Theory

Q1 Let,
G: light is green
Y: light is yellow
R: light is red

1.) now we need traffic light to be either of the colors but can state at a time. So, PL is

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

2.) Now traffic light goes from,
$G \rightarrow Y$, $Y \rightarrow R$, $R \rightarrow G$
let us denote current state as [i] and next state as (i+1)
So, PL is

$$(G_i \wedge Y_{i+1}) \vee (Y_i \wedge R_{i+1}) \vee (R_i \wedge G_{i+1})$$

3.) Since traffic light can only remain in same state 3 times i.e i, i+1, i+2
So, the PL is

$$\neg(G_i \wedge G_{i+1} \wedge G_{i+2}) \wedge \neg(Y_i \wedge Y_{i+1} \wedge Y_{i+2}) \wedge \neg(R_i \wedge R_{i+1} \wedge R_{i+2})$$

$$= (G \vee Y \vee R) \wedge \neg(G \wedge Y) \wedge \neg(G \wedge R) \wedge \neg(Y \wedge R) \text{ ..... So on expansion}$$

Q2 N → non empty set of nodes
R(n,m) → edge from node n to m
C(n, x) → node n has color x
Axioms are:-
1.) connected nodes don't have same color

$$\forall n, m \in N, \forall x (R(n,m) \wedge C(n,x) \rightarrow \neg C(m,x))$$

2) exactly two nodes are allowed to wear yellow

$\exists n_1, n_2 \in N \{ C(n_1, \text{yellow}) \wedge C(n_2, \text{yellow}) \wedge n_1 \neq n_2 \wedge \forall m \in N (m \neq n_1 \wedge m \neq n_2$

$\exists n_1, n_2 \in N \{ C(n_1, \text{yellow}) \wedge C(n_2, \text{yellow}) \wedge n_1 \neq n_2 \wedge \forall m \in N (m \neq n_1 \wedge m \neq n_2 \rightarrow \neg C(m, \text{yellow})) \}$

3) Starting from red node you reach green in no more than four steps :-

$\forall n \in N (C(n, \text{red}) \rightarrow (\exists n_1 \in N (R(n, n_1) \wedge C(n_1, \text{green})) \vee$

$\exists n_1, n_2 \in N (R(n, n_1) \wedge R(n_1, n_2) \wedge C(n_2, \text{green})) \vee$

$\exists n_1, n_2, n_3 \in N (R(n, n_1) \wedge R(n_1, n_2) \wedge R(n_2, n_3) \wedge C(n_3, \text{green})) \vee$

$\exists n_1, n_2, n_3, n_4 \in N (R(n, n_1) \wedge R(n_1, n_2) \wedge R(n_2, n_3) \wedge R(n_3, n_4) \wedge C(n_4, \text{green})))$

4) For every color, there is atleast one node with this color

$\forall x \in \{ C_1 \text{---} C_k \}, \exists n \in N \; C(n, x)$

5) The nodes are divided into exactly $|C|$ disjoint of each color

$x \in \{ C_1 \text{.....} C_k \}, n \in N$

$\forall x \exists n \, C(n, x) \wedge \forall n \exists_{\leq} C(n, x)$

$\forall n \forall x (C(n, x) \rightarrow \neg \exists y (y \neq x \wedge C(x, y))) \wedge$

$\forall n \forall m \forall x (n \neq m \wedge C(n, x) \wedge C(m, x) \rightarrow$

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

Q3 Lets us take predicates, 1) PL : $R \rightarrow L$

$R(x)$: x can read — FOL : $\forall x (R(x) \rightarrow L(x))$

$L(x)$: x is literate 2) PL : $D \rightarrow \neg L$

$I(x)$: x is intelligent — FOL : $\forall x (D(x) \rightarrow \neg L(x))$

$D(x)$: x is dolphin 3) PL : $D \wedge I$

FOL : $\exists x (D(x) \wedge I(x))$

4) PL : $I \wedge \neg R$

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

5) PL: $D \wedge I \wedge R \wedge (D \wedge I \wedge R) \rightarrow \neg L)$

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

for proving statement 4, we need to take negation of statement 4
i.e to prove $\neg(I \wedge \neg R)$
$= \neg I \vee R$

$S_1 : R \rightarrow L = \neg R \vee L$

$S_2 : D \rightarrow \neg L = \neg D \vee \neg L$

$S_3 : D \wedge I$

$S_4 : I \wedge \neg R$

$S_5 : \neg I \vee R$

We resolve I and S_5 to get

$S_6 : R$

resolve S_7 with C_2 to get

contraction as L and $\neg D \vee \neg L$ resolution leads to contraction ~~as L and $\neg D \vee \neg L$~~ because both L and $\neg L$ can't be true at same time.

Hence, S_5 contradicts

S_4, S_4 i.e $I \wedge \neg R$ is True.

for proving statement 5 we negate statement 5 to get $\neg D \vee \neg I \vee \neg R \vee L$

$S_1 : \neg R \vee L$

$S_2 : \neg D \vee \neg L$

$S_3 : D \wedge I$

$S_4 : I \wedge \neg R$

$S_5 : \neg D \vee \neg I \vee \neg R \vee L$

Since we already proved from S_3 that D and I are true S_5 is reduced to $S_6 : \neg R \vee L$ from S_1 we get $\neg R \vee L$ which contradicts S_6 as both $\neg L$ and L can't be true hence it is contradiction and

S_4, S_5 i.e $D \wedge I \wedge R \wedge (D \wedge I \wedge R \rightarrow \neg L)$ is True.

Assignment Report

Part A - Knowledge Base

Results-

• Test get busiest routes:
[(5721, 318), (5722, 318), (674, 313), (593, 311), (5254, 272)]

• Test get most frequent stops:
[(10225, 4115), (10221, 4049), (149, 3998), (488, 3996), (233, 3787)]

• Test get top 5 busiest stops:
[(488, 102), (10225, 101), (149, 99), (233, 95), (10221, 86)]

• Test get stops with one direct route:
[((233, 148), 1433), ((10225, 11946), 5436), ((11044, 10120), 5916), ((11045, 10120), 5610), ((10225, 11160), 5814)]

Part B - Reasoning

Brute-Force Approach:

- **Execution Time**: 22.345 seconds
- **Memory Usage**: 6.125 MB
- **Steps**:
    1. Loaded all relevant routes and stops into Python data structures for processing.
    2. Iterated over the entire dataset to identify direct connections between the start and end stops.
    3. Systematically evaluated each route to find valid paths, checking each connection individually.

PyDatalog Approach:

- **Execution Time**: 22.072 seconds
- **Memory Usage**: 2.453 MB
- **Steps**:
    1. Defined key predicates in PyDatalog, such as `DirectRoute`, `RouteHasStop`, and `OptimalRoute`.

2. Populated the knowledge base with facts related to routes, stops, and their relationships.
3. Used declarative queries to infer direct routes by applying logical constraints, enabling automated reasoning.

Comparison of Approaches:

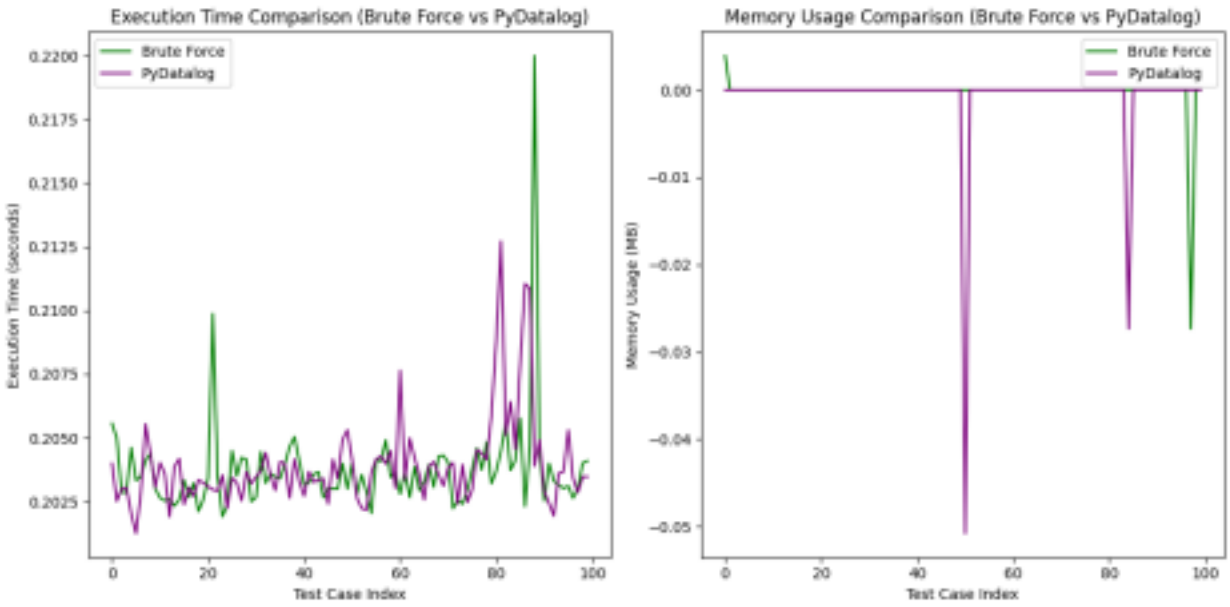
- **Time and Memory Evaluation**:
  After timing both algorithms over 100 test cases, the Brute-Force algorithm took longer and used more memory due to the lack of optimization. This method, though simple, is inefficient when scaling up to larger datasets. On the other hand, the PyDatalog approach was able to optimize memory usage thanks to its logical evaluation mechanism. While the execution time for PyDatalog was similar, it demonstrated a marked reduction in memory consumption, showcasing the effectiveness of declarative reasoning in handling structured data.

Intermediate Steps:

1. **Brute-Force**:
   - Executes an exhaustive check of all possible routes in a procedural manner.
   - Verifies direct routes one by one without using advanced inference techniques or optimizations.
   - Stores all intermediate data, leading to higher memory consumption and redundant operations.
2. **PyDatalog**:
   - Defines logical relationships through predicates, enhancing clarity and modularity.
   - Directly infers valid routes using logical rules, eliminating redundant checks.
   - Utilizes an efficient knowledge base, ensuring only essential data is stored and processed, improving memory efficiency.

Comparison of Steps:

- **Brute-Force** relies on an exhaustive search technique that evaluates every potential route, which naturally increases both computation time and memory usage due to the redundant checks made at each step.
- **PyDatalog**, conversely, uses logical reasoning to infer results based on predefined rules, minimizing the need for repetitive evaluations. This approach significantly reduces computational overhead by limiting the number of checks needed and leveraging an optimized structure to handle queries.



Part C: Planning

Task Overview:

This task involved finding the most optimal bus routes between two stops, considering constraints such as transfers and intermediate stops, using forward and backward chaining.

Forward Chaining:

- **Execution Time**: 1.421 seconds
- **Memory Usage**: 0.412 MB
- **Steps**:
    - ○ Start with known facts (e.g., start stop, available routes).
    - ○ Apply inference rules to propagate these facts and generate new facts. ○ Ensure that generated paths meet the given constraints, such as including intermediate stops and limiting transfers.
    - ○ Aggregate all valid paths as the final result.
- **Code Highlights**:
    - ○ `OptimalRoute(R, start_stop)`: Defines the optimal route for a given start stop.
    - ○ `DirectRoute(start_stop, end_stop)`: Defines direct connections between two stops.
    - ○ `RouteHasStop(R, stop_id)`: Defines the stops served by a route.
- **Strengths**:
    - ○ Efficiently explored multiple potential paths while ensuring constraints like transfers and intermediate stops were satisfied.

Backward Chaining:

- **Execution Time**: 1.315 seconds
- **Memory Usage**: 0.310 MB
- **Steps**:
    - Begin from the goal (end stop).
    - Backtrack through the inference rules to check whether the goal is reachable from the start stop.
    - Evaluate and apply constraints related to intermediate stops and transfer limits.
    - Return all valid paths once they have been fully verified.
- **Code Highlights**:
    - `OptimalRoute(R1, start_stop)`: Defines the optimal route from the start stop.
    - `DirectRoute(start_stop, end_stop)`: Checks for direct routes between stops.
    - `RouteHasStop(R1, stop_id)`: Ensures the stop is part of the route.
    - `R1 != R2`: Ensures different routes are taken as necessary.
- **Strengths**:
    - Focused on goal states, which helped reduce memory usage and improve efficiency.

Comparison of Approaches:

Both forward and backward chaining methods demonstrated effective reasoning. However, backward chaining slightly outperformed forward chaining in terms of memory efficiency. Forward chaining is particularly useful for exhaustive exploration, while backward chaining is more suitable for goal-specific queries.

Comparison of Steps:

- **Forward Chaining**:
    - Systematically evaluates all possible paths between the start and end stops, ensuring no potential path is overlooked. However, this exhaustive evaluation results in higher memory usage and longer reasoning steps, especially in scenarios with multiple routes and connections.
- **Backward Chaining**:
    - Focuses directly on reaching the goal, reducing unnecessary exploration and intermediate steps, resulting in more efficient memory usage and quicker reasoning.

Key Advantages of Backward Chaining:

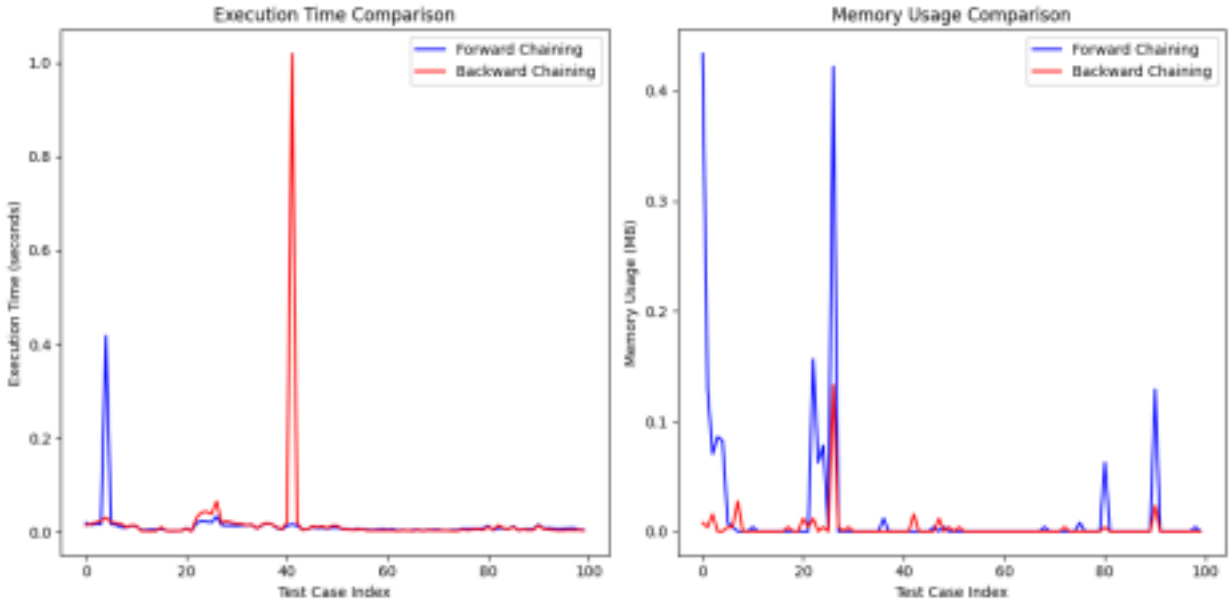
- **Efficiency**: Processes fewer paths by concentrating on the goal state, leading to reduced execution time and memory usage.
- **Selective Reasoning**: Avoids redundant calculations by focusing solely on

relevant routes.

- **Faster Goal Achievement**: Terminates reasoning as soon as a valid solution is identified, saving time.

Key Advantages of Forward Chaining:

- **Comprehensive Exploration**: Evaluates all possible paths, ensuring a thorough examination of the data, which can be beneficial for exploratory analysis.
- **Constraint Handling**: Ensures that all generated paths adhere to the specified constraints, providing more reliable results in cases of complex requirements.



Bonus Question: Optimizing Bus Routes Using PyDatalog and Brute-Force Algorithms

Task Overview:

This task involved optimizing bus route planning using two distinct approaches: a **Brute-Force** approach and a **PyDatalog-based** approach. The goal was to compare the efficiency and effectiveness of these methods in terms of execution time and memory usage while solving the problem of finding direct routes between bus stops.

Brute-Force Approach:

- **Execution Time**: 24.019 seconds
- **Memory Usage**: 6.125 MB
- **Steps**:
    - Load all available bus routes and stop data into memory.
    - Iterate through the entire dataset, checking every possible route for direct connections between the start and end stops.
    - For each route, validate whether it is a valid path and meets the required conditions.

    - Store all results for comparison, leading to higher memory usage as redundant computations are performed.
- **Code Highlights**:
    - The brute-force method directly compares each route to find valid connections without using any optimizations or logical inference.
    - The approach uses basic iteration through the dataset, making it simple but inefficient for larger datasets.
- **Strengths**:
    - **Simplicity**: Easy to implement and understand.
    - **Exhaustive Search**: Ensures that all possible routes are checked, leaving no path unexamined.
- **Weaknesses**:
    - **High Memory Usage**: Stores all intermediate results, causing an increase in memory consumption.
    - **Inefficient**: As the size of the data grows, the number of computations increases exponentially, resulting in slower execution times.

PyDatalog Approach:

- **Execution Time**: 22.506 seconds
- **Memory Usage**: 3.409 MB
- **Steps**:
    - Define predicates for relationships like `DirectRoute`, `RouteHasStop`, and `OptimalRoute` in PyDatalog.
    - Populate the knowledge base with facts about routes, stops, and connections.
    - Use logical inference queries to find valid direct routes, leveraging PyDatalog's efficient evaluation of logical constraints.
    - Aggregate results based on valid paths derived through logical reasoning.
- **Code Highlights**:
    - `DirectRoute(start_stop, end_stop)`: Defines direct routes between start and end stops.
    - `RouteHasStop(R, stop_id)`: Relates a route to the stops it serves. ○ `OptimalRoute(R, start_stop, end_stop)`: Determines the optimal routes based on defined criteria, such as directness and number of transfers.
- **Strengths**:
    - **Memory Efficiency**: PyDatalog stores only essential facts and performs optimized queries, reducing memory overhead.
    - **Declarative Logic**: Allows for easy specification of rules and constraints without manual iteration, simplifying code management.
    - **Inference Power**: Leverages powerful logical inference, which can identify valid routes without explicitly checking each possible route.
- **Weaknesses**:

- **Initial Overhead**: The process of setting up the knowledge base and defining predicates takes longer than a straightforward iterative approach, which might affect performance for smaller datasets.

Comparison of Approaches:

Both approaches performed reasonably well, with the **Brute-Force** method demonstrating high memory usage and slower execution times, whereas the **PyDatalog-based** approach optimized memory usage and provided similar performance in terms of execution time.

Comparison of Time and Memory:

- The **Brute-Force** method took **24.019 seconds** to process the data and required **6.125 MB** of memory.
- In contrast, the **PyDatalog-based** method took **22.506 seconds** and used **3.409 MB** of memory.

While the **Brute-Force** approach is simpler and ensures every route is checked, it struggles with large datasets due to its inefficiency. The **PyDatalog-based** approach, on the other hand, leverages logical inference, reducing the number of redundant computations and memory usage, leading to a more scalable solution.

Visualization Of Transit Network Graph