KING'S College LONDON

University of London

6CCS3PRJ Final Year Create the UI for a Traffic Control system - in collaboration with TFL

Final Project Report

Author: Ivaylo Krumov Hristov

Supervisor: Dr Theodora Koulouri

Student ID: 1435790

May 2, 2017

Abstract

This project's purpose is to create an interface between PTV Vissim and Artificial Intelligence planner. It consists of two separate tasks. First, traffic controllers data has to be extracted from Vissim's model and formatted in a correct PDDL format. The other part consists of extracting planner results and edit the corresponding network according to them.

The chosen programming language of the script that will execute those tasks is Python. Communication with the simulator is achieved through the COM interface. This behaviour is supported through the PyWin32 Python module, which is freely available in pip.

Although the script will be used by professionals, an intermediate JSON format of the extracted data will be created for better observation to be possible and also to offer more use-cases of the software.

Originality Avowal

Acknowledgements

I would like to thank for the support and guidence of my project supervisor, Dr. Theodora Koulouri. Throughout the length of the project she has always been responding to any of my problems and provided help for any problems I was encountering.

I would like also to thank Michael Cashmore, who offered me a very helpful introduction to the whole system.

Contents

Chapter 1

Introduction

1.1 Project Context

Greater London, England is a huge urban area with an estimated population of more than 8 million people, measured in mid-2014. The government body responsible for its traffic, TFL (Transport For London) faces every day with a big variety of issues in trying to organise their traffic plans. One of those problems is the big amount of congestion. In fact, London drivers had spent on average 101 hours in gridlock in 2015 alone[13]. One way of tackling this particular problem, is to create an effective traffic light plan, thus ensuring road flow to be as smooth as possible.

Currently, there are certain Urban Traffic Control Systems that are trying to solve the issue. Some have utilized a fixed-time schedule, which is static and stays the same unless a human agent decides to manually change the values. Other implementations have taken a more adaptive approach and by collecting live road data, such as car count on every street of a junction, the signal-timing schedule may change. What any of these systems can't provide a solution to, is a way to cope with any unexpected traffic incidents.

Currently, on this kind of events, it is again a duty of a human operator to make an educated decision and manually adjust the plans. Relying fully on humans for such complex task is highly problematic as not only it requires substantial amount of prior training and expertise, but it is also susceptible to human error. Therefore, a new system has been proposed. The proposed system makes use of artificial intelligence (AI) planning. Among the advantages of this approach is that the AI planner can cope with unusual scenarios such as accidents or planned road closures. Given a model of the traffic network, the planner is able to compute an optimised traffic lights plan. The aim of such system is to support human operators to make better decisions that in turn will alleviate traffic management problems.

1.2 Project Aim

Currently, the AI system will work with a simulation. For that to be possible, a realistic model for the simulator is created. From this same model it is a human operator's job to manually extract all the required data the AI plan needs to construct a sensible description of the network.

What the project is trying to accomplish is to get rid of those inefficient and error-prone parts of the system by creating an interface between the simulator and the planner. One of the goals to be achieved is that all the relevant data from the simulating software has to be collected and automatically put inside the file, which serves as an input to the planner. The next objective is to make it possible for the results of the plan to be applied directly to the simulation model. That way it will be easier to run evaluation tests on the new timing schedule.

1.3 Report Structure

The project will start with a background chapter, whose focus is to elaborate more on currently existing Urban Traffic Control systems, as well as describing how AI planning work, alongside examples of systems already using its approach. Moreover, key terms used throughout the project will be defined.

Following that, design and specification chapter will give a high-level description of the whole system. Additionally, the requirements of the project will be specified based on that context. Lastly, the thought-process behind choosing some the tools the project uses will be discussed alongside some of the alternative options considered.

Next major chapter is the implementation part. Its purpose is to give a detailed overview of the conducted development process. Certain blockers during the attempts to satisfy the requirements will be also be listed.

The last chapter is evaluation and future work. It will give an overall conclusion of the developed work by specifying whether or not the requirements have been met. Additionally, it will be discussed what features should be implemented in the future work of the project.

Chapter 2

Background

2.1 Urban Traffic Control Systems (UTC)

Urban Traffic Control is a term used to describe co-ordinating traffic signals in a network by the use of timing plans. Normally everything is controlled through a central computer. This means that the created plans are loaded automatically by the computer to the on-street equipment.

2.1.1 Fixed-Time Urban Traffic Control Systems

The most basic implementation of this concept is the fixed-time urban traffic control system. The way it works is that as a result of a collection and analysis of traffic data there are predefined series of plans on the central computer, each representing different time of the day. The optimal aim is to reduce the amount of waiting time by creating a "green wave", a sequence of leaving one signal and arriving at the next one just when it turns green. As described above the plans are automatically distributed on-street and the plans change cycle after cycle. In general adopting such system helped significantly reduce the amount of stops a car makes, thus saving energy and money. This solution has a few problems though. First, the data worked with is static. If there are any changes to the traffic patterns, a new set of data needs to be collected. This, however, is an expensive and time-consuming effort, thus the whole process becomes very ineffective.

2.1.2 SCOOT

An example of more "adaptive" system is Split Cycle Offset Optimisation Technique (SCOOT). It was first developed for general use in 1980 and it is a result of a collaboration work of

Transport Research Laboratory (TRL) and the UK traffic system industry [2]. Its goal is to minimize the average queues. In contrast to the fixed-time systems, SCOOT dynamically collects real-time data from on-street vehicle detectors. Worth noting is that it works with smaller scale models, ensuring faster adaptation to the designated area of operation. The way the systems works based on the information from the sensors it calculates all the queues in the area. An optimizer takes this as an input and calculates an adapted signal timing plan [15]. As calculating future traffic flow is proven to be very difficult to predict[5] in fear of the plan being already out-dated, the last step the system makes before actually applying the changes is to verify they are still better than the current schedule. One advantage of this adaptive nature of the system is that it removes the need of manually creating signal plans and keeping them up to date. However, the system still suffers from the limitation of not being able to adapt adequately to unpredicted events, such as traffic accidents. Although the technology is more than 30 years old, SCOOT is still currently used throughout the whole of UK, with London being one of the cities making use of its capabilities[14].

2.2 Artificial Intelligence Planning (AI Planning)

Artificial intelligence generally consists of three parts: input, planning and a plan.

The input is a model of the world. It is represented by its unique initial (I) and goal (G) state, and also a list of possible actions that an agent can make. All actions have properties called preconditions and effects. Once the planner is started, the solver will continuously switch states, starting from the initial one, until the specified target gets reached. Although different planners apply different algorithms, they all have in common that they attempt to reach the model's goal from its initial state. The way this is done is by constructing a directed graph. If we think of states as graph nodes, then, for each stage the plan is at a certain point, there is a finite amount actions, whose preconditions are met in that certain state. Undertaking a chosen action will apply its effects, thus changing the state to a new one.

In order for the agent to find its goal state more efficiently, each node is given a heuristic value, defined by how far is it from the desired target. Naturally, if a state of heuristic value equal to 0, then the current state is the G.

At the end of execution of the planner, when the goal state is reached, a plan is given. What it should consist of is the series of actions undertaken from the initial state to the goal state.

2.2.1 PDDL

Planning Domain Definition Language (PDDL), developed by Drew McDermott in 1998, is the standard modeling language for planning domains. One of the things the language did differently than its competitors at the time is to divide the input model into two files âĂŞ domain and problem files [8]. Figure 2.1 shows this structure with a simple diagram.

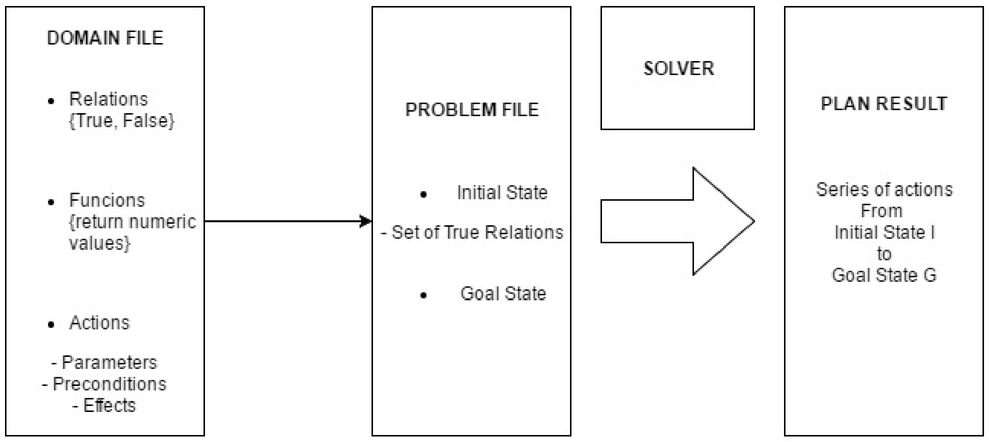


Figure 2.1: PDDL two-file architecture

The domain file is a collection of functions, predicates and a set of actions. The functions are the only way to introduce numeric values. Each has a name and when called return their value. On the other hand, predicates, also called propositions, is the set of relations the system can work with. Each relation can be either True of False. One good approach is to ensure possible mutual exclusion through those propositions. This is a crucial feature, as physical laws can be obliged: i.e. a robot cannot be in two rooms simultaneously. In terms of actions, each one of them has its own parameters, preconditions, and effects. The parameters are the objects involved in the action. The preconditions specify the set of relations that have to be True before the execution of the action has started. The effects are the set of state changes to be applied after the execution of the action. In practice, those effects can only be adding and deleting relations, or manipulating values of variables through the functions.

On the other hand, the problem file provides objects, the world's initial state and the desired goal state to be reached. Objects are the set of all the relevant the described world consists of. Each one of them has a name and a type. Both initial and goal states are just a set of relations among the objects. Only the true relations can be specified for the initial state, any non-mentioned ones are assumed to be false.

2.2.2 PDDL+

Although PDDL has seen a lot of improvements since the day it was introduced, it is still far too basic to deal with many of actual real-world problems. An extension of the language, called PDDL+ was introduced in 2006 by Fox and Long[6]. What PDDL+ offers more is support for flexible model of continuous change through the use of autonomous processes and events[6]. In simpler words, the extension supports properties of the domain, called “processes” and “events”. A process is simply a durative action, which continuously changes the state of the world. Events can be considered as another sort of the basic PDDL actions. They also have had their own parameters, preconditions, and effects. Intuitively, when the preconditions are met, the event gets triggered and effects of it are applied, which can dictate which processes should be active and which not. Such approach is more suitable for cases where is necessary to track the state of flow e.g. energy, money, liquid, etc.

Due to its capabilities, PDDL+ is already a key part of many huge projects trying to solve real-life problems.[7]

2.3 AI Planning in Urban Traffic Control

Although there are clear advantages of using artificial intelligence planning in traffic, its use as in today’s systems is little. By taking in mind the principles of planning and the capabilities of the tools it supports, possible applications include not only reducing traffic congestion, but also objectives such as reducing pollution or optimising public transport schedules.

2.3.1 Possible Planning Approaches

There are two possible approaches of planning for urban traffic control. It can be either microscopic or macroscopic. The microscopic path takes into consideration each vehicle as a separate object. There are many disadvantages to this approach, with scalability being the main one. On the other hand, macroscopic deals more with controlling the traffic flow itself rather than being on a single vehicles level[18].

2.3.2 SURTRAC

An already running system using planning is called SURTRAC. It concentrates on a day-to-day scenarios by aiming to reduce waiting time of cars. In contrast to the fixed-time or adaptive approaches, it has a fully decentralized organisation, with each intersection allocating its own signal timetable based on incoming vehicle flow. Any outflows are then communicated to neighbouring intersection[20]. Not only does this ensure appropriate reaction to any real-time issues, but also enables more reliable creation of green corridors, which was the initial issue trying to be solved by the first models of urban traffic control systems. The project started in a small area in City of Pittsburgh, Pennsylvania, US. Although the provided sample is not that big, the results are very convincing. The system reduced travel times more than 25% on average, and wait times reduced an average of 40% [16]. With this kind of outcome, the system is getting more trust by controlling more and more city's junctions. As of 2016, SURTRAC is installed at 50 intersections in Pittsburgh, with additional expansions currently in plan[16].

2.4 Traffic Simulation Software

The way most traffic simulators work is so: a network can be either created or imported. Then all of the network's properties can be configured, such as routes, links, traffic light algorithms and decision-making of the vehicles, on what distance to switch lanes, when to decide to which route to take, etc. The more freedom you have to introduce new constraints, the more accurate will be the simulation. With all of that information, the simulator calculates the expected traffic flow and visually represents it in 2D or 3D.

One of the biggest differences between the different traffic simulation software is the level of abstraction. There exist four different classes of traffic modeling: macroscopic, microscopic, mesoscopic and nanoscopic[4]. Macroscopic is the most abstract and concentrates on the average flow of big groups of vehicles and that is why it can handle simulation of huge networks. Microscopic models each vehicle in as many details as possible and it is mainly used to simulate just the smaller parts of a network. The mesoscopic one is something between the former two[19], while nanoscopic models vehicles in even bigger details, by dividing it into parts.

2.5 Traffic Properties

In order to better understand the requirements and implementation, key terms that a traffic simulator uses have to be defined. Only the relevant ones for the projects will be outlined.

2.5.1 Network and Links

The whole model that is first being loaded is called a Network. It contains everything - streets, signal heads, vehicle size, spawn rate, etc. The way streets work, is by defining Links. Link is a segment with a given length, direction, and number of lanes. Basically, if a street is going only eastbound and it has two lanes, only one link has to be created. Certain length for this link has to be given, it has an eastbound direction and also two lanes. On the other hand if on this same street, one of the lanes has the opposite direction, then two links have to be created, each with one lane and different directions. The way two links can be connected, let's say in a junction, is through a third link specified as a connection between them. Figure 2.2 helps to clarify these concepts.

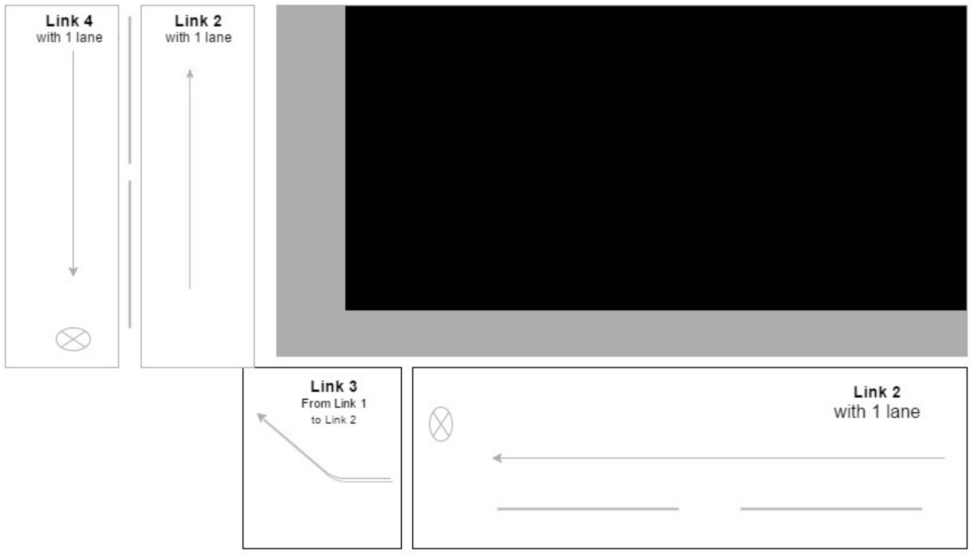


Figure 2.2: Two links connected to each other through a special third link. A street going both directions = two separate links

2.5.2 Traffic Plan and Stages

Specifying traffic light behaviour can be a bit more complicated. One way to do it is to use a simple fixed-time schedule. However, what the better simulators will let you do, is to determine your own algorithm for each of the intersection. To do this a "signal controller" is created and is assigned an algorithm, referred to as signal timing plan. On the other hand, a property called "Stage" specifies which signal heads should be green, and which - red. Based on certain logic in the plan, the simulator decides which stages should be active, and which not.

2.5.3 Signal Group

No matter if the signal timings are fixed-time or controlled by some more sophisticated algorithm, controlling each signal head individually can involve a lot of effort and is highly unnecessary. In practice, there are a lot of examples where signal heads with same behaviour are duplicated. Why that is necessary is because signals have to be visible to all of the road users. In order for creating a simulator model network to be less error-prone and reduce the redundant work for simulators, the software often provides an option to associate some heads into a group and assign only single behaviour to all of them. This traffic lights cluster is referred to as a signal group or “Phase”.

Chapter 3

Design & Specification

Before listing specific requirements for the project, the ones of the whole system it is part of should be mentioned first.

3.1 System Requirements and Overview

Here is given a brief description of the intelligent urban traffic control system, by specifying its high-level requirements.

1. A simulator creates a traffic flow simulation
2. The simulator passes simulation data onto the planner
3. The planner uses this data to create a planning model
4. The planner computes a signal timing plan (the plan specifies what changes need to be made to the traffic lights in a junction) for this model
5. The planner passes this plan to the simulator
6. The simulator applies the plan to make the specified changes to the simulation

The system's aim is to successfully address the shortcomings of existing systems, that is to deal with long queues caused by unplanned scenarios, such as car accidents. To give a better perspective on what the actual system is about and how each of its parts is connected, figure 3.1 is provided. Roughly, there are four steps. First, a specially-created network is loaded into the simulator. In the next step, by interacting with an interface, data from that model is extracted, which in combination with a PDDL domain file, are fed to the PDDL+ planner.

According to its aims, the planner uses the macroscopic modelling approach. As such the data required is a high-level one. The necessary information includes: road links, junctions, flows, saturation and max queues, as well as junction plans, stages, and phases. An advantage of this approach is that actual positions of the cars can be neglected, which would be the case in the macroscopic case.[18] Afterward, that planner starts computing and an optimized plan of the signal timings is generated. The results can then be represented in a human-friendly way to the human operator or they can also be exported to the original simulation model, so a safe evaluation of the whole process can be made.

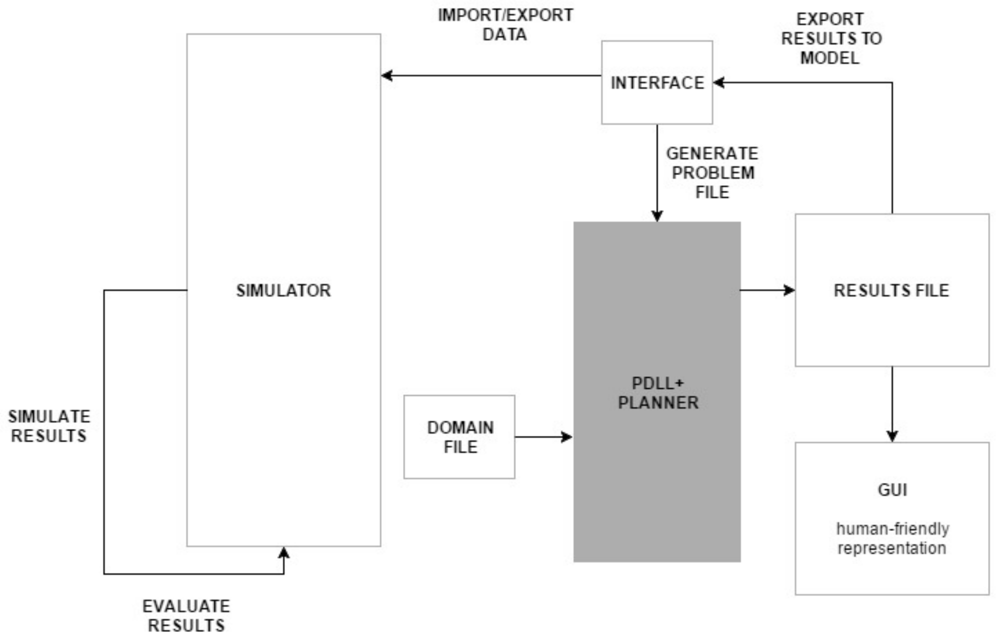


Figure 3.1: PDDL Urban Traffic Control System high level scheme

3.2 Project-Specific Requirements

The project relates to the second, fifth and sixth requirements of the whole system. Its purpose is to create the interface between the simulator and the planner. The work can be divided into two parts. As security and performance requirements are not necessary, each subsection has a different set of only functional features.

3.2.1 Simulator Data Extraction Requirements

- The user has to be able to choose a traffic model to work with, which will be loaded by a simulator software
- Signal timing plan cycle length has to be automatically extracted for each of the junctions
- Signal timings (schedule of stage changes) has to be automatically extracted for each of the junctions
- The initial stage has to be automatically extracted for each of the junctions
- The maximum stage has to be automatically extracted for each of the junctions
- Which phases are active for each of the stages has to be automatically extracted for each of the junctions
- All the gathered data has to be collected and organized in an intermediate form
- A PDDL problem file has to be created in a location and filled with the collected data in an appropriate format corresponding to the PDDL domain

3.2.2 Plan Results Application Requirements

- The user has to be able to choose a traffic PDDL plan results file
- The user has to be able choose the model which he wants to edit
- The script has to recognise the actions of the planner's results
- The script has to be able to recognise the involved objects and apply the desired changes to them
- The script should not break previous build of the model to be changed

3.3 How The Project Should Work

To elaborate a bit on the requirements, the expected behaviour of the project has to be described. The objective of the first part is to create a script that will import data from the simulator, that is relevant for the planner. Specifically, the desired extracted collection should consist of: junction timing plans, phases, and stages. This data has to be extracted to an intermediate form. After that, the exported details have to be put in a PDDL problem file, which has to oblige with a predefined PDDL domain. On the other hand, the second part of the project will work in the opposite direction: it has to apply the results from the planner directly to the simulator model. An overview of the described high-level behaviour can be observed in figure 3.2.

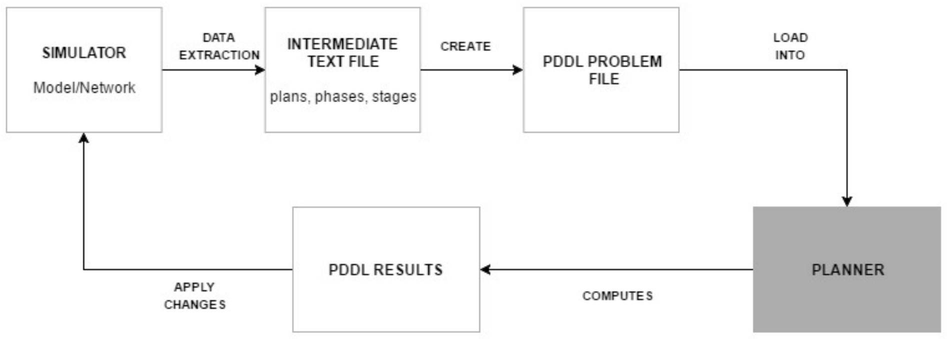


Figure 3.2: Project high level scheme

3.4 Simulator Software

As the planner will need a low level abstraction, and while nanoscopic class provides too many irrelevant details, the microscopic model sounds as the optimal model that is needed. That is why only a software running it should be considered.

3.4.1 PTV Vissim

The simulator for the purpose of the system is a software called PTV Vissim. Developed by PTV Group, it is the go-to software for the professionals in the sphere of traffic management. One of its advantages over other similar software is that it provides full control over the most parts of the simulation, meaning the user has more freedom and space to create a simulation as accurate as possible. On the con side of things, while the developers provide an option to apply for a free student licence, its functionality is heavily limited. What this means is that a purchase of the full software is necessary.

VAP

As already mentioned in Background, some more advanced simulators will let users define their own signal control logic. In Vissim the function is enabled by the Vechile Actuated Programming (VAP) language. As already discussed, controlling the logic for each signal head is really challenging and that is why heads with same behaviour are grouped together in Signal Groups. What the PTV Vissim simulator also provides is to group related to each other signal groups and to form a Signal Controller with the purpose of enabling easier signal traffic logic creation. When a signal controller of type “VAP” is created, two files have to be provided and assigned to it. The first one is of type “VAP”. It consists its logical behaviour. Also an add-on called VisVAP might be provided with the Vissim installation depending on the type of license of the program. The tool helps model developers to generate VAP files without using a text editor[9]. Without diving into too much detail, here is what a VAP file’s structure looks:

- Constants: Can be either user-defined or system ones that are defined by signal groups or interstages.
- Variables: automatically defined depending on the context of the logic. All variables are set to 0 at the start of the simulation execution. Timer variables are a bit special and cannot be changed manually by the user, unless special timer commands are called (START, STOP, RESET).
- Arrays: VAP support both one- and two-dimensional arrays.
- Subroutines: Each subroutine has its own execution, which defines the plan execution.

A sample VAP file content was put in the Appendix[A.1].

PUA

However, in order for the VAP signal controllers to work an additional “PUA” file has to be supplied. What the file should provide is:

- Signal groups: A list of the signal groups used in the file is specified with its name used inside the file and an id corresponding to its corresponding one from the network.
- Stages: A list of all the stages, with their corresponding signal groups that have to be green while they are active. Additionally the signal group that have to be red can be specified. Why that is optional is because it is assumed that if the signal group is not put in the “green” list, then it is in the “red” list.
- Starting Stage: the initial stage which will be active at the start of execution of the VAP file

- Interstages: An interstage is the state when switching from one specified stage to another. However, for now this transitional concept will be disregarded by the AI planner, as now it is applied that stage switching is immediate.

A sample PUA file content was put in the [AppendixA.4].

3.4.2 SUMO

One of the simulators that were first considered as it meets the initial requirement is called Simulation of Urban Mobility (SUMO). The reason why it was an interesting option is the fact that it is open-sourced, thus it is free to obtain it along all of its features. As required it uses microscopic simulation [10], meaning vehicles, pedestrians and public transport are modelled explicitly, thus providing the needed level of abstraction. One of the implemented features is to control the running simulation online through a service called Traffic Control Interface (TraCI). It works on a server/client architecture, with the user retrieving or manipulating object values, and requests being carried out through TCP, with support for a big range of programming languages.

3.5 Communication With The Simulator

While PTV Vissim does not provide SUMO's TraCI service, it has arguably a better solution to the problem: the chosen simulation software has support for COM interface. COM interface is a feature of Windows, which if supported by a software, enables the program to be used by an outside application. Although not too many programming languages support COM in their standard library, a huge amount of them can communicate with it through stable releases of easily-accessible third party libraries.

3.5.1 Vissim COM Interface

Figure 3.3 provides a brief overview of how Vissim can be interacted with through the Vissim COM. It also gives some examples on how some of the parts can be used. It is observable that not only does it support dynamically controlling simulations, but it is possible to collect or modify static network data. Another benefit of it is that when the software is installed, PTV Vissim comes with a lot of well-written manuals and examples regarding the usage of the software itself and also a guide how to communicate with it through the COM interface.

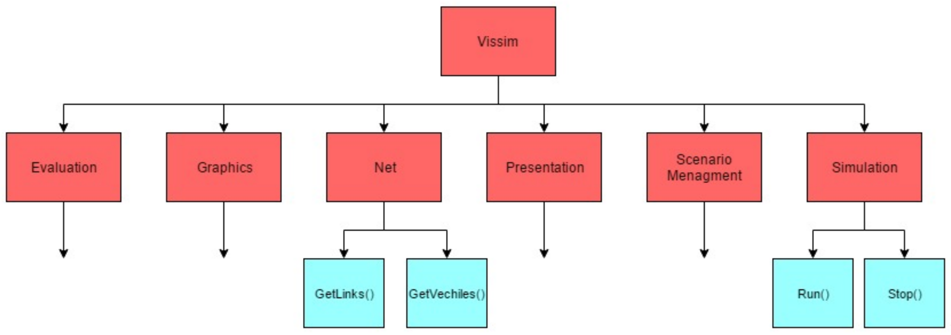


Figure 3.3: Vissim COM interface high-level scenario management

3.6 Programming Environment

In terms of the choice of a programming language there is only limitation, which is that it has to support COM interface. However, this constraint does not filter the list of options too much. Instead some other factors need to be considered to reach an optimal choice, such as the fact that the project does not run heavy computations, which means that there is no need for low-level abstraction.

3.6.1 Python

By taking these properties into mind, Python sounds very suitable and was eventually chosen as the programming language to be used for the project. It is an interpreted language, created in 1991. As such, the lines of code do not have to first be transformed into machine-language instructions before the program executes, which is the case of a compiled languages. Instead, the program will run using an interpreter. In general, that approach makes the technology more portable. However, there are also some disadvantages to it. Native machine code created by compilers is faster to run, but this is not such a big issue in the context of the project, as it has been already defined in the design that functional requirements take bigger priority than performance ones.

Compared to other interpreted languages, such as Java or C#, Python is less portable. The reason is that both Java and C# take a more of a hybrid approach than actual interpreted one. First they compile their high-level code into a specific sort of machine code before feeding that newly created sequence of instructions to an interpreter. The advantage is that a single copy of the compiled code has to be constructed, which will run on all the different systems, supported by their interpreters. However, Python provides better code readability by allowing developing concepts in less amount of code lines, which is something smaller-scale programs such as this project look for.

Python 2 vs Python 3

Currently there are two version of Python that are actively maintained - 2 and 3[11]. Although Python 3 is considered to be the only future of the language, there is one massive advantage of using Python 2: some of the libraries that exist still might not support or have a worse functional performance in version 3. Taking into consideration that the project will use external libraries and the fact that 2 is still maintained and has an enormous community, Python 2 should be considered the safer choice in this scenario.

Pip

One of the advantages of Python is that its installation will include its packet manager - pip. This enables installing third party libraries in most of the cases very easy. For example, the language does not support communicating with COM server in its standard library, but there is a third party one that provides that functionality, called “PyWin32”. Including it in the project’s environment in this case will be as easy as writing a single line in a terminal. What this also implies is that if during the implementation it is required to use any external libraries, it won’t create any significant overhead for the project to be completed.

PyCharm

PyCharm is a Python integrated development environment (IDE) software application created and actively maintained by JetBrains. Compared to standard text editors it provides developers with full set of tools designed to maximise productivity efficiency. Although the expected size of the project’s system will not be too big, the IDE is still a very helpful tool for any beginner in Python as it gives dynamic feedback on the code’s quality: it checks for bad variable naming, redundant library imports, etc. The software is also entirely free for any student in the United Kingdom.

3.7 Project Deployment

As any other software project, its possible distributions have to be considered before starting its actual implementation. As already mentioned above, pip is the Python package manager, from which it is possible to obtain external libraries. What it also supports though is a way to host projects. This means that after deployment of the work is done, it will be as easy and fast to obtain it.

3.8 Project Scalability

Even if the scripts are functioning well, with time the system might face new problems with similar requirements, which the script won't be able to solve directly. Considering all of this brings to mind some issues regarding scalability. That is why a decision was made to divide the project into separate modules. Each module's functions will have their own independent use-case, i.e. PUA module will have methods only for PUA files. That way not only will my project be used for my scripts, but the implementation of the already solved problems could be used in future work by just importing the project and using its methods for the desired goal.

3.9 Intermediate Structure Form Of Data

To increase significantly the use cases of the project, the collected data from the simulator first will be put in an intermediate format. That structure has to be readable by both humans, so all the data can be easily observed, and machines, so the PDDL problem file can be constructed from it. All in all, it should increase the quality of the product.

One of the better known document formats is Extensible Markup Language (XML). It is a markup language, and as such it gives a way to annotate text or arbitrary data so it can be computer-readable. How it works is that data is marked up with an opening and closing tag. A tag consists of the element's class name, surrounded by characters '<' and '>'. The closing tag adds a '/' in front of the name. Class-specific attributes can be added within the start tag as a key-value pair. Example: <street id="31">Oxford Street</street>. Since this format has been used as a standard data format for variety of applications, most programming languages have built some sort of support for it.

3.9.1 JSON

Although using XML provides developers with great advantages, there is another data format that has been the choice for most modern applications. It is called JavaScript Object Notation, or JSON for short. It works as a simple collection of key-value pairs. The value can be a standard type such as string or a number, but it can also be another JSON object or even a collection of this kind of objects (an array). In contrast to XML, the format is heavily inspired by the syntax of the programming languages and therefore that implies that most of them will have at least relative support for it. Also, in terms of human readability, it seems to outperform XML, as it omits the need for tags, which can be sometimes hard to follow. Although the technology is newer than XML, most programming languages have already built-in some sort of support for it. Moreover, there is an enormous amount of tools built for any kind of help working with the format.

3.10 Development Process

Test-driven development(TDD) is a special software development process that encourages implementing a requirement by first creating test cases before starting to write the code itself [3]. Although, creating tests at the start can be more time-consuming than what other development processes suggest, it encourages programmers to concentrate solely on the specified requirements and not lose time writing code that is outside of the scope. This is a key programming principle referred as “You aren’t gonna need it” (YAGNI). In general, the development cycle can be considered to consist of 5 steps:

1. A test is added
2. All the tests are ran with the idea that the new tests should fail. If not, then it is badly written and will pass even though the requirement is not met.
3. Code is written until the newly written test cases are satisfied.
4. All tests should be ran just to make sure that the new code has not broken anything working before. The step is repeated until all test cases pass.
5. At the end if necessary the code is cleaned up to satisfy certain code standards: efficiency, readability, etc. The process is called refactoring.

Why the proposed approach is appropriate for some parts of the project is because TDD is targeted for cases where the requirements are fully understood, but the way to solve them is unclear. For example it is already specified that traffic plans data has to be collected and formatted to JSON, but the way to do it is unknown until the implementation part starts.

3.10.1 Version Control System

For the purpose of the project, the Git Version Control System will be heavily used. This system provides an enormous amount of features that can hugely improve the development process of any scale of projects[17]. How it can help this project specifically, first it is a way to upload code on a cloud. This way it can be accessed from anywhere where Internet access is provided. Secondly, a methodology called “branching” is a way to structure your project so when working on a new feature, it will not break your previous stable version. Of course the system is not that simple to use, and in order to take most out of its features, a certain amount of experience working with it is required.

3.10.2 Branching Strategy

As already mentioned Git can take advantage of the fact that it can create different branches for the project. The way the branches will be organized is fully inspired by the Agile Development approach [12], used by both big technical companies and startup ones. First a clear list of needed features has to be created. Then a time-line has to be organized of when each of those feature will be developed. In simpler words, every few requirements are grouped together and set a deadline when their implementation has to be finished. Each one of those groups is called a sprint. When the actual coding starts, for each new feature:

- A new branch is created in the format “feature/feature-name”
- When the requirement is finished, a request is made to merge it with the branch with a stable build, which is called “development”

The cycle repeats itself until all of the requirements are met.

3.10.3 Development Backlog

First of all, the requirements have to be broken down into a list of tasks. Specifically the list looks like this:

1. Connect with Vissim through COM
2. Load a Network and extract basic non-formatted Signal Controllers specifications from it
3. Create the structure of the JSON file
4. Extract Signal Plans from VAP files in JSON
5. Extract Signal Stages from PUA files in JSON
6. Using the JSON file, create PDDL problem file

7. Apply the changes from the planner to a network

The next step is to divide requirements into sprints. The concept of sprints is briefly described in the Branching Strategy subsection. The way sprints are created is by estimating how much time each of the tasks will take and calculating a rough amount of available resources the developer team posses. Smaller and relevant to each other tasks get grouped together in the same sprints, while bigger ones can be the sole feature of a whole sprint. As the development team of this project consists of one person, the sprints will be smaller than usual. Taking everything into consideration, this is the optimal conclusion that was made:

- Sprint 1: Task 1. and 2.
- Sprint 2: Task 3.
- Sprint 3: Task 4.
- Sprint 4: Task 5.
- Sprint 5: Task 6.
- Sprint 6: Task 7.

Chapter 4

Implementation

To give a better representation of how the project was implemented, I will follow the process of developing tasks sprint by sprint by describing implementation choices and any blockers.

4.1 Sprint 1 - COM Setup And Initial Data Extraction

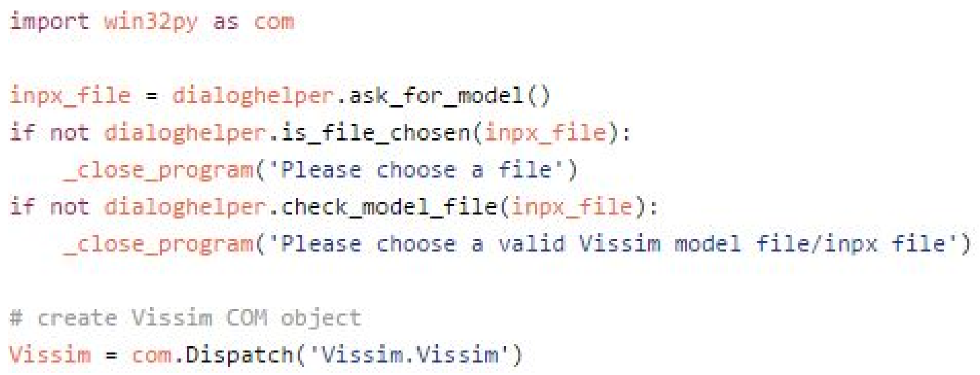
The first sprint is all about initial set up and testing the feasibility of the whole project. Tasks to be implemented:

- Connect with Vissim through COM
- Load a Network and extract basic non-formatted Signal Controllers specifications from it

4.1.1 Using Vissim's COM interface

First task for the project is finding a reliable way to communicate with the simulator. The standard library of Python does not come with support for COM interfaces. However, an external module called "PyWin32" offers this functionality. Installing it and putting it as a dependency was a straight forward process thanks to the python packet manager system, pip. Figure 4.1 shows an easy way of opening Vissim from a Python script. However, the initial student licence of the simulator did not support the COM functionality. That is how first blocker of the project was met and a purchase of a paid license was absolutely necessary for the project to progress.

Eventually the University provided a working license with COM support. The specified code finally ran without errors and communication between the script and the simulator was achieved. However, the license was working only with simple network models. Despite that, if extracting from those smaller networks is achieved, then working with bigger-scale ones should be successful too. The license also had the drawback that PTV Vissim can be run only inside King's College London network. Although, this means that full tests can't be run if working from home, the TDD approach can work around it by independently testing the functionality of each of the methods.



```
import win32py as com

inpx_file = dialoghelper.ask_for_model()
if not dialoghelper.is_file_chosen(inpx_file):
    _close_program('Please choose a file')
if not dialoghelper.check_model_file(inpx_file):
    _close_program('Please choose a valid Vissim model file/inpx file')

# create Vissim COM object
Vissim = com.Dispatch('Vissim.Vissim')
```

Figure 4.1: Sample code asking for network and opening PTV Vissim simulator

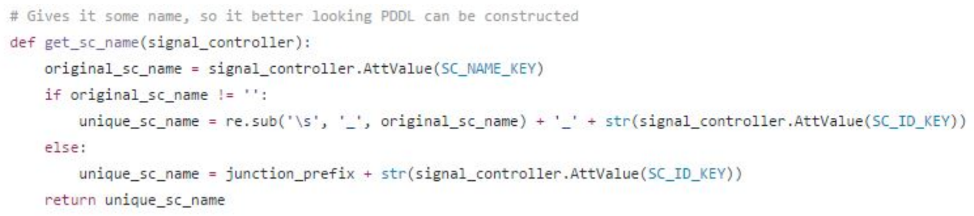
4.2 Sprint 2 - Create JSON structure

The objective of this implementation iteration was to come up with an optimal JSON structure and map the values extracted from the previous sprint to their corresponding keys. To satisfy the project's requirements and by familiarizing with the PDDL domain, here is a list of things it needs to satisfy my requirements:

1. Recognize separate junctions
2. Initial stage of each junction
3. Max stage of each junction
4. For each stage, which are the links that have green traffic light
5. How long is each stage
6. In what time does the last stage end, so the cycle of stages can restart

4.2.1 Junctions

PTV Vissim does not provide a direct way of identifying junctions. However, considering that signal groups can directly relate to each other only if they are on the same intersection, then signal controllers can be considered as junctions relatively safely, especially in the scope of this traffic signal project. What the planner needs to initialize a junction is only a unique name. Preferably the name has to be intuitive, so human operators can make sense of it when they see it mentioned in the generated PDDL problem file and the plan results. Moreover, it has to be possible for the purpose of the project's future work to map this unique name back to a specific signal controller, when results are exported back to the network. By observing the properties of the Vissim signal controller object, technically the only guaranteed to be unique attribute of the signal controller object is its id. However, as it is known from programming, giving variable a name that is or starts with a number is bad practice. That is why it was decided to serialize the signal controller's name and be passed later to the PDDL problem file. The decision was made with a few notes. The problems with that choice is that the attribute is marked as neither unique nor mandatory. That is why there are edge-cases where more than one signal controller will have the same name or it might not even be provided. To cope with both of the problems and also taking into mind the additional readability requirements the method shown on figure 4.2 was written to format the name. If the name attribute is not blank, then all whitespace characters are replaced with a "_" to cope with variable naming rules and also "_{id}" is appended to it, where the id is the unique identifier of the signal controller. And if the name is actually empty, then it is provided a placeholder of the format "__junction_{id}"



```
# Gives it some name, so it better looking PDDL can be constructed
def get_sc_name(signal_controller):
    original_sc_name = signal_controller.AttValue(SC_NAME_KEY)
    if original_sc_name != '':
        unique_sc_name = re.sub('\s', '_', original_sc_name) + '_' + str(signal_controller.AttValue(SC_ID_KEY))
    else:
        unique_sc_name = junction_prefix + str(signal_controller.AttValue(SC_ID_KEY))
    return unique_sc_name
```

Figure 4.2: Method that creates the name of the signal controller

4.2.2 Phases and Stages

After initializing the junctions, the planner also needs information about their stages: the initial one, the amount of them in total, which signal groups are green during each and where possible its duration too. For that to be possible first the locations of their VAP and PUA files have to be provided. Luckily, they are attributes of the signal controller object, hence accessible through it. Collection of phases is also that easily available. Signal groups do not have a direct access to the links they are operating on, but they might be extracted by following its collection of signal head objects, which provide that information. A graph is provided to give a better perspective of all the attributes discussed. However, mining the rest of the data will require a bit more effort by scrapping contents from the supply files.

4.2.3 JSON structure conclusion

Figure 4.3 shows the final structure of the JSON. The object will be an array of signal controllers. For each signal controller its id, type, name, VAP file location, PUA file location, the initial stage, the final stage, and stage timing changes will be provided. Also each controller will be associated with an array of signal groups. Each one of those phases will be represented by its id, its links, recognized with their names, and also in which stages will that phase be active (have a green light).

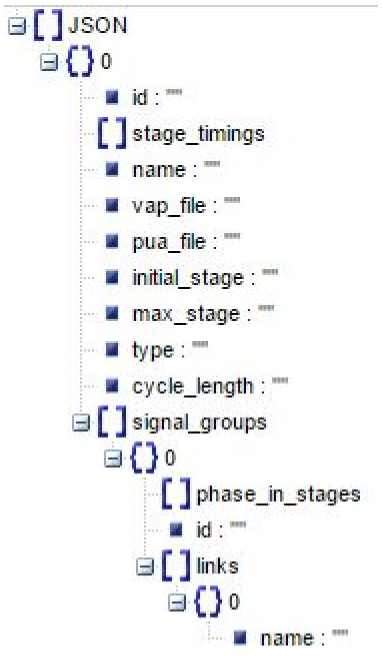


Figure 4.3: Structure of the JSON array

4.3 Sprint 3 - Extracting Data From VAP Files

The structure of the VAP files is described in section 4 of the Design and Specification chapter. The data that has to be extracted from those files is: stage timing schedule and cycle length. This information, however, is not available on every single network. An additional aspect thing to keep in mind too, is that the contents of the files is essentially a program, and it is known that programmers have different ways of structuring their code. That is why this part of the project will be working with just a very specific structure of the file, the one that TFL models are constructed with. Thanks to the fact the project is in collaboration with TFL, a small sample of networks created by them were provided. Even though the license of Vissim I was running is not able to run them, it was still possible to test the functionality of the implementation by creating and running unit tests, which embraces the Test-Driven Development approach.

4.3.1 Cycle Length Extraction

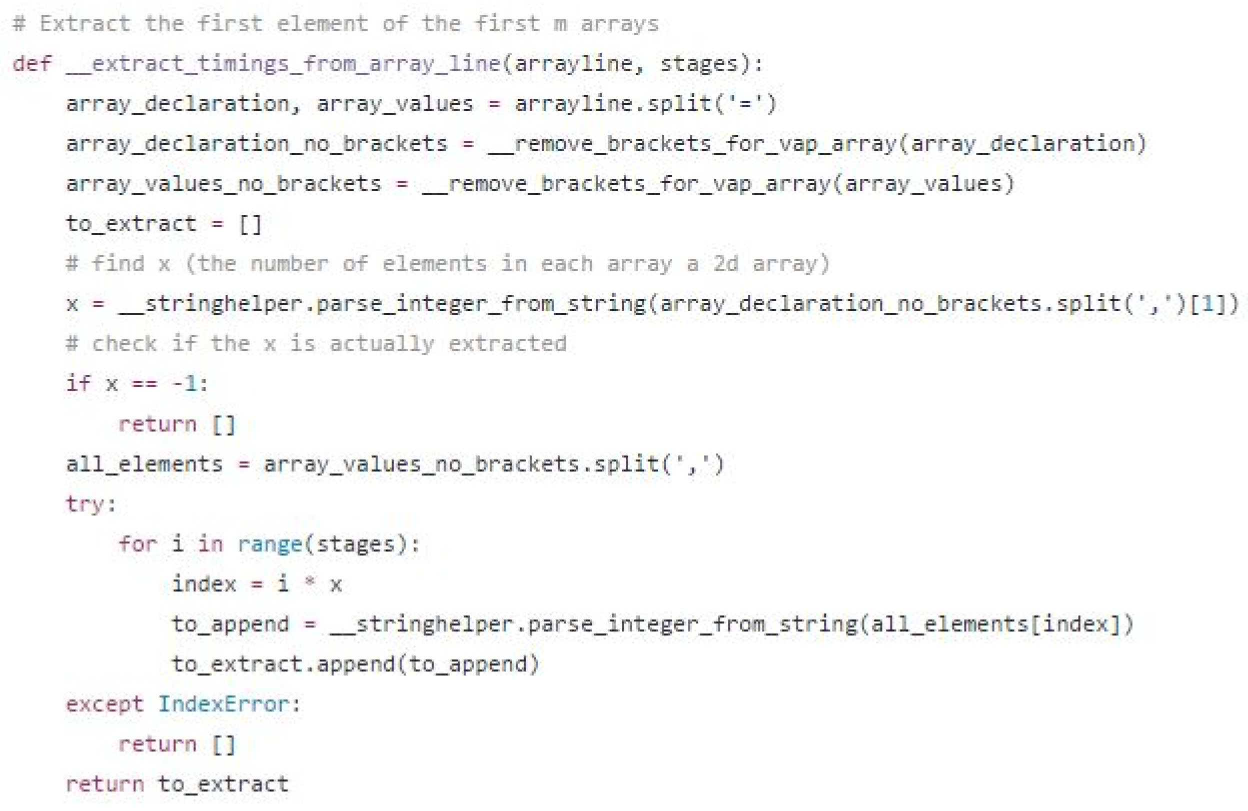
In the provided TFL models the cycle length was specified in the constants section, with format "CycleLength = some numeric value". That is why a function is called to scrape the lines of the file from the start of that section until the start of the next one. All the comment lines are disregarded and not added to that collection. The way a comment in VAP is made is with the line starting with "/*" and ending with "*/". All of those lines are then iterated until a regular expression of format "CycleLength =
d" is found. At the end the integer value is extracted and returned. If this information is not provided, then a negative value is returned as an indication of error.

4.3.2 Signal Timing Extraction

Again, TFL format-specific, the required data was put in a two-dimensional array, called Plan. The first element of the first m arrays is the time where the iterated stage, where m is the number of maximum stages. Just like the previous case, the lines of the whole ARRAY section, if found, and disregarding comments, are the first thing extracted. In those lines a regular expression of format "Plan [number, number] = [ ]" is looking for a match. If a match is made, then the arrays are abstracted and a collection is made of absolutely all items. Each first item is found by applying a formula with two variables, which are the size of the dimensions. This last function can be observed in the code snippet observed on figure 4.4.

4.3.3 Test Cases

The functionality of the methods was tested with a variety of differently-structured VAP files. The methods are expected to fail on cases where the files don't have content required or it is commented out. On other cases where the files have everything provided, the methods are expected to return sensible values.



```
# Extract the first element of the first m arrays
def __extract_timings_from_array_line(arrayline, stages):
    array_declaration, array_values = arrayline.split('=')
    array_declaration_no_brackets = __remove_brackets_for_vap_array(array_declaration)
    array_values_no_brackets = __remove_brackets_for_vap_array(array_values)
    to_extract = []
    # find x (the number of elements in each array a 2d array)
    x = __stringhelper.parse_integer_from_string(array_declaration_no_brackets.split(',')[1])
    # check if the x is actually extracted
    if x == -1:
        return []
    all_elements = array_values_no_brackets.split(',')
    try:
        for i in range(stages):
            index = i * x
            to_append = __stringhelper.parse_integer_from_string(all_elements[index])
            to_extract.append(to_append)
    except IndexError:
        return []
    return to_extract
```

Figure 4.4: Extract first elements from two-dimensional vap arrays

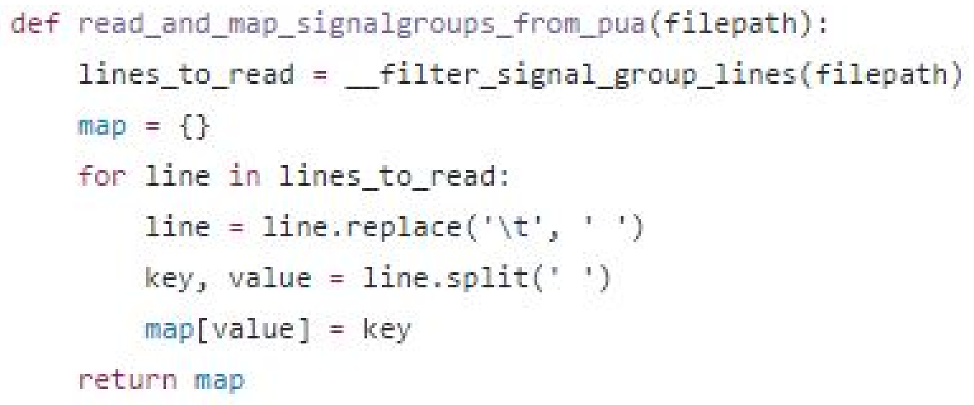
4.4 Sprint 4 - Extracting Data From PUA Files

Unlike VAP, PUA files have less freedom to be structured in different ways. Besides the design of interstages, which the project will abstract of anyways, the way content is organized is pretty consistent through all the simulator models. The data that has to be extracted from those files is: initial stage, maximum stage, what phases are active in what stages and what are the links corresponding to the phase. The approach to first extract only the lines of the files relevant to the wanted section was again followed, similarly to the VAP extraction methods. This time it was even easier as PUA does not allow comments inside the programming content areas. How a section is started is by the key of the section. After that there is the comment section. It ends when "$" is met. After that character every single line until the start of another section is regarded as instruction. The interstage section is an exception to the just defined rule, but it is disregarded for the purposes of the project anyways. From all the possible sections, the keys of the relevant ones for the projects are : "$SIGNAL_GROUPS", "$STAGES", "$STARTING_STAGE".

4.4.1 Signal Groups

The first section of each PUA file is where all the phases definitions are met. As both PUA and VAP constraint the identifier signal groups to start with a letter, the identifier that the simulator uses for it cannot be used as it is a number. In this first section of the file, a phase is declared by specifying their local name, only used in the PUA and VAP files, and also its corresponding id from the Vissim network. Those ids are separated by whitespace characters. It is essential to gather that pair so it is possible to identify the links that have green traffic lights for each of the stages, which data will be specified later.

What the method in figure 4.5 does is first extract the lines for the signal group section and then from that collection it filters out all the lines that are not in the right format. It has to be kept in mind that different developers have different opinions in terms of which whitespace characters should be used (intervals versus tabs). That is why the method converts those characters into only one kind, before splitting the line and constructing and putting in a map data structure the key-value pair of Vissim id : PUA id. At the end the whole constructed map data structure is returned.

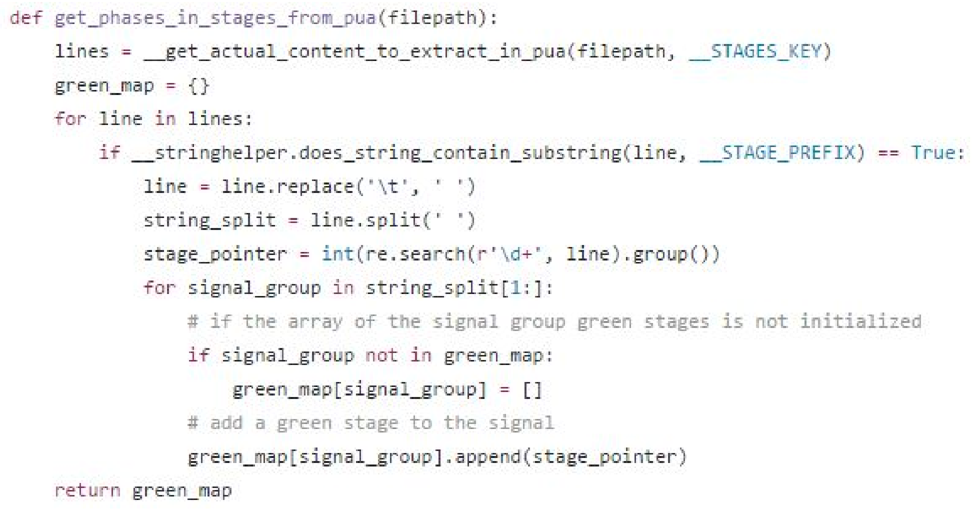


```
def read_and_map_signalgroups_from_pua(filepath):
    lines_to_read = __filter_signal_group_lines(filepath)
    map = {}
    for line in lines_to_read:
        line = line.replace('\t', ' ')
        key, value = line.split(' ')
        map[value] = key
    return map
```

Figure 4.5: Method that maps the Vissim to the local id pua uses

4.4.2 Phases In Stages

The next section is the one specifying all the stages. Along with its declaration in the format of “Stage_{stage-number}” it is also specified which of the phases are having green traffic light. It is optional to specify on the line after the declaration which of the phases have to be red, but even if that data is specified it will be disregarded as it is helpful only if interstages are considered in constructing the traffic plan, which is not the case for the current state of the system. Again, the execution of the method starts with getting all the lines from the desired section (“$STAGES”). Then it creates a map with keys being all the phase PUA keys, with corresponding value of a collection of all the stages the signal group should have green traffic light. Again different variations of whitespace are replace by only one, which is used to split the line into elements. At the end the constructed map is returned. The behaviour put in code is shown on the figure (4.6) below.

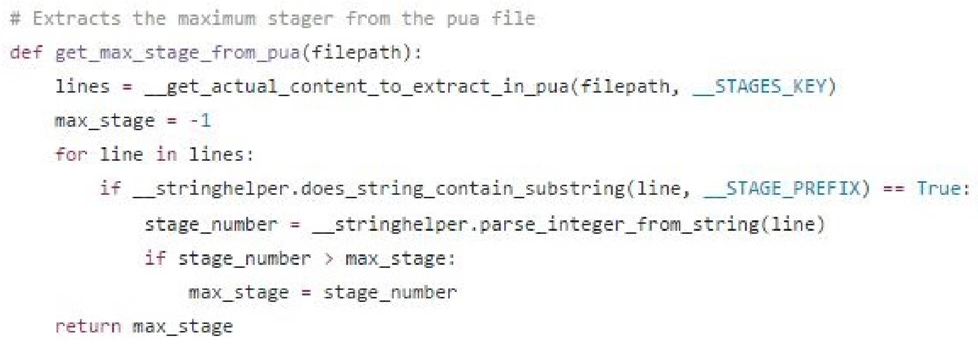


```python
def get_phases_in_stages_from_pua(filepath):
    lines = __get_actual_content_to_extract_in_pua(filepath, __STAGES_KEY)
    green_map = {}
    for line in lines:
        if __stringhelper.does_string_contain_substring(line, __STAGE_PREFIX) == True:
            line = line.replace('\t', ' ')
            string_split = line.split(' ')
            stage_pointer = int(re.search(r'\d+', line).group())
            for signal_group in string_split[1:]:
                # if the array of the signal group green stages is not initialized
                if signal_group not in green_map:
                    green_map[signal_group] = []
                # add a green stage to the signal
                green_map[signal_group].append(stage_pointer)
    return green_map
```

Figure 4.6: Method that extracts in what stages are phases active

4.4.3 Max stage

To extract the maximum stage again the STAGES section is observed. As the code snippet in figure 4.7 suggests, the method returns the number of the highest specified stage in this specific content area.

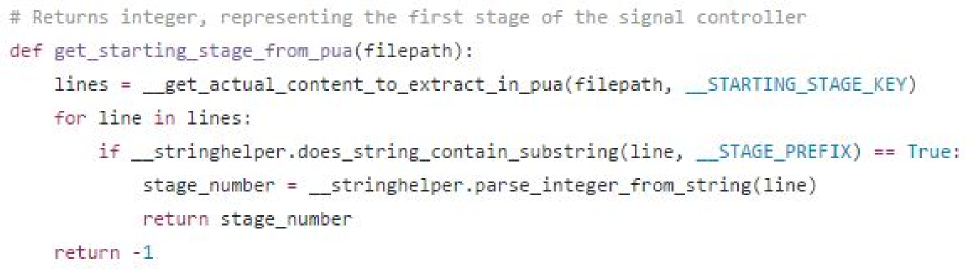


```python
# Extracts the maximum stager from the pua file
def get_max_stage_from_pua(filepath):
    lines = __get_actual_content_to_extract_in_pua(filepath, __STAGES_KEY)
    max_stage = -1
    for line in lines:
        if __stringhelper.does_string_contain_substring(line, __STAGE_PREFIX) == True:
            stage_number = __stringhelper.parse_integer_from_string(line)
            if stage_number > max_stage:
                max_stage = stage_number
    return max_stage
```

Figure 4.7: Method that extracts the max stage of a signal controller

4.4.4 Initial Stage

To extract the initial stage, the "$STARTING_STAGE" section is observed. Every time it consists of only one content line of format "Stage_{stage-number}". The integer from that line is extracted and returned. The procedure put into code is shown on figure 4.8.



```
# Returns integer, representing the first stage of the signal controller
def get_starting_stage_from_pua(filepath):
    lines = __get_actual_content_to_extract_in_pua(filepath, __STARTING_STAGE_KEY)
    for line in lines:
        if __stringhelper.does_string_contain_substring(line, __STAGE_PREFIX) == True:
            stage_number = __stringhelper.parse_integer_from_string(line)
            return stage_number
    return -1
```

Figure 4.8: Method that extracts starting stage of a signal controller

4.4.5 Test cases

To ensure correct functionality of the methods, they are tested against a big range of differently-structured PUA files. One of the test covers the edge-case where correctly-structured content is provided, but put in the comments section. For such files the expected behaviour to return either nothing or negative values, indicating no data is provided, depending on the type of the method's return value. The idea is that no matter what, the comments should be disregarded. Also, both tab and space used as indentation have been covered by the tests. Moreover, the behaviours are also tested for cases such as empty files, files with the keys needed but no content, and also few files that have correct.

4.5 Sprint 5 - Create PDDL problem file

After all the necessary data was gathered from the network, VAP and PUA files, it is put in a JSON file. How it works is that all the collecting is separated between different modules and put together in the main script. Besides calling the methods from the modules, what also this script does is to serialize all the data in a JSON structure and create a JSON file. With all of this done, the next step for the project is to create a PDDL problem file based on the collected data. The method that handles that requirement essentially translates a JSON file with a specific structure to PDDL. As already defined the JSON object is a collection of signal controllers. For each of them a PDDL object of type junction is made with its provided name. For each of those intersection these are the PDDL properties that have to be defined:

- (= (current_stage {junction}) {intial-stage})
- (= (max_stage {junction}) {max-stage})
- (= (phase_in_stage {link} {junction}) {stage-number})

Both the values of current and max stage are included in the JSON on keys “initial_stage” and “max_stage” respectively. For the phases in stages it has to go deeper in the JSON by following each signal group along with the names of the links they operate on.

If the values of the stage properties are negative or the collection of the phases is empty, then that means that the information was either not provided or not fetched properly. In this case a PDDL comment is made indicating that the information is not provided. The reason why this procedure was considered necessary is so human operators can effortlessly detect this kind of issues.

4.5.1 Test cases

To ensure correct translation from JSON to PDDL, the functionality of the involved methods is tested against one JSON file with all the values needed, and also one file with no or error values instead. The desired behaviour is on good values of certain keys, then it serves as a parameter to a corresponding PDDL statement. On the other hand, the error values no actual statements are printed, but a comment line indicating that the information needed was not provided.

4.6 Sprint 6 - Apply Changes From Planner Results

The last feature to be implemented involved making sense of the output of the planner and applying the proposed changes to the simulator network. As for now there is no way for the results of the planner to indicate which model they worked on. That is why besides providing the PDDL plan, the user will be asked also for the desired network to be changed. Before loading the network though, a method is called to process the PDDL file and export its content in a map with key-value pairs, where key is the name of the junction project, and value is a collection of integers, representing the proposed schedule of switching between the intersection’s stages. The way function works is to check every line of the plan file and look for a regular expression of format “{certain-time-in-seconds} : (switchtrafficsignal {junction-name})”. After that it abstracts itself from the irrelevant sections of the line and leaves only the time in seconds and the junction name. Then a map is created with a key of the intersection name and a value of a collection of the specified times mentioned for that junction. Here is the first catch - there is no way to be sure what the format of the junction name will be, that is why it is assumed that it will be the same structure it was defined on the section where the JSON structure was being implemented.

Assuming the previous step was successful, here comes the more catchy part. The signal timing changes proposed should be applied to the VAP file by editing its Plan array if there is one. It should also be considered that due to safety features, the original VAP file should not be destroyed if it has to be used again or to be compared with the new one. That is why a new file has to be made and not override the original one. After the user chooses a model to be modified, the script looks for the signal controller associated with the junction name from the PDDL. The way to do it is to work backwards from the implementation of making a unique name described previously. Assuming that all the signal controllers are found correctly, then the script accesses their VAP files. As already mentioned it is a good idea to not override the original file, that is why its contents are simply copied. The only thing that is changed is the Plan array. The way it does it is when copying this specific line it finds the first element of each of the arrays through a regular expression matching. Instead of putting the original values, they are substituted with the new values extracted from the plan results. The actual applying put in code is showed on figure 4.9.

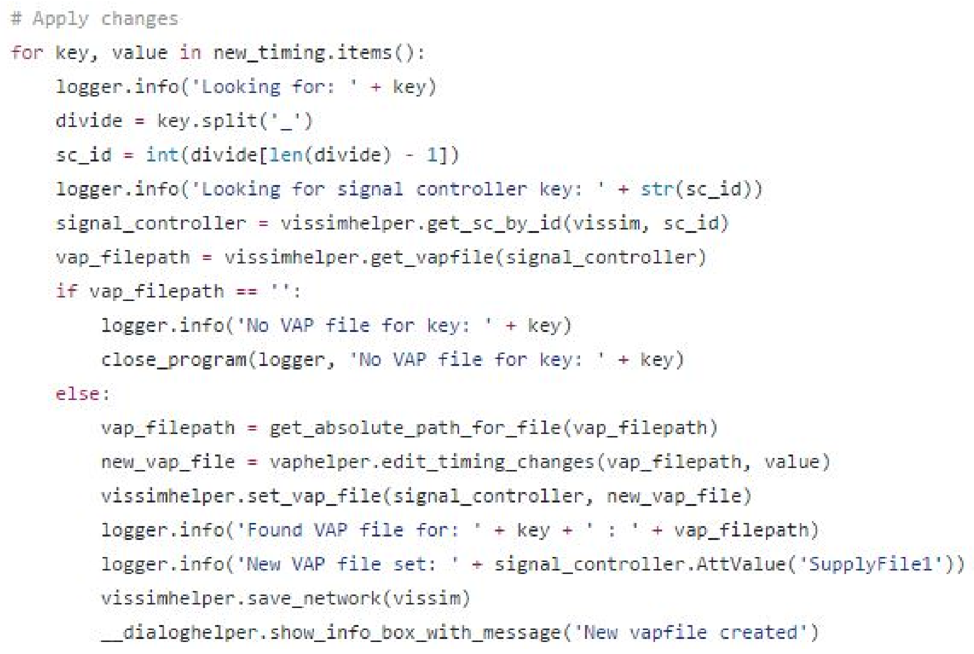

```
# Apply changes
for key, value in new_timing.items():
    logger.info('Looking for: ' + key)
    divide = key.split('_')
    sc_id = int(divide[len(divide) - 1])
    logger.info('Looking for signal controller key: ' + str(sc_id))
    signal_controller = vissimhelper.get_sc_by_id(vissim, sc_id)
    vap_filepath = vissimhelper.get_vapfile(signal_controller)
    if vap_filepath == '':
        logger.info('No VAP file for key: ' + key)
        close_program(logger, 'No VAP file for key: ' + key)
    else:
        vap_filepath = get_absolute_path_for_file(vap_filepath)
        new_vap_file = vaphelper.edit_timing_changes(vap_filepath, value)
        vissimhelper.set_vap_file(signal_controller, new_vap_file)
        logger.info('Found VAP file for: ' + key + ' : ' + vap_filepath)
        logger.info('New VAP file set: ' + signal_controller.AttValue('SupplyFile1'))
        vissimhelper.save_network(vissim)
        __dialoghelper.show_info_box_with_message('New vapfile created')
```

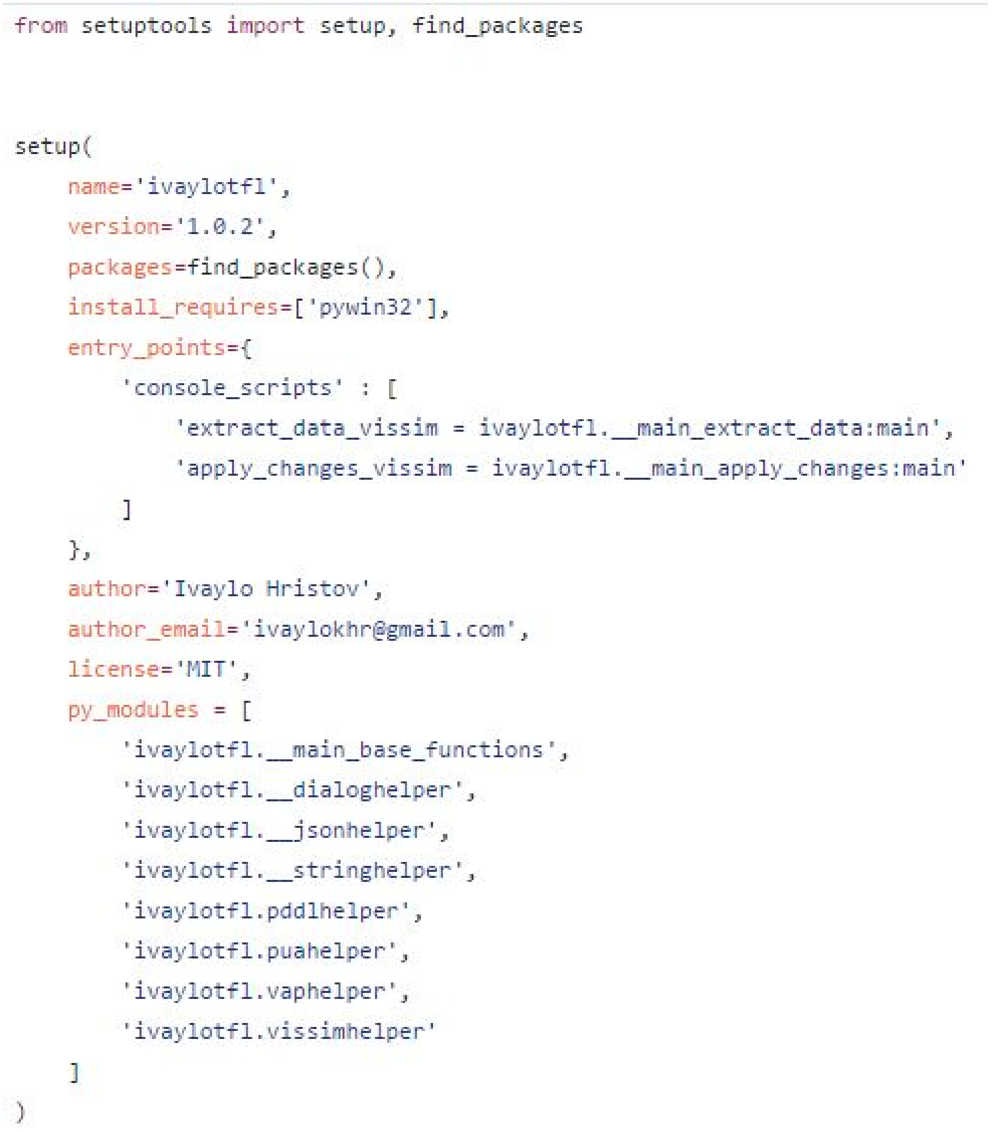
Figure 4.9: The method of applying the new timing

4.6.1 Test cases

To ensure correct changes to the model, the functionality of the methods that links each signal controller to its new timing schedule is tested against two files. One is with the expected structure of plan results. The other is another good sample file, but it is not results file, but a problem one. The tested behaviour is to return a correct map in the first scenario and raise an error on the second one.

4.7 Project Deployment

The actual last step of the project was to implement a way to deploy the project so it could be distributed easily through Python's packet manager. The task involved constructing and calling a specific script. Figure 4.10 shows the exact contents of that script. To simplify what it does, first a certain "setuptools" library has to be imported. It is included in the standard packages the Python installation comes with and it does not have to be downloaded from the manager. Then its "setup" method has to be called by providing all the required options you want to put. There are certain properties that are more important than the others and they have to be explained. First of all "name" is the name you are giving to your own project. This is the identifier which pip uses and it will be used by the users who want the work. "Install_requires" is another key option - it specifies all the dependencies you use in the project. In simpler words, it is a collection of all the used external libraries, which will have to be installed in addition. The "py_modules" specify the collection of all the python files implemented and used inside the package. Lastly, "entry_points" defines all the executables of the projects. It works as a key-value pair, where the key is a command, which will call the script file or some specific method put in the value. That way after someone downloads the project through pip, it will take him only one more command to start a script. In this case the commands are "extract_data_vissim" and "apply_changes_vissim" which will call the "main" method of respectively the "__main_extract_data" and "__main_apply_changes" Python script files.



```python
from setuptools import setup, find_packages


setup(
    name='ivaylotfl',
    version='1.0.2',
    packages=find_packages(),
    install_requires=['pywin32'],
    entry_points={
        'console_scripts' : [
            'extract_data_vissim = ivaylotfl.__main_extract_data:main',
            'apply_changes_vissim = ivaylotfl.__main_apply_changes:main'
        ]
    },
    author='Ivaylo Hristov',
    author_email='ivaylokhr@gmail.com',
    license='MIT',
    py_modules = [
        'ivaylotfl.__main_base_functions',
        'ivaylotfl.__dialoghelper',
        'ivaylotfl.__jsonhelper',
        'ivaylotfl.__stringhelper',
        'ivaylotfl.pddlhelper',
        'ivaylotfl.puahelper',
        'ivaylotfl.vaphelper',
        'ivaylotfl.vissimhelper'
    ]
)
```

Figure 4.10: Implementation of a script that deploys the project and hosts it on pip

Chapter 5

Professional and Ethical Issues

While working on the project I have taken into consideration that it has to meet the professional and ethical standards suggested by the Code of Conduct & Code of Good Practice issued by the The British Computer Society[1]. I have applied those principles in my project, where it is appropriate.

For the project to work, a Python module called ”PyWin32” was installed. The library is under Python Software Foundation License and its use in the project has been conducted accordingly to its terms and conditions.

Chapter 6

Evaluation And Future Work

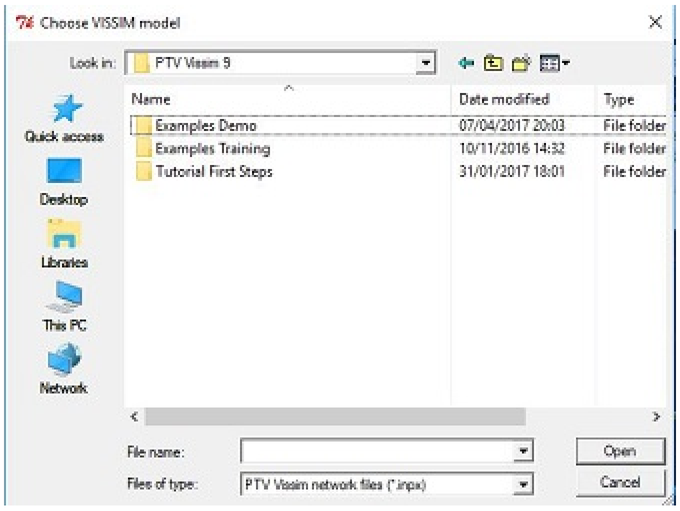
This chapter is going to evaluate the success of the work by going through the degree of satisfaction of each of the requirements set earlier in the paper.

6.1 Simulator Data Extraction Requirements Evaluation

6.1.1 Communication With The Simulator

This first requirement was all about enabling a way for the script to communicate with the simulator so it can export or modify its information. Additionally,

- The user has to be able to choose a traffic model to work with, which will be loaded by a simulator software.



Name	Date modified	Type
Examples Demo	07/04/2017 20:03	File folder
Examples Training	10/11/2016 14:32	File folder
Tutorial First Steps	31/01/2017 18:01	File folder

File name:
Files of type: PTV Vissim network files (*.inpx)

Figure 6.1: A dialog asking the user to load a model

The requirement was fully satisfied. As shown in figure 6.1, the first thing the user sees when he launches the script is a window asking him about the model he wants to load. The default filter is set to be “*.inpx” as this is the default extension of the Vissim’s networks. If the user chooses a valid mode, then through the COM interface, given that PTV Vissim is installed on the machine, the simulator is opened and it starts loading the chosen model. As the COM interface is Microsoft-Windows-specific, this requirement alongside the whole project will only work on this specific operating system.

6.1.2 Collect Signal Timing Plan Data

For the implementation of both of the desired features specified below data is put in the supplied VAP file of the signal controller. It is assumed that those files were created by TFL and their own guidelines have been followed.

1. **Signal timing plan cycle length has to be automatically extracted for each of the junctions**

   In order for the information to be extracted, then a variable called CycleLength has to be initialised in the VAP file of the signal controller. If this is not the case a negative value will be exported, indicating an error. From the samples I have been given, all TFL models work under the described constraints. If it is indeed a TFL model and there is no such variable, than that should mean that this information was not needed in the first place.

2. **Signal timings (schedule of stage changes) has to be automatically extracted for each of the junctions**

   This information is available in a two-dimensional array called Plan in the VAP file, given the signal controller is of type “VAP”. Else, an empty array will be outputted as the value of this property.

Both of the behaviours have been extensively unit-tested against a variety of differently-structured files. As a conclusion, the requirements can be considered as satisfied.

6.1.3 Collect Stages Data

All the information needed in the listed below requirements is present in the other kind of supply files - PUA. In contrast to the VAP ones, the structure of those files do not vary that much. If the information needed is not present it indicates only that the structure is malformed and the simulation won't work anyways.

1. **The initial stage has to be automatically extracted for each of the junctions**

   If the information is provided, it will be exported. If not, the outputted value will be negative, indicating an error.

2. **The maximum stage has to be automatically extracted for each of the junctions**

   The script manages to count the number of stages and return it. On error, a negative value is returned.

3. **Which phases are active for each of the stages has to be automatically extracted for each of the junctions**

   The script maps the active signal groups for each of the stages. It also links those signal groups to specific links as required from the PDDL domain. If there is no information in which stage the signal group is active, no matter if that was intended or not, an empty array is outputted.

All those requirements are represented by functions, whose behaviour were unit-tested extensively against a variety of differently structured PUA files. As conclusion, all of them should be considered as satisfied.

6.1.4 Structure The Information In JSON

What the actual requirement says is:

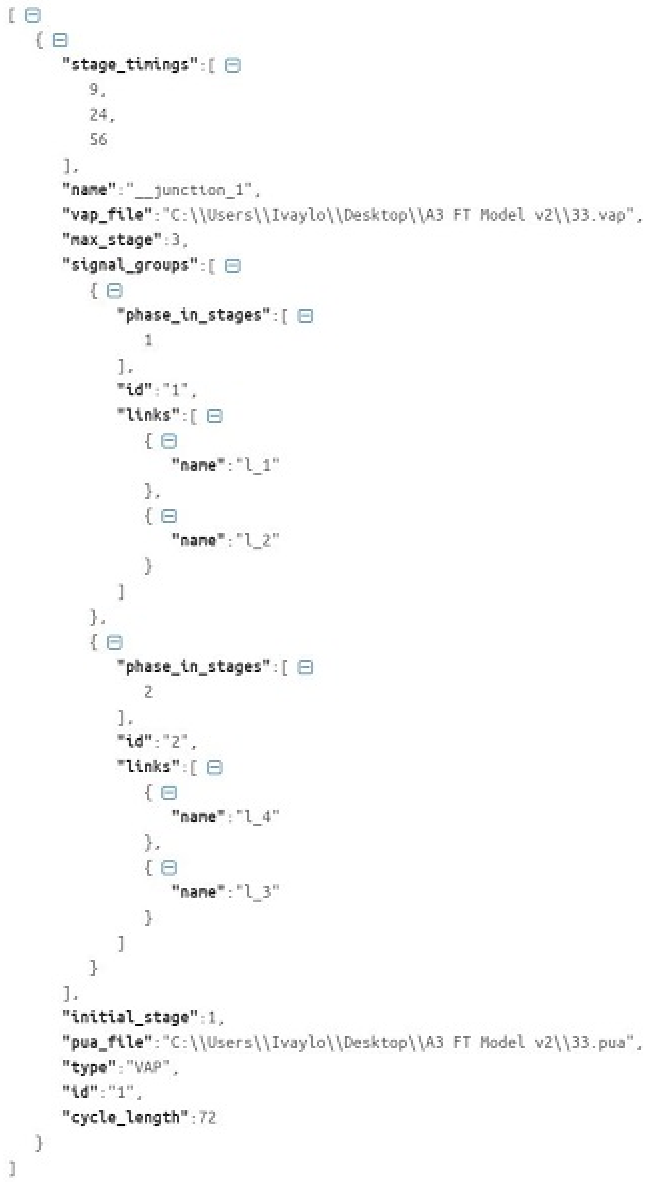
- All the gathered data has to be collected and organized in an intermediate form

The file is created in the folder of the chosen model. Figure 6.2 shows such JSON file filled with all the collected data from the previous requirements. Thus the requirement is fulfilled.

6.1.5 Create PDDL Problem File

The final requirement of this part is:

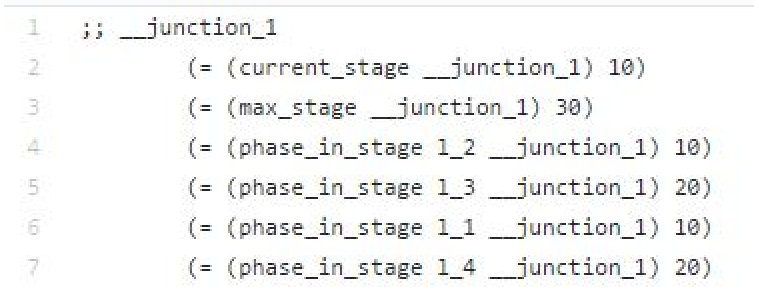
- A PDDL problem file has to be created in a location and filled with the collected data in an appropriate format corresponding to the PDDL domain



```
[
   {
      "stage_timings":[
         9,
         24,
         56
      ],
      "name":"__junction_1",
      "vap_file":"C:\\Users\\Ivaylo\\Desktop\\A3 FT Model v2\\33.vap",
      "max_stage":3,
      "signal_groups":[
         {
            "phase_in_stages":[
               1
            ],
            "id":"1",
            "links":[
               {
                  "name":"l_1"
               },
               {
                  "name":"l_2"
               }
            ]
         },
         {
            "phase_in_stages":[
               2
            ],
            "id":"2",
            "links":[
               {
                  "name":"l_4"
               },
               {
                  "name":"l_3"
               }
            ]
         }
      ],
      "initial_stage":1,
      "pua_file":"C:\\Users\\Ivaylo\\Desktop\\A3 FT Model v2\\33.pua",
      "type":"VAP",
      "id":"1",
      "cycle_length":72
   }
]
```

Figure 6.2: A sample of data put in JSON

Effectively, it involved translating the collected JSON data from network to PDDL. After the user chooses desired location and name of the file as shown on Figure the file is created and filled with content. Example PDDL file is shown on figure 6.3. The behaviour was tested through unit tests, making sure the values from the right JSON keys are correctly formatted in PDDL. Thus the requirement is satisfied.



```
;; __junction_1
        (= (current_stage __junction_1) 10)
        (= (max_stage __junction_1) 30)
        (= (phase_in_stage 1_2 __junction_1) 10)
        (= (phase_in_stage 1_3 __junction_1) 20)
        (= (phase_in_stage 1_1 __junction_1) 10)
        (= (phase_in_stage 1_4 __junction_1) 20)
```

Figure 6.3: A sample of constructed PDDL problem file

6.2 Applying Plan Changes Requirements Evaluation

The second part of the project involved of translating plan results and applying changes to the simulation model.

6.2.1 Choosing Files To Work With

The first two requirements of this part are:

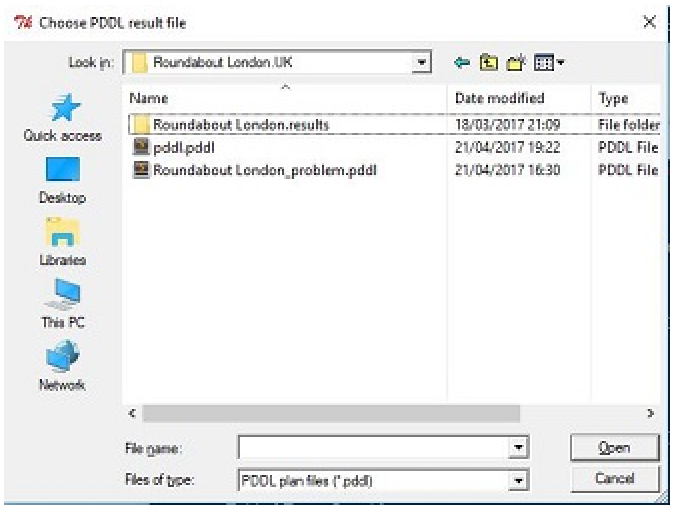
- The user has to be able to choose a traffic PDDL plan results file
- The user has to be able choose the model which he wants to edit

The first thing the user sees is a dialog asking him to provide a PDDL results file like in figure 6.4 . When the script verifies the file is correct, by looking for a specific content, the user will get another prompt like in figure 6.5. As such, the requirements can be considered satisfied.

6.2.2 Recognising Changes

The key requirements of this specific part of the project are:

- The script has to recognise the actions of the planner's results
- The script has to be able to recognise the involved objects and apply the desired changes to them


Choose PDDL result file

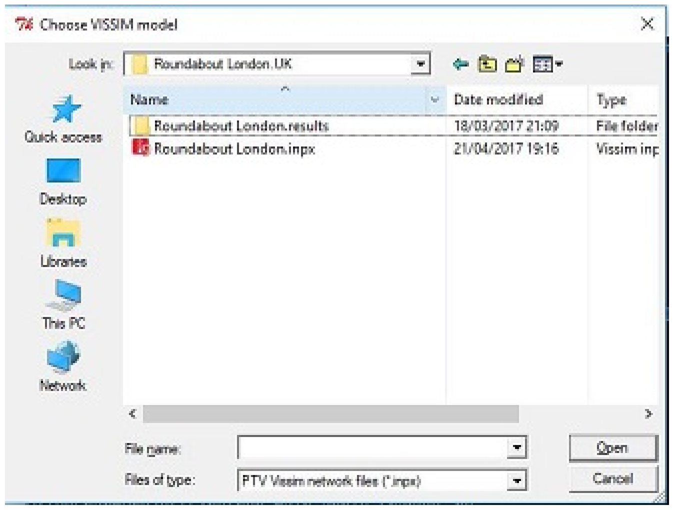
Look in: Roundabout London.UK

Name	Date modified	Type
Roundabout London.results	18/03/2017 21:09	File folder
pddl.pddl	21/04/2017 19:22	PDDL File
Roundabout London_problem.pddl	21/04/2017 16:30	PDDL File

File name:

Files of type: PDDL plan files (*.pddl)

Figure 6.4: Dialog asking for a PDDL results file to be provided


Choose VISSIM model

Look in: Roundabout London.UK

Name	Date modified	Type
Roundabout London.results	18/03/2017 21:09	File folder
Roundabout London.inpx	21/04/2017 19:16	Vissim inp

File name:

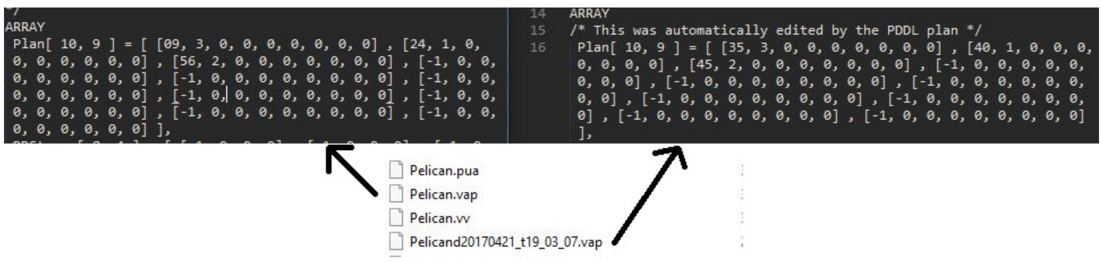
Files of type: PTV Vissim network files (*.inpx)

Figure 6.5: Dialog asking which Vissim network should be edited

The behaviour is tested against a sample results file and also a file that does not have the desired content. On the first case, a correct information is derived, while for the other scenario an error is raised. It is however mandatory that the results file was produced for a problem file created by the other part of the project. But considering that there is no other feasible way to recognise the network objects involved in the plan file, this requirement can be considered fulfilled.

6.2.3 Recognising Changes

The last requirement of the whole project is considering not losing unwanted files. Specifically it says:



```
*/
ARRAY
 Plan[ 10, 9 ] = [ [09, 3, 0, 0, 0, 0, 0, 0, 0] , [24, 1, 0,
 0, 0, 0, 0, 0, 0] , [56, 2, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0,
 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0,
 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0,
 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0,
 0, 0, 0, 0, 0, 0] ],
```

```
ARRAY
/* This was automatically edited by the PDDL plan */
 Plan[ 10, 9 ] = [ [35, 3, 0, 0, 0, 0, 0, 0, 0] , [40, 1, 0, 0, 0,
 0, 0, 0, 0] , [45, 2, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0,
 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0,
 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0,
 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0] , [-1, 0, 0, 0, 0, 0, 0, 0, 0]
 ],
```

```
Pelican.pua
Pelican.vap
Pelican.vv
Pelicand20170421_t19_03_07.vap
```

Figure 6.6: New and original file existing together

- The script should not break previous build of the model to be changed

It means that the newly created VAP files content should not destroy their previous versions. Figure 6.6 shows how the script created a new file with the desired new plan and associated the signal controller with it without deleting the original file. This should satisfy the requirement fully.

6.3 Project Obtainment

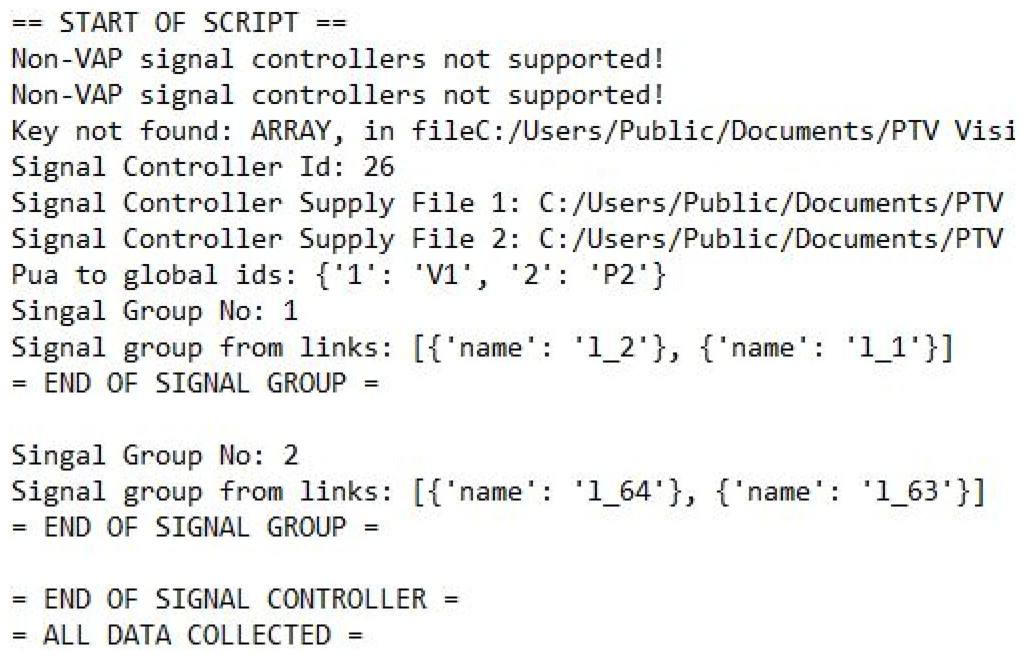
The problem of finding a way to acquire a hold onto the scripts fast and easy was solved by making it possible to download it through the Python package manager. This way the work can be run with only two command lines in terminal environment, assuming that it has installed Python and is able to download its only dependency "PyWin32", which will run only on Windows, as the COM interface is only supported by this specific operating system.

6.4 Project Scalability

With the way project was done, it has a potential to stand the test of time. All the requirements of the scripts have been met and also collecting data in an intermediate form will enable operators to use the data for purposes outside of the current domain. What else has to be considered is that when the project is downloaded through the Python package manager, not only is there an easy access to its scripts, but also the implemented methods will be available, thus it will be possible to use them to solve a solution for a future problem the system will face.

6.5 User Experience

The project was created taking in mind to provide positive experience to the end-user. Besides interaction on choosing to open and save files, the scripts generate log files during their execution. Those files follow all the program's instructions and provides some sensible representation of them. This way if there is a scenario, where the script does something unexpected, the human operator can track the error and fix the problem that is causing it. Example is shown on figure 6.7, where the fetched data is displayed, while also noting some signal controllers are not type "VAP", thus nothing was serialized for them.



```
== START OF SCRIPT ==
Non-VAP signal controllers not supported!
Non-VAP signal controllers not supported!
Key not found: ARRAY, in fileC:/Users/Public/Documents/PTV Visi
Signal Controller Id: 26
Signal Controller Supply File 1: C:/Users/Public/Documents/PTV
Signal Controller Supply File 2: C:/Users/Public/Documents/PTV
Pua to global ids: {'1': 'V1', '2': 'P2'}
Singal Group No: 1
Signal group from links: [{'name': 'l_2'}, {'name': 'l_1'}]
= END OF SIGNAL GROUP =

Singal Group No: 2
Signal group from links: [{'name': 'l_64'}, {'name': 'l_63'}]
= END OF SIGNAL GROUP =

= END OF SIGNAL CONTROLLER =
= ALL DATA COLLECTED =
```

Figure 6.7: Log file sample content

6.6 Future Work

Although all the requirements have been met, there are aspects of the work that can be improved as the part of the future work.

6.6.1 Better Error Handling

Although the script can handle some scenarios where something unexpected happened, the situation needs to be further improved. As an example if the user has no Vissim installed on their computer, they will be warned after choosing a model file and eventually the program will close itself. However, if the simulator is installed, but it is not able to connect to its license provider software, the program will not close itself correctly and no additional warnings are given of why it did not work.

6.6.2 Better User-Friendliness

To a certain extent functionality that the scripts provides are quite obvious, especially for a professional user who knows what to expect from it. The newly created files have appropriate names indicating they were not there before and also when change is made to the VAP files a comment is made revealing that the next line is created automatically. However, a few dialog providing certain options, e.g. “Open newly created file”, could save some time and enable better user experience.

6.6.3 Export In Different Formats

JSON works fine, but it might be in the case in the future where a certain framework that will collaborate with the system will not support JSON, but XML for instance. In that case it will not be that hard to translate quickly JSON to XML, hence there are definitely free solutions to the problem on the Internet, but it will be good to remove any necessary overhead by implementing those features natively.

6.6.4 Graphical Interface

If additional time was available for this project, I would consider making a full graphical user interface giving as much information and control as possible to the user. Everything that is happening should be displayed. Right now when the script does some computations in the back, the user is not notified on this. An additional feature of the interface could be to provide a list of options and flags - basically to be up to the user to select what exact data they want extracted and in what format. Or in context to the second part of the project, a dialog might be put after the application of the changes have finished to ask if a simulation should be started.

6.6.5 More Test Cases

This is probably the most important future task. Currently the functionality of the project is tested with sample networks provided from Vissim training demos, some of which were altered by creating signal controller with the supply files (PUA and VAP) provided by TFL. However, there has not been a full run of the scripts with an actual whole TFL network or actual plan results. The reason is that the Vissim’s software license used as environment throughout developing the project did not support the scale of the TFL models.

References

[1] Bcs code of conduct. URL `http://www.bcs.org/category/6030`.

[2] Traffic advisory leaflet. 1995. URL `http://www.ukroads.org/webfiles/tal04-95.pdf`.

[3] K. Beck. *Test-driven Development: By Example.* 2003.

[4] S. A. Boxill and L. Yu. An evaluation of traffic simulation models for supporting its development, 2000.

[5] T. de Castella. Does pressing the pedestrian crossing button actually do anything? September 2013. URL `http://www.bbc.co.uk/news/magazine-23869955`.

[6] M. Fox and D. Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27, 2006.

[7] M. Fox, D. Long, and D. Magazzeni. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 44, June 2012.

[8] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl - the planning domain definition language, October 1998.

[9] P. Group. Ptv vissim - modules. URL `http://vision-traffic.ptvgroup.com/fileadmin/files_ptvvision/Downloads_N/0_General/2_Products/2_PTV_Vissim/EN_PTV_Vissim_Modules.pdf`.

[10] D. Krajzewicz, G. Hertkorn, P. Wagner, and C. RÃűssel. Sumo (simulation of urban mobility) an open-source traffic simulation, 2002.

[11] B. Peksag. Should i use python 2 or python 3 for my development activity?, 2017. URL `https://wiki.python.org/moin/Python2orPython3`.

[12] D. Radigan. The agile coach: Feature branching your way to greatness. URL `https://www.atlassian.com/agile/branching`.

[13] G. Roberts. Uk drivers spend 30 hours on average in congestion. 2016. URL `http://www.fleetnews.co.uk/news/fleet-industry-news/2016/03/15/uk-drivers-spent-30-hours-in-congestion-in-2015`.

[14] D. I. Robertson and R. D. Bretherton. Research on the transyt and scoot methods of signal coordination. *ITE JOURNAL*, January 1986.

[15] D. I. Robertson and R. D. Bretherton. Optimizing networks of traffic signals in real time-the scoot method. 1991.

[16] S. F. Smith, G. J. Barlow, X.-F. Xie, and Z. B. Rubinstein. Smart urban signal networks: Initial application of the surtrac adaptive traffic signal control system, 2013.

[17] L. Torvalds and J. Hamano. Git: Fast version control system, 2010. URL `https://git-scm.com/`.

[18] M. Treiber and A. Kesting. Traffic flow dynamics. *ITE JOURNAL*, 2013.

[19] M. Vallati. Ai planning for urban traffic control: Moving from objects to flows. 2016.

[20] X.-F. Xie, S. F. Smith, and G. J. Barlow. Schedule-driven coordination for real-time traffic network control. 2012.