Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Assignment no 2: **Logics (and Planning) - FOL Bus Trip Planner**

Name: Robert-Daniel Gavrilă
Group: 30433
Email: danielgavrila2@gmail.com

Teaching Assistant: Marius Stoica
Stoica.Ma.Marius@student.utcluj.ro

Contents

Introduction

1.1 My opinion about this Assignment

- This assignment represented a balance between maintaining the core idea of **Planning in First Order Logic** and building a real-world software system.

- While many academic projects remain purely theoretical, I intentionally aimed to combine **AI planning** with **software engineering**, to build something unique and what could contribute for solving a real problem.

- In an era dominated by AI coding tools, originality and problem formalization remain essential skills. One advantage, is that now is more easier to bring an ideea to reality then ever.

- **First Order Logic (FOL)** is a the bridge between human and machine interpretation, something which can be written and understood by both, and I think that it is a subject which is not covered sufficiently in modern AI tools.

- The idea came naturally, in one day, while I was waiting for the bus after the laboratory: *can my bus routes be planned using logic?*

- After a week of work, the answer proved to be **yes**, within reasonable constraints.

- As a short disclaimer, this application will not run if you do not own a key for the Tranzy's public API. To solve this problem, I will come with a demo video, and if needed I can provide my API key, for live testing the application at laboratory.

- I will provide you also the prover and mace4 files, in order to keep the things transparently in the whole grading process.

Puzzle Formalization

2.1 Puzzle Idea

- Planning a trip in a foreign city is difficult due to route changes and lack of centralized data.
- Many cities lack public APIs or structured datasets.
- Cluj-Napoca provides sufficient public transport data to experiment with logical planning.
- I have used Tranzy's API to fetch data related to the busses and the routes.
- **Goal**: Find an efficient path between two stations with minimal transfers and tickets.
- The solution uses Python to generate FOL inputs for **Prover9** and **Mace4**.

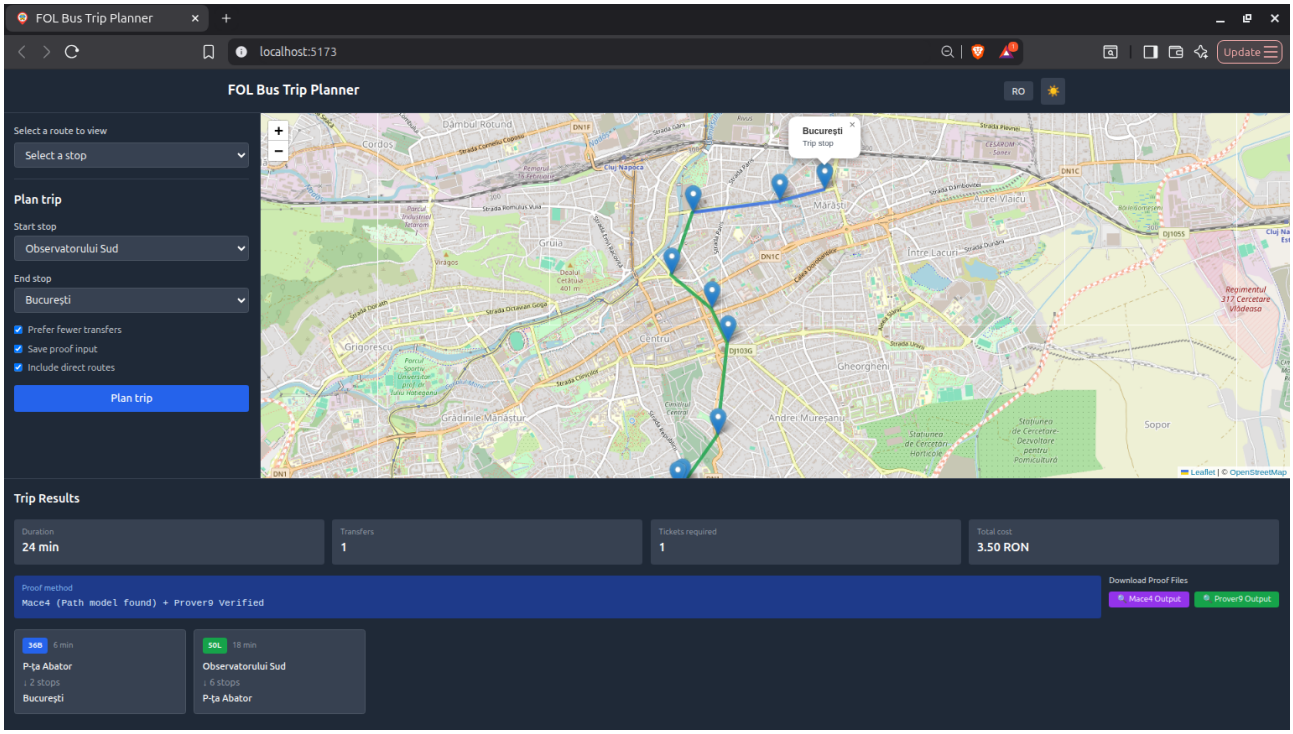


Figure 2.1: Planning Application Interface

2.2 First Order Logic Engine

This component is the logical core of the application. It transforms graph-based transport data into formal logical statements and delegates reasoning to Prover9 and Mace4.

Due to the age and limitations of these tools, several optimizations were required:

- Prover9 and Mace4 struggle with large domains and deep graphs.
- The flag `set(production).` is supported only by Prover9 (2017+), not Mace4.
- Graph pruning using BFS and Dijkstra drastically reduces problem size.
- Node remapping ensures small domains, improving performance and reliability.

2.2.1 Generate FOL Existence (Mace4)

- The function **generate_fol_existence** is responsible for constructing a **First Order Logic** description of existence and consistency, which is later evaluated using **Mace4**.
- Its primary role is not to prove a path, but to verify that the generated facts describing the trip do not lead to contradictions and that all involved stations are reachable within the considered subgraph.

```
def generate_fol_existence(self, path, include_direct_routes=True):
    lines = ["formulas(assumptions)."]
    reachable_nodes = set()

    for seg in path:
        lines.append(f"connected({seg['from']},{seg['to']},r{seg['route']}).
    ")
        reachable_nodes.add(seg["from"])
        reachable_nodes.add(seg["to"])

    if include_direct_routes:
        for i in range(len(path)-1):
            a = path[i]["from"]
            b = path[i+1]["from"]
            c = path[i+1]["to"]
            lines.append(f"connected({a},{b},r_direct).")
            lines.append(f"connected({a},{c},r_direct).")

    for n in reachable_nodes:
        lines.append(f"reachable({n}).")

    lines.append("end_of_list.")
    return "\n".join(lines)
```

This function constructs a **consistency model**. Mace4 does not prove goals; instead, it checks whether the asserted facts can coexist in a finite model.

```
formulas(assumptions).

connected(11,9,r29).
connected(9,10,r29).
connected(10,6,r29).
connected(6,7,r29).
connected(7,4,r29).
connected(4,5,r29).
connected(5,8,r29).
connected(8,2,r4).
connected(2,12,r4).
...
connected(8,2,r_direct).
connected(8,12,r_direct).
connected(2,12,r_direct).
connected(2,1,r_direct).
connected(12,1,r_direct).
connected(12,3,r_direct).
connected(1,3,r_direct).
connected(1,0,r_direct).

reachable(8).
reachable(1).
reachable(6).
reachable(12).
reachable(7).
reachable(4).
reachable(2).
reachable(9).
reachable(5).
reachable(10).
reachable(11).
reachable(0).
reachable(3).
reachable(0).

end_of_list.
```

Interpretation:

- The generated Mace4 input consists exclusively of ground facts, grouped under **formulas(assumptions).**
- The logic shows us:
  - Existence of connections between stations
  - Existence of reachability facts for all stations involved
  - No negations, no implications, no goals
- If Mace4 **finds at least one model**, the set of facts is internally consistent and admissible for further reasoning. If not, the path is rejected before reaching the deductive phase.

2.2.2 Generate FOL Verification (Prover9)

- The function **generate_fol_verification** constructs the deductive core of the system and produces the input file for **Prover9**.
- The difference between **Mace4** and **Prover9** is that the **Prover** is used to prove that a sequence of steps logically leads from the start station to the destination, using formal inference rules.
- The key idea is to model the trip as a discrete sequence of steps:
  - **step(N, X)** means that at step **N**, the traveler is at station **X**
  - **succ(N, M)** defines the successor relation between steps
  - **uses(N, R)** specifies which route is used at each step
- The axiom used resumes the rule which is used by the planner:

  $\forall(N, M, Y, R)\ (step(N, X) \ \wedge \ succ(N, M) \ \wedge \ uses(M, R) \ \wedge \ connected(X, Y, R)) \rightarrow step(M, Y)$
- The goal clause asserts that the final step reaches the destination station: **step(k, destination)**

```python
def generate_fol_verification(self, path):
    lines = ["set(production)."]
    lines.append("formulas(assumptions).")
    lines.append("assign(max_weight, 30).")
    lines.append("assign(max_proofs, 1).")
    lines.append("assign(max_seconds, 30).")

    for seg in path:
        lines.append(f"connected({seg['from']},{seg['to']},r{seg['route']}).
    ")

    for i in range(len(path)):
        lines.append(f"succ({i},{i+1}).")

    lines.append(f"step(0,{path[0]['from']}).")

    for i, seg in enumerate(path):
        lines.append(f"uses({i+1},r{seg['route']}).")

    lines.append(
        "all N all M all X all Y all R "
        "(step(N,X) & succ(N,M) & uses(M,R) & connected(X,Y,R) -> step(M,Y))
    ."
    )

    lines.append("end_of_list.")
    lines.append("formulas(goals).")
    lines.append(f"step({len(path)},{path[-1]['to']}).")
    lines.append("end_of_list.")
    return "\n".join(lines)
```

This encoding performs **deductive planning**. The axiom represents forward chaining over steps.

```
set(production).
formulas(assumptions).

assign(max_weight,13).
assign(max_proofs,1).
assign(max_seconds,13).
assign(sos_limit,26).

connected(25,23,r29).
connected(23,24,r29).
connected(24,20,r29).
connected(20,21,r29).
connected(21,18,r29).
connected(18,19,r29).
connected(19,22,r29).
connected(22,16,r4).
connected(16,27,r4).
connected(27,15,r4).
connected(15,17,r4).
connected(17,14,r4).

succ(0,1).
succ(1,2).
succ(2,3).
succ(3,4).
succ(4,5).
succ(5,6).
succ(6,7).
succ(7,8).
succ(8,9).
succ(9,10).
succ(10,11).
succ(11,12).

step(0,25).

uses(1,r29).
uses(2,r29).
uses(3,r29).
uses(4,r29).
uses(5,r29).
uses(6,r29).
uses(7,r29).
uses(8,r4).
uses(9,r4).
uses(10,r4).
uses(11,r4).
uses(12,r4).

(all N all M all X all Y all R (step(N,X) & succ(N,M) & uses(M,R) & connected(X,Y,R) -> step(M,Y))).

end_of_list.

formulas(goals).
step(12,14).
end_of_list.
```

- This is an input generated by the script according to our rules.
- It consists of some flags which are increasing the performance of the prover, in order to assure the consistency of our application.
- That's why we are limiting only at one proof with **assign(max_proofs,1)** or limiting the weight of our proof with **assign(max_weight, 13)**.
- Without these settings, the Prover9 didn't worked at all with heavy tasks, like a route with multiple changes and many routes.
- I have tried over 10 configurations for Prover9, and this was the only that worked consistently. You can check in the commits on github, the previous versions without these flags.

2.3 Tools

2.3.1 Prover9

Prover9 is used as a **first-order automated theorem prover** whose role in this project is deductive verification. Formally, the planner checks:

$$\exists \langle s_0, s_1, \ldots, s_n \rangle : step(0, s_0) \wedge \ldots \wedge step(n, s_n)$$

The final proof confirms that the planned path is valid under the axioms.

- From a logical perspective, **Prover9** answers the question:
  - *Given these assumptions, does the destination necessarily follow?*
- The planning problem is reduced to a logical problem. If Prover9 succeeds, the path is not just found heuristically, but **logically justified** step by step, which guarantees the existance of it.
- The explicit use of quantifiers, implications, and resolution makes this phase fundamentally different from graph search. While BFS finds a path, Prover9 proves that the path must reach the destination under the given rules, which is our main goal obviously.

2.3.2 Mace4

Mace4 is a **finite model finder**. It does not prove reachability but verifies that no contradictions exist in the route facts.

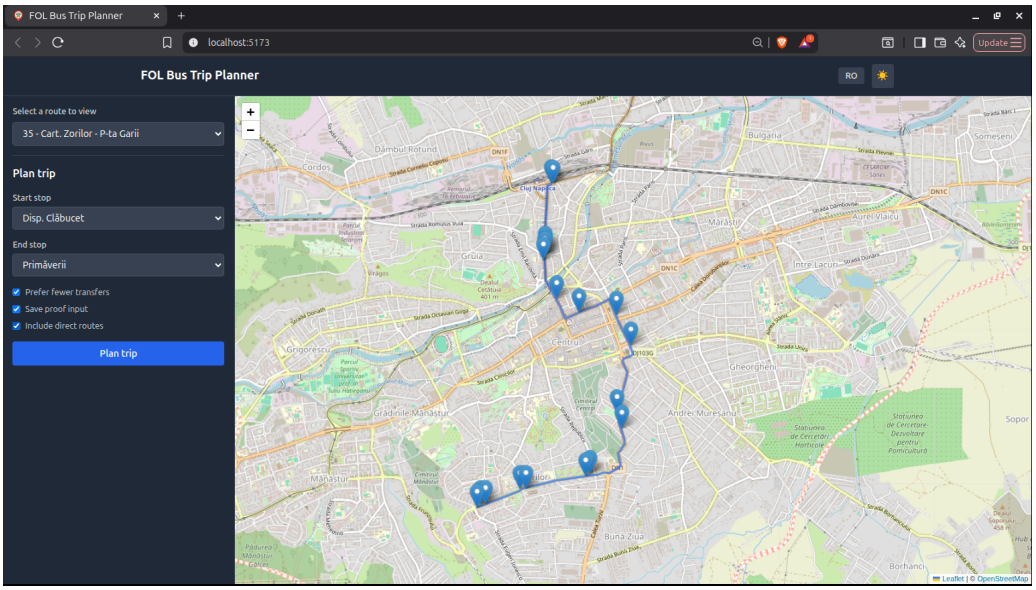
- Instead of proving goals, **Mace4** answers the question:
  - *Is there at least one finite interpretation in which all given facts are true?*
- In my project, **Mace4** is used to validate that the reduced subgraph and reachability assertions do not contradict each other.
- Prevents Prover9 from running on invalid inputs

If a model is found, the assumptions are satisfiable.

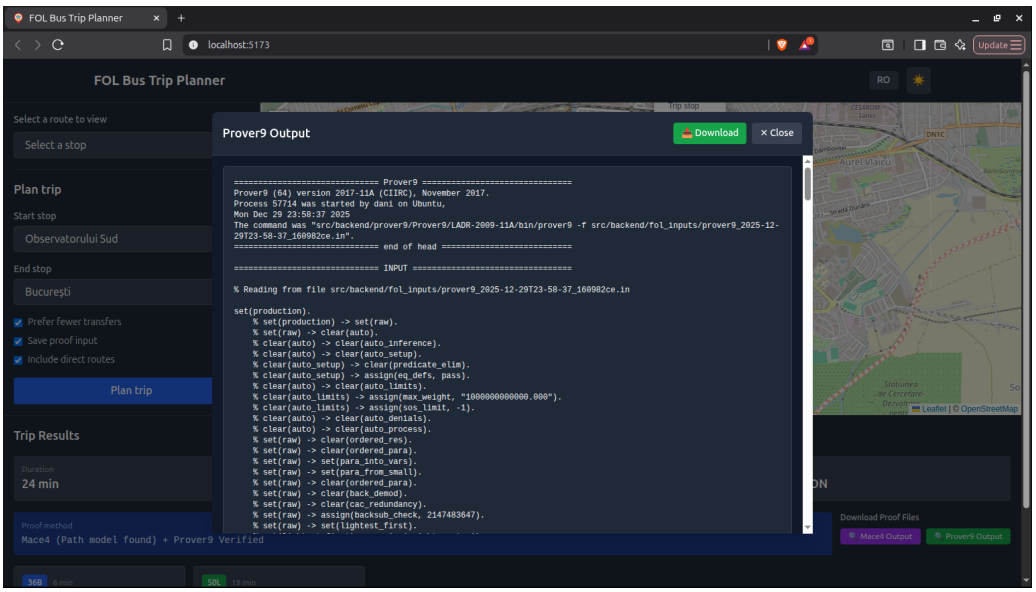
Application

3.1 Features and Workflow

- The **prooving core** is embedded in a full-stack web application, built using **Python (FastApi)** and **TypeScript (React)**, designed using best-practices, while keeping the main ideea in front.
- The **User Interface** consists of a dashboard, containing two main features:
  1. **See your bus route**
     - In this panel, we can **choose the route**, and it will appear on map, among its stops, where if we click on them we can see the stop name.
     - These information are projected on the city map.

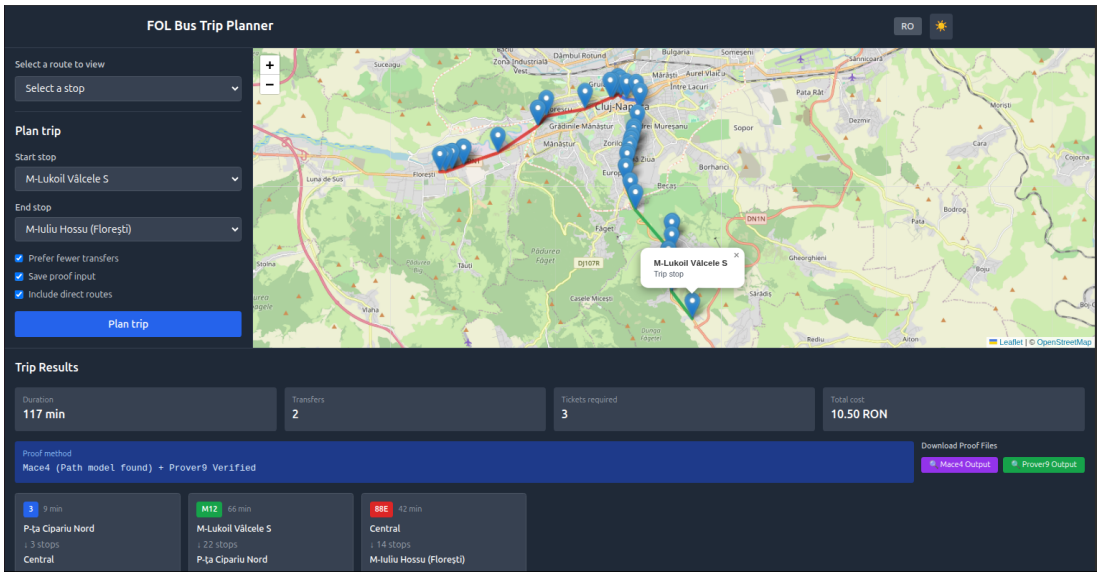


  2. **Plan a trip**
     - We have to choose the starting and the ending stops, from the related panel.
     - Then, we have some options, regarding the output:
       (a) **Prefer fewer transfers**: The output will consist of trips with minimum number of changes.
       (b) **Save proof input**: The files containing the proof, will be saved. We can check and download those files within the application dashboard.
       (c) **Include direct routes**: To avoid the complexity, we can choose to remove the proves for direct routes, as they connects directly the stations **(Knoledge base)**.



- We can get more details about the trip in the backpanel, where we get information about:

  1. **Duration** (min): the estimated time of the trip (we have estimated a time of 3 minutes between stops).
  2. **Number of transfers**: how many busses do we need to change.
  3. **Number of tickets**: We have a whole engine for calculating the number of required tickets. As we know, in Cluj-Napoca, one ticket costs **3.5 RON** and it is available for **45 min.**
  4. **Information about the routes**: We can see which route do we need to take, from the starting to the ending/intermediate stop, with the estimated ride time.



- Another features are represented by **Dark Mode** and **multiple language support** (Romanian and English).

3.2 Activity Diagram

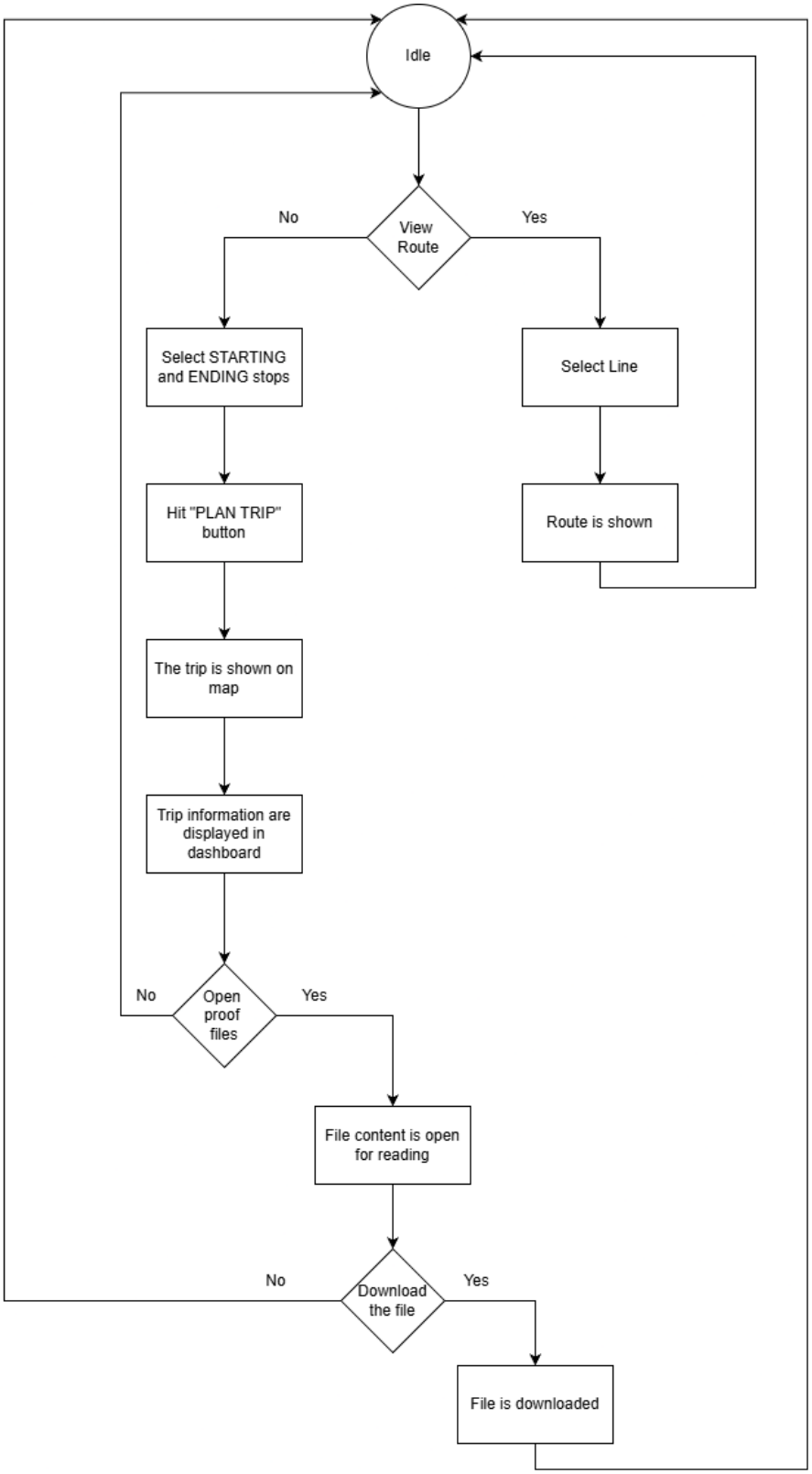


Figure 3.1: Activity diagram, showcasing the forkflow of the application

3.3 Study Case - How to use the app

- Before runnign anything, we have to install the application. It can be done easily, by following the guide published on my github page, at this repository: **FOL-Bus-Trip-Planner.**
- To run your application on your machine, you have to request an API key from **Tranzy**, which you will need to put in your **.env** file.
- If you want only to run the Prover9 and Mace4 files, you will need to use a slightly modified Prover9 version, because the original LADR-2009 was not working, so I switched to LADR-2017, which worked on my machine. It can be installed from this github repository: **Prover9 - Github**.

We choose randomly to analyze a trip from **Observatorului Nord** to **Cluj Arena**, which I done frequently, and because we don't have a direct connection, it requires transfers.

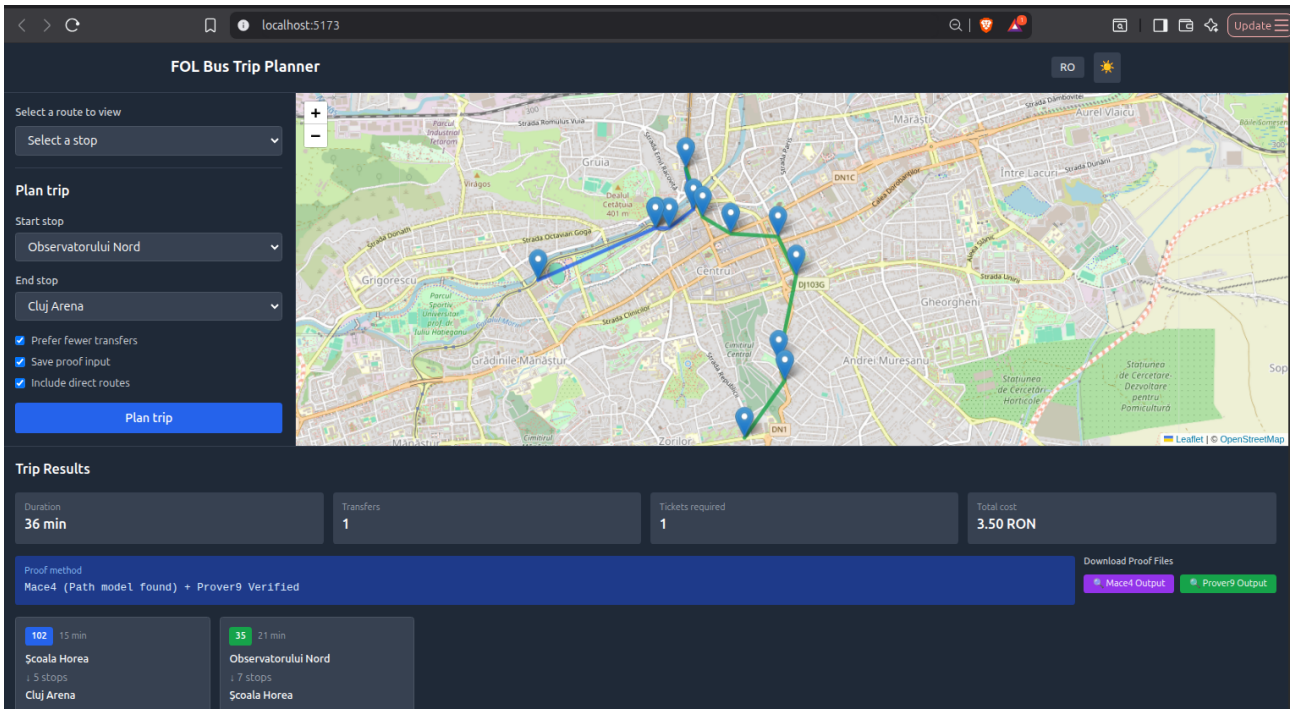


Figure 3.2: Screenshot with the planned trip

As we can see, the planner tripped for us a route. Now with this result, we can compare with Google Maps, to check if our result is consistent or not, at the first appearance.

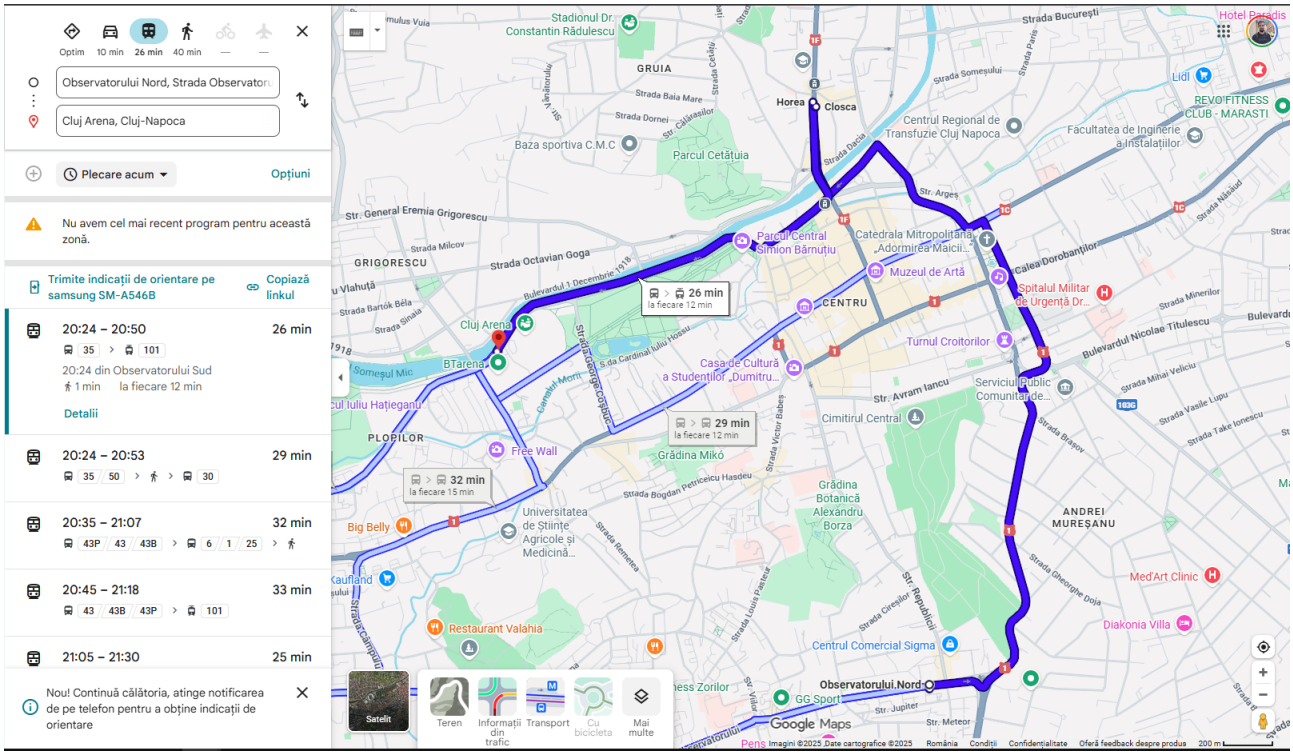


Figure 3.3: Screenshot with trip planned by Google Maps

So, as we can see, our planner done a pretty decent job, and it is comparable with the Google Maps output. The second step is to analyze the output files. We can download the files by hitting the **download** button after we have opened the file.

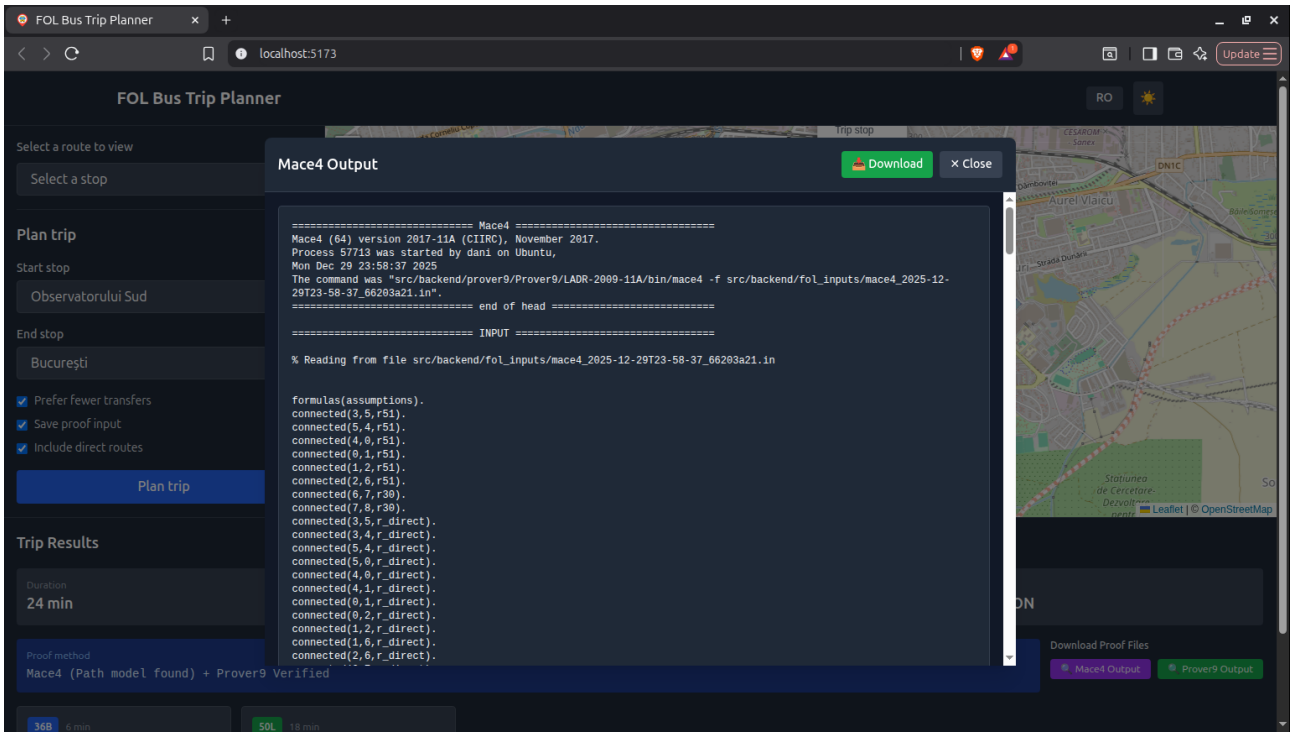


Figure 3.4: Screenshot with the opened result file for Mace4

3.4 Interpreting the results

- After downloading the files, we will get one input and one output file.
- To run the file we have to perform the following steps:
  - Locate the **prover9/mace4** binary file, which is usually found in **bin**.
  - Navigate to that directory, with the file, and run the following command:
  - **./prover9 -f <input_file>**
  - **./mace4 -f <input_file>**
- **Interpretation:**
  1. **Prover9**
     - If the prover succeeded then, we will see at the end of the generated file/response the following message: **THEOREM PROVED**.
     - The same file contains also the Proof which is performed step-by-step, the statistics and also the input.
     - Usually if already went done, then the theorem is proved, and it means that everything worked fine. Else, somewhere was a problem, and we need to rewrite the input file.
  2. **Mace4**
     - If Mace4 succeeded then, we will see at the end of the generated file/response the following message: **Exiting with x models**.
     - The same file contains also the steps and the generated models.
     - A model consists of the domain size, which is an integer, the interpretation and the final relation which is represented as a matrix.
- Due to the limitations and the size of the files, I will not paste them here, but I will attach some examples, in the archive.
- The size of my files is large, because the problem was a really challenge to formalize and to integrate successfully with my app.
- Probably a more powerful theorem prover, like **Z3**, would perform better on large knowledge bases with complex rules.

Bibliography

- **Tranzy.ai** - public transport API
- **ai4reason** - Prover9
- **Prover9 Documentation**
- A.Groza, Modelling Puzzles in First Order Logic
- J. Grundy, T. Melham, S. Krstic, S. McLaughlin, Tool Building Requirements for an API to First-Order Solvers
- N. Shalit, M. Fire, D. Kagan, E. Ben-Elia, Short Run Transit Route Planning Decision Support System Using a Deep Learning-Based Weighted Graph
- S. Kartoirono, I. Riadi, F. Furizal, A. Azhari, Improved Breadth First Search For Public Transit Line Search Optimization