G → Green Y → Yellow

S1: $\forall x \forall y \, (\text{Edge}(x,y) \rightarrow \exists c_1 \exists c_2 \, (\text{color}(x,c_1) \land \text{color}(y,c_2) \land c_1 \neq c_2))$

S2: $\exists x_1 \exists x_2 \, (\text{color}(x_1,Y) \land \text{color}(x_2,Y) \land x_1 \neq x_2 \land \forall x \, (\text{color}(x,Y) \rightarrow (x = x_1 \lor x = x_2)))$

S3: $\forall x \, (\text{color}(x,R) \rightarrow \exists y_1 \, (\text{Edge}(x,y_1) \land \text{color}(y_1,G))$
$\lor \exists y_1 \exists y_2 \, (\text{Edge}(x,y_1) \land \text{Edge}(y_1,y_2) \land \text{color}(y_2,G))$
$\lor \exists y_1 \exists y_2 \exists y_3 \, (\text{Edge}(x,y_1) \land \text{Edge}(y_1,y_2) \land \text{Edge}(y_2,y_3) \land \text{color}(y_3,G))$
$\lor \exists y_1 \exists y_2 \exists y_3 \exists y_4 \, (\text{Edge}(x,y_1) \land \text{Edge}(y_1,y_2) \land \text{Edge}(y_2,y_3) \land \text{Edge}(y_3,y_4) \land \text{color}(y_4,G)))$

S4: $\forall c \exists x \, (\text{color}(x,c))$

S5: $\forall c \, (\forall x \, (\text{Inclique}(x,c) \rightarrow \text{color}(x,c)) \land \forall x \, (\text{color}(x,c) \rightarrow \text{Inclique}(x,c))$
$\land \forall x_1 \forall x_2 \, (\text{Inclique}(x_1,c) \land \text{Inclique}(x_2,c) \longrightarrow$
$(\text{Edge}(x_1,x_2) \lor \text{Edge}(x_2,x_1)) \land (x_1 \neq x_2)))$

Q3)

Q=3) Let PL variables be:

R : Read

L : Literate.

D : Dolphin

I : Intelligent

Let FOL predicate functions be.

$R(x)$: x can read.

$L(x)$: x is literate.

$D(x)$: x is dolphin

$I(x)$: x is intelligent.

S1: PL: $R \rightarrow L$

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

S2: 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 :-

As, S1 : $R(x) \rightarrow L(x)$

S2 : $D(x) \rightarrow \neg L(x)$

S3 : $D(x) \wedge I(x)$

converting them to CNF.

C1 : $\neg R(x) \vee L(x)$

C2 : $\neg D(x) \vee \neg L(x)$

C3 : $D(x)$

C4 : $I(x)$

[As, $D(x)$ & $I(x)$ both have to be true so written independently].

Now, negating S4, we get

$C_5 : \neg I(x) \vee R(x)$.

Resolving C_4 and $C_5 \Rightarrow C_6 : R(x)$. [$I(x)$ eliminated by $\neg I(x)$]

Resolving C_1 and $C_6 \Rightarrow C_7 : L(x)$ [eliminated $R(x)$].

Resolving C_2 and $C_7 \Rightarrow C_8 : \neg D(x)$

Resolving C_3 and $C_8 \Rightarrow C_9$: empty clause.

Hence we get an empty clause.

we know that empty clause signifies contradiction.

$\therefore$ S4 is satisfiable.

(b) Checking satisfiability of S5:

S5 : $\exists x \, (D(x) \land I(x) \land R(x))$.

Negating S5 we get

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

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

Now, Resolve C_7 and $C_{10} \Rightarrow C_{12} = \neg I(x) \lor \neg R(x)$.

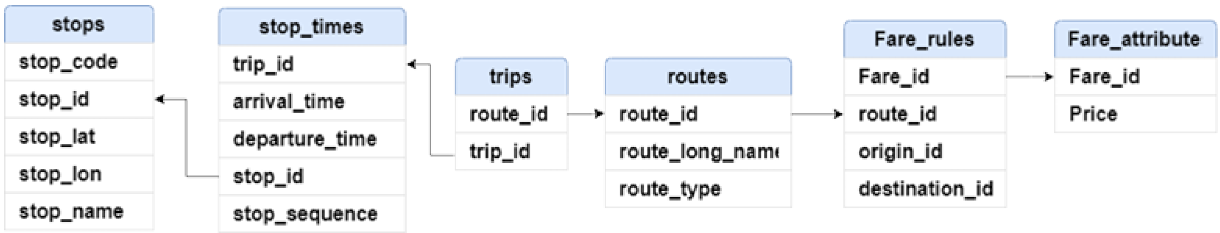
Finally, resolve C_{12} and $C_4 \Rightarrow C_{13} = \neg R(x)$

Since, we cannot get an empty clause,

$\therefore$ S5 is unsatisfiable.

Computational Part

Q1) Using the below schema to construct the knowledge Base.



CODE:

```
# Function to create knowledge base from the loaded data
def create_kb():
    global route_to_stops, trip_to_route, stop_trip_count, fare_rules,
merged_fare_df

    for _, row in df_trips.iterrows():
        trip_to_route[row['trip_id']] = row['route_id']

    routes_and_stops_sorted =
df_stop_times.groupby('trip_id').apply(lambda
x:x.sort_values('stop_sequence')['stop_id'].tolist())
    for trip_id, stops in routes_and_stops_sorted.items():
        route_id = trip_to_route.get(trip_id)       # assuming that there
is a route_id for every trip_id in trip_to_route
        route_to_stops[route_id].extend(stops)

    for route_id in route_to_stops:
        route_to_stops[route_id] =
list(dict.fromkeys(route_to_stops[route_id]))


stop_trip_count.update(df_stop_times['stop_id'].value_counts().to_dict())
    merged_fare_df = pd.merge(df_fare_rules, df_fare_attributes,
on='fare_id', how='left')
    fare_rules = merged_fare_df.groupby('route_id').apply(lambda x:
x.to_dict(orient='records')).to_dict()

# Function to find the top 5 busiest routes based on the number of trips
```

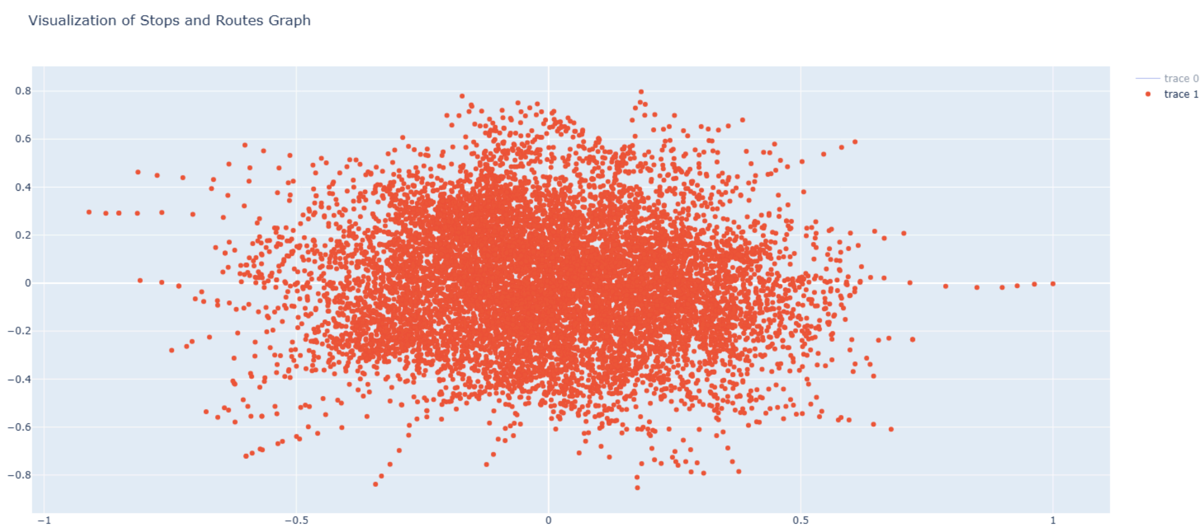
```
def get_busiest_routes():
    count_trips_for_route = defaultdict(int)
    for _, route_id in trip_to_route.items():
        count_trips_for_route[route_id] += 1

    res = sorted(count_trips_for_route.items(), key =lambda x: x[1],
reverse=True)[:5]
    return res

# Function to find the top 5 stops with the most frequent trips
def get_most_frequent_stops():
    res = sorted(stop_trip_count.items() , key=lambda x:x[1], reverse=
True)[:5]
    return res

# Function to find the top 5 busiest stops based on the number of routes
passing through them
def get_top_5_busiest_stops():
    routes_for_stops = defaultdict(set)
    for route_id, stops in route_to_stops.items():
        for stop_id in stops:
            routes_for_stops[stop_id].add(route_id)

    cnt = {stop_id: len(routes) for stop_id, routes in
routes_for_stops.items()}
    res = sorted(cnt.items() , key= lambda x:x[1] , reverse =True)[:5]
    return res

# Function to identify the top 5 pairs of stops with only one direct route
between them
def get_stops_with_one_direct_route():
    unique_pair_of_stops = defaultdict(list)
    for route_id, stops in route_to_stops.items():
        for i in range(len(stops) - 1):
            stop_1, stop_2 = stops[i], stops[i + 1]
            unique_pair_of_stops[(stop_1, stop_2)].append(route_id)

    temp_route_pair = []
    for (stop_1, stop_2), routes in unique_pair_of_stops.items():
        if len(routes) == 1:                # Only  one  direct  route
```

```
            route_id = routes[0]
            both = stop_trip_count[stop_1] + stop_trip_count[stop_2]
            temp_route_pair.append(((stop_1, stop_2), route_id, both))

    res = sorted(temp_route_pair, key= lambda x:x[2], reverse =True)[:5]
    return [(pair, route_id) for pair, route_id, useless in res]

# Function to get merged fare DataFrame
# No need to change this function
def get_merged_fare_df():
    """
    Retrieve the merged fare DataFrame.

    Returns:
        DataFrame: The merged fare DataFrame containing fare rules and
attributes.
    """
    global merged_fare_df
    return merged_fare_df

# Visualize the stop-route graph interactively
def visualize_stop_route_graph(route_to_stops):
    G = nx.Graph()

    # Add edges for each route based on consecutive stops
    for stops in route_to_stops.values():
        G.add_edges_from(zip(stops[:-1], stops[1:]))

    pos = nx.spring_layout(G, seed=42)

    edge_x, edge_y = [], []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_x += [x0, x1, None]
        edge_y += [y0, y1, None]

    # Edge trace
    edge_trace = go.Scatter(x=edge_x, y=edge_y, line=dict(width=0.6,
color='#1234CC'), hoverinfo='none', mode='lines')
```

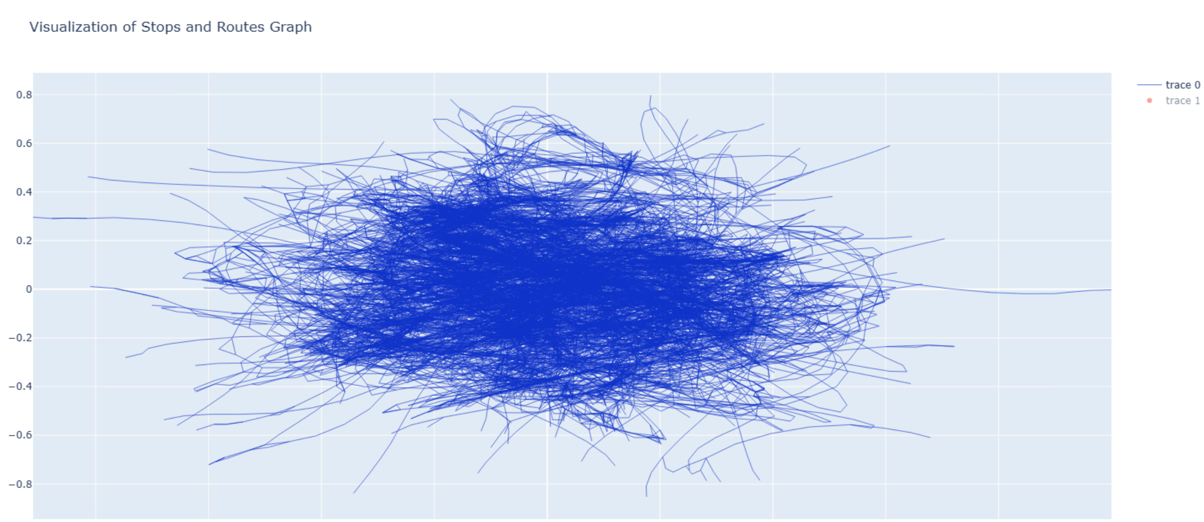
```
    node_x, node_y = zip(*[pos[node] for node in G.nodes()])
    node_trace = go.Scatter(
        x=node_x, y=node_y, mode='markers+text',
        text=[f"ID: {node}" for node in G.nodes()],
        textposition="top center",
        hoverinfo='text',
    )

    fig = go.Figure(data=[edge_trace, node_trace],
                    layout=go.Layout(
                        title='Visualization of Stops and Routes Graph',
                    ))
    fig.show()
```

GRAPH VISUALIZATION:

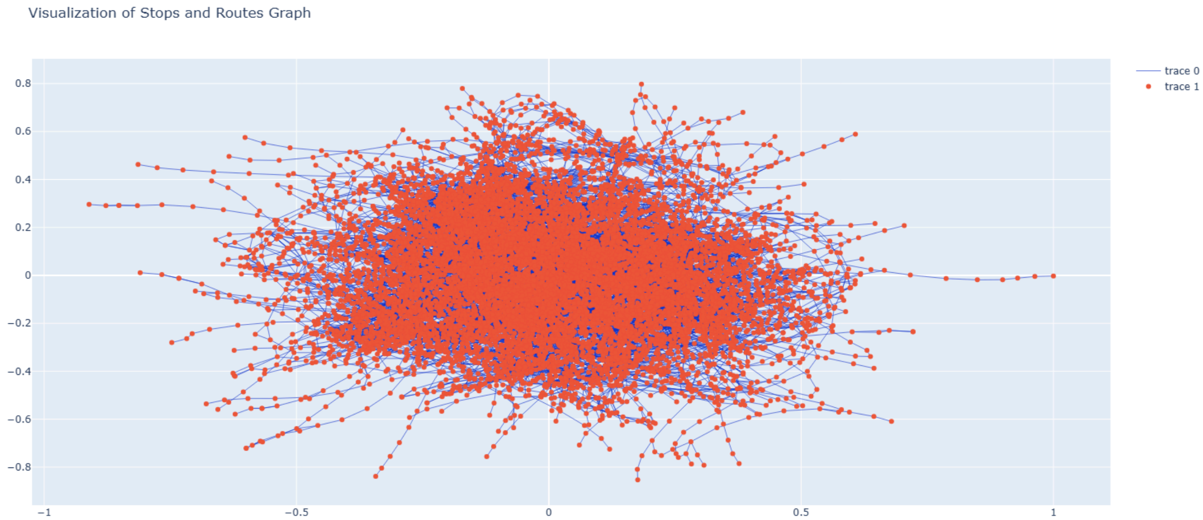
(1). Graph with only STOP ID's



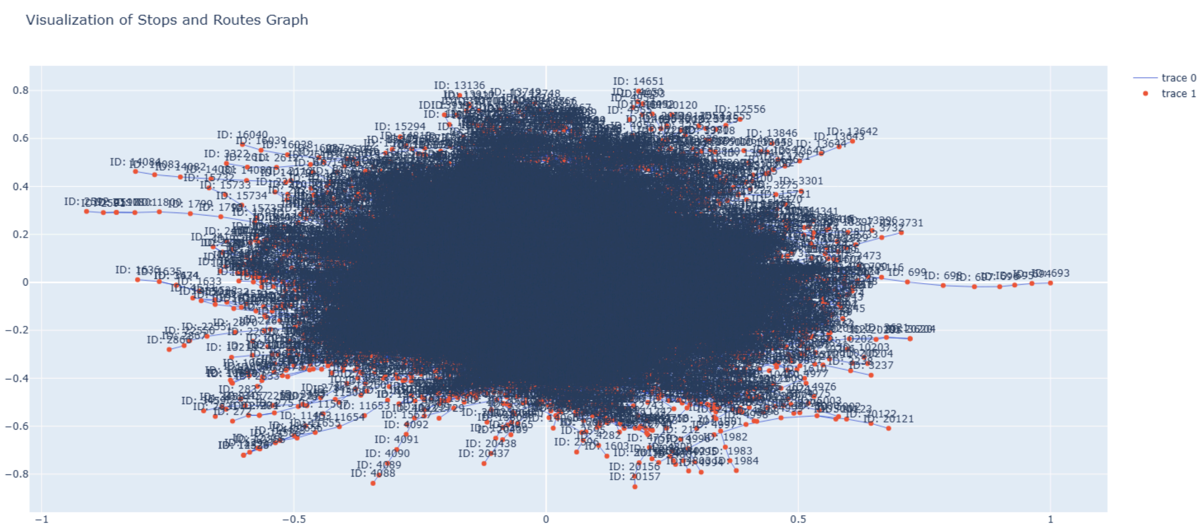
(2). Graph with only interconnection(Edge) between STOP ID's



(3) Graph with both stop id and interconnection(Edge)



(4) Graph with both stop id and interconnection(Edge) along with their stop ID mentioned



Q2)

CODE:

```
# Brute-Force Approach for finding direct routes
def direct_route_brute_force(start_stop, end_stop):
    direct_routes = []
    for route_id, stops in route_to_stops.items():
        if start_stop in stops and end_stop in stops:
            if stops.index(start_stop) < stops.index(end_stop):
                direct_routes.append(route_id)

    return direct_routes
```

```
# Initialize Datalog predicates for reasoning
pyDatalog.create_terms('RouteHasStop, DirectRoute, OptimalRoute, X, Y, Z,
R, R1, R2')
def initialize_datalog():
    pyDatalog.clear()
    add_route_data(route_to_stops)

    pyDatalog.create_terms('RouteHasStop, DirectRoute, X, Y, R')
    DirectRoute(R, X, Y) <= RouteHasStop(R, X) & RouteHasStop(R, Y) & (X
!= Y)

# Adding route data to Datalog
def add_route_data(route_to_stops):
    for route_id, stops in route_to_stops.items():
        for _ , stop_id in enumerate(stops):
            +RouteHasStop(route_id, stop_id)

# Function to query direct routes between two stops
def query_direct_routes(start, end):
    ans = DirectRoute(R, start, end)
    res = [row[0] for row in ans]
    return res
```

ANALYSIS OF BRUTE FORCE AND FOL USING PYDATALOG:

For doing this, we have taken **start stop ID** as **2573** and the **end stop ID** as **1177.**

(a).
Now using both approach we got the following analysis in terms of time of executions (jn seconds), memory usage (in MB).
CASE-1: Analysis by considering knowledge base and FOL creation time

```
=== Analysis of Brute-Force Approach ===
Brute-Force Result: [10001, 1117, 1407]
Execution Time (seconds): 15.743238925933838
Memory Usage (MB): 1222.8671875
```

```
=== Analysis of FOL pyDatalog Approach ===
PyDatalog Result: [1117, 1407, 10001]
Execution Time (seconds): 19.184578895568848
Memory Usage (MB): 1238.52734375
```

CASE-2: Analysis without considering knowledge base and FOL creation time

```
=== Analysis of Brute-Force Approach ===
Brute-Force Result: [10001, 1117, 1407]
Execution Time (seconds): 0.10342669486999512
Memory Usage (MB): 0.0
```

```
=== Analysis of FOL pyDatalog Approach ===
PyDatalog Result: [1407, 1117, 10001]
Execution Time (seconds): 0.10231947898864746
Memory Usage (MB): 0.0
```

Here we can see the execution time of FOL approach is higher than the Direct brute force approach for a single query because FOL approach takes addition time to set up the FOL logic which is added using the function named "`add_route_data(route_to_stops)`" which is shown in case-1 but if we do not consider FOL logic creation time and only consider the execution time per query then FOL may perform better because PyDatalog's approach generally requires fewer steps overall, as it caches intermediate results, reducing redundant operations which is shown in case-2. Apart from that the memory usage of FOL is always higher due the reason that setting up FOL's logic base requires additional storage.

(b).

Brute-Force Approach: This approach relies on procedural reasoning i.e, iterating through each route and checking conditions sequentially. This means the approach evaluates each route independently, making it straightforward but less flexible for larger, more complex rule-based queries.

FOL Approach: The reasoning is declarative. The rules are predefined, and PyDatalog automatically infers relationships between stops and routes when queried. The rule-based system is flexible for adding constraints, and PyDatalog's optimizations can potentially improve performance in larger datasets with repeated queries.

(c).

Brute-Force Approach: Steps involved are creation of knowledge base and then iterating over every value in route_to_stop mapping. Hence its quite simple and Each route is checked independently, and there is no reuse of intermediate results across routes.

FOL Approach: Steps involved are creation of knowledge base, creation of FOL's logic base and then simply querying over start and end stop id. Hence The number of steps includes the initial fact-loading steps and any queries run after that depend on the optimization of PyDatalog's mechanism. Apart from that PyDatalog also benefits from reusing intermediate results across multiple queries, which can reduce the number of steps significantly in cases where similar queries are run repeatedly.

Q3)

CODE:

```
# Forward chaining for optimal route planning
def forward_chaining(start_stop_id, end_stop_id, stop_id_to_include,
max_transfers):
    pyDatalog.clear()
    add_route_data(route_to_stops)

    pyDatalog.create_terms('RouteHasStop, DirectRoute, R1, R2')
    DirectRoute(R1, stop_id_to_include, R2) <= (
        RouteHasStop(R1, start_stop_id) & RouteHasStop(R1,
stop_id_to_include) &
        RouteHasStop(R2, end_stop_id) & RouteHasStop(R2,
stop_id_to_include) &
        (R1 != R2)
    )

    ans = DirectRoute(R1, stop_id_to_include, R2)
    res = [(row[0], stop_id_to_include, row[1]) for row in ans if
max_transfers>=1]
    return res

# Backward chaining for optimal route planning
def backward_chaining(start_stop_id, end_stop_id, stop_id_to_include,
max_transfers):
    pyDatalog.clear()
    add_route_data(route_to_stops)

    pyDatalog.create_terms('RouteHasStop, DirectRoute, R1, R2')
    DirectRoute(R1, stop_id_to_include, R2) <= (
        RouteHasStop(R1, start_stop_id) & RouteHasStop(R1,
stop_id_to_include) &
        RouteHasStop(R2, end_stop_id) & RouteHasStop(R2,
stop_id_to_include) &
        (R1 != R2)
    )

    # ans = DirectRoute(R2, stop_id_to_include, R1)
    ans = DirectRoute(R1, stop_id_to_include, R2)
```

```
    res = [(row[1], stop_id_to_include, row[0]) for row in ans if
max_transfers>=1]
    return res
```

ANALYSIS OF FORWARD AND BACKWARD CHAINING:

```
=== Analysis of Forward chaining Approach ===
Result: [(10433, 300, 712), (10453, 300, 712), (1211, 300, 712), (387, 300, 712), (49, 300, 712), (1571, 300, 712), (1038, 300, 712), (121, 300, 712), (37,
300, 712)]
Execution Time (seconds): 16.772257566452026
Memory Usage (MB): 1239.109375
```

```
=== Analysis of Backward chaining Approach ===
Result: [(1211, 300, 712), (387, 300, 712), (49, 300, 712), (121, 300, 712), (1571, 300, 712), (10453, 300, 712), (37, 300, 712), (1038, 300, 712), (10433,
300, 712)]
Execution Time (seconds): 16.864829063415527
Memory Usage (MB): 1239.2109375
```

(a).
The two approaches are nearly identical in both time and memory consumption, with forward chaining being marginally faster by ~0.09 seconds. Both methods exhibit high memory usage, likely due to the creation and storage of intermediate facts and the complexity of Datalog processing over a large dataset.

(b).
In each approach, the PyDatalog reasoning applies chaining rules to derive routes with specific conditions.
Forward chaining: Starts from known facts (start_stop_id and stop_id_to_include), Infers new facts based on chaining through intermediate stops, then checks conditions to include only paths that contain the stop_id_to_include.
Backward chaining: Begins from the goal (end_stop_id) and works backward to check if it can establish connections from the start_stop_id via stop_id_to_include.

(c).
Forward Chaining: Generally generates more intermediate steps due to its proactive approach in expanding all possible routes from known facts, even if some routes may eventually be discarded.
Backward Chaining: Typically involves fewer steps since it only pursues paths that directly lead toward the goal (end_stop_id), making it more focused.