Artificial Intelligence Assignment 2

Parth Sandeep Rastogi,2022352

Theory

Variables

Predicate Definition

Q1

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

1) Traffic light is either green, yellow or red

$(G_t \vee Y_t \vee R_t)$

for exclusivity that its never more than one state at a time

we add clauses $-(G_t \wedge Y_t)$, $-(G_t \wedge R_t)$, $-(Y_t \wedge R_t)$ which show 2 state can't happen together at same time

hence $(G_t \vee Y_t \vee R_t) \wedge (-(G_t \wedge Y_t)) \wedge (-(G_t \wedge R_t)) \wedge (-(Y_t \wedge R_t))$

Q2

2) traffic switches from green to yellow, yellow to red and red to green

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

3) traffic light cannot remain in same state for more than 3 consecutive cycles

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

Here to can have any value this show that at all time from t to t+3 the traffic light can not be in same state

Q1)

Q2 Constants :-

Red, Yellow, Green are colors from C which are treated as a constant

Predicates :-

Node (x) means x is node from N

Color (c) mean if c is color from C

Edge (x, y) means edge exist from x to y

colored (x, c) means x is colored with c

Connected (x, y) means x and y are in same component

Actions/functions :- (connected ~~edge~~

Dist (x, y) ⇒ shortest distance between x and y

1) Connected nodes dont have same color

$$\forall x \forall y \, (Node(x) \wedge Node(y) \wedge Edges(x,y) \rightarrow \forall c \, (color(c) \wedge (Colored(x,c) \rightarrow \neg Colored(y,c))))$$

2) Exactly two nodes colored yellow

$$\exists x \exists y \, (Node(x) \wedge Node(y) \wedge x \neq y \wedge Colored(x, yellow) \wedge Colored(y, yellow) \wedge \forall z \, (Node(z) \wedge colored(z, Yellow) \rightarrow (z = x \vee z = y))))$$

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

$$\forall x \, (Node(x) \wedge Colored(x, Red) \rightarrow \exists y \, (Node(y) \wedge Colored(y, Green) \wedge [Dist(x,y) \leq 4])))$$

4) Each color has at least 1 node

$\forall c \; (\text{Color}(c) \rightarrow \exists x (\text{Node}(x) \wedge \text{Colored}(x,c))$

5) here we introduce predicate Clique (x, c)

$\rightarrow \forall c \; \exists x \; \text{Color}(c) \wedge \text{Node}(x) \wedge \text{Clique}(x,c)$

in each color there exist atleast one node x belonging to its clique

So And way 1 :-

$\forall x \forall y \forall c_1 \forall c_2 \; (\text{Node}(x) \wedge \text{Node}(y) \wedge \text{Color}(c_1) \wedge \text{Color}(c_2)) \rightarrow$
$((\text{Colored}(x,c_1) \wedge \text{Colored}(y,c_2) \wedge c_1 \neq c_2) \rightarrow$
$((\neg \text{Clique}(x_0, c_2)) \wedge (\neg \text{Clique}(y,c_1))$

if 2 nodes x, y have different colors then x can't belong to clique with color of y and vice versa indicating cliques are disjoint

Way 2-

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

if 2 node belong to same clique then have same color

Q3 Using FOL

Predicates

L(x) : x is literate
R(x) : x can read
I(x) : x is intelligent
D(x) : x is a dolphin

1) $\forall x\,(R(x) \rightarrow L(x))$
2) $\forall x\,(D(x) \rightarrow \neg(L(x)))$
3) $\exists x\,(D(x) \wedge I(x))$
4) $\exists x\,(I(x) \wedge \neg R(x))$
5) $\exists x\,(D(x) \wedge I(x) \wedge R(x)) \wedge \forall y\,(D(y) \wedge I(y) \wedge R(y) \rightarrow \neg L(y))$

Using PL

Predicates
D : Dolphin
L : Literate
I : Intelligent
R : Can Read

1) $R \rightarrow L$
2) $D \rightarrow \neg L$
3) Can't be shown with Proposition
4) Can't be shown with Proposition
5) Can't be shown with Proposition

FOL Resolution

Statement & Satisfiability

facts we know

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

~~if to~~ now to check

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

negate the goal

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

$\forall x (\neg I(x) \vee R(x))$ for goal to be satisfyable this should be non satisfiable

now removing $\forall$ and assuming as $\exists x$ as c

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

~~from D(d)~~

from D(c) and $\neg D(x) \vee \neg L(x)$ we have $\neg L(c)$

from I(c) and $\neg I(x) \vee R(x)$ we have R(c)

from R(c) and $\neg R(x) \vee L(x)$ we have L(c)

$\neg L(c)$ and L(c) which are contradiction,

hence statement 4 is satisfiable

Satisfiability of 5th statement

$\forall x (\neg R(x) \vee L(x))$ — (1)

$\forall x (\neg D(x) \vee \neg L(x))$ — (2)

D(c) $\wedge$ I(c) — (3) assuming some c which satisfy

~~D(d)~~ I(d) $\wedge$ R(d) — (4) assuming some d

D(k) $\wedge$ I(k) $\wedge$ R(k) — (5) k is some constant

$\forall y (\neg D(y) \vee \neg I(y) \vee \neg R(y) \vee \neg L(y))$ — (6)

negating the goal

L(k)

Converting to clause

¬ R(x) ∨ L(x)
¬ D(x) ∨ ¬ L(x)
D(c)
I(c)
I(d)
¬R(d)
D(k)
I(k)
R(k)
¬ D(x) ∨ ¬ I(x) ∨ ¬ R(x) ∨ ¬ L(x)

~~from R(k) and ¬ D(x) ∨~~

from
R(k) and ¬ R(x) ∨ L(x)
we have L(k)
which is not a contradiction
hence ~~is~~ not satisfiable

Computational

Q1)Data Loading and Knowledge Base Creation

Step 1) loaded and saved in the dataframes

```
df_stops = pd.read_csv('GTFS/stops.txt')
df_routes = pd.read_csv('GTFS/routes.txt')
df_stop_times = pd.read_csv('GTFS/stop_times.txt')
df_fare_attributes = pd.read_csv('GTFS/fare_attributes.txt')
df_trips = pd.read_csv('GTFS/trips.txt')
df_fare_rules = pd.read_csv('GTFS/fare_rules.txt')
```

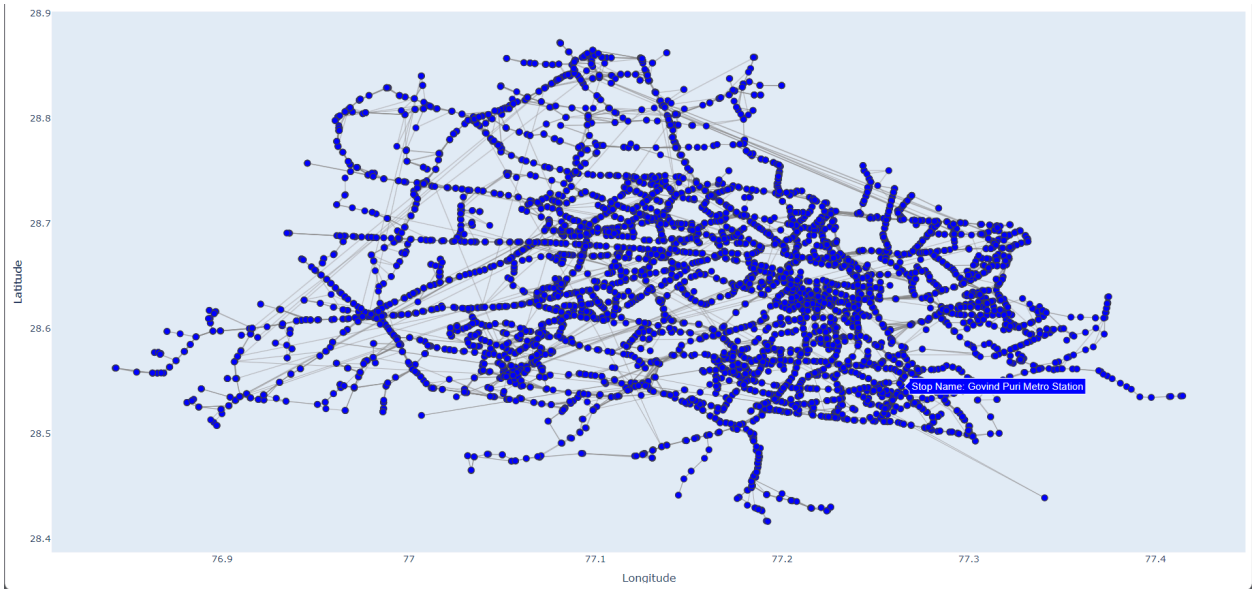
Step 2) Kb create function

```
global route_to_stops, trip_to_route, stop_trip_count, fare_rules, merged_fare_df
for _, it in df_trips.iterrows():
    tid = it['trip_id']
    rid = it['route_id']
    trip_to_route[tid] = rid
for _, it in df_stop_times.iterrows():
    tid = it['trip_id']
    sid = it['stop_id']
    rid = trip_to_route.get(tid)
    if sid not in route_to_stops[rid]:
        route_to_stops[rid].append(sid)
for sid in df_stop_times['stop_id']:
    stop_trip_count[sid] += 1
for _, it in df_fare_rules.iterrows():
    rid = it['route_id']
    fid = it['fare_id']
    fare_rules[rid] = fid
merged_fare_df = pd.merge(df_fare_rules, df_fare_attributes, on='fare_id', how='left')
```

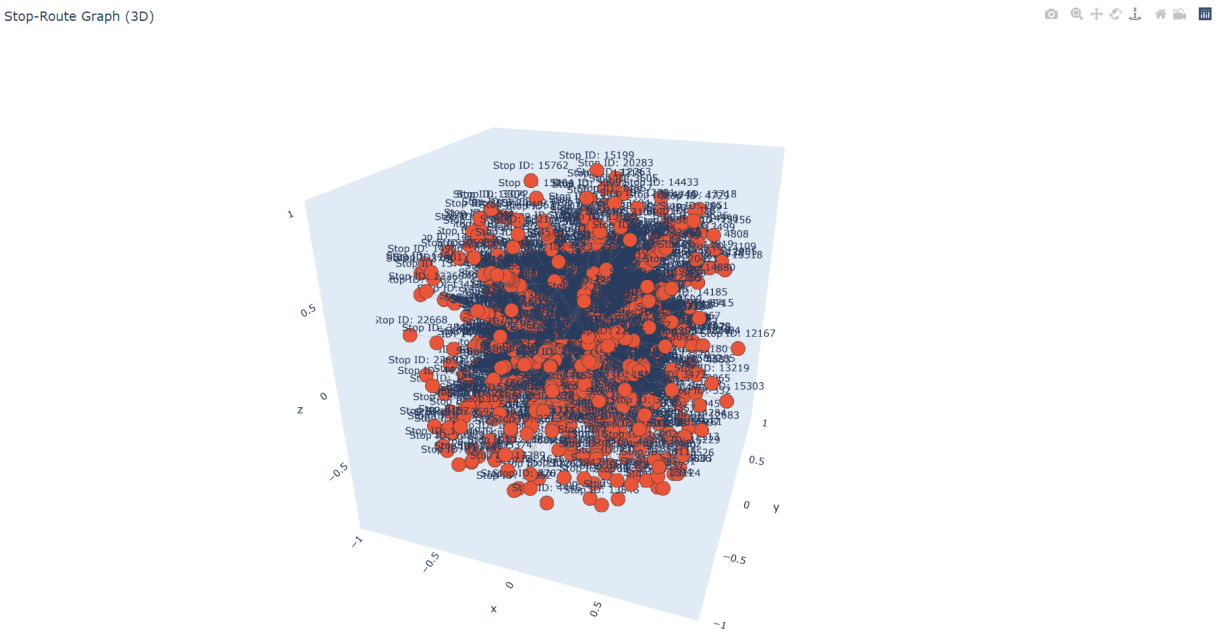
-Mapping Trip IDs to Route IDs: The first loop goes through each trip in df_trips and creates a dictionary (trip_to_route) where each trip_id is mapped to its corresponding route_id.
-Mapping Stops to Routes: The second loop iterates through df_stop_times, retrieving the route_id for each trip_id from the trip_to_route dictionary. It then adds each stop_id to the appropriate route's stop list in route_to_stops, if it's not already present(and as its already sorted as per the sequence hence we don't have to sort
-Counting Trips for Each Stop: The third loop counts how many times each stop_id appears in df_stop_times and increments the count in the stop_trip_count dictionary.

-Merging Fare Rules with Fare Attributes: The fourth loop populates the fare_rules dictionary with mappings from route_id to fare_id based on df_fare_rules. Finally, the df_fare_rules is merged with df_fare_attributes on fare_id to create merged_fare_df, combining both datasets.

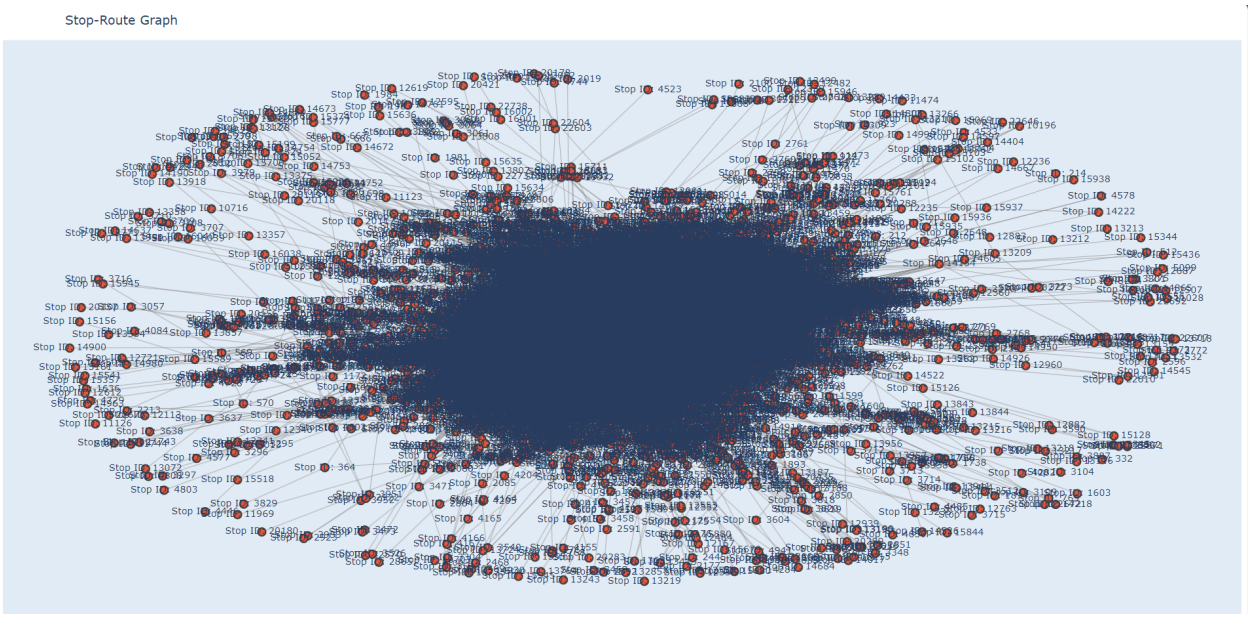
Graph constructed from stops.txt and route to stops



3 d graph that is unaware of latitude and longitude just takes Route to stops



2D graph that is unaware of latitude and longitude just takes Route to stops



Q2)Reasoning

1. Brute Force Approach

Explanation and Process

- **Goal**: Identify direct routes between two stops by iterating through all routes and checking if both stops exist in each route.
- **Approach**:
    - For each route in the list, check if both the start and end stops exist in that route.
    - If they do, append the route ID to the result.

(a) Time Complexity and Memory Usage for 10000 iter

- **Execution Time**: 16.52 seconds
- **Memory Usage**: 0.000672 MB

Since this approach iterates over all routes and performs checks on each route's list of stops, its time complexity is approximately O(N×M), where:

- N is the number of routes.
- M is the average number of stops per route.

Memory usage is low because we're only storing a small set of route IDs matching the direct route criteria.

(b) Intermediate Steps

1. **Loop Over Routes**: Traverse each route in the list.
2. **Check Stops**: For each route, check if both start and end stops are present.
3. **Store Result**: If both stops exist in the route, store the route ID as a valid direct route.

(c) Comparison of Steps

- **Total Steps**: The brute force approach performs approximately N×Mchecks, as it iterates through all stops in each route.

```
def direct_route_brute_force(start_stop, end_stop):
    ans = []
    for route_id, stops in route_to_stops.items():
        if (start_stop in stops) and (end_stop in stops):
                ans.append(route_id)
    return ans
```

2. First-Order Logic (FoL) Query-Based Approach

Explanation and Process

- **Goal**: Use logical predicates to query for direct and optimal routes, utilizing definitions of relationships among stops and routes.
- **Predicates Used**:
    - DirectRoute(X, Y, R): A direct route exists if X and Y are both on route R.
- **Query Process**:
    - Use DirectRoute to check for direct connections.

(a) Time Complexity and Memory Usage for 10000 iterations

- **Execution Time**: 44.82 seconds for querying direct routes
- **Memory Usage**: 1.39 MB

In this implementation, the time complexity varies depending on how efficiently the query can search through the relationships. Given that it processes all relationships among stops and

routes, the complexity can be approximated as O(N×M), with additional overhead for logical inference.

(b) Intermediate Steps

1. **Define Predicates**: Set up predicates that represent relationships between stops and routes.
2. **Execute Direct Route Query**: Use the `DirectRoute` predicate to find all direct routes.
3. **Execute Optimal Route Query**: Use the `OptimalRoute` predicate to find transfer routes with minimum transfers.

(c) Comparison of Steps

- **Steps**: The FoL approach likely involves more steps due to additional logical inference and data relationships it checks, especially when querying for optimal routes. We need to add route has stop then from route has stop we have to define direct route .

```
def query_direct_routes(start, end):
    Ans = DirectRoute(start, end, R).data
    ans2 = [it[0] for it in Ans]
    return ans2
```

Q3) Reasoning

1. Implementation and Constraints

We defined the following key predicates for our PyDatalog reasoning logic:

- `DirectRoute(X, Y, R)`: There is a direct route R between stop X and stop Y.
- `OptimalRoute(X, Y, R1, Z, R2)`: An optimal route exists between X and Y with one interchange at stop Z, involving two routes R1 and R2(where R1≠R2).
- Additional predicates, like `RouteHasStop(R, X)`, help determine if a route has a specific stop.

2. Forward Chaining

Code Explanation:

- Forward chaining starts at the **starting stop** and incrementally explores all possible paths toward the **end stop**.
- At each stop, it:
    - Checks if the current route meets the via stop constraint.
    - Tracks the number of interchanges to ensure only one interchange is included.
    - Prunes paths exceeding the maximum number of interchanges.

Performance: for 8000 iteration

- **Time Taken**: 420.69 seconds
- **Memory Usage**: 3.37 MB

Reasoning:

1. **Path Exploration**: Starts from the starting stop and explores each path step-by-step, which may result in revisiting some paths unnecessarily.
2. **Intermediate Steps**:
    - Paths that don't meet constraints (lack of mid stop or transfer !=max_transfers) are pruned.
    - All feasible paths satisfying constraints are recorded starting from the start node .first direct route from start to intermediate then intermediate to end . number of steps are S-I X I-E where S-I is number of route bw start and intermediate . and I-E is number of route bw intermediate and end

```
def forward_chaining(start_stop_id, end_stop_id, stop_id_to_include, max_transfers):
    Ans = OptimalRoute(start_stop_id, end_stop_id, R1, stop_id_to_include, R2).data
    ans2=[(route[0], stop_id_to_include, route[1]) for route in Ans]
    return ans2
```

3. Backward Chaining

Code Explanation:

- Backward chaining begins at the **end stop** and works backward, attempting to trace an optimal route back to the **starting stop**.
- It:
    - Identifies paths that reach the starting stop with exactly one interchange and a via stop.
    - Avoids exploring paths that cannot satisfy the via stop constraint early.

Performance: for 8000 iteration

- **Time Taken**: 418.85 seconds
- **Memory Usage**: 3.52 MB

Reasoning:

1. **Path Exploration**: Begins at the destination and explores paths backward, which can often rule out ineligible paths sooner.
2. **Intermediate Steps**:
    - Paths not satisfying the via stop constraint are pruned early.
    - Paths that don't meet the single interchange constraint are discarded.
    - But in our case the issue is that the backward chaining should get one path then but end but for our case we have to find all path which will lead to it be same as forward chaining but just construction from reverse
    - number of steps are E-I X I-S where E-I is number of route bw end and intermediate . and I-S is number of route bw intermediate and start
    - This also similar to forward chaining

```
def backward_chaining(start_stop_id, end_stop_id, stop_id_to_include, max_transfers):
    Ans = OptimalRoute(end_stop_id, start_stop_id, R1, stop_id_to_include, R2).data
    ans2 = [(route[0], stop_id_to_include, route[1]) for route in Ans]
    return ans2
```

Q4) Bonus

3. PDDL Planning

- Execution Time: 743.01 seconds
- Memory Usage: 2.35 MB
- Methodology:

    - Formulates the problem in PDDL, defining actions (`BoardRoute`, `TransferRoute`) and goals (`OptimalRoute`), with PyDatalog modeling each predicate.
    - The reasoning involves evaluating all possible actions and transitions, thus taking more computation time due to the extensive rule-checking within the PDDL-style logic.
- Info:
    - The PDDL approach is more flexible and explicitly structured for defining complex constraints.
    - Memory usage is slightly optimized as it uses a compact logic model for constraints and it uses smaller steps to query that the uses less memory
    - The time complexity is significantly higher due to 3 distinct query checking which is more than 2 of the forward and backward chaining . and also in forward and backward there are 2 join in pddl there are 3 joins
    - Intermediate steps are that first we find all route that we can board on start stop and then we find all pair of routes that transfer at intermediate steps and then all route that deboard at end stop hence 3 steps

Number of items at intermediate of each public test case for all 3 forward , backward and pddl

forward chaining tc1

route from start to middle

Number : 1

route from middle to end

Number : 1

Total end to end route

Number : 1

forward chaining tc2

route from start to middle

Number : 9

route from middle to end

Number : 1

Total end to end route

Number : 9

backward chaining tc1

route from end to middle

Number : 1

route from middle to start

Number : 1

Total end to end route

Number : 1

backward chaining tc2

route from end to middle

Number : 1

route from middle to start

Number : 9

Total end to end route

Number : 9

pddl planning tc 1

Starting with Boarding Route from the initial stop...

Number of initial boarding options: 7

Finding possible transfer routes...

Number of transfer options: 6

Finding possible final boardings to reach the end stop...

Number of final boarding options: 11

Combining first board and transfer options...

Number of combined options combining first board and transfer at middle : 2

Combining all

Number of final options: 1

pddl planning tc 1

Starting with Boarding Route from the initial stop...

Number of initial boarding options: 30

Finding possible transfer routes...

Number of transfer options: 2652

Finding possible final boardings to reach the end stop...

Number of final boarding options: 4

Combining first board and transfer options...

Number of combined options combining first board and transfer at middle : 459

Combining all

Number of final options: 9

Should all return same path:

In this case they will all have same path as optimality is not checked and all have just exactly one intermediate it wont affect as we are trying to attain all possible route not the most optimal route