GRP 12: Toronto Daytrip Planner

Darwin C (20267164)

Jasper N (20285349)

Sijing Z (20243635)

Yu Xuan L (20288359)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 5, 2022

Contents

1 Abstract

Using the data set found on transitfeeds.com[1], our project evaluates and provides the most optimal ways one can get from one point to another within Toronto, using just the public transport system. Our project takes into consideration many different variables including transfers, budget, time constraints, etc. As the user you will input your starting location, desired ending location, current time, desired arrival time, travel budget, age, extra visiting locations (eg. Toronto Zoo), and other miscellaneous options. Using this information, our program will show you a list of trains, buses, and streetcars you need to take at which times in order to reach your location within your budget and time constraints.

2 Propositions

2.1 Budget Propositions:

- **Adult** = true if the user does not fall into the child or seniors
- **Kid** = true if user falls into the kids age category (≤ 12)
- **Youth** = true if the user falls into the youth category (13-19)
- **Senior** = true if the user falls into the senior category (> 65)
- **Presto** = true if the user has an active PRESTO account with sufficient balance to complete the trip with
- **Presto_Adult** = true if the user holds a PRESTO card and is an adult
- **Presto_Other** = true if the user holds a PRESTO card and not an adult (other age groups excluding kids)
- **Normal_Adult** = true if the user is not a PRESTO card holder and an adult
- **Normal_Other** = true if the user is not a PRESTO card holder and not an adult (other age groups excluding kids)

2.2 Time Propositions:

- **Valid_Start_Time** = true if the trip's starting time is after the user's desired departure time (the transit is not leaving before the user even wants to start their trip)
- **Valid_End_Time** = true if the trip's ending time is before the user's desired arrival time (the transit is not arriving at its destination later than desired)
- **Rush_Hour** = true if the trip time is within the time periods of 7 am - 10 am and 4 pm - 7 pm.
- **More_Than_Fifty_Strtcar_Bus** = true if the trip is made up of $> 50\%$ busses or street cars transit types

2.3 Preference Propositions:

- **Prefer_Subway** = true if the user prefers using the subway (given by user input)
- **Prefer_Bus** = true if the user prefers using the bus (given by user input)
- **Prefer_Streetcar** = true if the user prefers using the streetcar (given by user input)
- **Prefer_Walking** = true if the user prefers walking (given by user input)
- **Mostly_Subway** = true if the trip is mostly using the subway (percentage calculation)
- **Mostly_Bus** = true if the trip is mostly using the bus (percentage calculation)
- **Mostly_Streetcar** = true if the trip is mostly using the streetcar (percentage calculation)
- **Mostly_Walking** = true if the trip is mostly walking (percentage calculation)
- **No_Pref** = true if no preference on transit type (given by user input)

2.4 Solution Propositions:

- **Within_preference** = true if the trip's transit types are matching user's preferences
- **Within_time_cons** = true if the trip is within the time constraints defined by the user
- **Within_budget_cons** = true if the trip's budget constraints are met
- **Solution**$_{(a,b)}$ = true if all the user-defined requirements (all of the above propositions) are met and a route is found from destination a to b.

3 Constraints

In order for the program to find the optimal trip, our propositions must all meet the requirements for a valid trip plan. These requirements will be made possible using constraints.

3.1 Budget Constraints:

- Kids ages 0-12 ride for free.
- When you pay your fare using a PRESTO card or PRESTO Ticket, you get a two-hour transfer that allows you to enter and exit the TTC as much as you'd like within a two hour period.

TTC Pricing		
Fare Type	Adult	Senior(65+) Youth (13-19)
Single Fare	$ 3.25	$2.30
PRESTO	$3.20	$2.25
PRESTO Day Pass	$13.50	$13.50

If the user does not fall into the youth or senior or kid category, then the user must be an adult:

$$\text{Adult} \rightarrow (\neg \text{ Youth } \wedge \neg \text{ Senior } \wedge \neg \text{ Kid })$$

If the user falls into the youth category, then the user is not an adult or senior or kid:

$$\text{Youth} \rightarrow (\neg \text{ Adult } \wedge \neg \text{ Senior } \wedge \neg \text{ Kid })$$

If the user falls into the senior category, then the user is not an adult or youth or kid:

$$\text{Senior} \rightarrow (\neg \text{ Youth } \wedge \neg \text{ Adult } \wedge \neg \text{ Kid })$$

If the user falls into the kid category, then the user is not an adult or youth or senior:

$$\text{Kid} \rightarrow (\neg \text{ Youth } \wedge \neg \text{ Adult } \wedge \neg \text{ Senior })$$

If the user has an active PRESTO card with sufficient funds for the trip and also an adult, then the trip price will be calculated using PRESTO pricing:

$$(\text{ PRESTO } \wedge \text{ Adult }) \rightarrow \text{PRESTO\_Adult}$$

If the user has an active PRESTO card with sufficient funds for the trip and also a youth or a senior, then the trip price will be calculated using PRESTO pricing:

$$(\text{ PRESTO } \wedge (\text{ Youth } \vee \text{ Senior })) \rightarrow \text{PRESTO\_Other}$$

3.2 Time Constraints:

If the valid trips found are within the time limit specified by the user:

$$\text{Valid\_Start\_Time} \land \text{Valid\_End\_Time} \rightarrow \text{Within\_Time\_Constraint}$$

Rush hour implies there can't be more than 50% slow transit types within the trip:

$$\text{Rush\_Hour} \rightarrow \neg\ \text{More\_Than\_Fifty\_Strtcar\_Bus}$$

Rush hour and more than 50% slow transit types means that the trip will not be within the time constraint:

$$\text{Rush\_Hour} \land \text{More\_Than\_Fifty\_Strtcar\_Bus} \rightarrow \neg\ \text{Within\_Time\_Constraint}$$

3.3 Second Layer Budget Constraint:

If the calculated total price is lower than the user's budget, then Within_time_cons is true:

$$\text{Within\_time\_cons}$$

3.4 Preference Constraints:

If the user's preferred transit type is walking, then it can't be subway, street car, or the bus:

$$\text{Prefer\_Walking} \rightarrow \neg(\ \text{Prefer\_Subway} \land \text{Prefer\_Bus} \land \text{Prefer\_Streetcar}\ )$$

If the user's preferred transit type is subway, then it can't be walking, street car, or the bus:

$$\text{Prefer\_Subway} \rightarrow \neg(\ \text{Prefer\_Walking} \land \text{Prefer\_Bus} \land \text{Prefer\_Streetcar}\ )$$

If the user's preferred transit type is street car, then it can't be subway, walking, or the bus:

$$\text{Prefer\_Streetcar} \rightarrow \neg(\ \text{Prefer\_Subway} \land \text{Prefer\_Bus} \land \text{Prefer\_Walking}\ )$$

If the user's preferred transit type is bus, then it can't be subway, street car, or walking:

$$\text{Prefer\_Bus} \rightarrow \neg(\ \text{Prefer\_Subway} \land \text{Prefer\_Walking} \land \text{Prefer\_Streetcar}\ )$$

If the trip uses made up of a transit type and the user's preference is also the same as that transit type, then within preference is satisfied:

$$\text{Mostly\_Subway} \land \text{Prefer\_Subway} \rightarrow \text{Within\_preference}$$

$$\text{Mostly\_Bus} \land \text{Prefer\_Bus} \rightarrow \text{Within\_preference}$$

$$\text{Mostly\_Streetcar} \land \text{Prefer\_Streetcar} \rightarrow \text{Within\_preference}$$

$$\text{Mostly\_Walking} \land \text{Prefer\_Walking} \rightarrow \text{Within\_preference}$$

Having no preferences means that all preferences can't be true and within preference is true:

$$\text{No\_pref} \rightarrow \text{Within\_preference}$$

3.5 Solution Constraints:

A solution is only valid if it is within time constraints, within budget, and within preference or has no preference on the transit type at all:

$$\text{Within\_budget\_cons} \land \text{Within\_time\_cons} \land (\text{ Within\_preference} \lor \text{no\_pref }) \rightarrow \text{Solution}_{(a,b)}$$

4 Model Exploration

4.1 External Libraries in the Model

- **Geopy:** Used in the user input script to find the bus stop closest to the user inputted location.
- **Sqlite3:** Used to store the data about the trips, routes, and stops so accessing the data is fast.
- **Os:** Used to clear the terminal when doing error checking in the user input section.
- **Platform:** Used to detect operating system so since the clear terminal command is different across different platforms.
- **Itertools:** Used to create all permutations of each generated route.
- **Json:** Used to convert binary numbers to python dictionary and python dictionary to binary since dictionaries cannot be stored in SQL databases.
- **Bauhaus:** Used to create the logical model that the permutations are tested against.

4.2 Python Section

The code required to output a valid trip is quite complicated and requires a lot more explanation than what is provided inside the documentation and inside of the code. So here is a summarized explanation of how our code functions.

All of the data for every single stop, route, and trip is stored within 4 different databases. These databases are created using the python library "sqlite3" which is the python version of the popular database language SQL. The first database is the "trips" database, stored in the "trips.db" file. Each "route" has multiple trips every day and we need a way to store which stops it traverses through and at what time of day the route passes each and every stop. Stored within the "trips" database is every single trip for every route in the TTC system. Each trip has multiple variables and these include:

- trip_id (the number used to identify the particular trip)
- route_id (the number used to associate the trip with its parent route)
- times_list (an ordered list of times at which the trip visits each stop during the day)

- stops_list (an ordered list of stop_ids at which the trip visits each stop during the day
- trip_headsign (the text displayed on the headsign of the bus/streetcar/subway)
- direction (the direction the route is heading, this is used later in the code)

The next database is the "routes" database stored in the "routes.db" file. Since routes change depending on the time of day, maybe skipping certain stops at the end of the day, we are not able to just store all the stops that the route visits in an ordered list. Instead, we store all of the stops that the route visits throughout the entire day in a dictionary to avoid any duplicating stops. The routes database contains all the routes in the TTC system and has the following variables:

- route_id (the number used to identity the particular route)
- long_name (the name of the route)
- route_type (indicates what type of transportation, 0 - streetcar, 1 - subway, 3 - bus)
- trip_id_list (a list of trips associated with the particular route_id, this variable was only used to create the variable below)
- unique_stops (a dictionary of all of the stops that the route visits through all of its trips)

The next database is the "stops" database stored in the "stops.db" file. Each stop contains the following variables:

- stop_id (the number used to identify this particular stop)
- stop_name (the name of the stop)
- stop_lat (the latitude of the stop)
- stop_lon (the longitude of the stop)
- route_id_dict (a dictionary of all the routes that pass through the current stop)

The final database is the "subway" database which is stored in the "subway.db" file. This database contains all of the stops that subways go to which significantly cuts down the runtime complexity of our code later in the explanation. Each subway stop contains the following variables:

- stop_id (the number used to identify this particular stop)
- stop_name (the name of the stop)
- lat (the latitude of the stop)
- lon (the longitude of the stop)
- lane (the subway line that the current stop is on, lines 1 through 4)

Using these 4 databases we're able to generate a route from any point in Toronto to another using just public transportation.

The code starts by getting the user to input some information that is used for both the path-finding and the logical constraints later on in the code. First, the code will request for a starting location. You are given 3 different options, the first of which is inputting a general location, which could be just as simple as typing in "cn tower", "zoo", or "university of toronto", etc. The program will then proceed to find the closest bus stop to the mentioned location as long as the mentioned location exists and is within the boundaries of Toronto. The second option is to input the exact name of the stop you are currently at. These can be found on the official TTC website. (Note: the database we found is 2 years old so there may be some discrepancies between the real-world name and the name stored inside of our database). The last option is to input your current longitude and latitude, after which the code will return the closest bus stop to the stated location. The code then repeats the same process with the destination stop.

Next, the code will request for the starting time of the user's trip. Two options will be presented: 1. The current time, the code will get the current time of the user's machine, and 2. A specified time that's error-checked to ensure that it's a valid time during the day. Then the code will ask for the ending time of your trip and it is error-checked to make sure it is a valid time after the starting time.

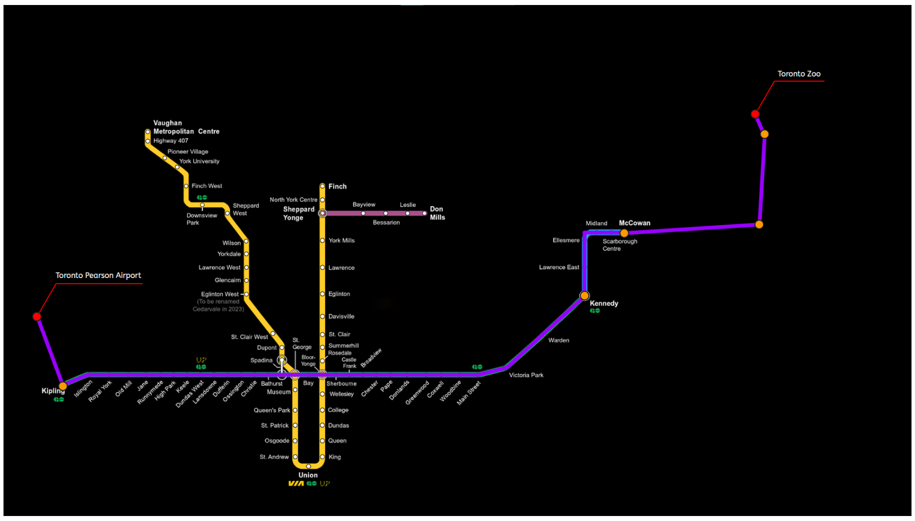
Next, the code will ask for your age, budget, and if you own a presto card. All of these will be used to calculate if you can afford the proposed trip as each of these variables affect how much your trip will cost.

Once all of this information is inputted properly, the code will then begin to calculate a valid trip from the inputted starting location and ending location. It will first attempt to find a direct route from start to end, and if this returns false then the code will move to the second method.

The code will find the subway stops closest to the starting location and ending location and then calculate how the user will navigate the subways, which includes transferring between lines. Next, the code will attempt to find a direct route from the starting location to the subway line. If no direct route is found, the code will try to navigate to all the nearby stops leading to the subway line, stops that have more than 3 routes running through it. The code will continue to search using this method until a valid path is found. The same is performed on the other end of the subway line and the code will find a path from the subway line to the ending location.

Once a path is found, all the different permutations of the proposed route are fed through our propositional logic to verify that there exist trips that operate within the time period that the user has provided, and to calculate if the user has enough budget to afford the trip.

Here is a path that our program would propose if you took public transport from the Toronto Pearson Airport to the Toronto Zoo:



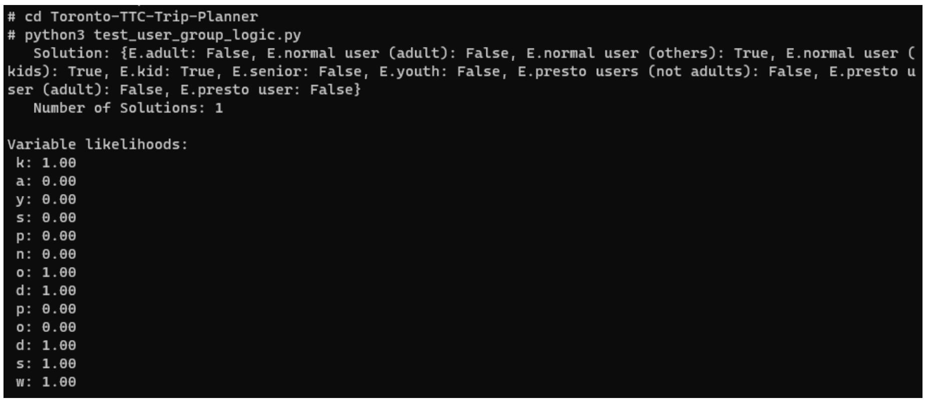
The path-finding algorithm described above runs with a reasonable time complexity, however in order to provide a trip to input into the logical model we would have to generate all permutations of possible trips in combination of all permutations of all the possible times at which the bus navigates each stop. As you can imagine this creates thousands of permutations and for even the most trivial routes such as taking a subway, transferring to a different line, and then going to another subway stop creates over one-hundred-thousand permutations which then have to be run through the logical model. This would cause the program to run for multiple hours without returning a solution. So we have decided to select a couple of test permutations to show that the logical model using the bauhaus library functions as promised.

4.3 Logical Model

Our logic model is split up into 4 parts (which does imply we have separate models for different parts of the program which then get their outputs combined for a final solutions output) to evaluate 2 different constraint sets targeted towards mainly the budget (split into 2 parts), time and additional constraints.

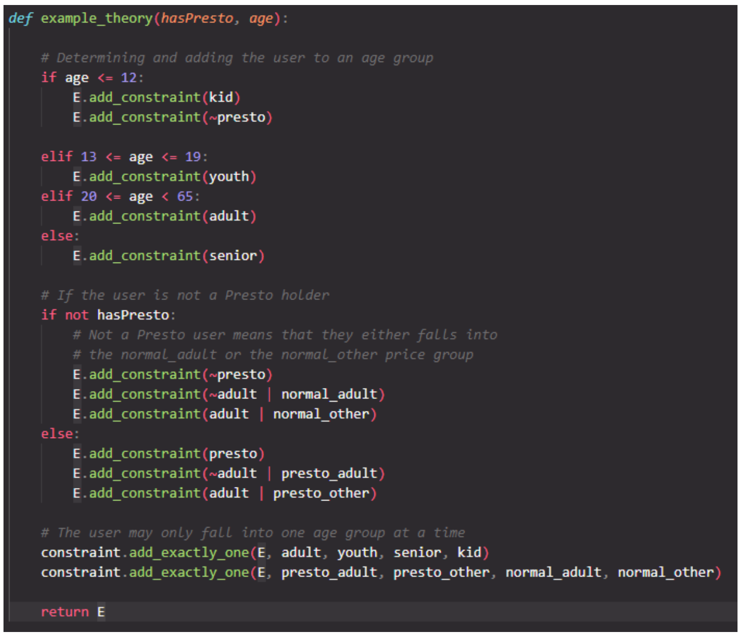
4.3.1 Budget Constraint (Layer 1)

During the development of the budget logic set (the base layer of our budget logic), many issues were run into and we used this as an opportunity to test and learn.



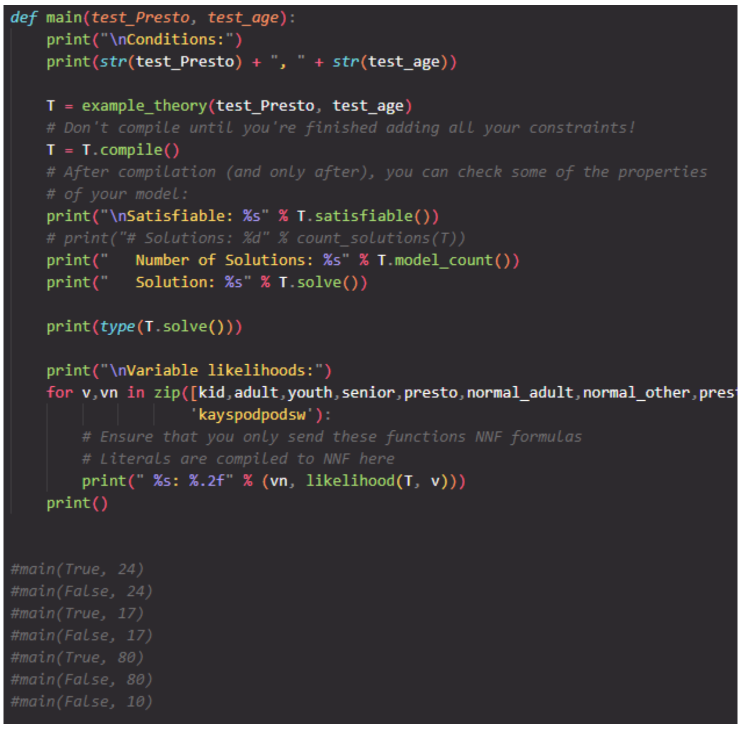
```
# cd Toronto-TTC-Trip-Planner
# python3 test_user_group_logic.py
   Solution: {E.adult: False, E.normal user (adult): False, E.normal user (others): True, E.normal user (
kids): True, E.kid: True, E.senior: False, E.youth: False, E.presto users (not adults): False, E.presto u
ser (adult): False, E.presto user: False}
   Number of Solutions: 1

Variable likelihoods:
 k: 1.00
 a: 0.00
 y: 0.00
 s: 0.00
 p: 0.00
 n: 0.00
 o: 1.00
 d: 1.00
 p: 0.00
 o: 0.00
 d: 1.00
 s: 1.00
 w: 1.00
```

After many trials and errors, we have finally grasped how the logic library works and cleaned up our constraints to be precise and describe exactly the user's age group and their corresponding price group.



```python
def example_theory(hasPresto, age):

    # Determining and adding the user to an age group
    if age <= 12:
        E.add_constraint(kid)
        E.add_constraint(~presto)

    elif 13 <= age <= 19:
        E.add_constraint(youth)
    elif 20 <= age < 65:
        E.add_constraint(adult)
    else:
        E.add_constraint(senior)

    # If the user is not a Presto holder
    if not hasPresto:
        # Not a Presto user means that they either falls into
        # the normal_adult or the normal_other price group
        E.add_constraint(~presto)
        E.add_constraint(~adult | normal_adult)
        E.add_constraint(adult | normal_other)
    else:
        E.add_constraint(presto)
        E.add_constraint(~adult | presto_adult)
        E.add_constraint(adult | presto_other)

    # The user may only fall into one age group at a time
    constraint.add_exactly_one(E, adult, youth, senior, kid)
    constraint.add_exactly_one(E, presto_adult, presto_other, normal_adult, normal_other)

    return E
```

This code was tested against all situations and should only return one solution with the detailed solution indicating what price group the user falls into. Our test code is below.



```
def main(test_Presto, test_age):
    print("\nConditions:")
    print(str(test_Presto) + ", " + str(test_age))

    T = example_theory(test_Presto, test_age)
    # Don't compile until you're finished adding all your constraints!
    T = T.compile()
    # After compilation (and only after), you can check some of the properties
    # of your model:
    print("\nSatisfiable: %s" % T.satisfiable())
    # print("# Solutions: %d" % count_solutions(T))
    print("   Number of Solutions: %s" % T.model_count())
    print("   Solution: %s" % T.solve())

    print(type(T.solve()))

    print("\nVariable likelihoods:")
    for v,vn in zip([kid,adult,youth,senior,presto,normal_adult,normal_other,pres
                    'kayspodpodsw'):
        # Ensure that you only send these functions NNF formulas
        # Literals are compiled to NNF here
        print(" %s: %.2f" % (vn, likelihood(T, v)))
    print()


#main(True, 24)
#main(False, 24)
#main(True, 17)
#main(False, 17)
#main(True, 80)
#main(False, 80)
#main(False, 10)
```

An issue that was noticed during the testing was that we could not call main() multiple times, or else it would only work for the first test case and quickly fail on the second test. This might be the fact that we have made the model already for the first test case and the constraint and propositions were carried over to the second test, as a result not being able to find any solutions.

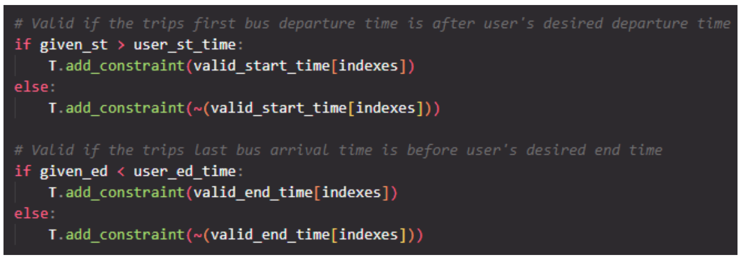
With the completion of the base layer logic for our budget, we can extract the data we need from the solution (only 1 as that represents the only price group the user may fall into) and put into a simple calculation for price.

Using a simple predefined price for each price group, we now have the information to calculate the price for every variation of trips (going from user's defined starting point to their destination).

4.3.2 User Preferences, Time & Budget Constraint (Layer 2)

To use the price group data from the first logic model, we then began working on the time constraint so we could calculate the amount that the user would need to spend for a working trip option.

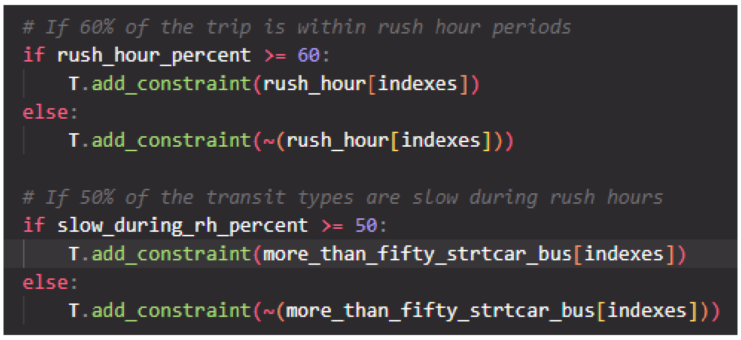
Our initial plan to simply check if the time of a trip (in hours) is within the total trip time (also in hours) was quickly scrapped as our data shows us the specific timing of every bus stop (in 24hr clock format, 12:00). It was quickly found that we could check whether the starting time of a possible trip is after the user planned departure time along with checking if the ending time of the last stop is before the user planned arrival time to achieve the same results.



```python
# Valid if the trips first bus departure time is after user's desired departure time
if given_st > user_st_time:
    T.add_constraint(valid_start_time[indexes])
else:
    T.add_constraint(~(valid_start_time[indexes]))

# Valid if the trips last bus arrival time is before user's desired end time
if given_ed < user_ed_time:
    T.add_constraint(valid_end_time[indexes])
else:
    T.add_constraint(~(valid_end_time[indexes]))
```

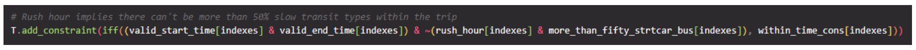
With basic time propositions set up, we moved on to implementing the "rush_hour" time constraint. Initially, we only expected to have one proposition for this part of the logic, but we soon realized that this was inadequate as there is no indication that having a trip during rush hour would 100% add a specific amount of extra travel time, and at the same time we could not change the transit's expected arrival times. The solution we came up with was to then see how much percentage of the trip was spent on transits that are known to be slow during rush hours. The code to determine the "rush_hour" proposition was also changed from detecting whether the trip runs through the rush hours to how much percentage of the trip is in rush hour for better accuracy.

Research suggested that rail systems (subway) and walking are the most efficient methods of transportation during rush hours, which then became part of our parameter for an additional proposition named "more_than_fifty_strtcar_bus". This proposition will work with the initial "rush_hour" proposition in a set of constraints that will mark a trip as valid if and only if one of it were to be true, limiting the trip so that if the majority of it happens within rush hour periods, then only the trips with a low amount of busses and street cars transit types are valid.



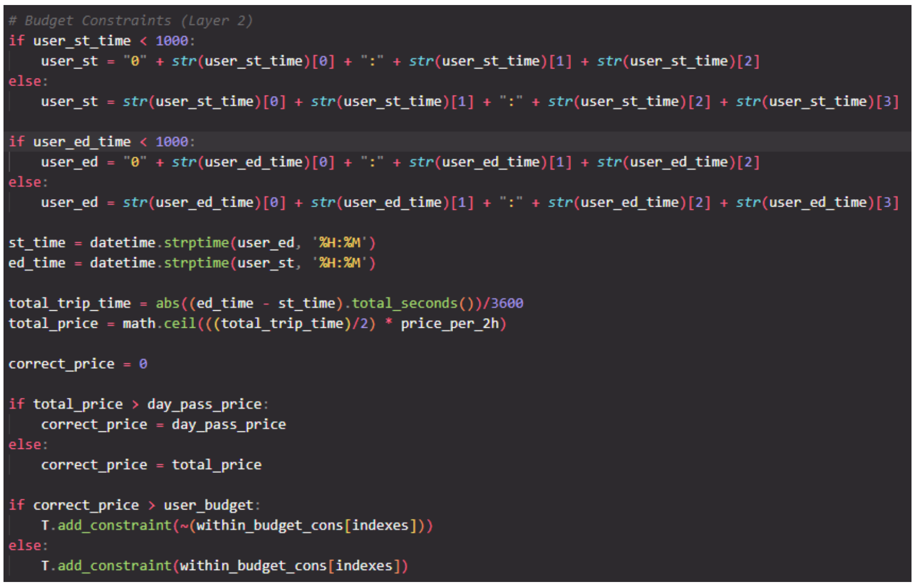
```python
# If 60% of the trip is within rush hour periods
if rush_hour_percent >= 60:
    T.add_constraint(rush_hour[indexes])
else:
    T.add_constraint(~(rush_hour[indexes]))

# If 50% of the transit types are slow during rush hours
if slow_during_rh_percent >= 50:
    T.add_constraint(more_than_fifty_strtcar_bus[indexes])
else:
    T.add_constraint(~(more_than_fifty_strtcar_bus[indexes]))
```

Having these implemented logically as our time propositions, our "within_time" proposition and constraints is now set up.



```python
# Rush hour implies there can't be more than 50% slow transit types within the trip
T.add_constraint(iff((valid_start_time[indexes] & valid_end_time[indexes]) & ~(rush_hour[indexes] & more_than_fifty_strtcar_bus[indexes]), within_time_cons[indexes]))
```

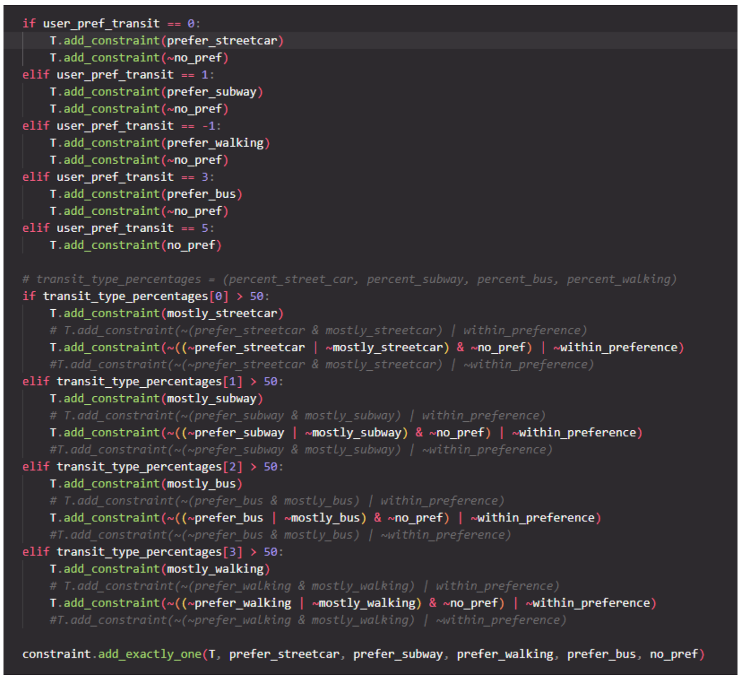
Now knowing that a trip is valid in terms of their starting and ending time, we ran a quick calculation on this time to find the number of hours needed to complete the trip. This result was used then to get the amount of money the user needs to pay in respect to their personal price group (given by the first layer budget logic). The data was put to comparison with the user's budget to see whether the expected price is under the user's specified budget. This completes the logic implementation of the "within_budget" constraint.



```python
# Budget Constraints (Layer 2)
if user_st_time < 1000:
    user_st = "0" + str(user_st_time)[0] + ":" + str(user_st_time)[1] + str(user_st_time)[2]
else:
    user_st = str(user_st_time)[0] + str(user_st_time)[1] + ":" + str(user_st_time)[2] + str(user_st_time)[3]

if user_ed_time < 1000:
    user_ed = "0" + str(user_ed_time)[0] + ":" + str(user_ed_time)[1] + str(user_ed_time)[2]
else:
    user_ed = str(user_ed_time)[0] + str(user_ed_time)[1] + ":" + str(user_ed_time)[2] + str(user_ed_time)[3]

st_time = datetime.strptime(user_ed, '%H:%M')
ed_time = datetime.strptime(user_st, '%H:%M')

total_trip_time = abs((ed_time - st_time).total_seconds())/3600
total_price = math.ceil(((total_trip_time)/2) * price_per_2h)

correct_price = 0

if total_price > day_pass_price:
    correct_price = day_pass_price
else:
    correct_price = total_price

if correct_price > user_budget:
    T.add_constraint(~(within_budget_cons[indexes]))
else:
    T.add_constraint(within_budget_cons[indexes])
```

Many different iterations of these two parts of the logic were explored as we work towards a more clear and understandable model in which only does what we want it to do, classifying a trip to be valid or not. Following this ideology, we decided to implement a set of final solution propositions and a dedicated "solutions" propositions variable to classify a trip as feasible given the constraints.

To increase the complexity in terms of our customizability of the trip planning process, the user's transit type preference parameter was thought of and implemented into our logic model.

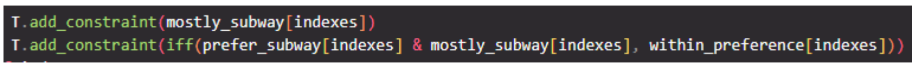


```python
if user_pref_transit == 0:
    T.add_constraint(prefer_streetcar)
    T.add_constraint(~no_pref)
elif user_pref_transit == 1:
    T.add_constraint(prefer_subway)
    T.add_constraint(~no_pref)
elif user_pref_transit == -1:
    T.add_constraint(prefer_walking)
    T.add_constraint(~no_pref)
elif user_pref_transit == 3:
    T.add_constraint(prefer_bus)
    T.add_constraint(~no_pref)
elif user_pref_transit == 5:
    T.add_constraint(no_pref)

# transit_type_percentages = (percent_street_car, percent_subway, percent_bus, percent_walking)
if transit_type_percentages[0] > 50:
    T.add_constraint(mostly_streetcar)
    # T.add_constraint(~(prefer_streetcar & mostly_streetcar) | within_preference)
    T.add_constraint(~((~prefer_streetcar | ~mostly_streetcar) & ~no_pref) | ~within_preference)
    #T.add_constraint(~(~prefer_streetcar & mostly_streetcar) | ~within_preference)
elif transit_type_percentages[1] > 50:
    T.add_constraint(mostly_subway)
    # T.add_constraint(~(prefer_subway & mostly_subway) | within_preference)
    T.add_constraint(~((~prefer_subway | ~mostly_subway) & ~no_pref) | ~within_preference)
    #T.add_constraint(~(~prefer_subway & mostly_subway) | ~within_preference)
elif transit_type_percentages[2] > 50:
    T.add_constraint(mostly_bus)
    # T.add_constraint(~(prefer_bus & mostly_bus) | within_preference)
    T.add_constraint(~((~prefer_bus | ~mostly_bus) & ~no_pref) | ~within_preference)
    #T.add_constraint(~(~prefer_bus & mostly_bus) | ~within_preference)
elif transit_type_percentages[3] > 50:
    T.add_constraint(mostly_walking)
    # T.add_constraint(~(prefer_walking & mostly_walking) | within_preference)
    T.add_constraint(~((~prefer_walking | ~mostly_walking) & ~no_pref) | ~within_preference)
    #T.add_constraint(~(~prefer_walking & mostly_walking) | ~within_preference)

constraint.add_exactly_one(T, prefer_streetcar, prefer_subway, prefer_walking, prefer_bus, no_pref)
```

With this addition to our model, when the user is entering their trip details, an additional parameter of the user's preferred type of transit will be accounted for. Trips we got from the algorithm will then be labelled with the amount (in percentage) of each transit type in each one. Fitting this data into the model will then filter out all the trips that have $> 50\%$ of the trip in a transit type that are not within the user preference group and mark them as invalid in terms of the proposition "within_preference".

With the 3 main sections implemented, we begin piecing it all together into a single model (as each were individually tested and ran until this point). The tests were all run on a single trip instance, which was not what our output from the algorithm is (which caused issues later on…).

At this stage, our model was giving us a hard time as many test cases were simply just not correct. At many instances, the logic model would return false when the trip is clearly feasible. This led us to reevaluate our constraints and encouraged us to make changes to the structure of how the constraints are defined, including defining and using a custom "iff" function to replace the many logical "iff"s implemented with "and" and "or". This certainly helped clean up the code and made everything a lot more readable, eliminating many of the syntax issues we were running into (missing/misplaced brackets).



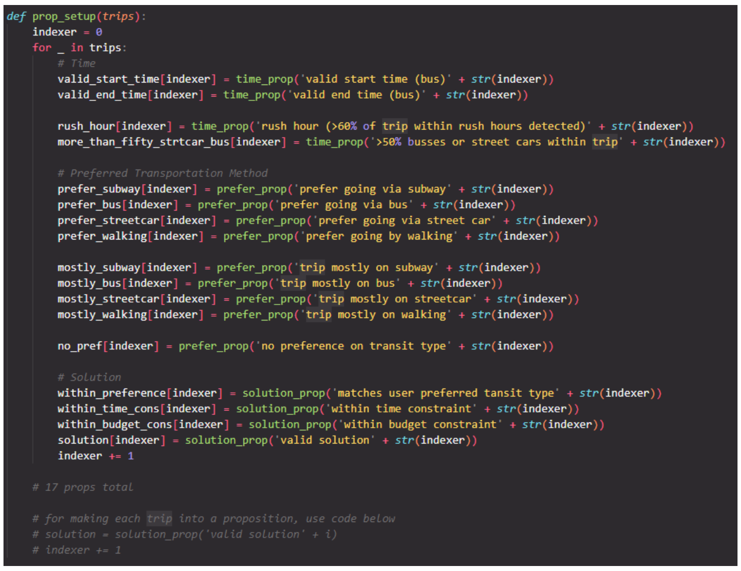
```
T.add_constraint(mostly_subway[indexes])
T.add_constraint(iff(prefer_subway[indexes] & mostly_subway[indexes], within_preference[indexes]))
```

At this point, we have finished fixing up the constraints and have a working and correct logic model, but for only one trip instance. To make this model apply to all trips that go from point A to point B, we realized that having one set of constraints was simply not enough as every instance of a trip would need to be checked using their own set of propositions and constraints.

To achieve this, we have decided to create a dictionary of propositions with their keys being the number instance of the trip we are looking at (for the first trip option it would be 0 and so on).

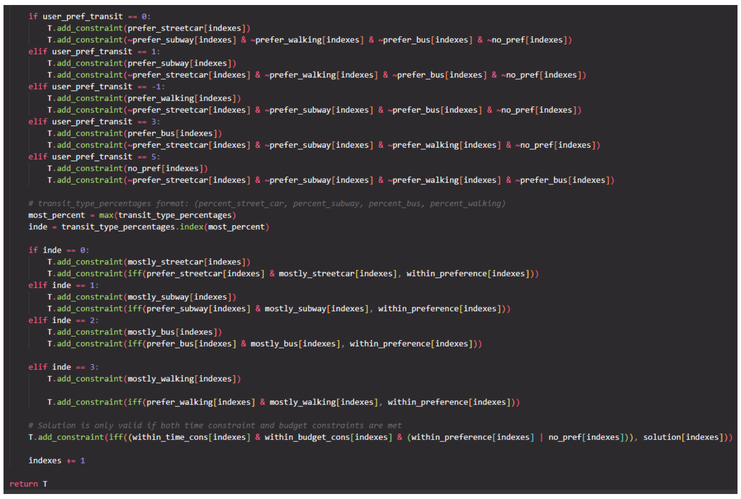
This idea of proposition generation allows us to create a set of propositions and make constraints for each and every trip option, which will then evaluate and see the number of viable solutions in all the trip options, achieving our end goal.

To make the propositions of each instance of a trip unique, we used the same index to label which trip a proposition may belong to. This is achieved using a for loop and a counter variable, which was then appended to each and every propositions that were created.



```python
def prop_setup(trips):
    indexer = 0
    for _ in trips:
        # Time
        valid_start_time[indexer] = time_prop('valid start time (bus)' + str(indexer))
        valid_end_time[indexer] = time_prop('valid end time (bus)' + str(indexer))

        rush_hour[indexer] = time_prop('rush hour (>60% of trip within rush hours detected)' + str(indexer))
        more_than_fifty_strtcar_bus[indexer] = time_prop('>50% busses or street cars within trip' + str(indexer))

        # Preferred Transportation Method
        prefer_subway[indexer] = prefer_prop('prefer going via subway' + str(indexer))
        prefer_bus[indexer] = prefer_prop('prefer going via bus' + str(indexer))
        prefer_streetcar[indexer] = prefer_prop('prefer going via street car' + str(indexer))
        prefer_walking[indexer] = prefer_prop('prefer going by walking' + str(indexer))

        mostly_subway[indexer] = prefer_prop('trip mostly on subway' + str(indexer))
        mostly_bus[indexer] = prefer_prop('trip mostly on bus' + str(indexer))
        mostly_streetcar[indexer] = prefer_prop('trip mostly on streetcar' + str(indexer))
        mostly_walking[indexer] = prefer_prop('trip mostly on walking' + str(indexer))

        no_pref[indexer] = prefer_prop('no preference on transit type' + str(indexer))

        # Solution
        within_preference[indexer] = solution_prop('matches user preferred tansit type' + str(indexer))
        within_time_cons[indexer] = solution_prop('within time constraint' + str(indexer))
        within_budget_cons[indexer] = solution_prop('within budget constraint' + str(indexer))
        solution[indexer] = solution_prop('valid solution' + str(indexer))
        indexer += 1

    # 17 props total

    # for making each trip into a proposition, use code below
    # solution = solution_prop('valid solution' + i)
    # indexer += 1
```

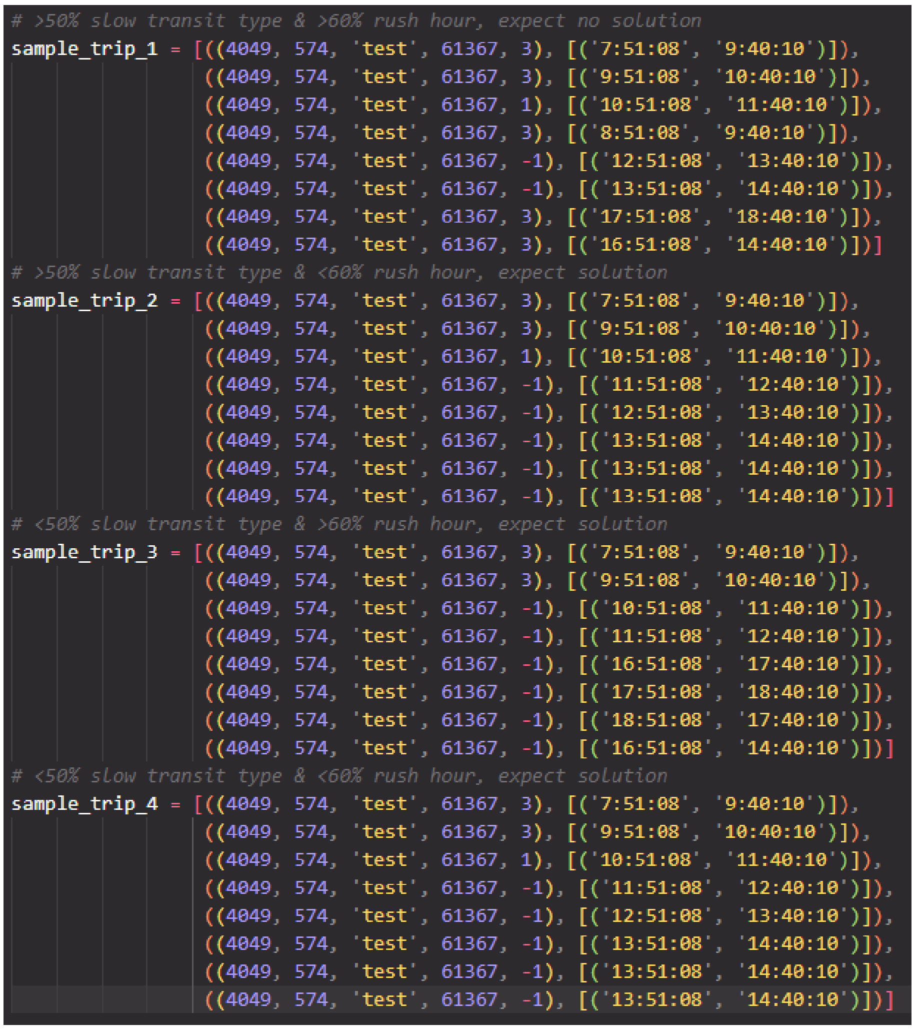
This same concept was then translated for use when adding the constraints, where a for loop with a counter variable was used to access all the propositions in their corresponding dictionaries. The generation of these constraints caused some small issues and made us reevaluate the constraints again, but with some debugging, the logic model was finally working with our test data, a screenshot can be found below showing a portion of the code that was modified to be used in the generator. This marks a milestone in the development of this program.



```python
    if user_pref_transit == 0:
        T.add_constraint(prefer_streetcar[indexes])
        T.add_constraint(~prefer_subway[indexes] & ~prefer_walking[indexes] & ~prefer_bus[indexes] & ~no_pref[indexes])
    elif user_pref_transit == 1:
        T.add_constraint(prefer_subway[indexes])
        T.add_constraint(~prefer_streetcar[indexes] & ~prefer_walking[indexes] & ~prefer_bus[indexes] & ~no_pref[indexes])
    elif user_pref_transit == -1:
        T.add_constraint(prefer_walking[indexes])
        T.add_constraint(~prefer_streetcar[indexes] & ~prefer_subway[indexes] & ~prefer_bus[indexes] & ~no_pref[indexes])
    elif user_pref_transit == 3:
        T.add_constraint(prefer_bus[indexes])
        T.add_constraint(~prefer_streetcar[indexes] & ~prefer_subway[indexes] & ~prefer_walking[indexes] & ~no_pref[indexes])
    elif user_pref_transit == 5:
        T.add_constraint(no_pref[indexes])
        T.add_constraint(~prefer_streetcar[indexes] & ~prefer_subway[indexes] & ~prefer_walking[indexes] & ~prefer_bus[indexes])

    # transit_type_percentages format: (percent_street_car, percent_subway, percent_bus, percent_walking)
    most_percent = max(transit_type_percentages)
    inde = transit_type_percentages.index(most_percent)

    if inde == 0:
        T.add_constraint(mostly_streetcar[indexes])
        T.add_constraint(iff(prefer_streetcar[indexes] & mostly_streetcar[indexes], within_preference[indexes]))
    elif inde == 1:
        T.add_constraint(mostly_subway[indexes])
        T.add_constraint(iff(prefer_subway[indexes] & mostly_subway[indexes], within_preference[indexes]))
    elif inde == 2:
        T.add_constraint(mostly_bus[indexes])
        T.add_constraint(iff(prefer_bus[indexes] & mostly_bus[indexes], within_preference[indexes]))

    elif inde == 3:
        T.add_constraint(mostly_walking[indexes])

        T.add_constraint(iff(prefer_walking[indexes] & mostly_walking[indexes], within_preference[indexes]))

    # Solution is only valid if both time constraint and budget constraints are met
    T.add_constraint(iff((within_time_cons[indexes] & within_budget_cons[indexes] & (within_preference[indexes] | no_pref[indexes])), solution[indexes]))

    indexes += 1

return T
```

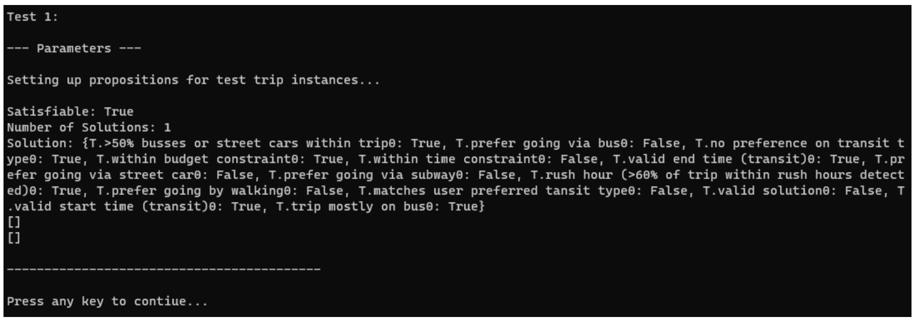
4.4 Test Cases

For test case 1, we started off with a simple set of artificial trips that we know for certain the answers to. The image below shows the 4 samples we created, we will explain their uses and expected outputs below.

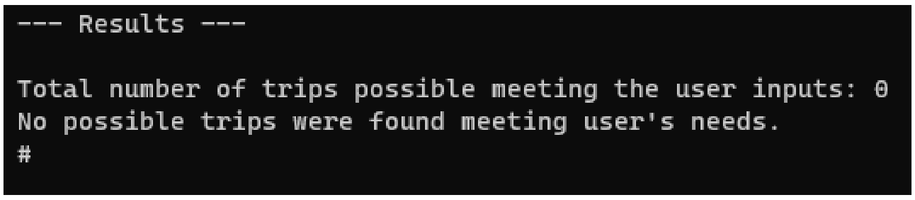


```
# >50% slow transit type & >60% rush hour, expect no solution
sample_trip_1 = [((4049, 574, 'test', 61367, 3), [('7:51:08', '9:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('9:51:08', '10:40:10')]),
                 ((4049, 574, 'test', 61367, 1), [('10:51:08', '11:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('8:51:08', '9:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('12:51:08', '13:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('17:51:08', '18:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('16:51:08', '14:40:10')])]
# >50% slow transit type & <60% rush hour, expect solution
sample_trip_2 = [((4049, 574, 'test', 61367, 3), [('7:51:08', '9:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('9:51:08', '10:40:10')]),
                 ((4049, 574, 'test', 61367, 1), [('10:51:08', '11:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('11:51:08', '12:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('12:51:08', '13:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')])]
# <50% slow transit type & >60% rush hour, expect solution
sample_trip_3 = [((4049, 574, 'test', 61367, 3), [('7:51:08', '9:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('9:51:08', '10:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('10:51:08', '11:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('11:51:08', '12:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('16:51:08', '17:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('17:51:08', '18:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('18:51:08', '17:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('16:51:08', '14:40:10')])]
# <50% slow transit type & <60% rush hour, expect solution
sample_trip_4 = [((4049, 574, 'test', 61367, 3), [('7:51:08', '9:40:10')]),
                 ((4049, 574, 'test', 61367, 3), [('9:51:08', '10:40:10')]),
                 ((4049, 574, 'test', 61367, 1), [('10:51:08', '11:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('11:51:08', '12:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('12:51:08', '13:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')]),
                 ((4049, 574, 'test', 61367, -1), [('13:51:08', '14:40:10')])]
```

Starting off with sample_trip_1, this array simulates a set of steps it requires to reach from point A to point B in one possible iteration of the trip. This sample trip is set up so that it will not be a valid trip in the end due to the fact that it has over 50% transit types that are slow during rush hours while also having 60% of the trip happening during rush hours. The timings of each step in the trip can be found at the very right of each step, with the first time in the tuple indicating the departure time and the right being the arrival time of the transit that this step is using. Running this sample trip on its own gave us a model still with 1 solution, however parsing through the model output, we can see that it does not hold any valid trips.



```
Test 1:

--- Parameters ---

Setting up propositions for test trip instances...

Satisfiable: True
Number of Solutions: 1
Solution: {T.>50% busses or street cars within trip0: True, T.prefer going via bus0: False, T.no preference on transit t
ype0: True, T.within budget constraint0: True, T.within time constraint0: False, T.valid end time (transit)0: True, T.pr
efer going via street car0: False, T.prefer going via subway0: False, T.rush hour (>60% of trip within rush hours detect
ed)0: True, T.prefer going by walking0: False, T.matches user preferred tansit type0: False, T.valid solution0: False, T
.valid start time (transit)0: True, T.trip mostly on bus0: True}
[]
[]

------------------------------------------

Press any key to contiue...
```

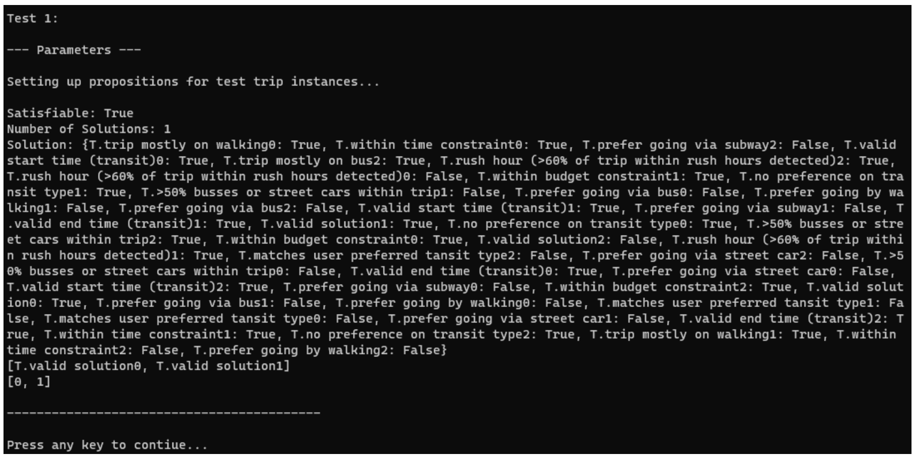


```
--- Results ---

Total number of trips possible meeting the user inputs: 0
No possible trips were found meeting user's needs.
#
```

Using the same methodology, we created 3 more test cases, each testing a different part of the logic model (screenshot of the code can be found above). All the test cases were run individually and then returned the correct expected results.

To take the test now to a closer level to a real world situation, we combined the sample_trips all into one array named "test_1" to simulate a normal scenario where we have many different iterations to complete a trip, all of which are found through permutations by the algorithm.

The results were as expected, the solutions output has now grown in size due to the number of propositions the model now has and was then searched through for the data we need. The image below shows the sample run.

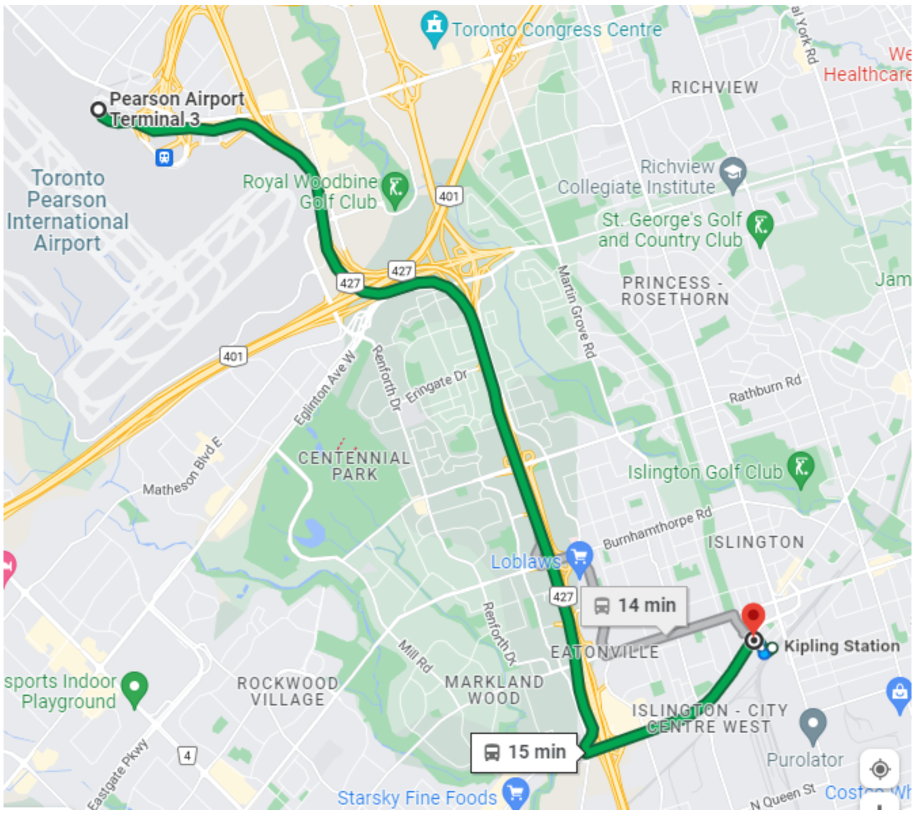


```
Test 1:

--- Parameters ---

Setting up propositions for test trip instances...

Satisfiable: True
Number of Solutions: 1
Solution: {T.trip mostly on walking0: True, T.within time constraint0: True, T.prefer going via subway2: False, T.valid
start time (transit)0: True, T.trip mostly on bus2: True, T.rush hour (>60% of trip within rush hours detected)2: True,
T.rush hour (>60% of trip within rush hours detected)0: False, T.within budget constraint1: True, T.no preference on tra
nsit type1: True, T.>50% busses or street cars within trip1: False, T.prefer going via bus0: False, T.prefer going by wa
lking1: False, T.prefer going via bus2: False, T.valid start time (transit)1: True, T.prefer going via subway1: False, T
.valid end time (transit)1: True, T.valid solution1: True, T.no preference on transit type0: True, T.>50% busses or stre
et cars within trip2: True, T.within budget constraint0: True, T.valid solution2: False, T.rush hour (>60% of trip withi
n rush hours detected)1: True, T.matches user preferred tansit type2: False, T.prefer going via street car2: False, T.>5
0% busses or street cars within trip0: False, T.valid end time (transit)0: True, T.prefer going via street car0: False,
T.valid start time (transit)2: True, T.prefer going via subway0: False, T.within budget constraint2: True, T.valid solut
ion0: True, T.prefer going via bus1: False, T.prefer going by walking0: False, T.matches user preferred tansit type1: Fa
lse, T.matches user preferred tansit type0: False, T.prefer going via street car1: False, T.valid end time (transit)2: T
rue, T.within time constraint1: True, T.no preference on transit type2: True, T.trip mostly on walking1: True, T.within
time constraint2: False, T.prefer going by walking2: False}
[T.valid solution0, T.valid solution1]
[0, 1]

------------------------------------------

Press any key to contiue...
```

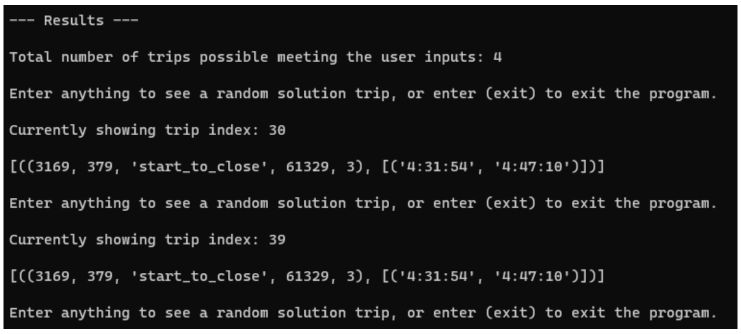
At the bottom of the output, we can see the marked solutions that were found to be true. These 2 solutions were sample_trip_2 and sample_trip_3, which are both expecting valid solutions.

After this, a few more combinations of the trips were tested to ensure that the propositions were all indeed returning the right values inside the solution output and that the final valid solutions were marked correctly. This then takes us to the final actual test with real world data.

To put our logic model and program as a whole to test, we looked and found a route that is simple yet realistic. Going from Pearson Airport Terminal 3 to Dundas St West at Aukland Rd.

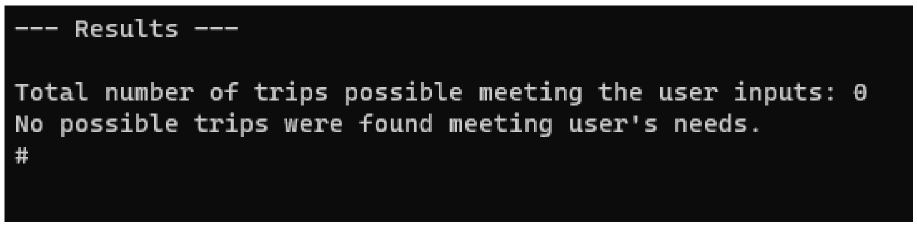


We then set up the other parameters to be all valid (budget, time...). The program then outputted the correct trip that we expected, along with a few more that we didn't even catch (there turns out to be 3 other solution trips that met our constraints that we did not expect but are indeed valid!).



```
--- Results ---

Total number of trips possible meeting the user inputs: 4

Enter anything to see a random solution trip, or enter (exit) to exit the program.

Currently showing trip index: 30

[[(3169, 379, 'start_to_close', 61329, 3), [('4:31:54', '4:47:10')]]]

Enter anything to see a random solution trip, or enter (exit) to exit the program.

Currently showing trip index: 39

[[(3169, 379, 'start_to_close', 61329, 3), [('4:31:54', '4:47:10')]]]

Enter anything to see a random solution trip, or enter (exit) to exit the program.
```

After this successful test, we began playing around with different parameters, mainly ones that weren't satisfiable, which includes unrealistic timings, insufficient budget (although this one was a bit hard to check for this specific test case as the trip is so short).... All of which still does return a valid logic solution to the model but however fails to produce a true valid solution.



```
--- Results ---

Total number of trips possible meeting the user inputs: 0
No possible trips were found meeting user's needs.
#
```

If you want to try out the real world example from above follow these steps:

1. Answer no to the "Run test" query.
2. Select the first method and input "Toronto Airport".
3. Select the second option that is provided to you.
4. Answer Yes to is this the stop you are currently at.
5. Next select the second method and input "Kipling Station".
6. Answer No to is this the stop you want to go to and then select the alternative stop that is labeled "DUNDAS ST WEST AT AUKLAND RD".
7. Next select "Specific Time" for the starting time and enter 04:30 or if you want to test different times against our model you can enter a different time.
8. Next enter in 04:50 for the ending time or any other time to test the model.
9. For the next 4 steps you can enter whichever values to test our model.

Do not try to run the program with any other trips. Even though it will output the valid trips, it will take your computer at least 10 hours to iterate through all the possible permutations if the program decides to at any point use a subway line. This example specifically does not run through any subway stations.

We have provided a test case of a complicated route and a test case with a direct route using our user input method to test the routes against our logical model, but the output is a bit difficult to understand. So let us take one of the outputs from the second test case and explain how to understand it.

[((3169, 3854, 'direct', 61438, 3), [('4:31:54', '4:47:10')])]

Each tuple within the list represents a "step" in the route. Since in this case we were able to find a direct route from point A to point B, there is only one step in the route. There are two elements within this tuple. The first element is a tuple giving us information about the locations, and method of travel. The first two numbers here, in this case 3169 and 3854, represent different stops, 3169 being Pearson Airport Terminal 3 and 3854 being Dundas St West at Aukland Rd. The next element, the string ‘direct’, is telling us that it is a direct route from the desired starting stop to the ending stop. There can be 5 possible strings in this section:

- ‘direct’ - There is a direct route from the starting stop to the ending stop
- ‘start_to_close’ - There is a direct route from the starting stop to a stop within 200 meters of the ending stop.
- ‘close_to_end’ - There is a direct route from a stop within 200 meters of the starting stop to the ending stop.
- ‘close_to_close’ - There is a direct route from a stop within 200 meters of the starting stop to a stop within 200 meters of the ending stop.
- ‘walking’ - The ending stop is within 200 meters of the starting stop.

The next element is the number representing the route you take to get from the first stop to the second stop. In this case it is 61438, which corresponds to the route “Airport Express”. This element will not be present if the previous element stated that the step is accomplished by ‘walking’.

The final element in the first tuple represents the transportation type. In this case it is 3, which corresponds to buses. There can be 4 possible numbers in this section:

- -1 - ‘walking’
- 0 - ‘streetcar’
- 1 - ‘subway’
- 3 - ‘bus’

Next, let’s move onto the second part of the output. This portion is fairly self-explanatory, the first element within the tuple in the list is the time at which you leave the first station, and the second element is the time which you arrive at the second station. Note that all of these times will be displayed in 24-hour format.

Do note that this specific example is for a direct route for which there is only one step in your trip. If there were multiple steps in the trip, the program will output a list of all of the steps in order.

5 First-Order Extension

Predicate logic: For each proposition, we can represent it with a predicate that holds the current time p and the arrival time q, The universal discourse A would be the different possible stops and locations in Toronto. An example of which would be the following:

EAST - 10 VAN HORNE towards VICTORIA PARK STATION$_{(p,q)}$

WEST - 10 VAN HORNE towards DON MILLS STATION$_{(p,q)}$

SOUTH - 102 MARKHAM RD towards WARDEN STATION$_{(p,q)}$

The constraints above that use these prepositions can be replaced with their predicate counterparts using all possible times for current time and all possible times for the desired arrival time. Here are some examples:

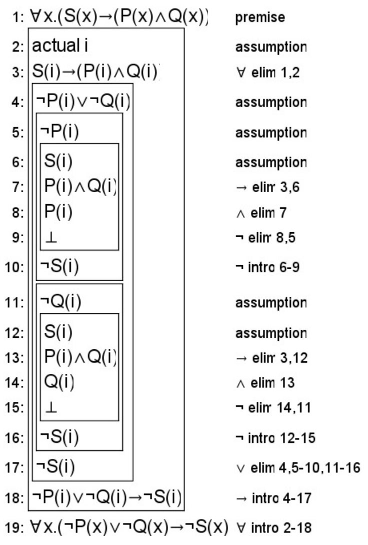
- B = Budget Limit
- R = Rush Hour
- L = Time Limit
- G = Successful Trip

1. $\forall$p $\exists$q (T(p,q) $\land$ B):
   For all current time p there exists an arrival time q where you can go on the trip "EAST - 10 VAN HORNE towards VICTORIA PARK" within the budget limit.
2. $\forall$p $\neg\exists$q (T(p,q) $\land$ $\neg$B):
   For all current time p there does not exist an arrival time q where you can go on the trip without exceeding the budget limit.
3. $\forall$p $\exists$q ((T(p,q) $\land$ G) $\rightarrow$ (B $\land$ L)):
   For all current time p and all arrival time q, if you go on a trip and it is a successful trip, then the trip will be within the Budget Limit and the Time Limit.
4. $\forall$p $\exists$q ((T(p,q) $\land$ ($\neg$B $\lor$ $\neg$L)) $\rightarrow$ $\neg$G):
   For all current time p and all arrival time q, if you go on a trip and you are not within Budget Limit or Time Limit, then you will not have a successful trip.
5. $\exists$p $\exists$q ((T(p,q) $\land$ G) $\rightarrow$ (R $\land$ L $\land$ B)):
   There exists current time p and there exists arrival time q where if you go on a trip and it is a successful trip, then you have to do so during rush hour while being within the Time Limit and Budget Limit.

6 Jape Proofs

6.1 Proof 1:

$\forall x.(\mathbf{S(x)}\rightarrow(\mathbf{P(x)}\wedge\mathbf{Q(x)})) \vdash \forall x.(\neg\mathbf{P(x)}\vee\neg\mathbf{Q(x)}\rightarrow\neg\mathbf{S(x)})$

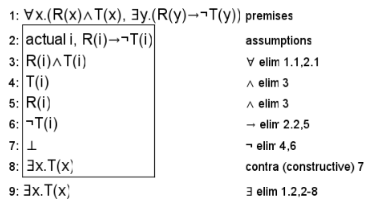


```
1: ∀x.(S(x)→(P(x)∧Q(x))        premise
2: actual i                     assumption
3: S(i)→(P(i)∧Q(i)              ∀ elim 1,2
4: ¬P(i)∨¬Q(i)                  assumption
5: ¬P(i)                        assumption
6: S(i)                         assumption
7: P(i)∧Q(i)                    → elim 3,6
8: P(i)                         ∧ elim 7
9: ⊥                            ¬ elim 8,5
10: ¬S(i)                       ¬ intro 6-9
11: ¬Q(i)                       assumption
12: S(i)                        assumption
13: P(i)∧Q(i)                   → elim 3,12
14: Q(i)                        ∧ elim 13
15: ⊥                           ¬ elim 14,11
16: ¬S(i)                       ¬ intro 12-15
17: ¬S(i)                       ∨ elim 4,5-10,11-16
18: ¬P(i)∨¬Q(i)→¬S(i)           → intro 4-17
19: ∀x.(¬P(x)∨¬Q(x)→¬S(x)       ∀ intro 2-18
```

x: starting time
P: time limit S: successful trip Q: budget limit
We know that for all possible trips, a successful trip would imply that we meet both the time constraint and budget constraints. As such, here we prove that for all trips any exceeded limit would mean that it will not be a successful trip.

6.2 Proof 2:

$\forall x.\mathbf{R(x)}\wedge\mathbf{T(x)},\exists.(\mathbf{R(y)}\rightarrow\neg\mathbf{T(y)}) \vdash \exists.\mathbf{T(x)}$

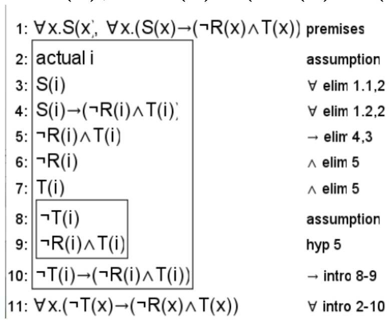


```
1: ∀x.(R(x)∧T(x), ∃y.(R(y)→¬T(y))  premises
2: actual i, R(i)→¬T(i)            assumptions
3: R(i)∧T(i)                       ∀ elim 1.1,2.1
4: T(i)                            ∧ elim 3
5: R(i)                            ∧ elim 3
6: ¬T(i)                           → elim 2.2,5
7: ⊥                               ¬ elim 4,6
8: ∃x.T(x)                         contra (constructive) 7
9: ∃x.T(x)                         ∃ elim 1.2,2-8
```

x: starting time y: ending time
R: rush hour T: time limit
At any departure time, there is a time limit and the trip is in rush hour. There exists arrival time, rush hour implies not meeting the time limit, indicating that there is a departure time that can meet the time limit.

6.3 Proof 3:

$\forall x$.S(x), $\forall x$.S(x)$\rightarrow$($\neg$R(x)$\wedge$T(x)) $\vdash$ $\forall x$.($\neg$T(x)$\rightarrow$($\neg$R(x)$\wedge$T(x))

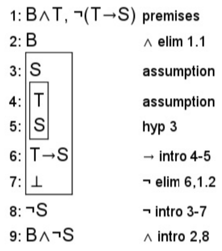


```
1: ∀x.S(x), ∀x.(S(x)→(¬R(x)∧T(x)) premises
2: actual i                       assumption
3: S(i)                           ∀ elim 1.1,2
4: S(i)→(¬R(i)∧T(i))              ∀ elim 1.2,2
5: ¬R(i)∧T(i)                     → elim 4,3
6: ¬R(i)                          ∧ elim 5
7: T(i)                           ∧ elim 5
8: ¬T(i)                          assumption
9: ¬R(i)∧T(i)                     hyp 5
10: ¬T(i)→(¬R(i)∧T(i))            → intro 8-9
11: ∀x.(¬T(x)→(¬R(x)∧T(x))        ∀ intro 2-10
```

x: trip
S: successful trip R: rush hour(true if not in rush hour) T: time limit
For a successful trip at any departure time, a successful trip at any departure time means that it is within the time limit and not in rush hour, which indicates that for an arbitrary departure time, not within the time limit means it is in rush hour and has a time limit.

6.4 Proof 4:

B$\wedge$T,$\neg$(T$\rightarrow$S) $\vdash$ B$\wedge\neg$S

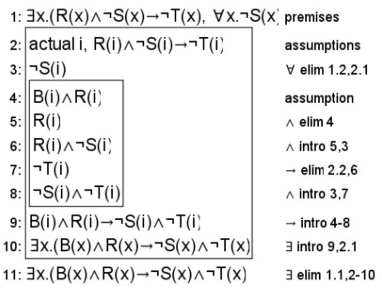


```
1: B∧T, ¬(T→S) premises
2: B           ∧ elim 1.1
3: S           assumption
4: T           assumption
5: S           hyp 3
6: T→S         → intro 4-5
7: ⊥           ¬ elim 6,1.2
8: ¬S          ¬ intro 3-7
9: B∧¬S        ∧ intro 2,8
```

B: budget S: successful T: time limit
The trip with sufficient budget and within the time limit, but without indicating success within the time limit, proves that the budget meets the condition and the trip is not successful.

6.5 Proof 5:

$\exists$ **x.(R(x)** $\wedge$ **¬S(x)→¬T(x)),**$\forall$ **x.¬S(x)** $\vdash$ **∃x.**(B(x)∧R(x)→¬S(x)∧¬T(x))



```
 1: ∃x.(R(x)∧¬S(x)→¬T(x), ∀x.¬S(x   premises
 2: | actual i, R(i)∧¬S(i)→¬T(i        assumptions
 3: | ¬S(i)                            ∀ elim 1.2,2.1
 4: | | B(i)∧R(i                       assumption
 5: | | R(i)                           ∧ elim 4
 6: | | R(i)∧¬S(i                      ∧ intro 5,3
 7: | | ¬T(i)                          → elim 2.2,6
 8: | | ¬S(i)∧¬T(i)                    ∧ intro 3,7
 9: | B(i)∧R(i)→¬S(i)∧¬T(i             → intro 4-8
10: | ∃x.(B(x)∧R(x)→¬S(x)∧¬T(x)        ∃ intro 9,2.1
11: ∃x.(B(x)∧R(x)→¬S(x)∧¬T(x)          ∃ elim 1.1,2-10
```

x: departure time
T: time limit B: Budget limit S: successful trip R: rush hour(true if not in rush hour)
There exists a trip at a departure time in which an unsuccessful trip while not during Rush hour still results in the required time limit not being met. Thus there must exist a trip at a departure time in which meeting both the budget limit and meeting the Rush hour constraint still resulted in us not having a successful trip and not meeting the time limit.

7 Conclusion

We have succeeded in creating a model for a Toronto Daytrip Planner by utilizing the concepts and ideas that were discussed over the course of CISC 204. While we were unfortunately unable to optimize our time complexity for our algorithm to be able to run on any processing platform regardless of specifications. We have nonetheless created an algorithm that is logically capable of planning a day trip in the city of Toronto. One that will meet the budget and time set by the user while taking into consideration of all the different factors that come with traveling in Toronto.