AI Assignment 2

Vikranth Udandarao
2022570

Theory

1.

Let:

$G(t)$: The traffic light is green at time t.

$Y(t)$: The traffic light is yellow at time t.

$R(t)$: The traffic light is red at time t.

a. "At any given moment, the traffic light is either green, yellow, or red."

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

b. "The traffic light switches from green to yellow, yellow to red, and red to green."

$(G(t) \rightarrow Y(t+1)) \wedge (Y(t) \rightarrow R(t+1)) \wedge (R(t) \rightarrow G(t+1))$

c. "The traffic light cannot remain in the same state for more than 3 consecutive cycles."

For any state S:

$\neg(S(t) \wedge S(t+1) \wedge S(t+2) \wedge S(t+3))$

should satisfy the condition:

$\{ \neg(G(t) \wedge G(t+1) \wedge G(t+2) \wedge G(t+3)) \wedge \neg(Y(t) \wedge Y(t+1) \wedge Y(t+2) \wedge Y(t+3)) \wedge \neg(R(t) \wedge R(t+1) \wedge$
$R(t+2) \wedge R(t+3)) \}$

Assuming S(t) is the 1st cycle and S(t+1) is the next cycle.

2.

Let:

color(u, c): color of node u is c

edge(u, v): Node u and node v are directly connected

path(x, y, n): there exists a path from x to y of length $\leq$ n

inClique(x, q): node x is in a clique q

cliqueColor(q, c): clique q has color c

Node(u): Node u is a node in graph

Rule 1: "Connected nodes don't have the same color."
If two nodes are connected by an edge, they cannot share the same color.

∀u,v[Node(u)∧Node(v)∧edge(u,v)]→¬∃c[color(u,c)∧color(v,c)]

Rule 2: "Exactly two nodes are allowed to wear yellow."

∃u∃v[Node(u)∧Node(v)∧u≠v∧color(u,yellow)∧color(v,yellow)∧∀z¬(color(z,yellow)∧z≠u∧z≠v)]

Rule 3: "Starting from any red node, you can reach a green node in no more than 4 steps."

∀x(Node(x)∧color(x,red)→∃y1, y2, y3, y4 (Node(y1)∧Node(y2)∧Node(y3)∧Node(y4)∧ edge(x,y1)∧edge(y1
,y2)∧edge(y2,y3)∧edge(y3 ,y4)∧color(y4,green))

Rule 4: "For every color in the palette, there is at least one node with this color."

∀c∃u(Node(u)∧color(u,c))

Rule 5: "The nodes are divided into exactly |C| disjoint non-empty cliques, one for each color."
Clique for each color: Nodes with the same color should form a clique, meaning that any two nodes with the same color are connected to each other.

∀c,c′ ∃x(inClique(x,q)) ∧(∀y,y′ ∈inClique(y,c) ∧inClique(y′ ,c′))→c≠c′

3.
Let:
R: Individual can read.
L: Individual is literate.
D: Individual is a dolphin.
I: Individual is intelligent.

canRead(x): x can read.
literate(x): x is literate.
dolphin(x): x is a dolphin.
intelligent(x): x is intelligent.

a. Whoever can read is literate:

Propositional Logic: $R \rightarrow L$

First Order Logic: $\forall x\ (canRead(x) \rightarrow literate(x))$

b. Dolphins are not literate:

Propositional Logic: $D \rightarrow \neg L$

First Order Logic: $\forall x\ (dolphin(x) \rightarrow \neg\ literate(x))$

c. Some dolphins are intelligent:

Propositional Logic:

It is not expressible in PL without representing specific dolphins as individual variables as quantifiers can't be expressed in PL.

For specific dolphins:

Let dolphins be D1, D2, D3, ...

Let intelligent be I1, I2, I3, ...

$(D1 \vee I1) \wedge (D2 \vee I2) \wedge (D3 \vee I3)$ ...

First Order Logic: $\exists x\ (dolphin(x) \wedge intelligent(x))$

d. Some who are intelligent cannot read:

Propositional Logic:

It can't be expressed in PL as quantifiers can't be expressed.

For specific dolphins:

$(I \wedge \neg R)$

First Order Logic: $\exists x\ (intelligent(x) \wedge \neg\ canRead(x))$

e. 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 literate:

Propositional Logic:

It can't be expressed in PL as quantifiers can't be expressed.

For specific dolphins:

$(D \wedge I \wedge R) \wedge (I \wedge R \rightarrow \neg L)$

First Order Logic: $\exists x\, (dolphin(x) \wedge intelligent(x) \wedge canRead(x)) \wedge \forall y\, (dolphin(y) \wedge intelligent(y) \wedge canRead(y) \rightarrow \neg\, literate(y))$

(I) For S4

We have to prove contradiction – i.e., $\forall x\, (intelligent(x) \rightarrow canRead(x))$.

From S2, we know all dolphins are not literate, i.e., $\forall x\, (dolphin(x) \rightarrow \neg literate(x))$.

From S1, if a being can read, it must be literate, i.e., $\forall x\, (canRead(x) \rightarrow literate(x))$.

If we assume $\forall x\, (intelligent(x) \rightarrow canRead(x))$, this would imply that all intelligent beings must be literate (from S1 & S2).

But this conflicts with the idea in S3 that some intelligent beings are dolphins, & dolphins can't be literate by S2.

Contradiction - This contradiction implies that S4 is satisfiable with the given premise.

$\Rightarrow$ 4th statement is satisfiable.

(II) For S5

Goal - Prove negation of S5 is false, i.e., no dolphin exists that is both intelligent and can read, OR there exists an intelligent dolphin who can read & is literate.

Steps -

Assume there exists an intelligent dolphin who can read and is literate.

From S1, any being who can read must be literate.

From S2, no dolphin can be literate, meaning any dolphin who can read would violate this rule.

This assumption leads to contradiction.

⇒ 5th statement is unsatisfiable as it cannot be true under these premises.

We have to prove contradiction, i.e., $\forall x$ (Intelligent(x) $\rightarrow$ Read(x))

From statement 2, we know all dolphins are not literate, i.e.,
$\forall x$ (dolphin(x) $\rightarrow$ ¬literate(x))

From statement 1,
$\forall x$ (Read(x) $\rightarrow$ literate(x))

Converting to CNF - [Rule: (R $\rightarrow$ L) := (¬R $\vee$ L)]

S1) ¬Read(x) $\vee$ literate(x)
S2) ¬dolphin(x) $\vee$ ¬literate(x)
S3) dolphin(x) $\wedge$ Intelligent(x)
S4) Intelligent(x) $\wedge$ ¬Read(x) $\Leftarrow$ To prove

Now,
Assume Intelligent(x) to be true, add to KB.

Then,
Resolve S3: dolphin(x) $\wedge$ Intelligent(x) giving
S6: dolphin(x)

Resolve S4 using S6, giving: ¬literate(x) $\rightarrow$ S7

Resolve S1 using S7, giving: ¬Read(x) $\rightarrow$ S8

We get,
All conditions are satisfied, and no contradiction to any existing fact, thus our assumption is true.

$\therefore$ Intelligent(x) $\rightarrow$ S8

From S8 and S4 we get,
Intelligent(x) $\wedge$ ¬Read(x)

Thus proving S4

5th statement:

Goal: To prove negation of S5 is false, i.e., no dolphin exists (i.e., both intelligent and can read), or there exists an intelligent dolphin who can read and is literate.

Steps:
To prove: ∃x (dolphin(x) ∧ intelligent(x) ∧ read(x))
∀x (intelligent(x) ∧ read(x) → ¬literate(x))

Solution:
From the previous,
S = {S1, S2, S3, S4, S6, S7, S8, S9}
all satisfy the conditions and are known to be true.

Now, from S6, S8, and S9, we get
{dolphin(x) ∧ intelligent(x) ∧ ¬read(x)}

which contradicts the "there exists x" statement in S5.

Also,
intelligent(x) ∧ read(x) cannot be entailed from the knowledge base, as it contradicts S8 and S9.

To CNF for 2nd part,
[¬dolphin(x) ∨ ¬intelligent(x) ∨ ¬literate(x)]

which is also unsatisfiable from: S6, S7, S9.

Thus, S5 cannot be true under the known facts, making it unsatisfiable.

Computational

The `create_kb` function constructs a knowledge base by organizing data from various datasets to establish key relationships among routes, trips, stops, and fare rules:

1. **Mapping Trips to Routes**: It iterates through the `df_trips` DataFrame, mapping each `trip_id` to its corresponding `route_id`.
2. **Mapping Routes to Stops**: Using `df_stop_times`, it assigns a list of stops to each route in the order they are encountered, using `stop_sequence` and `stop_id`. It also counts the number of trips per stop.
3. **Ensuring Unique Stops**: For each route, it removes duplicate stops and orders them by `stop_sequence`.
4. **Setting Up Fare Rules**: The function creates a dictionary of fare rules indexed by `route_id` using data from `df_fare_rules`.
5. **Merging Fare Data**: Finally, it merges `df_fare_rules` and `df_fare_attributes` into `merged_fare_df` for a unified fare information view.

The `get_busiest_routes` function identifies the top 5 busiest routes based on the number of trips:

1. **Counting Trips per Route**: It creates a dictionary (`route_trip_count`) that increments the count for each route every time it encounters a `trip_id` associated with that route.
2. **Sorting and Selecting Top 5**: After counting trips for each route, it sorts the routes by trip count in descending order and retrieves the top 5 routes with the highest trip counts.
3. **Return**: It returns a list of tuples, each containing a `route_id` and its corresponding `trip_count`.

The `get_most_frequent_stops` function identifies the top 5 stops with the highest number of trips:

1. **Sorting Stops by Trip Count**: It sorts the `stop_trip_count` dictionary (which holds the count of trips for each stop) in descending order based on trip count.
2. **Selecting Top 5 Stops**: It retrieves the top 5 stops with the most trips.
3. **Return**: It returns a list of tuples, each containing a `stop_id` and its corresponding `trip_count`

The `get_top_5_busiest_stops` function finds the top 5 stops with the most routes passing through them:

1. **Mapping Stops to Routes**: It creates a dictionary (`stop_to_routes`) where each `stop_id` maps to a set of `route_ids` passing through that stop, ensuring each route is counted only once per stop.
2. **Counting Routes per Stop**: It calculates the number of unique routes for each stop and stores these counts in `stop_route_count`.
3. **Selecting Top 5**: It sorts the stops by route count in descending order and retrieves the top 5.
4. **Return**: It returns a list of tuples, each containing a `stop_id` and its `route_count`.

The `get_stops_with_one_direct_route` function identifies the top 5 pairs of stops that are connected by exactly one direct route:

1. **Counting Direct Connections**: It creates a nested dictionary (`stop_pairs`) to store pairs of consecutive stops for each route. For each pair, it records the route that connects them and increments the count.
2. **Filtering Single-Route Pairs**: It extracts pairs connected by only one unique route, ensuring there's no other route that connects the same stop pair.
3. **Sorting and Selecting Top 5**: It sorts these pairs by the combined trip frequency of both stops and retrieves the top 5 pairs with the highest trip frequency.
4. **Return**: It returns a list of tuples, where each tuple contains a stop pair and the `route_id` connecting them.

The `visualize_stop_route_graph_interactive` function creates an interactive graph visualization of the stops and routes using Plotly and NetworkX:

1. **Graph Creation**: It initializes an undirected graph (`G`) using NetworkX, where nodes represent stops and edges represent direct routes between consecutive stops.
2. **Positioning Nodes**: It calculates a spring layout (`pos`) to position nodes in a visually appealing way.
3. **Plotting Edges**: It creates the edge traces for the graph by adding x and y coordinates for each edge and labels them with route information.
4. **Plotting Nodes**: It creates the node traces for each stop with x and y positions and adds hover text to display the stop ID.
5. **Interactive Visualization**: It combines edge and node traces into a Plotly figure, sets up the layout, and saves the visualization as an HTML file (`stop_route_graph.html`) for interactive exploration in a web browser. It also displays the plot directly if running in an environment with Plotly visualization support.

The `direct_route_brute_force` function identifies all direct routes connecting two specified stops using a brute-force approach:

1. **Iterate Over Routes**: It loops through each route in `route_to_stops`, checking if both `start_stop` and `end_stop` are present in the list of stops for that route.
2. **Identify Direct Routes**: If both stops are found within a route, the route is added to `direct_routes` as it provides a direct connection between the stops.
3. **Return**: It returns a list of `route_ids` that directly connect the `start_stop` and `end_stop`.

The `initialize_datalog` function sets up Datalog predicates to facilitate reasoning about routes and stops:

1. **Clearing Terms**: It clears any previously defined terms in `pyDatalog` to ensure a fresh setup.
2. **Defining Predicates**: It establishes a `DirectRoute` predicate, which defines a direct connection between two stops `X` and `Y` within a route `R`, based on the existence of the `RouteHasStop` relationship for each stop and the condition that `X` is not equal to `Y`.
3. **Populating Knowledge Base**: It calls `create_kb()` to initialize the global data structures and `add_route_data()` to load route and stop data into Datalog, making it ready for logical queries.
4. **Confirmation Print**: It outputs a confirmation message indicating that key terms have been initialized.

The `add_route_data` function loads route and stop data into Datalog for logical reasoning:

1. **Iterating Over Routes and Stops**: It loops through each route in `route_to_stops` and then iterates through the stops associated with each route.
2. **Defining `RouteHasStop` Predicate**: For each `route_id` and `stop_id` pair, it asserts a `RouteHasStop` predicate in Datalog, establishing that the specified route includes the specified stop.
3. **Enabling Reasoning**: This setup allows for querying relationships between stops and routes in the Datalog framework, supporting logical reasoning tasks based on these associations.

The `query_direct_routes` function retrieves routes that directly connect two specified stops using Datalog queries:

1. **Querying Datalog**: It uses the `DirectRoute` predicate to ask if there exists a route `R` that connects the given `start` and `end` stops directly.
2. **Checking Results**: If no direct routes are found, it returns an empty list.
3. **Extracting and Sorting Route IDs**: For valid results, it extracts the `route_id` from each answer, sorts them, and returns a sorted list of route IDs that directly connect the two stops.

The `forward_chaining` function uses forward chaining logic to find optimal routes between two stops, allowing for one intermediate transfer:

1. **Defining Optimal Routes**: It sets up a Datalog rule for `OptimalRoute` to identify routes where a transfer occurs at a specified `stop_id_to_include`. This rule connects the `start_stop_id` to the intermediate stop via one route (`R1`) and then connects the intermediate stop to the `end_stop_id` via another route (`R2`).
2. **Querying Datalog**: It queries Datalog for `OptimalRoute` paths that match the criteria, retrieving valid route pairs that connect `start_stop_id` to `end_stop_id` through the intermediate stop.
3. **Storing Valid Paths**: It appends each valid path as a tuple (`route_id1`, `stop_id`, `route_id2`) to the `paths` list.
4. **Return**: It returns the list of paths if any are found, or an empty list otherwise.

The `backward_chaining` function finds optimal routes between two stops, allowing for a transfer at a specified stop, by applying backward chaining logic:

1. **Defining Optimal Routes**: It defines an `OptimalRoute` rule, allowing routes that connect `start_stop_id` to `end_stop_id` through an intermediate stop (`Z`). This rule requires a direct route `X` from `start_stop_id` to `Z` and a direct route `Y` from `Z` to `end_stop_id`.
2. **Querying for Valid Paths**: It queries for `OptimalRoute` paths using `stop_id_to_include` as the transfer point and retrieves valid route pairs that satisfy this transfer condition.
3. **Storing Paths**: For each valid path, it appends a tuple with (`route_id1`, `stop_id`, `route_id2`) to the `paths` list, ensuring it avoids duplicate routes.
4. **Return**: It returns a list of paths if any are found; otherwise, it returns an empty list.

The `pddl_planning` function uses a PDDL (Planning Domain Definition Language)-inspired approach to find routes with optional transfers:

1. **Defining Actions**: It defines two actions using Datalog:
    - `board_route`: Allows boarding a route at a given stop (`R` at `X`).
    - `transfer_route`: Describes a transfer action, where a passenger switches from route `R1` to `R2` at a specified transfer stop `Z` (connecting `start_stop_id` to `end_stop_id`).
2. **Constructing the Route Plan**: It formulates a route plan combining the actions:
    - Boarding at `start_stop_id`.
    - Transferring at `stop_id_to_include`.
    - Boarding the next route to reach `end_stop_id`.

3. **Collecting Paths**: For each valid action sequence, it constructs a path (`route1`, `stop_id`, `route2`) and appends it to `paths`.
4. **Return**: It returns the list of valid paths if any are found, or an empty list otherwise, providing possible routes with the specified transfer.

The `prune_data` function filters fare data to include only routes within a specified fare limit:

1. **Applying Fare Limit**: It filters `merged_fare_df` by selecting rows where the fare price (`price` column) is less than or equal to `initial_fare`.
2. **Return**: It returns a new DataFrame (`pruned_df`) containing only the routes that meet the fare constraint, helping to limit route options based on budget.

The `compute_route_summary` function pre-computes and organizes route information, focusing on fare and stop details:

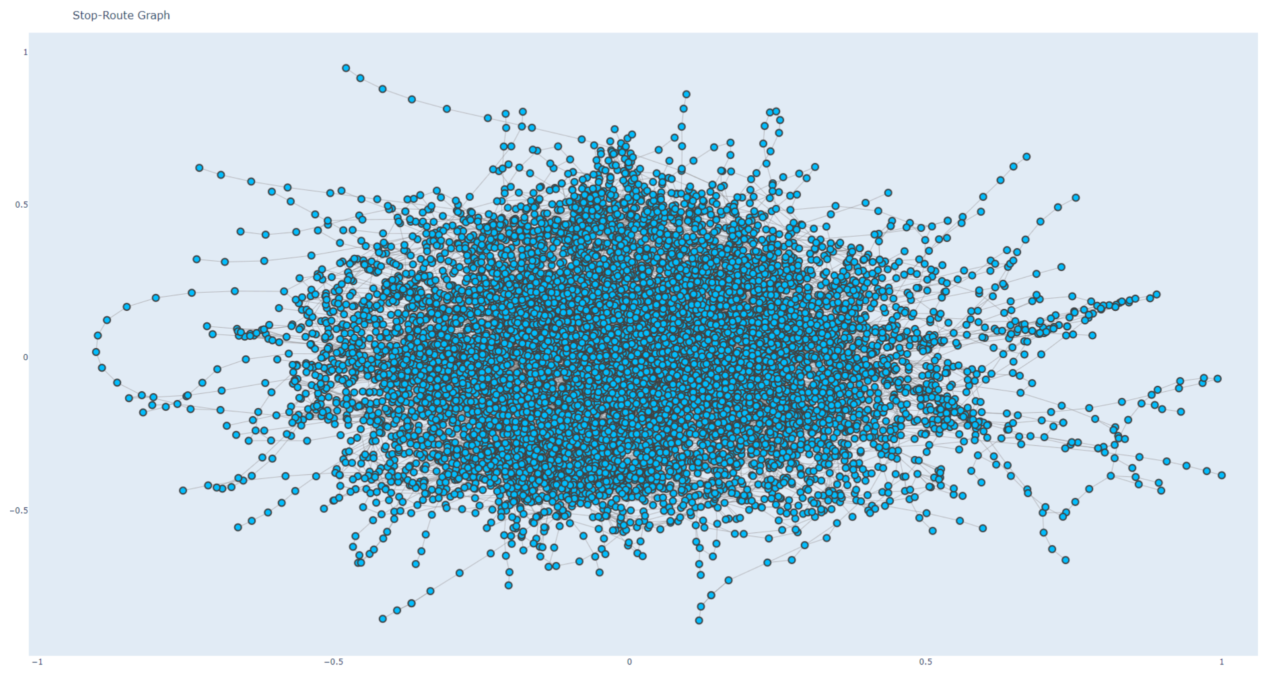
1. **Grouping by Route**: It groups `pruned_df` by `route_id`, isolating fare and stop information for each route.
2. **Calculating Minimum Fare and Stop Set**: For each route:
    - **Minimum Fare**: It determines the lowest fare for the route (`min_price`).
    - **Stop Set**: It collects a unique set of stops (`origin_id` and `destination_id`) for that route.
3. **Return**: It builds and returns a dictionary, `route_summary`, where each route ID maps to its minimum fare and set of stops, streamlining data access for subsequent route planning steps.

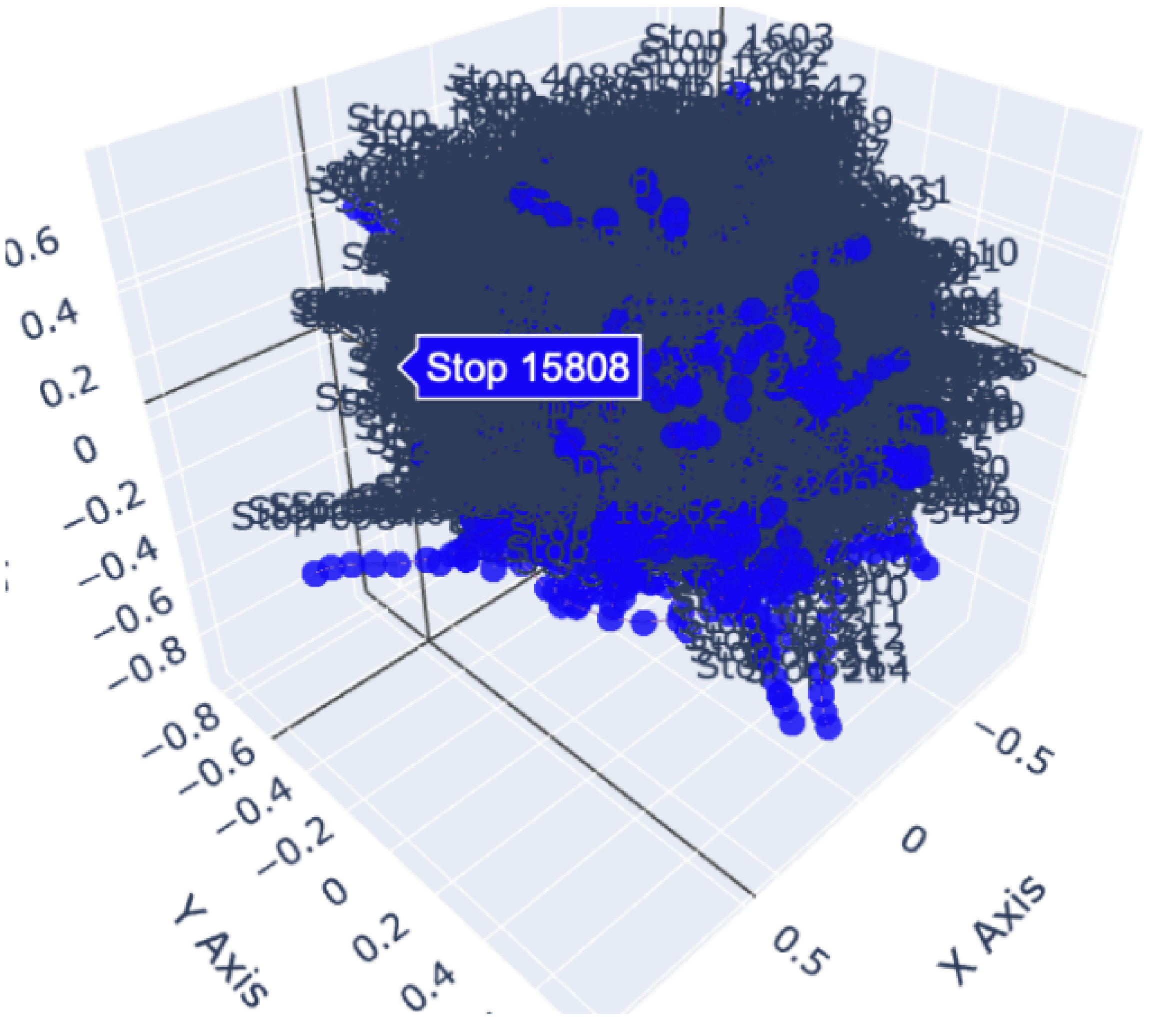
The `bfs_route_planner_optimized` function uses Breadth-First Search (BFS) to find an optimal route between two stops, considering fare and transfer constraints:

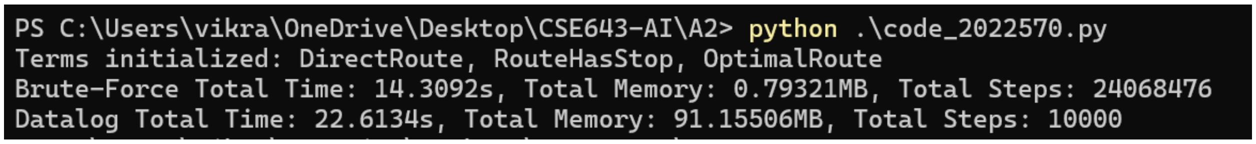
1. **Initializing BFS Queue**: It starts with a queue containing the `start_stop_id`, initial fare, an empty path, and zero transfers.
2. **Processing Each Node**: For each stop dequeued:
    - **Check for Destination**: If the current stop is `end_stop_id`, it returns the path taken.
    - **Transfer Limitation**: It skips routes if the transfer count exceeds `max_transfers`.
3. **Exploring Routes and Stops**: For each route:
    - **Fare and Stop Check**: It only considers routes where the current stop is in the route and the fare fits within the remaining budget.
    - **Path Update**: It generates new paths with updated fare, path, and transfer counts.
4. **Marking Visited Stops**: It tracks visited stops and their remaining fare to avoid reprocessing.

5. **Return**: If no path is found, it returns an empty list; otherwise, it returns a path listing the route and stop pairs taken to reach the destination.

The Time, Memory and Number of Steps can be found here Results

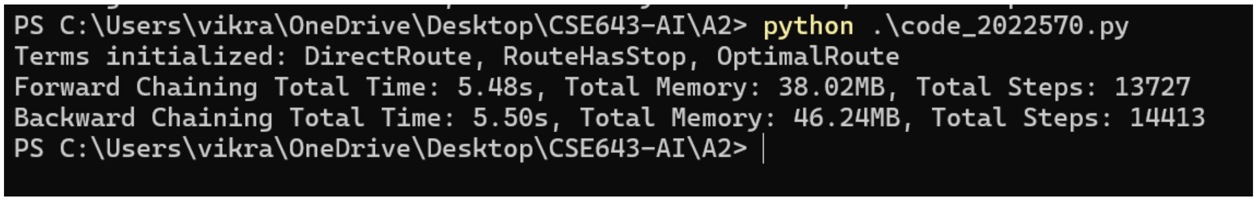






```
PS C:\Users\vikra\OneDrive\Desktop\CSE643-AI\A2> python .\code_2022570.py
Terms initialized: DirectRoute, RouteHasStop, OptimalRoute
Brute-Force Total Time: 14.3092s, Total Memory: 0.79321MB, Total Steps: 24068476
Datalog Total Time: 22.6134s, Total Memory: 91.15506MB, Total Steps: 10000
```

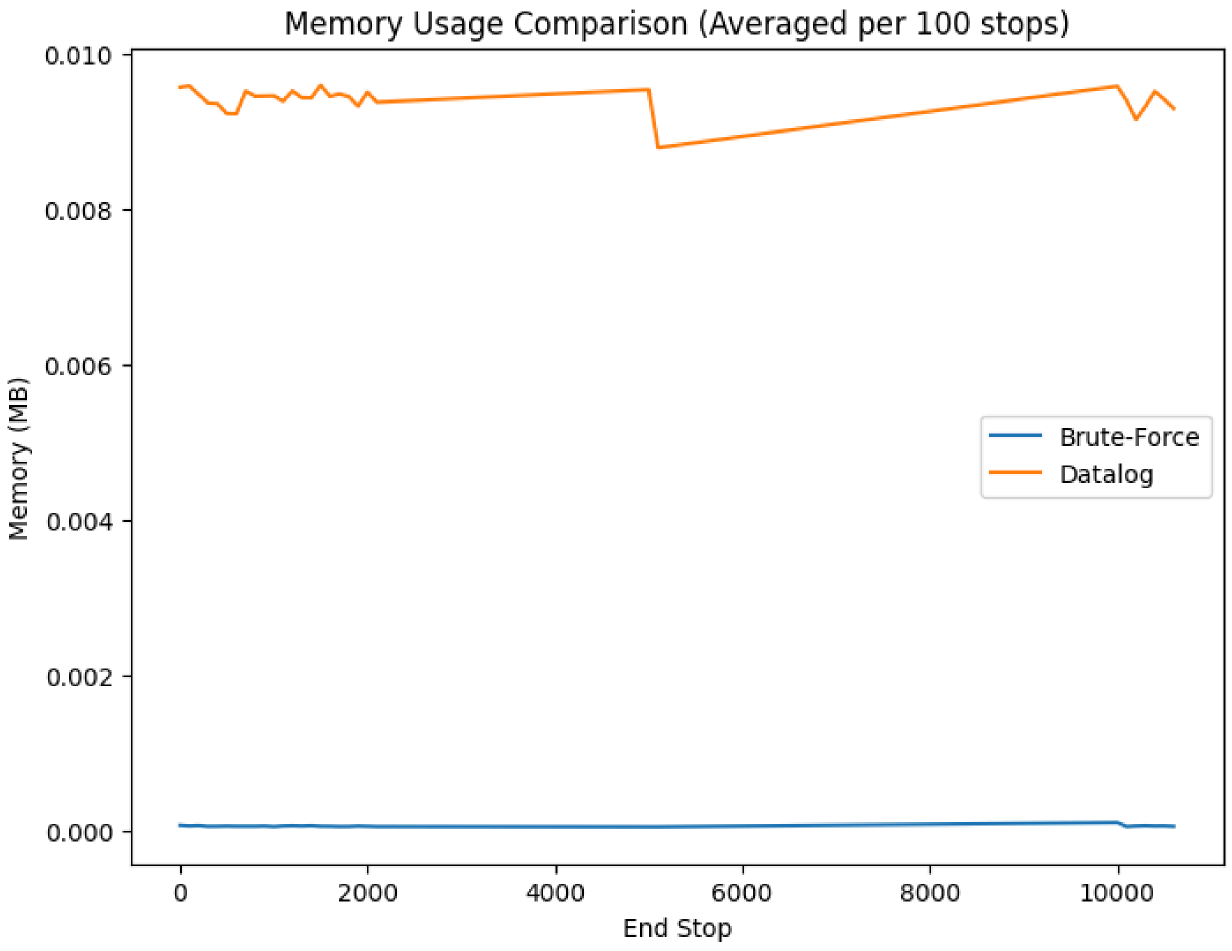
The brute-force approach used 0.79 MB of memory, processing each step sequentially with minimal storage. In contrast, Datalog required 91.1 MB due to its handling of complex queries and storing intermediate results. The higher memory usage in Datalog reflects the overhead of managing advanced logic operations and rule-based querying. Brute-force took 14.3 seconds, as it exhaustively checked every combination without optimization. In comparison, query Datalog took 22.6 seconds, likely due to the added complexity of handling advanced logic and querying multiple facts

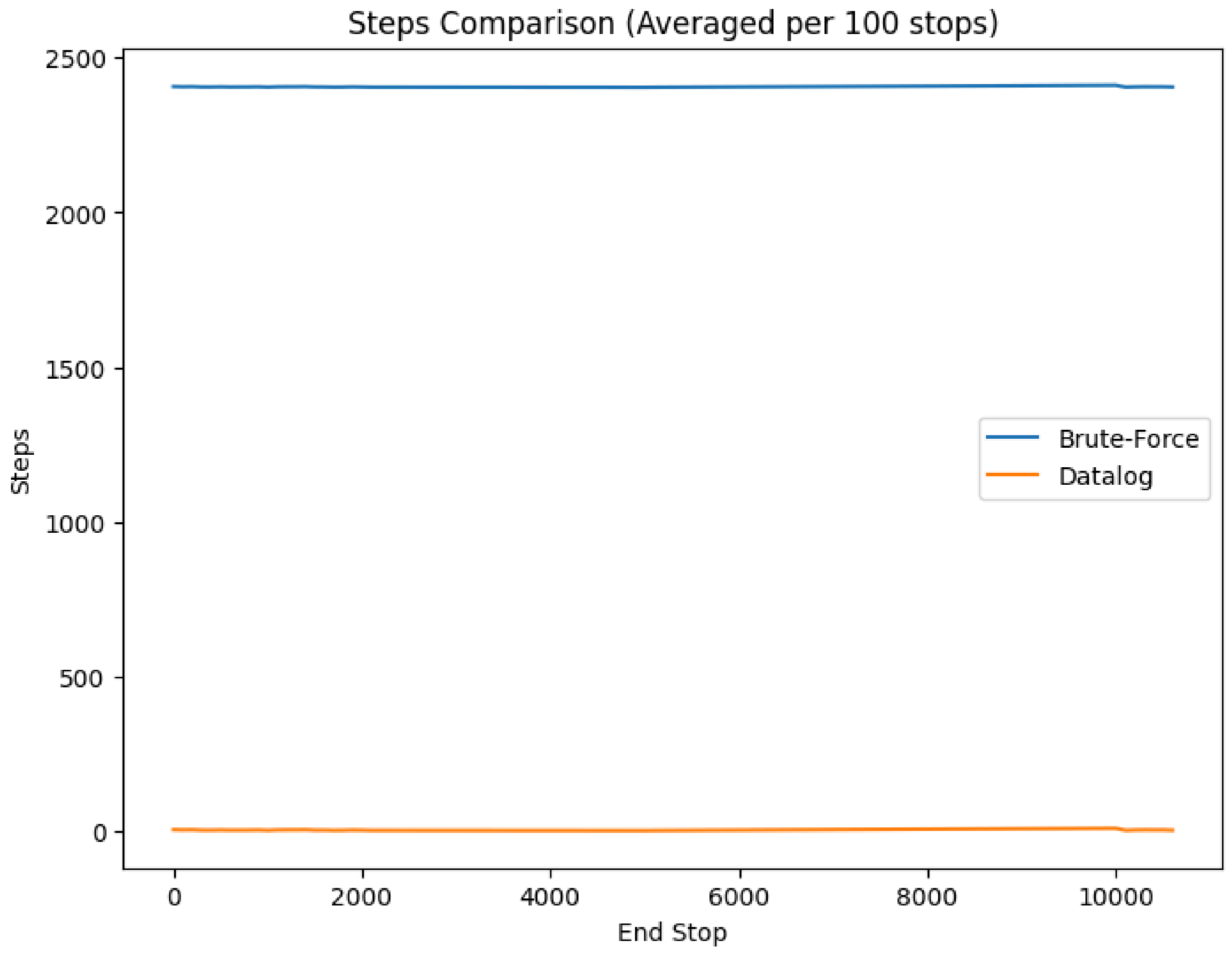


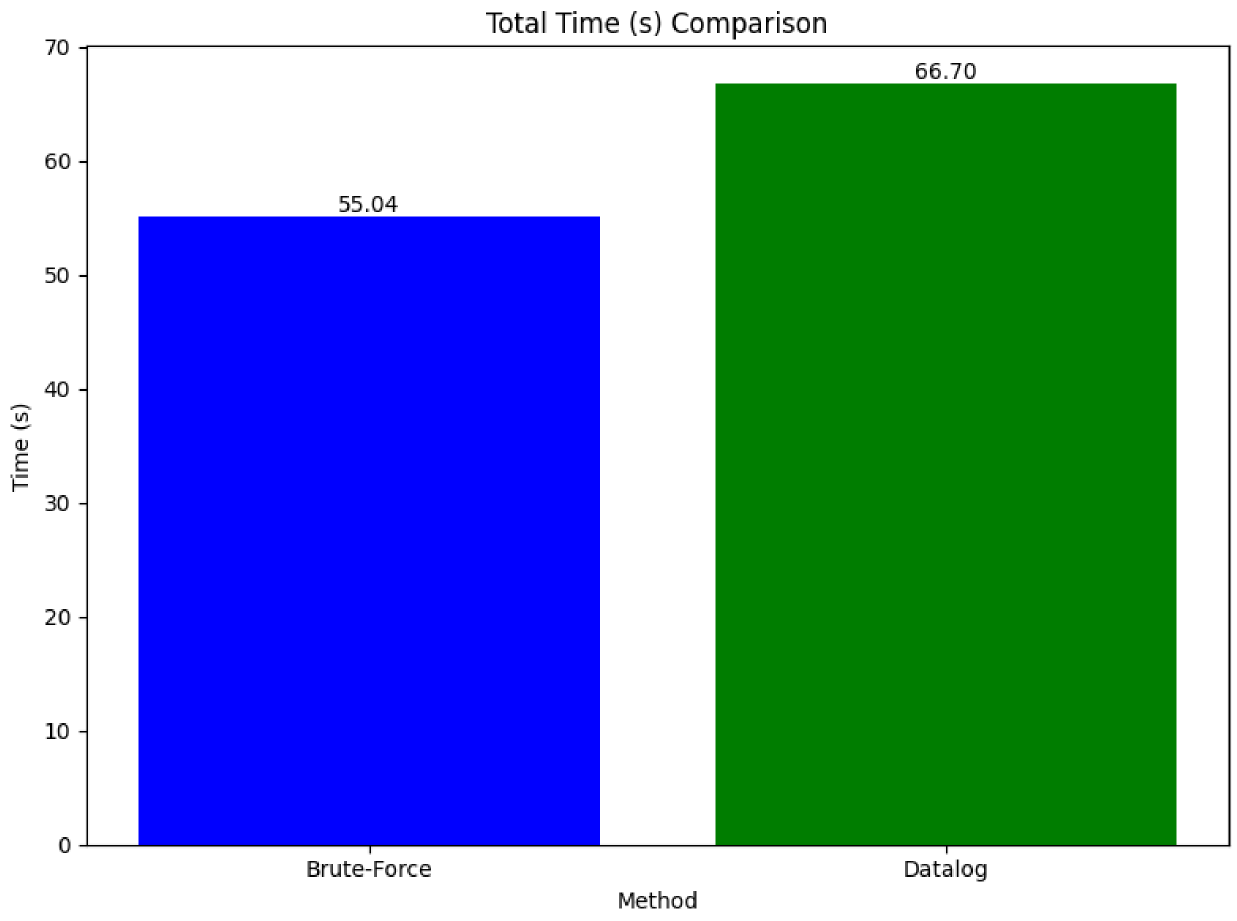
```
PS C:\Users\vikra\OneDrive\Desktop\CSE643-AI\A2> python .\code_2022570.py
Terms initialized: DirectRoute, RouteHasStop, OptimalRoute
Forward Chaining Total Time: 5.48s, Total Memory: 38.02MB, Total Steps: 13727
Backward Chaining Total Time: 5.50s, Total Memory: 46.24MB, Total Steps: 14413
PS C:\Users\vikra\OneDrive\Desktop\CSE643-AI\A2>
```

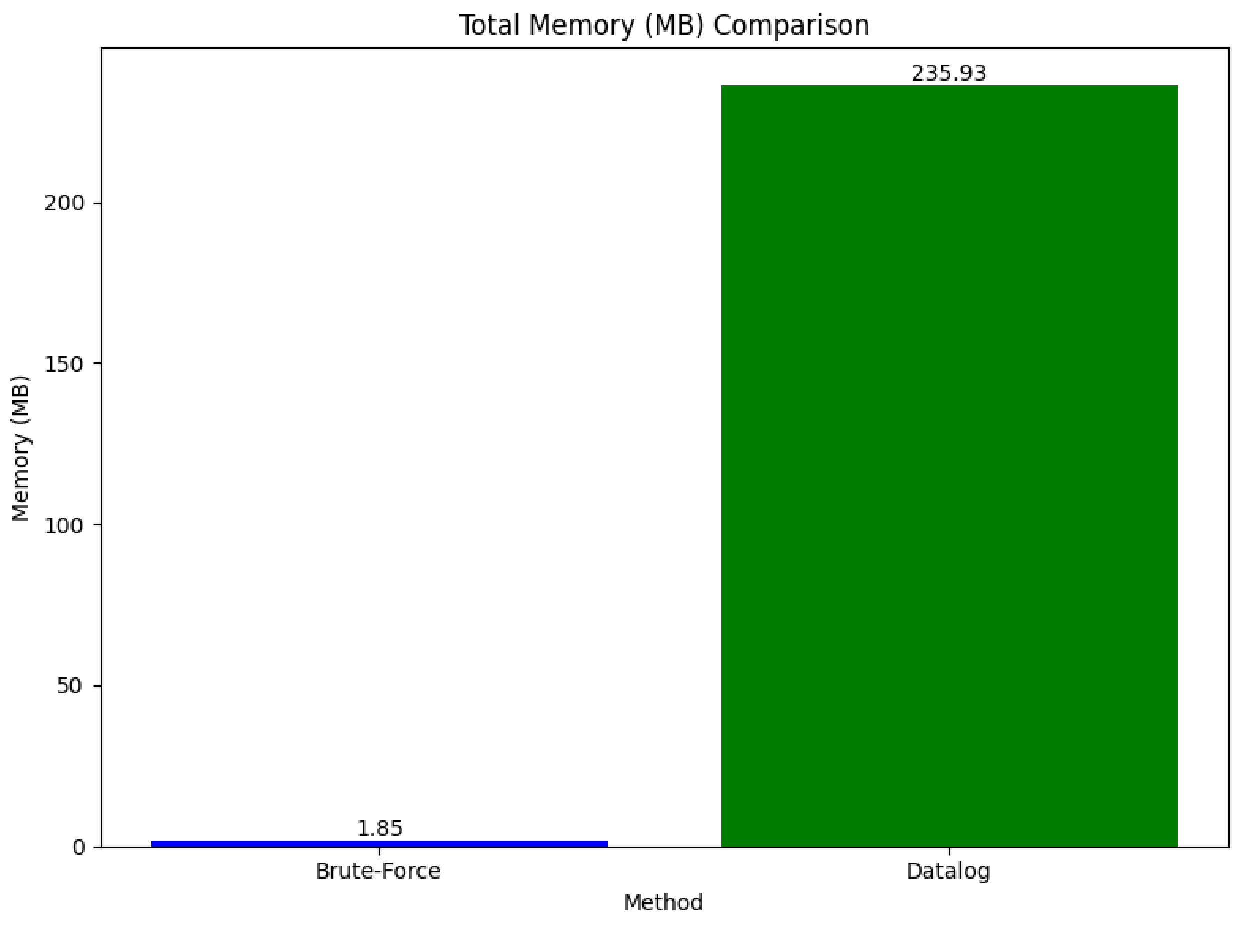
We can see that backward chaining took 5.50 seconds, slightly longer than forward chaining, which completed in 5.48 seconds. The minimal difference in time is likely due to the inherent differences in the two approaches. Forward chaining operates by starting from known facts and applying rules to infer new information progressively, often making it faster as it moves forward through the process without needing to backtrack. In contrast, backward chaining begins at the goal and works backward to verify which facts support it, requiring it to explore different paths and validate conditions at each step. This extra layer of verification in backward chaining might explain the slight increase in time, although both methods are quite efficient when compared to brute-force approaches.

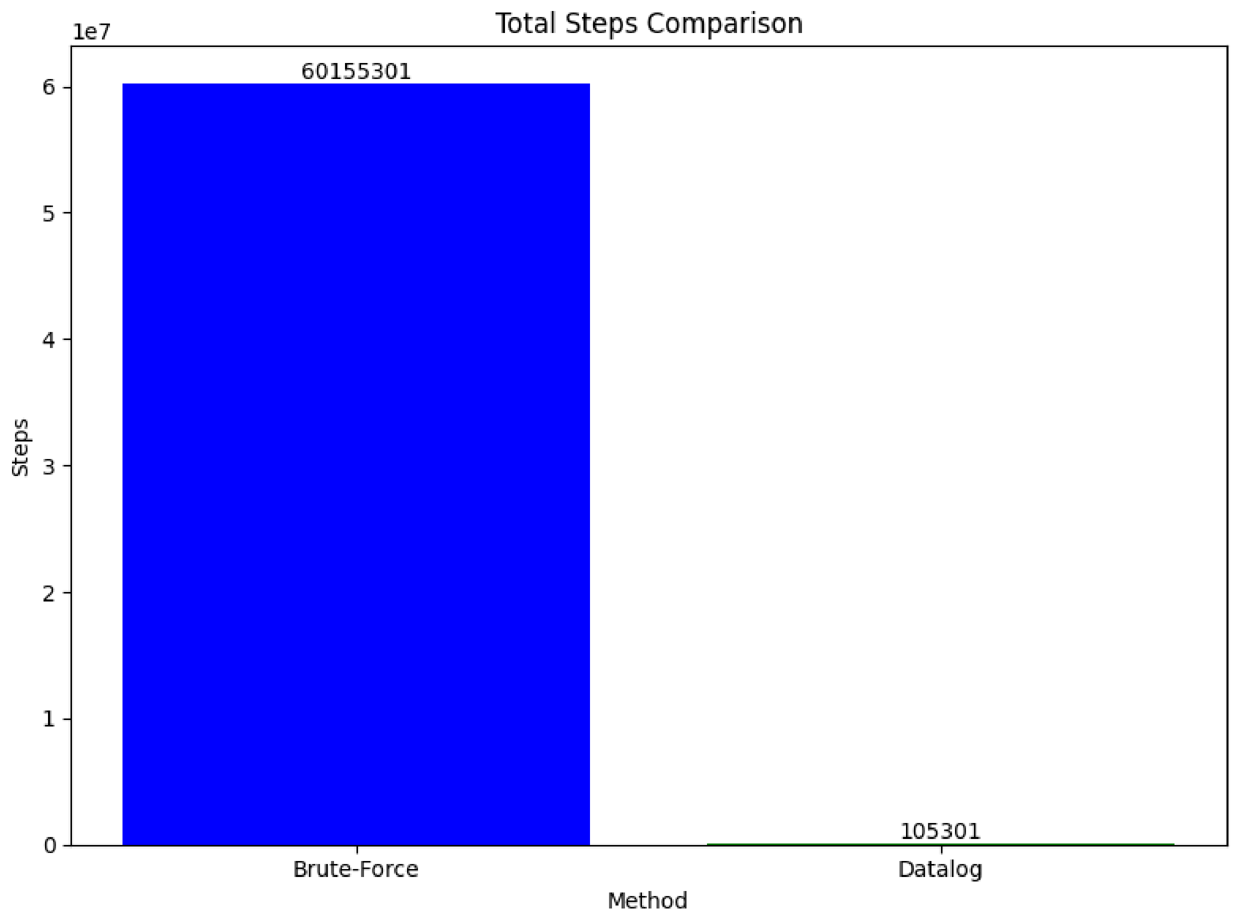
We can see that backward chaining used 46.24 MB of memory, compared to forward chaining's 38.02 MB. The higher memory usage in backward chaining is likely due to its approach of working backward from the goal, which requires storing multiple potential paths and intermediate results as it traces back through the problem. This method may need to retain more information to validate different possible routes and ensure that the goal can be supported by the initial facts. In contrast, forward chaining starts with known facts and applies rules progressively, storing fewer intermediate results, which accounts for its lower memory consumption. Although forward chaining is more memory-efficient, backward chaining's higher memory usage is a tradeoff for its ability to handle more complex rule systems with greater precision.

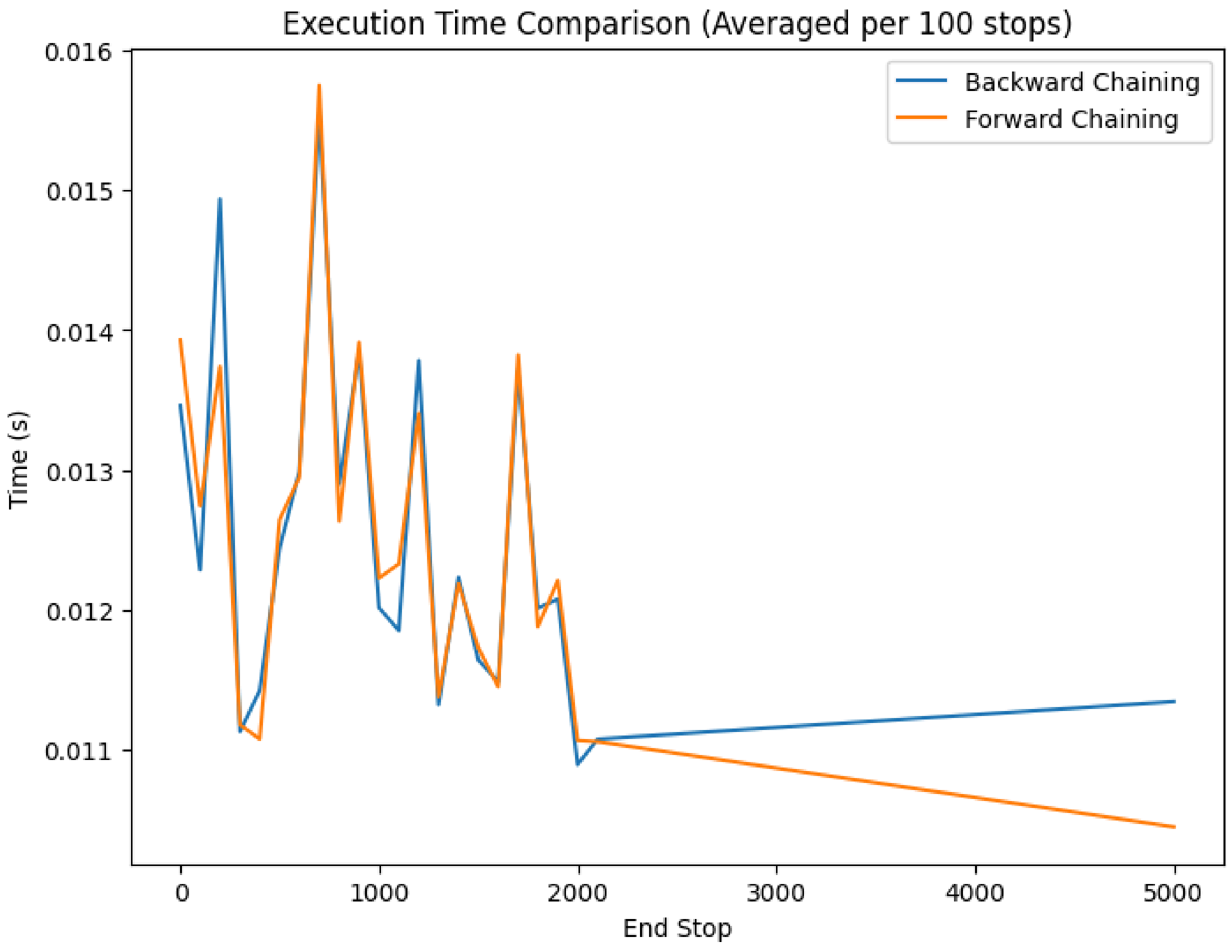

Memory Usage Comparison (Averaged per 100 stops)
Memory (MB)
0.010
0.008
0.006
0.004
0.002
0.000
0
2000
4000
6000
8000
10000
End Stop
Brute-Force
Datalog

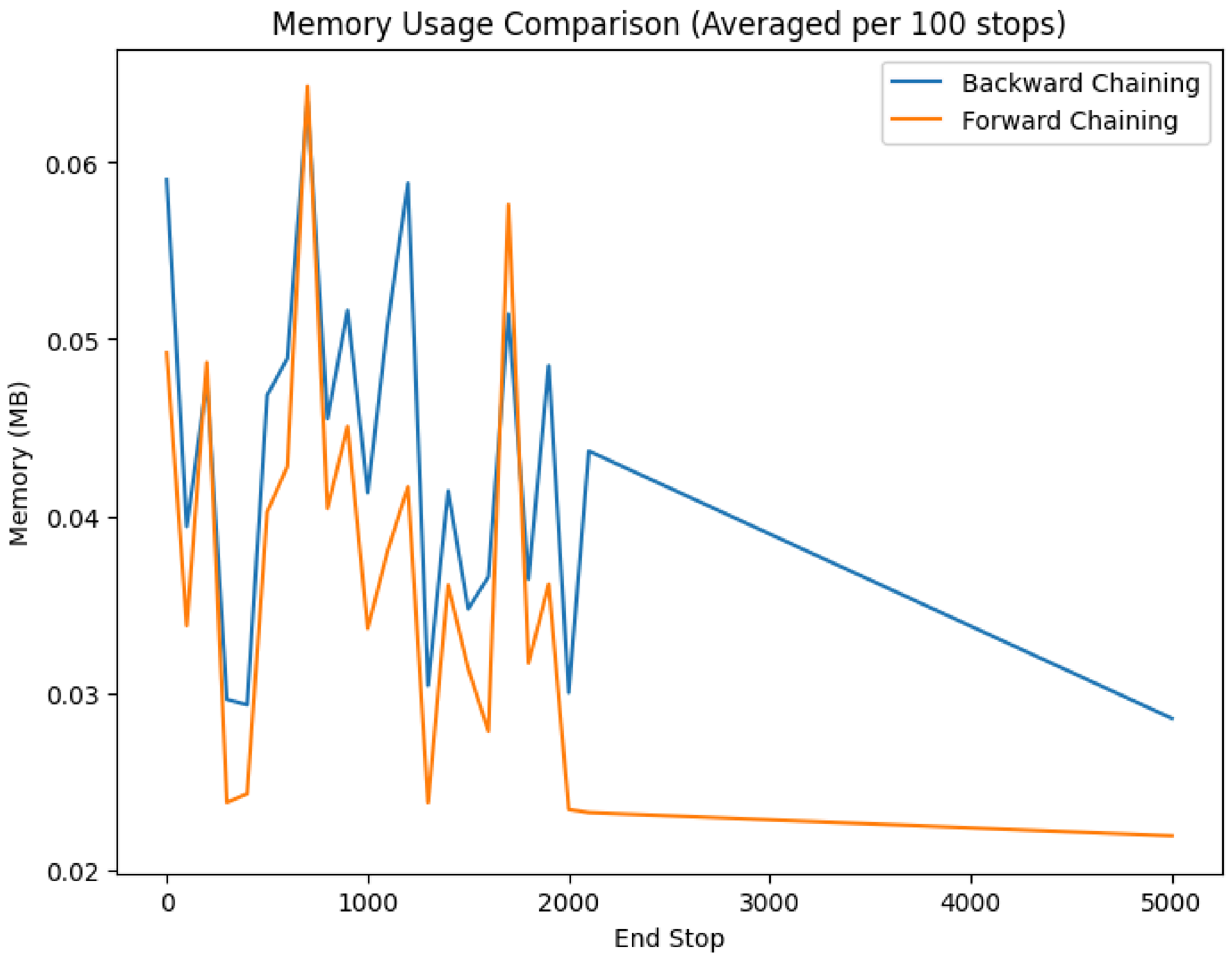

Steps Comparison (Averaged per 100 stops)
2500
2000
1500
1000
500
0
Steps
0
2000
4000
6000
8000
10000
End Stop
Brute-Force
Datalog

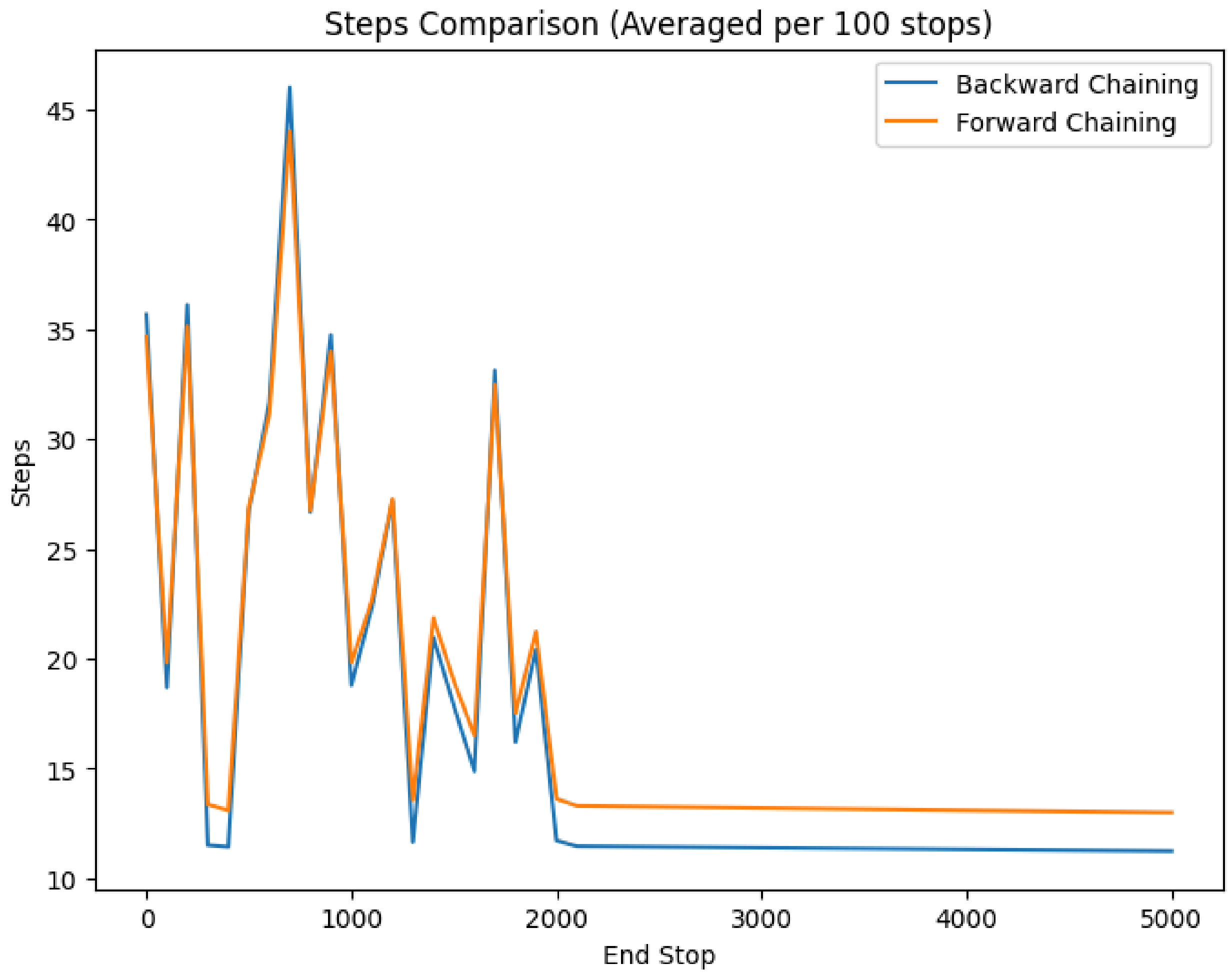

Total Time (s) Comparison
55.04
66.70
Time (s)
Brute-Force
Datalog
Method
0
10
20
30
40
50
60
70

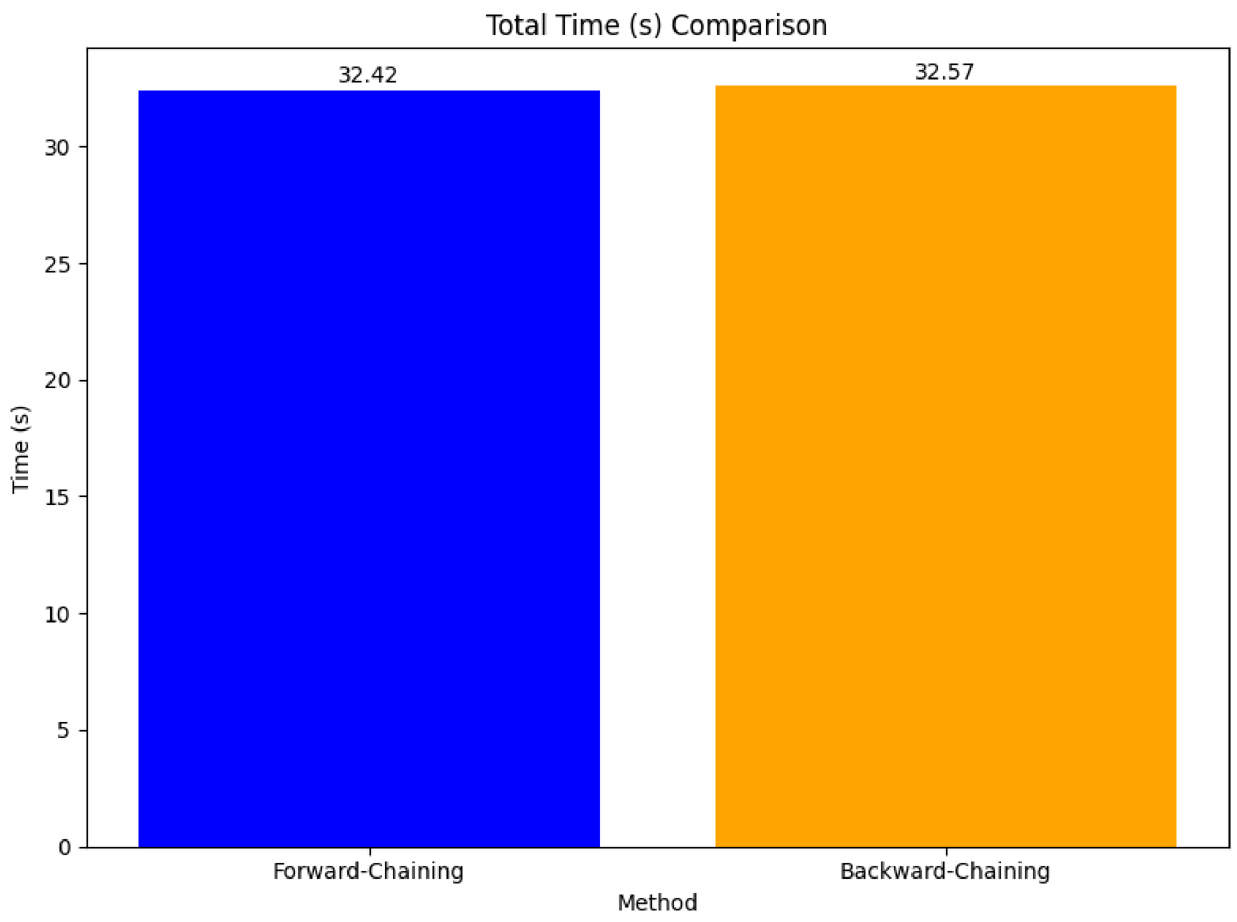

Total Memory (MB) Comparison
235.93
1.85
Memory (MB)
0
50
100
150
200
Brute-Force
Datalog
Method

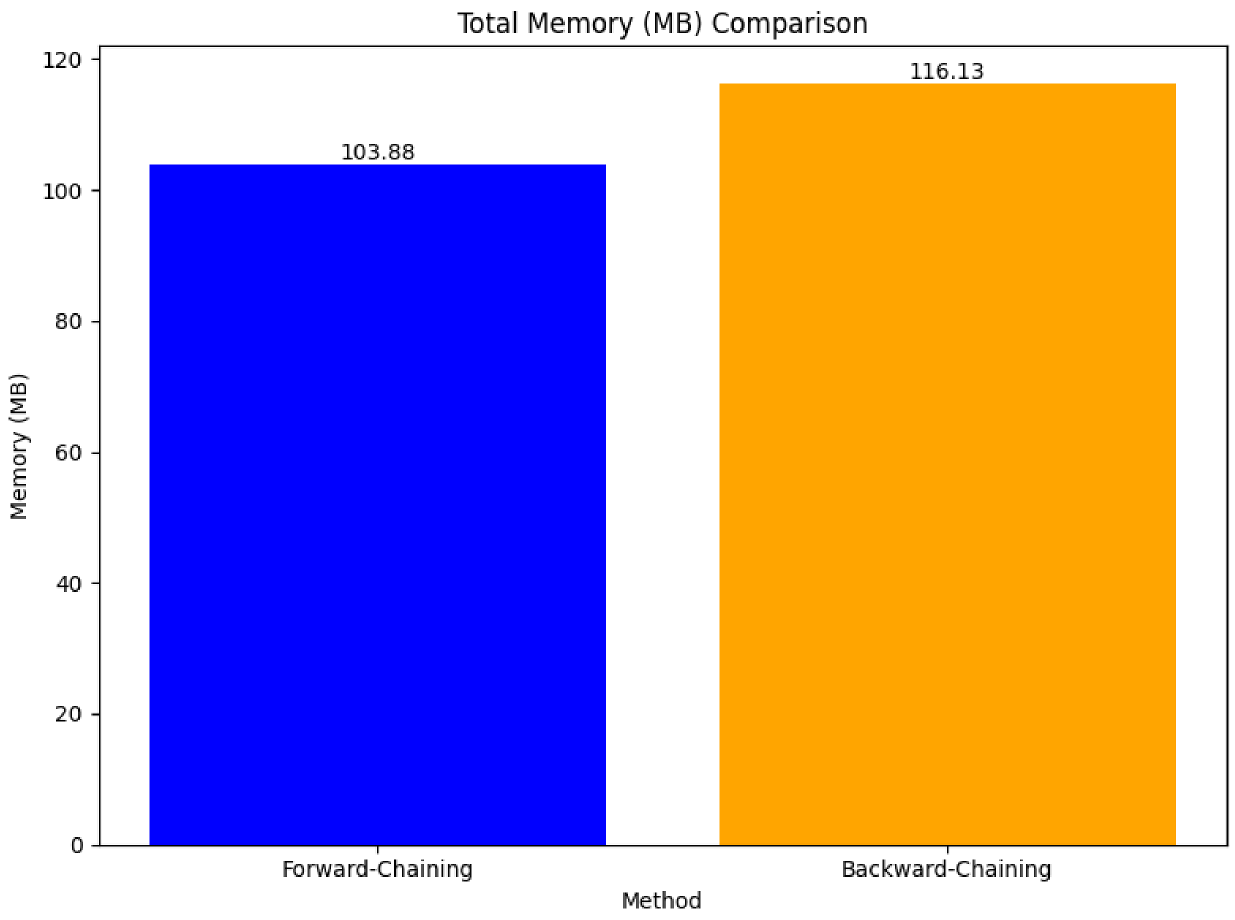

Total Steps Comparison
1e7
6
5
4
3
2
1
0
Steps
60155301
105301
Brute-Force
Datalog
Method

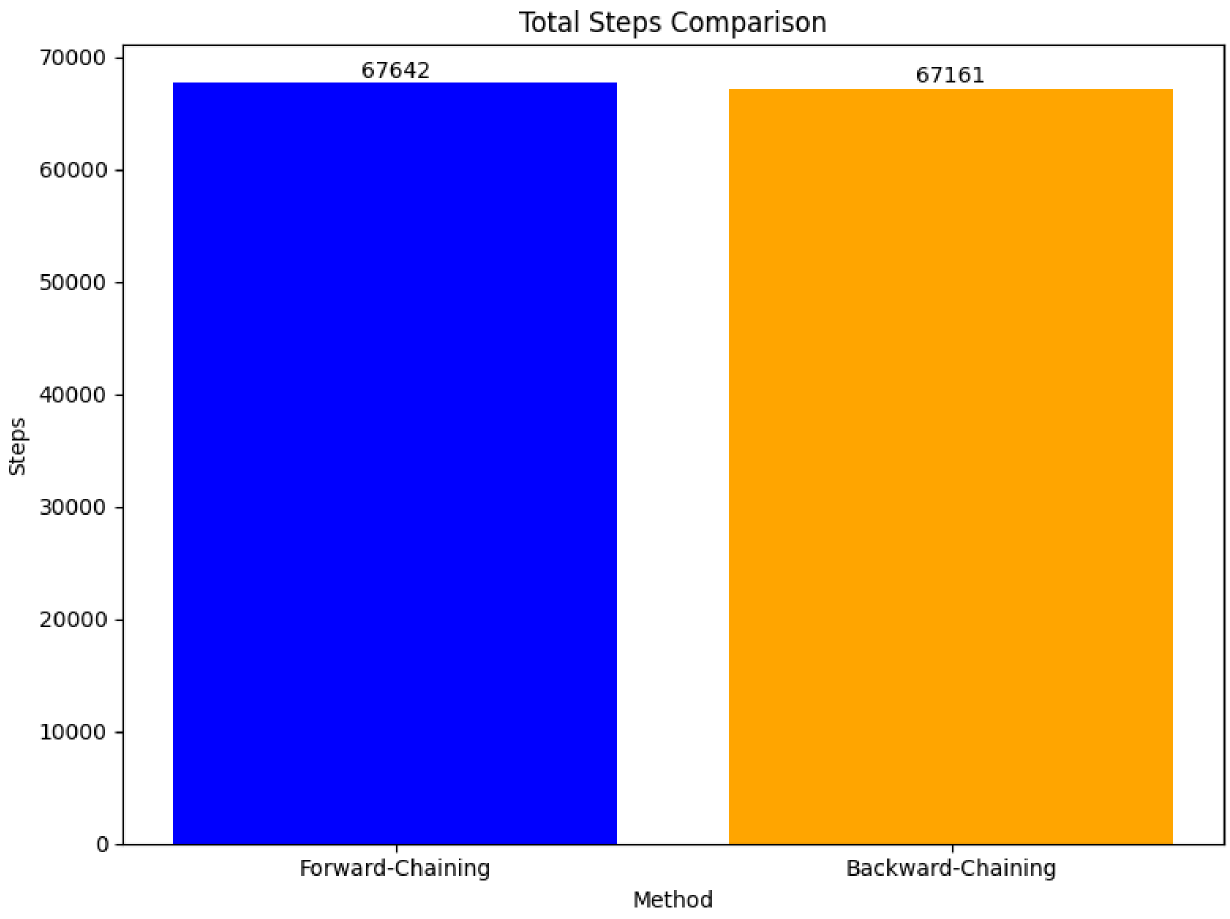

Execution Time Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
Time (s)
End Stop
0.016
0.015
0.014
0.013
0.012
0.011
0
1000
2000
3000
4000
5000

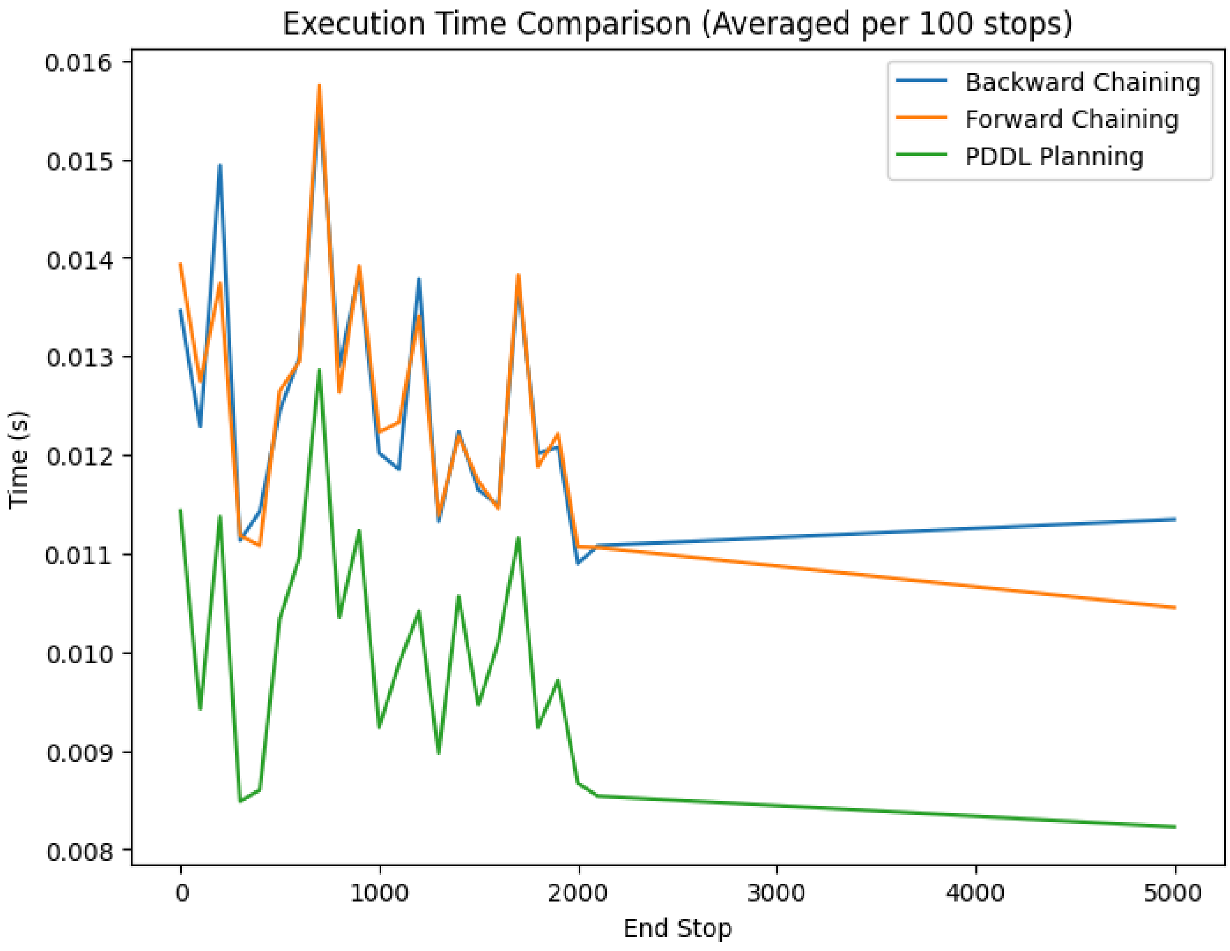

Memory Usage Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
Memory (MB)
0.02
0.03
0.04
0.05
0.06
End Stop
0
1000
2000
3000
4000
5000

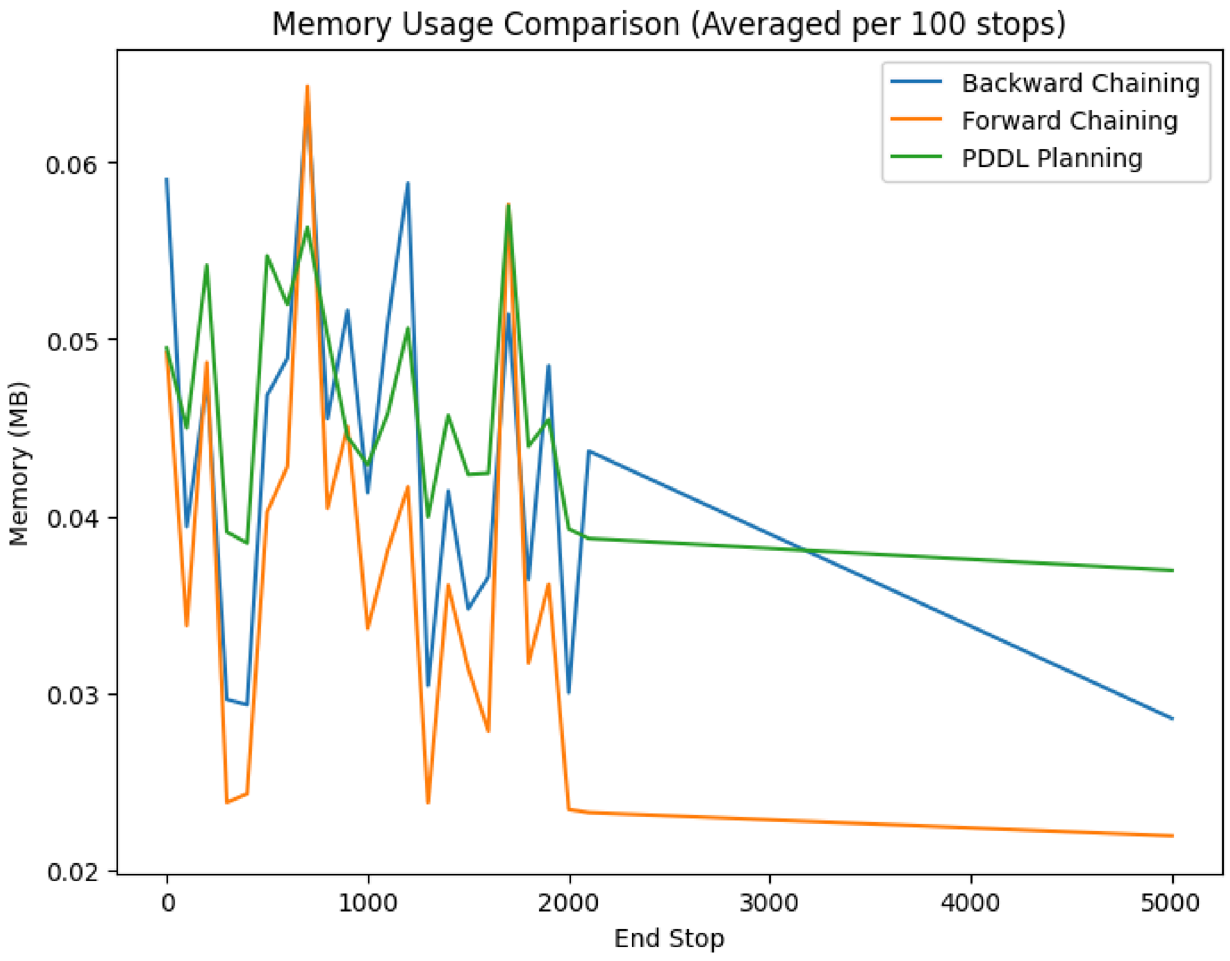

Steps Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
Steps
End Stop
45
40
35
30
25
20
15
10
0
1000
2000
3000
4000
5000

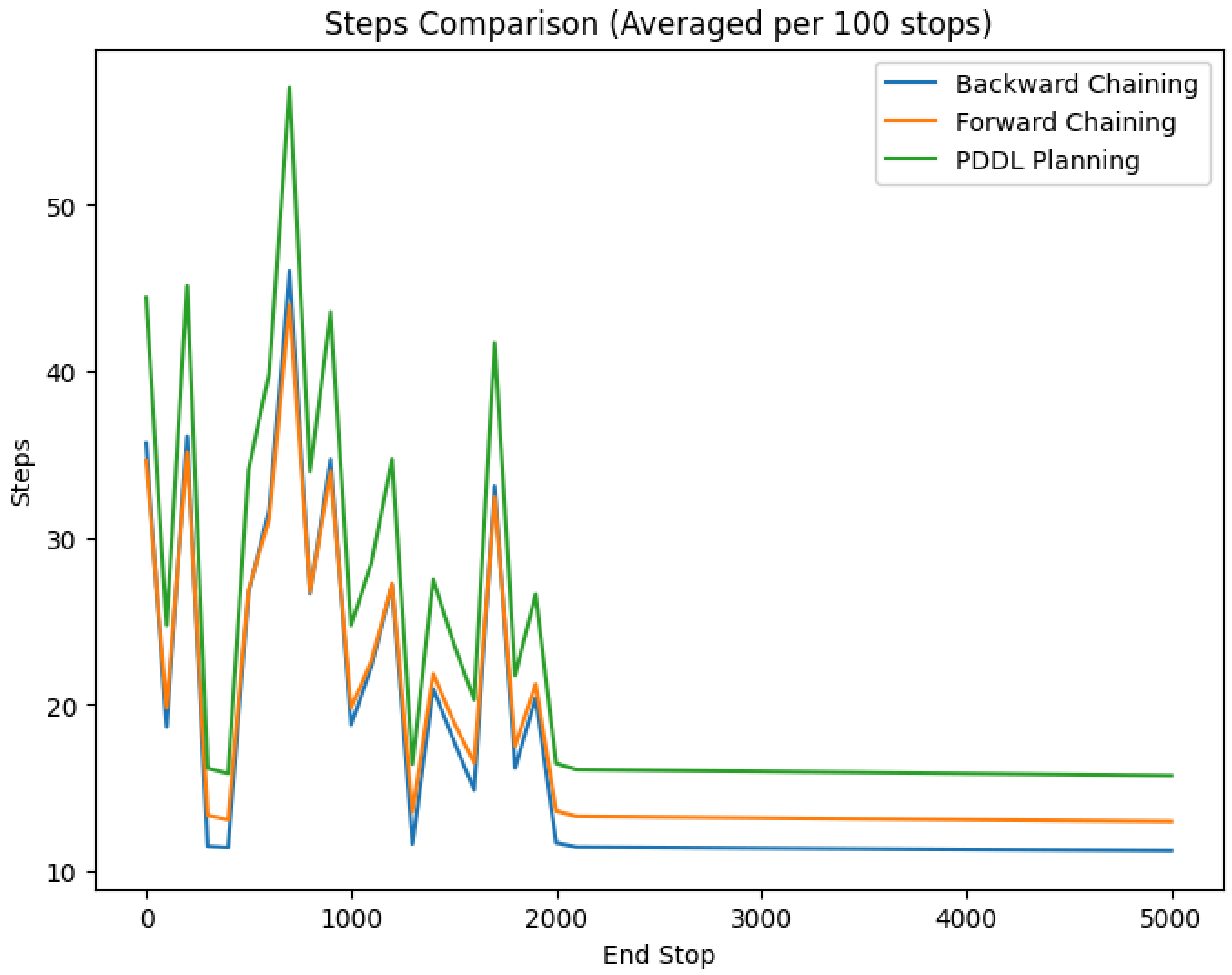

Total Time (s) Comparison
32.42
32.57
Time (s)
0
5
10
15
20
25
30
Forward-Chaining
Backward-Chaining
Method

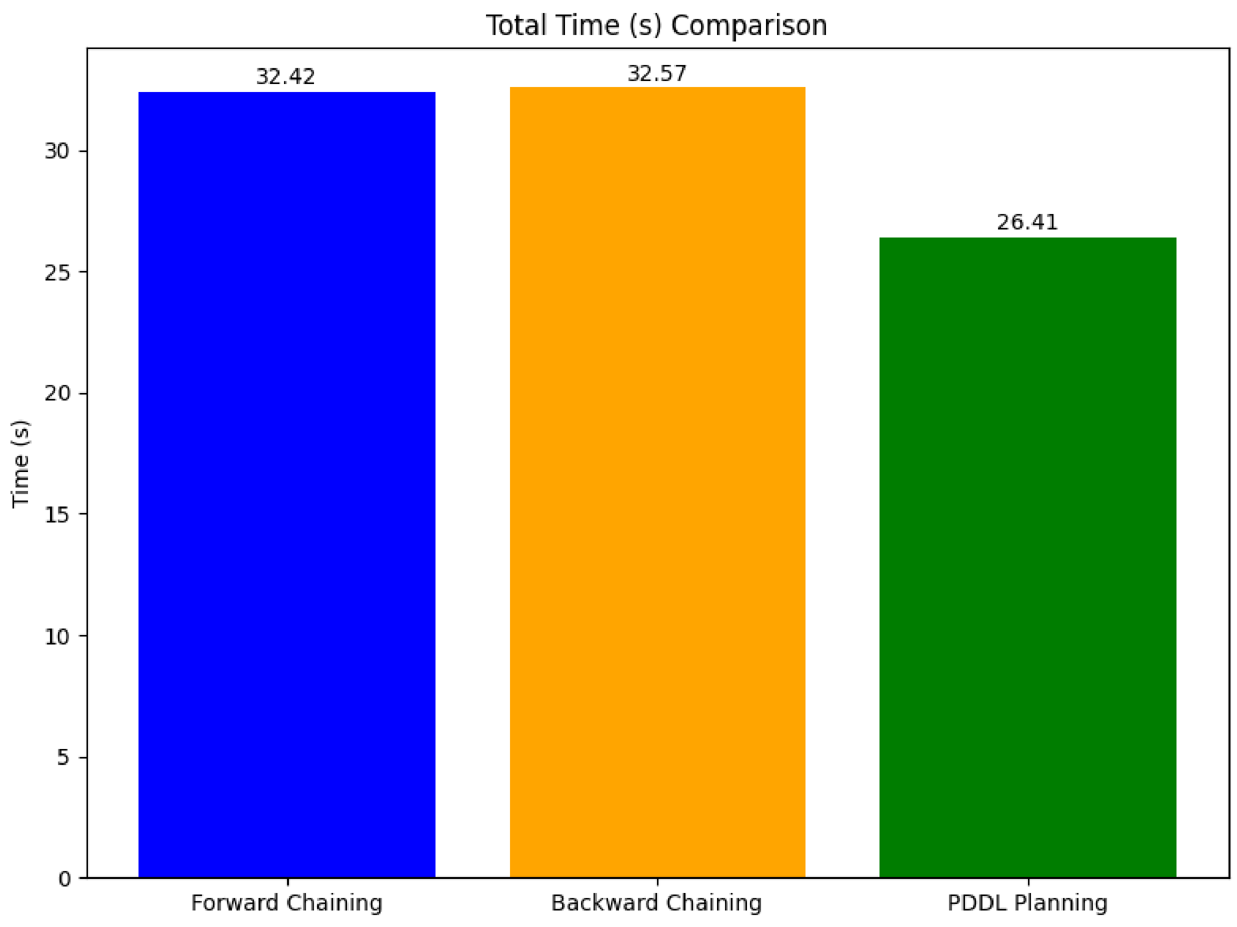

Total Memory (MB) Comparison
Memory (MB)
0
20
40
60
80
100
120
103.88
116.13
Forward-Chaining
Backward-Chaining
Method

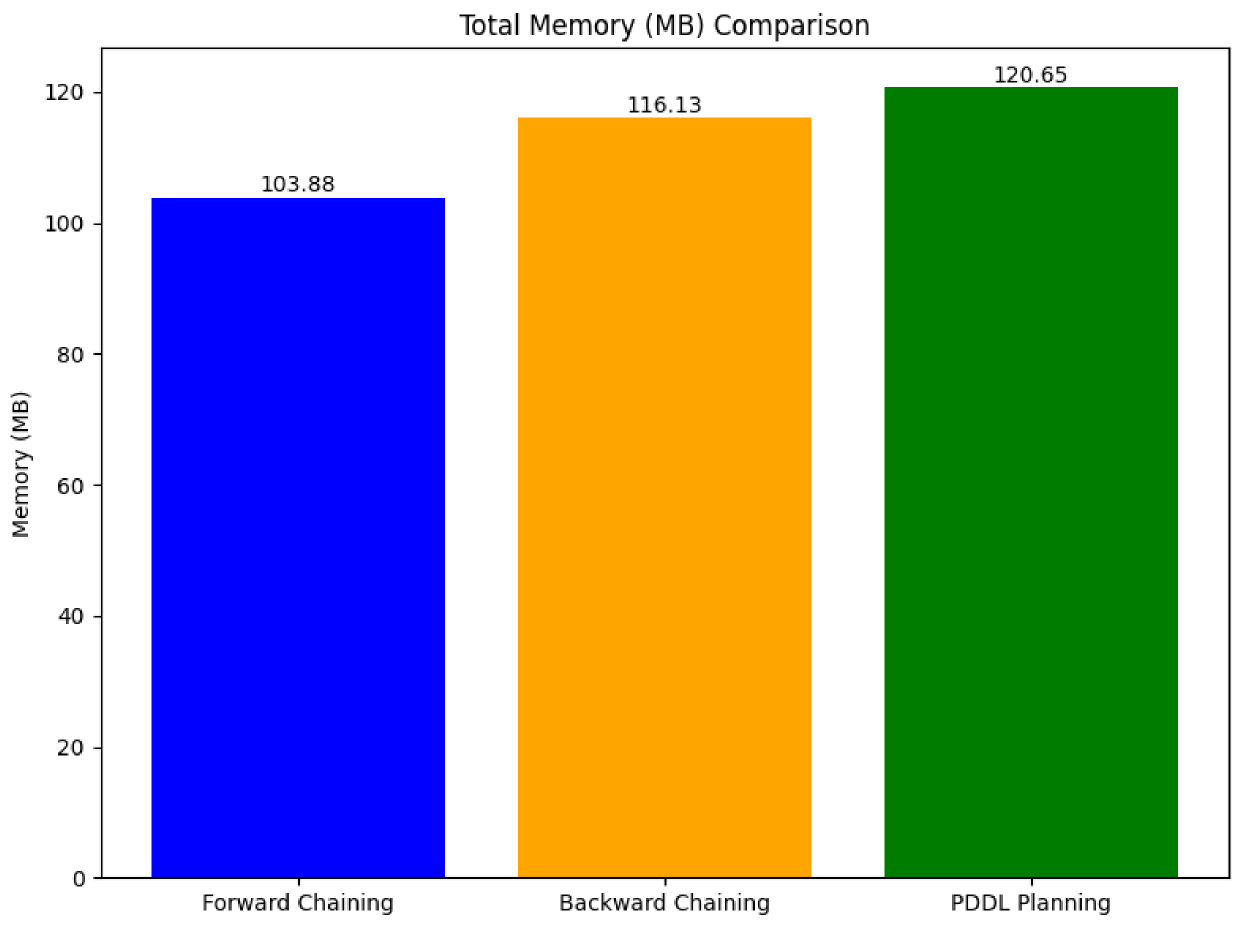

Total Steps Comparison
67642
67161
Steps
70000
60000
50000
40000
30000
20000
10000
0
Forward-Chaining
Backward-Chaining
Method

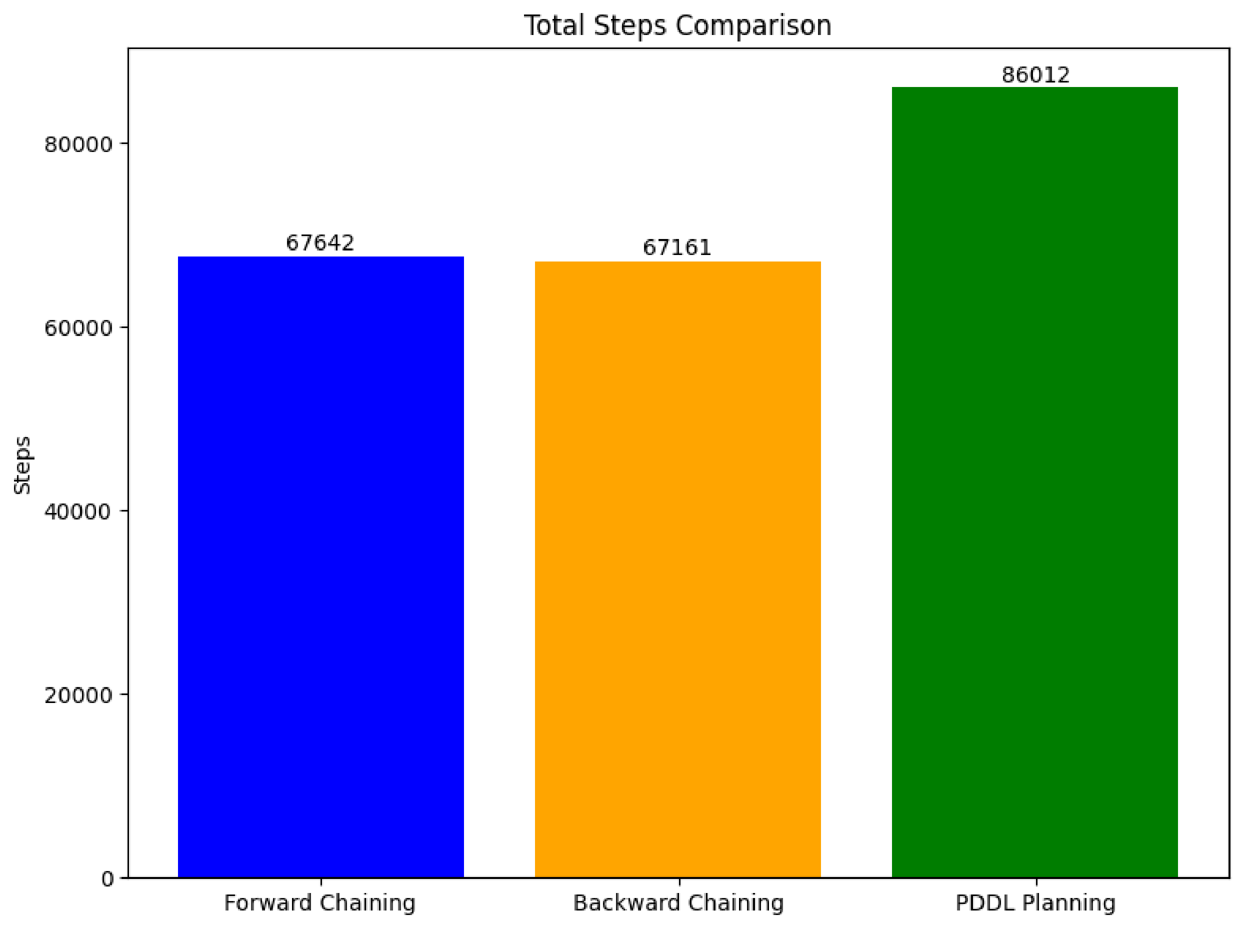

Execution Time Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
PDDL Planning
Time (s)
End Stop
0.016
0.015
0.014
0.013
0.012
0.011
0.010
0.009
0.008
0
1000
2000
3000
4000
5000


Memory Usage Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
PDDL Planning
Memory (MB)
0.02
0.03
0.04
0.05
0.06
End Stop
0
1000
2000
3000
4000
5000


Steps Comparison (Averaged per 100 stops)
Backward Chaining
Forward Chaining
PDDL Planning
Steps
End Stop
10
20
30
40
50
0
1000
2000
3000
4000
5000


Total Time (s) Comparison
Time (s)
32.42
32.57
26.41
0
5
10
15
20
25
30
Forward Chaining
Backward Chaining
PDDL Planning


Total Memory (MB) Comparison
Memory (MB)
0
20
40
60
80
100
120
103.88
116.13
120.65
Forward Chaining
Backward Chaining
PDDL Planning


Total Steps Comparison
Steps
80000
60000
40000
20000
0
67642
67161
86012
Forward Chaining
Backward Chaining
PDDL Planning