POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMATICA E BIOINGEGNERIA
MASTER THESIS

ADAPTIVE CAR NAVIGATION SYSTEM FOR SMART CITIES

M.Sc Thesis of:
Leonardo Arcari

Supervisor:
Prof. Gianluca Palermo

Academic year 2018/2019

Abstract

CAR NAVIGATION SYSTEM technologies recently established across the globe as an imperative utility for modern navigation on road networks. The rising wave of self-driving cars along with an increasing demand for real-time traffic data is expected to generate massive growth in the number of routing requests and processing on large graphs representing the urban network. This trend imposes larger and more powerful computing infrastructures composed of HPC resources. In the context of smart cities, new, dynamic solutions are required in order to deliver high-quality car navigation services, powered by municipal traffic monitoring data, capable of handling such a vast expected demand with reasonable employment of financial resources.

In this thesis, an adaptive car navigation system for smart municipalities is presented, proposing a methodology to approach navigation services design along with capacity planning analysis through system modelling. An efficient, configurable C++ library, named *ARLib*, is introduced, implementing state-of-the-art algorithms addressing the Alternative Route Planning (ARP) problem, a key part of car navigation systems. In this context, *ARLib* exhibits its flexibility by exposing a variety of parameters to manage the tradeoff between execution time and quality of results. Moreover, since no ultimate solution exists, an auto-tuning process is presented to proactively choose the optimal heuristic for each query, leveraging a predictive model providing recommendations based on request features. This gives the possibility to have the highest quality reachable within the available latency budget. Furthermore, a Queueing Petri Net model of the car navigation system is proposed to study its behavior from performance evaluation perspective in a computer simulation environment. This aspect helps to assess and size the infrastructure for the navigation service deployment through capacity planning analysis. Finally, an empirical evaluation of the presented methodology is reported, by applying the advised approach in a case study on the city of Milan.

Estratto in Lingua Italiana

I SISTEMI DI NAVIGAZIONE, negli ultimi anni, si sono affermati in tutto il mondo come uno strumento fondamentale per la navigazione moderna sulla rete stradale. Il fenomeno in aumento delle auto a guida autonoma, insieme alla sempre più crescente domanda di aggiornamenti sul traffico in tempo reale, fanno prevedere una crescita imponente del numero di richieste di percorsi e di elaborazioni su enormi grafi che rappresentano le reti stradali cittadine. Questo trend impone la presenza di impianti informatici più grandi e potenti, composti da risorse HPC. Nell'ambito delle Smart City, al fine di fornire servizi di navigazione di qualità, sfruttando i dati cittadini di monitoraggio del traffico, sono necessarie soluzioni nuove e dinamiche, capaci di gestire una tale domanda prevista con un impiego ragionevole di risorse finanziarie.

In questo lavoro di tesi, si presenta un sistema di navigazione adattivo per Smart City, proponendo una metodologia per la progettazione di un servizio di navigazione insieme ad un'analisi per il dimensionamento delle risorse attraverso un modello del sistema. Si introduce una libreria C++ efficiente e configurabile, chiamata *ARLib*, che implementa alcuni algoritmi dallo stato dell'arte per risolvere il problema della Pianificazione di Percorsi Alternativi, una componente chiave dei sistemi di navigazione su strada. In questo contesto, *ARLib* mostra la sua flessibilità esponendo una serie di parametri per gestire il tradeoff tra tempo di esecuzione e qualità del risultato. Inoltre, poiché non esiste un approccio definitivo al problema, si presenta un processo di auto-tuning per scegliere automaticamente, in maniera proattiva, l'euristica ottimale per ogni query, sfruttando un modello predittivo che fornisce raccomandazioni basate sulle caratteristiche delle richieste. Questo permette di ottenere la massima qualità raggiungibile nel tempo di calcolo a disposizione. In aggiunta, si propone un modello del sistema di navigazione con reti di code e reti di Petri combinate, per studiare il comportamento del servizio dal punto di vista delle performance in un ambiente di simulazione. Ciò permette di valutare e dimensionare l'infrastruttura adeguata ad erogare il servizio di navigazione attraverso capacity planning. In ultimo, si riporta una validazione empirica della metodologia presentata, applicando l'approccio consigliato ad un caso studio sulla città di Milano.

Ringraziamenti

RITAGLIO questo umile spazio, fra le pagine di un arduo lavoro di tesi, oramai voltosi al termine, per esprimere in *lingua natia* un profondo ringraziamento alle persone che, con la loro vicinanza, hanno dato valore al tempo speso in questa fatica. A ciascuno di voi dedico un breve pensiero, ben sapendo quante e quali storie ci hanno legati, storie che tuttavia richiederebbero un'intera antologia per poter essere degnamente ricordate.

Ringrazio i miei genitori, che da tutta la vita, con il loro affetto, mi hanno spronato e supportato nei miei sogni, donandomi i mezzi e le occasioni per seguirli. A voi devo la persona che oggi sono.

Ringrazio il mio relatore, Gianluca, per la fiducia e le opportunità che mi ha offerto durante questo anno di ricerca, fornendomi il suo prezioso aiuto senza il quale questo lavoro non mi avrebbe dato la soddisfazione che oggi provo.

Ringrazio mia sorella Elena, mio fratello Daniele, e i miei nonni Selene, Santino e Giovanna per la loro vicinanza e la pazienza nell'ascoltarmi, per un anno, vaneggiare di scoperte e piccoli traguardi.

Ringrazio gli amici e fratelli di una vita, Francesco, Umberto, Paolo e Carlotta, per avermi dimostrato con la loro presenza che un'amicizia autentica trascende le difficoltà e accompagna, inarrestabile, i passi della tua esistenza.

Ringrazio gli amici e compagni di corso, Emiliano, Emanuele, Edoardo, Samuele, Andrea, Leandro, Andrea e Michele, per avermi spronato nell'*ars ingegneristica* e aver reso gli anni al Politecnico un felice ricordo da portarmi per il resto della vita.

Ringrazio gli amici e compagni d'Appa, Gio, Michelle, Peppo, Samsung, Ssette e Sbrisolino per essere stati la mia casa negli anni vissuti a Milano, per non avermi fatto sentire solo per un istante e aver lasciato un'impronta indelebile nella mia storia.

In ultimo, ringrazio in maniera speciale la mia ragazza, Giulia, per avermi sostenuto e rincuorato negli attimi più cupi e aver dato pienezza, con la sua presenza, ai momenti più felici che questo anno mi ha regalato.

Leonardo
Milano, 29 Marzo 2019

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In recent times, car navigation system technology established itself as an imperative utility used across the globe as an essential tool for modern navigation on land. Car navigation systems are nowadays pervasive in people lives, either because directly installed by their car manufacturer, either because purchased from after-market solutions or simply because available as GPS navigation apps on everybody's smartphone.

According to some research [1], global automotive navigation systems market size is expected to observe an exponential growth in the next eight years, with an anticipated compound annual growth rate of 9.95% during the forecast period. Most important driving factors are the increase in the number of vehicles worldwide and raising demand for real-time traffic data.

Indeed, both passenger and commercial vehicle drivers share the same interest to reach their destinations as soon as possible. In a world where less and less daily activities can be accomplished by moving on foot, the former would rather spend as less time as possible in their cars, transferring from one place to another. On the other hand, driving faster to destination means for the latter delivering a better service to their customers and raising their generated daily revenue.

In both cases, people are putting more and more trust in car navigation systems, relying on computer-suggested routes as opposed to their driving experience. Such a scenario will be even more true with the expected worldwide spread of self-driving cars. Indeed, drivers will eventually turn into actual passengers, leaving to their car, and hence to its car navigation system, the burden to choose a path for them.

In this framework, several opportunities arise in order to transform the current traffic situation for the better and leverage traffic data availability to optimize driving conditions.

For instance, smart cities might employ their historical traffic monitoring data to provide their citizens with an accurate car navigation service that, at the same time, would make their local drivers happier and the whole city less congested.

From an algorithmic perspective, the problem of finding the best path in terms of expected arrival time can be solved into two subsequent steps: find a set of alternative routes from source to destination points and use probabilistic distributions of segments travel time to estimate the fast path to follow. These two steps are known in the literature as Alternative Route Planning (ARP) [2, 3] and Probabilistic Time Dependent Routing (PTDR) [4, 5] problems.

An efficient methodology for a self-adapting PTDR solution was extensively investigated in other works [6]. In this thesis, I am going to focus on the ARP phase, presenting an original methodology for an adaptive alternative route planning module of a car navigation system.

Furthermore, considering the rising wave of self-driving cars, and automotive navigation systems in general, the number of navigation requests will increase rapidly together with the need of real-time updates and processing on large graphs representing the urban network. This trend imposes larger and more powerful computing infrastructures composed of HPC resources.

In this context, this thesis also presents a car navigation system model representing the full service deployed in an HPC environment.

Motivation

Given the complexity in efficiently solving the alternative route planning problem, researches have been investigating several approaches to discover a one-fits-all solution.

Indeed, while computing a single shortest path is a well-defined problem, with a solid historical background, most notably represented by Dijkstra's Algorithm, ARP problem has been formulated in several ways. Moreover, since finding an exact solution to the ARP problem was demonstrated to be computationally intractable, multiple heuristics have been presented so far.

However, heuristics typically work by introducing some assumptions in order to simplify the problem and efficiently search for a solution. As a result, one heuristic might work better for some kind of input while another suits better different scenarios.

The main work of this thesis aims at studying ARP problem state-of-the-art algorithms and carry out an adaptive car navigation system proactively choosing the optimal approach based on input characteristics.

Moreover, in the interest of providing such a service to a smart city community, I propose a car navigation system model in order to study its behavior from an extra-functional perspective and correctly size it for some municipality population.

Contributions

The main outcome of this work of thesis is a methodology to combine several alternative route planning algorithms into an adaptive car navigation system proactively choosing the best heuristic for each routing requests leveraging a machine learning predictive model. Moreover, an approach to car navigation system modelling is proposed by means of Queuing Petri Nets formalism, enabling an effective study of its extra-functional properties through a stochastic simulation and a capacity planning analysis.

The whole set of decisions has been validated on New York City and Milan urban areas to show the applicability of presented approaches in a real-life context.

In particular, the thesis contributions are the following:

- An alternative route planning C++ library, named *ARLib*, for a real car navigation system. While proofs-of-concept exist, mostly from algorithms authors, no comprehensive solution was available to process ARP queries with efficiency in mind. ARLib is a fast, configurable C++ library leveraging Boost.Graph, a widely-adopted, long-time-maintained library for graph representation, enabling users to change algorithm and problem parameters with minimal effort. I publicly released ARLib source code, along with documentation and tutorials with the hope to provide a reliable tool to software developers interested in routing applications.
- A deep investigation of ARP algorithms behavior and execution time versus quality trade-offs. I performed several experiments to explore ARP wide design space, highlighting those configurations under which considered algorithms prevail or are dominated by others.
- A methodology on implementing and training a machine learning model for best algorithm prediction depending on request characteristics. I presented feature engineering, model selection and prediction steps in order to train and use online a recommender module to proactively decide the optimal ARP heuristic for a specific input.
- A car navigation system design along with a Queuing Petri Net model for performance evaluation and capacity planning analysis. With the objective of delivering a valid solution for a smart city wishing to offer a navigation service, the ability to study system behavior and size it according to citizens demand is mandatory for an effective applicability analysis.
- A simulation of the proposed service deployment starting from open data collected by local authorities. On top of this, I evaluated the proposed approach, validating the system-level model against real data in two different service plan scenarios.

In the remainder of this thesis, I will write using the first-person plural to acknowledge the support from advisor and colleagues. However, I take responsibility for all the decisions and choices described in this thesis, since I was the main investigator.

Publications

From this work of thesis the following article was derived and is, at the time of writing, currently under peer review:

- Leonardo Arcari, Gianluca Palermo "ARLib: an Alternative Route Planning Library for Boost.Graph," *SoftwareX*.

Outline

This thesis structure follows the macro steps I accomplished during its development. With the exception of chapter 2, introducing the theoretical background for this work, each chapter ends with an experimental evaluation supporting the claims made across its sections. This structure has been preferred to highlight that each subsequent step was made considering the experimental evidence collected from the previous one.

chapter 2 provides the theoretical background laying under this thesis. Starting from notions of Graph Theory, it moves to Alternative Route Planning problem formulation and most notable solution quality metrics. Moreover, it provides an overview of state-of-the-art alternative route planning algorithms. Furthermore, it presents the topic of computer systems modelling and two notable formalisms from the literature: Queueing Networks and Petri Nets.

chapter 3 presents a first approach to a car navigation system design, describing the core modules composing it. As the main focus of this thesis, we concentrate on Alternative Route Planning step, introducing *ARLib*, an efficient, configurable C++ alternative routing library. Moreover, a brief description of library design, architecture and APIs are provided, along with a working example to show its ease of usage. Furthermore, we conduct an ARP design space exploration to study strengths and weaknesses of considered ARP algorithms. Finally, a first adaptive policy is depicted to statically choose ARP problem parameters and heuristics to tailor the navigation service to some provider constraints.

chapter 4 further extends the car navigation system presented before by introducing an auto-tuning module, based on machine learning models, recommending the optimal ARP algorithms depending on incoming request characteristics. Feature engineering, model selection and prediction phases are deeply described in this chapter, together with a policy to extend a basic car navigation system pipeline with adaptive capabilities.

chapter 5 introduces a Queueing Petri Net model of the system described in previous chapters along with a methodology for efficient capacity planning analysis. Furthermore, an overload-tolerant extension of the model is presented in order to cope with unexpected workload spikes in an HPC environment.

chapter 6 conducts a case study on the Milan municipality. Starting from real historical traffic data, we identify the expected workload in most congested hours and propose two possible service plans to accommodate routing requests targeting two different objectives both in terms of service quality and financial availability.

Finally, chapter 7 concludes the thesis, by summarising the findings and limitations of the proposed approach and by stating recommendations for future works.

CHAPTER 2

Background

In this chapter, an overview of the fundamental topics, on top of which this work elaborates on, are reported. In particular, we considered previous works on Alternative Route Planning problem and System Modelling, since these two topics are the most relevant for this thesis. In fact, the former plays a fundamental role in the development of a car navigation system pipeline, to compute those candidate paths that might be suggested to the service users. On the other hand, the latter provides us with the tools to capture a car navigation system details from the performance evaluation perspective in order to effectively study the service behavior in a simulation environment.

2.1 Alternative Route Planning problem

Definition 2.1.1. *Graph: an ordered pair $G = (V, E)$, where $E \subseteq V \times V$ is the set of edges (u, v) which are 2-element subsets of V, i.e. an edge is an association between two vertices.*

To avoid ambiguity, this type of graph is considered *undirected* and *simple*. A graph is *undirected* if its edges have no orientation, which means the edge (u, v) is identical to the edge (v, u). An undirected graph is then *simple* if it has neither multiple edges, connecting the same pair of vertices, nor loops.

A road network, though, might have edges directly connecting one point in the city to another which might indeed be one-way only. Moreover, loops are certainly possible within a city network, letting, in fact, a walker to start from a corner of a building and walking themselves around the building reaching the same point they left.

Definition 2.1.2. *Directed graph: a graph in which edges have orientations. An edge (u, v) is considered to be directed from u to v; v is called the target or head of the edge, while u is called source or tail of the edge.*

With these fundamental definitions from the graph theory in place, a formal road network definition can be given to model the typical domain assumptions.

Definition 2.1.3. *Road network: let V denote a set of vertices (or nodes) that represent road intersections and other points of interest. A road network is a directed, non-simple, graph $G = (V, E)$, where $E \subseteq V \times V$ is the set of edges (u, v), each representing a road segment that connects nodes u and v.*

In the context of route planning then, a common question is what sequence of roads to follow in order to move from current point s to a destination t, minimizing some metric of interest like the traveled distance or time or again the fuel cost of the driven car. To include these requirements we extend the Definition 2.1.3 with the notion of *path* and *weight*.

Definition 2.1.4. *Path: in symbols, $p(s \to t)$, from node s to t is a connected and cycle-free sequence of edges $p(s \to t) = \langle (s, u), \ldots, (v, t) \rangle$.*

The above definition is, formally, referring to *simple* paths. In this work, paths are always considered to be simple.

Definition 2.1.5. *Weight: function: a function $w : E \to \mathbb{R}^+$ assigning to each edge (u, v) a weight w_{uv}, which captures the cost of moving from u to v.*

From the notion of weight, it is simple to derive a formulation of the length of a path p, in symbols $\ell(p)$ as the sum of the weights of the edges composing the path p, in symbols:

$$\ell(p) = \sum_{\forall (u,v) \in p} w_{uv}. \tag{2.1}$$

Now that all the basic concepts to describe a road network are in place, it is possible to move forward in the definition of the Alternative Route Planning (ARP) problem. As previously mentioned, the focus of this thesis work is on building an adaptive car navigation system for a smart city, wishing to optimize the municipality traffic conditions. A fundamental part of the process to achieve that goal is to find a number k of *alternative paths* from a source node s to a target node t on which to compute the expected travel time.

The problem of finding the *shortest path* between a pair of vertices have always been a matter of interest both in research and industry fields. Many solutions to solve it have been proposed along the history, most notably the Dijkstra's Algorithm [7] which executes in $O(|V|^2)$ time. While the shortest path problem has a clear formulation and understanding, i.e. finding the path with minimum length, the problem of finding k alternative paths is less trivial. Following a naive approach, every path connecting a source node s to a target node t could be considered an alternative path from s to t, although, not necessarily very useful. After all, the main point of finding more paths connecting two nodes is that they might be better from other perspectives, like the expected arrival time. If an alternative path is too much longer than the shortest one, most likely also the expected arrival time will be higher, since the free-flow travel time would still be constrained by the road speed limit. Therefore, it is desirable for an alternative path to be as short as possible.

On the other hand, a good alternative route should not overlap too much with the others. Indeed, it could be tempting to consider as a good alternative path a small variation of the shortest one, by taking a small detour at any point and then joining back. Most certainly, though, the two paths discovered this way would share the same traffic condition, making the expected travel time almost identical, hence losing the chance to find a faster road, longer in terms of weight but less congested.

To come to a formal definition of a good alternative path, first, the notion of similarity is introduced.

$$Sim(p, p') = \frac{\sum_{(u,v) \in p \cap p'} w_{uv}}{\ell(p')}. \tag{2.2}$$

The similarity is also known as *overlap ratio*, thus $0 \leq Sim(p, p') \leq 1$, where $Sim(p, p') = 0$ holds if p shares no edge with p' and $Sim(p, p') = 1$ if $p \equiv p'$. Using this similarity measure, the definition of *alternative path* is given.

Definition 2.1.6. *Alternative path: let P be a set of paths from s to t and $\theta \in [0, 1)$ be a similarity threshold. A path p is alternative to P if and only if*

- *p is also from s to t*
- *$\forall p_i \in P : Sim(p, p_i) \leq \theta$*

This formulation settles the latter point of the aforementioned analysis on how alternative paths should appear with respect to each other, ensuring that valid solutions overlap at maximum by a factor θ, to be tuned according to the desired quality metric. It should be noted that the similarity definition in Equation (2.2) is asymmetric. The rationale behind this choice originates from the solution construction process that ARP algorithms follow: the concept of alternative path is always defined between one *candidate* alternative path and a given set of alternative paths, having the shortest path being the first added.

With the notion of alternative path, the Alternative Route Planning problem, also known in the literature as *k-Shortest Paths with Limited Overlap* (k-SPwLO) [2], can be defined.

Definition 2.1.7. *ARP problem: Given a source node s, a target node t and a threshold θ, a query ARP(s, t, θ, k) returns a set $P = \{p_0, \ldots, p_{k-1}\}$ of k paths from s to t, such that:*

- *p_0 is the shortest path from s to t,*
- *$\forall p_i, p_j \in P$ with $i \neq j : Sim(p_i, p_j) \leq \theta$*

The first point of Definition 2.1.7 guarantees that the shortest path $p_0(s \to t)$ is always recommended. In the process of computing the path with soonest expected arrival time, in fact, the shortest path, for distance or average travel time, is the most promising candidate. Although, harsh traffic conditions might lead to slowdowns and other paths could be better from this point of view. The second point instead, assures that the recommended paths in P are sufficiently dissimilar to each other. Again, this property is desirable when looking for a path that is not affected by traffic congestion in some area of the city. Having too similar alternative paths would lead to solutions with comparable drive time, uninteresting from the end-user perspective that would rather drive along the route they are customary to instead of saving a minimal amount of time through a road they do not know.

2.2 Quality

In the previous section, the provided formal definition for the ARP problem solves only one of the two requirements that a good alternative routes set should satisfy. While the necessary limitation on the overlap ratio is addressed directly, still, no constraint on the lengths of the recommended alternative paths is set. Indeed, a solution having k paths completely non-overlapping would satisfy the ARP problem definition, but the lengths of the alternative routes might be so greater than the shortest path one to result completely uninteresting from the arrival time point of view. Therefore, in order to rank different solutions, a quality metric is due to take into account also the length of all the alternative routes with respect to the shortest one.

In the literature, several metrics are proposed to judge the quality of an ARP solution. The first one [8], reported below, rates a solution according to the average ratio between an alternative path and the shortest one. Formally, let P be a solution to the ARP problem, with $k = |P|$ and p_0 the shortest path:

$$\textbf{spDifference} := \frac{1}{k-1} \sum_{p_i \in P \setminus \{p_0\}} \frac{\ell(p_i)}{\ell(p_0)} . \tag{2.3}$$

Because Equation (2.3) measures the average difference with respect to the shortest path, the relation $\textbf{spDifference} \geq 1$ always holds and the lower the difference, the better the quality. In fact, considering that each path in any ARP solution overlaps with each other with a maximum factor of θ, routes with lengths closer to the shortest one should be preferred.

In other research works [3,9], a different set of metrics has been proposed to measure both the average length of the alternative routes and the amount of distance that routes share with each other. Looking at the problem from a whole different perspective, the metrics described hereinafter leverage the notion of *Alternative Graph*.

Definition 2.2.1. *Alternative graph (AG): let $G = (V, E)$ be a graph and $w : E \to \mathbb{R}^+$ be its weight function. For a given source node s and a target node t an AG $H = (V', E')$ is a graph with $V' \subseteq V$ such that for every edge $e \in E'$ there exists a simple s-t-path in H containing e and no node is isolated. Furthermore, for every edge (u, v) in E' there must be a path from u to v in G; the weight of the edge w_{uv} must be equal to the path's weight.*

Moreover, a *reduced* AG is defined as an AG in which every node has in-degree $\neq 1$ or out-degree $\neq 1$ and thus provides a very compact encoding of all alternatives contained in AG. In this work, only reduced AG is considered, hence AG always refers to a reduced AG.

Given a solution $P = \{p_0, \ldots, p_{k-1}\}$ to the ARP problem, as formulated in Definition 2.1.7, it is simple to build an AG. Indeed, an AG $H = (V', E')$ is such that, given V_i and E_i the set of vertices and edges respectively in path $p_i \in P$, $V' = \cup_i V_i$ and $E' = \cup_i E_i$.

Knowing how to move from an ARP solution to an AG H, the following metrics are introduced to measure the quality of the alternative paths:

$$\mathbf{totalDistance} := \sum_{(u,v)\in E'} \frac{w_{uv}}{d_H(s,u) + w_{uv} + d_H(v,t)} \tag{2.4}$$

$$\mathbf{averageDistance} := \frac{\sum_{(u,v)\in E'} w_{uv}}{d_G(s,t) \cdot \mathbf{totalDistance}} \tag{2.5}$$

where d_G denotes the shortest path distance in the original graph G, while d_H the shortest path distance in the AG H. The $\mathbf{averageDistance}$ measures the extent to which the routers defined by the AG are non-overlapping, the higher the better, reaching its maximal value of k when the AG consists of k disjoint paths. Note that scaling by $d_H(s,u) + w_{uv} + d_H(v,t)$ is necessary because otherwise, long, non-optimal paths would be encouraged. The $\mathbf{averageDistance}$, on the other hand, measures the path quality directly as the average stretch of an alternative path, using a way of averaging that avoids giving high weight to large numbers of alternative paths that are all very similar.

The latter metric, which is always greater than or equal to 1, resembles Equation (2.3), measuring how long the alternative routes are with respect to the shortest one. The former metric, conversely, returns a value of the dissimilarity of the alternative paths. As opposed to the constraints set by Definition 2.1.7, Equation (2.4) does not consider only an upper bound on the overlap factor, making the comparison among different ARP solutions possible. The last question is how to combine Equation (2.4) and Equation (2.5) in order to impose an ordering among valid alternative routes sets. It is suggested to employ a linear combination of the two.

$$\mathbf{AGQuality} := \mathbf{totalDistance} - \alpha(\mathbf{averageDistance} - 1) \tag{2.6}$$

In $\mathbf{quality}$, α quantifies the penalization factor that long alternative paths have on the overall solution goodness, and it is usually set to 1. Finally, because $\mathbf{totalDistance}$ value is driven by the number of paths k, it is better to normalize it to compare the quality of solutions with respect to k more clearly.

$$\mathbf{AGQuality}\ \text{(normalized)} := \frac{1}{k}\,\mathbf{totalDistance} - \alpha(\mathbf{averageDistance} - 1) \tag{2.7}$$

Whether it is better to adopt $\mathbf{spDifference}$ and similarity threshold in the search approach, or jointly minimizing $\mathbf{totalDistance}$ and $\mathbf{averageDistance}$ is hard to judge. Indeed, it looks more like an arbitrary decision, based on what it is considered good for an ARP solution.

Literature works supporting $\mathbf{spDifference}$ quality metric suggest to fix a threshold on the overlap ratio that will make the alternatives search discard all those too-similar paths while aiming at minimizing the average length. Indeed, it is easy to find non-overlapping alternative that, although, force severe detours, in the end making them completely unappealing from a driver perspective.

On the other hand, works supporting $\mathbf{AGQuality}$ suggest that best solutions are those where alternatives overlap the least, while marginally penalizing solutions with longer routes. Moreover, in order to judge the goodness of $\mathbf{AGQuality}$, a survey on human drivers was carried out and the more human-suggested paths matched the computed ones, the better.

While aiming at very dissimilar routes may lead to more human-understandable solutions, in the context of discovering less-congested and faster paths, this might not be the primary focus.

To conclude, only extensive real-world testing could break the tie between these two approaches. In this work, we choose to follow the $\mathbf{spDifference}$ quality metrics as an arbitrary choice, sure that the proposed methodology would seamlessly apply if $\mathbf{AGQuality}$ was selected.

Having a satisfactory quality metric in place, a natural question to pose would be whether there exists a way to *efficiently* optimize it, that is, for instance, an algorithm that discovers an AG that maximizes Equation (2.7) in polynomial time. Unfortunately, it is proven that optimizing a meaningful combination of Equation (2.4) and Equation (2.5) is NP-hard. Therefore, it is necessary to restrict to heuristics to compute solutions to the ARP problem.

2.3 Algorithms

In this section, a survey on the ARP algorithms from the literature is proposed to show the different available approaches to achieve an approximate solution, with a major focus on those that were chosen as candidates for the evaluation in this work.

2.3.1 OnePass+

The first presented algorithm belongs to the class of k-Shortest Paths with Limited Overlapping (k-SPwLO) methods [2, 8]. This family of algorithms develops on the classical *k-Shortest Paths* approach, following the idea that also slightly suboptimal paths are good. Historically, k-Shortest Path heuristics used to compute solutions that "looked bad" from a human perspective, since often too similar to each other. k-SPwLO algorithms build on top of them to address this key issue. In particular, OnePass+ [8] searches the graph space expanding paths from the source node s until k paths to t are discovered. In the process of exploring the search space, candidate routes are pruned according to two pruning criteria.

Definition 2.3.1. *k-SPwLO pruning criterion (1): let P be the set of already recommended paths. If p is an alternative path to P with respect to a threshold θ, then $Sim(p', p_i) \leq \theta$ holds for every subpath p' of p and all $p_i \in P$. If the condition is not met by a soon-to-be expanded path, the path is pruned.*

OnePass+ algorithm employs a min-priority queue in order to examine paths in increasing order of their length. Each time a new path is recommended, i.e., added to the result set P, an update procedure takes place for all remaining incomplete paths $p(s \rightarrow n)$ in the priority queue. The algorithm terminates when either k paths are added to the result set or all paths from s to t qualifying Definition 2.3.1 are examined. Moreover, in the search process, a second pruning criterion is considered.

Definition 2.3.2. *k-SPwLO Pruning criterion (2): let P be a set of paths from a source node s to a target node t and p_i, p_j be two paths from source s to some node n. If*

$\ell(p_i) < \ell(p_j)$ and $\forall p \in P : Sim(p_i, p) \leq Sim(p_j, p)$ hold, then path p_j cannot be part of the shortest alternative path to P and p_i is said to dominate p_j, in symbols, $p_i \prec_P p_j$.

The second pruning criterion can be used to compute the shortest alternative to a set of paths as follows. Let P be the set of paths for which we want to compute the shortest alternative path, and P_n be the set of paths from s to a node n created during the expansion of paths from s. If set P_n contains a path $p'(s \to n)$ such that:

a) p' is longer than any path $p_n \in P_n \setminus \{p'\}$ and

b) for every path $p \in P$ the overlap ratio $Sim(p', p)$ is higher than the ratio $Sim(p_n, p)$ for all paths $p_n \in P_n \setminus \{p'\}$,

then p' can be pruned.

By means of these two pruning criteria, OnePass+ can efficiently compute a set of alternative paths. Although, because Definition 2.3.2 strongly depends on the paths in the set P at the moment it is applied, a path p' pruned while computing the i-th alternative to P might still have been part of the best alternative at step $i + 1$. While a straightforward solution to this problem would be to restore the pruned paths every time a new alternative is recommended, this introduces strong delays in the algorithm execution time. Therefore, OnePass+ never resets the pruned sub-paths, leading to an approximate, but faster to compute solution.

2.3.2 Penalty

The second approach considered in this work is named Penalty [3, 9, 10] and follows a simple idea: computing alternative paths by iteratively running shortest path queries and adjusting the weight of the edges on the resulted s-t-paths. The rationale behind this method is straightforward. The key point of ARP problem is to find paths such that a traffic analysis might rank a longer path better from the arrival time viewpoint. Therefore, by penalizing the recommended paths, the next shortest path is likely to be different, but not completely. In fact, some sub-paths might still be shorter than a full detour. The basic steps of Penalty are the following:

- A shortest s-t path p is computed with Dijkstra's algorithm or any speedup variation of it.
- The discovered path p is added to the alternatives set P and it is penalized by increasing the weight of its edges.
- A new s-t query is executed and the new shortest path p' is found.
- If p' is short and different enough from the previously recommended alternatives, then it is added to P.

Indeed, the crucial point of this algorithm is how and when the edge weights are incremented. In this work, the following weight adjustment policy is considered.

The increase on the weights should be of small magnitude in order to keep the resulting $\mathbf{averageDistance}$ low. To penalize an edge of an s-t-path, only a small fraction of its initial weight is added. Let p be the *penalty factor*, such that $0.1 \leq p \leq 1$, and e be an edge of an s-t-path to penalize, then the new edge weight is computed as the following: $w_e^{new} = w_e + p \cdot w_e^{old}$. Generally speaking, the higher the penalty factor is, the more the new shortest path is likely to differ from the last one. On the other hand, the lower the penalty factor is, the more shortest path queries can be performed, reducing the risk of losing meaningful paths.

The number of weight adjustments should be restricted as it could lead to the loss of good alternatives. A working scenario supporting this statement is a city with only one very fast highway into it and many alternative paths through the city center. Allowing an unbounded number of weight increments, the cost of the highway would grow after every s-t path query. Therefore, after several updates, subsequent shortest paths would take a detour much longer than the highway, that a human driver would hardly choose. To overcome this problem, the number of increases is limited for edges in already recommended paths.

Finally, to reduce the overlapping between the computed alternatives, it is useful to extend the weight adjustment to their neighbors. The rationale behind this is to avoid the *saw tooth effect*, i.e. alternative routes that present many minimal detours around a previously recommended path, forking from it at some node and rejoining immediately after. Therefore, when increasing the weight of the edges in a shortest s-t-path, the weights of edges that leave and join its vertices should be additionally penalized by a factor $0.1 \leq r \leq 1$. This step, named *rejoin-penalty*, contributes to high $\mathbf{totalDistance}$. The cost increment step is not uniform across the edges, though. To reduce the possibility of computing alternative paths that tend to rejoin with the previously recommended, heavier weights are put on those *outgoing* edges that are closer to the target t, while heavier weights are put on those *incoming* edges that are closer to the source s. Formally, let p_{st} be the shortest s-t-path just penalized, then:

$$w_{uv}^{new} = w_{uv} + (0.1 + r \cdot d_s(u)/d_s(t)) \cdot w_{uv}^{old}, \ \forall (u,v) \in E : u \in p_{st}, \ v \notin p_{st} \tag{2.8}$$

$$w_{uv}^{new} = w_{uv} + (0.1 + r \cdot d_t(v)/d_t(s)) \cdot w_{uv}^{old}, \ \forall (u,v) \in E : u \notin p_{st}, \ v \in p_{st} \tag{2.9}$$

2.3.3 ESX

The last ARP algorithm taken into account is again from the family of k-SPwLO methods, but it follows an iterative approach, similarly to Penalty. It is named Edge Subset Exclusion (ESX) [8] and its main idea is to compute a set of alternative routes by subsequently excluding edges from the road network. The basic steps are the following:

- Given a source node s and a target node t in a graph, the algorithms first adds the shortest path p_0 to the result set P.
- Then, ESX removes an edge of p_0 from the road network and a new shortest s-t-path p_c is computed on the updated road network.
- If the overlap of the candidate path p_c with p_0 does not violate the similarity threshold θ, then p_c is added to P, as stated in Definition 2.3.1.
- Otherwise, the algorithm proceeds by removing more edges from the road network. If P contains more than one path, ESX removes an edge from path $p \in P$ for which the similarity $Sim(p_c, p)$ is the highest.

- The process is repeated until a path that does not violate the similarity threshold is found. The algorithm terminates when the desired number of alternatives are found or when there are no more edges to remove.

The critical step in ESX is the policy according to which an edge is elected for exclusion. A good selection strategy would choose that edge which is most important for the network connectivity between s and t and hence causing a relevant detour if removed. Deleting such edge from the graph would cause the subsequent shortest path queries to return alternative routes potentially dissimilar enough from the already recommended ones. The policy adopted by ESX employs a heuristic based on a local check. Given an edge $e(a, b)$ on some path $p \in P$, let $E_{inc}(a)$ be the set of all incoming edges $e(n_i, a)$ to a from some node $n_i \in N \setminus \{b\}$ and $E_{out}(b)$ be the set of all outgoing edges $e(b, n_j)$ from b to some node $n_j \in N \setminus \{a\}$. First the heuristic computes the set P_s which contains the shortest paths from every node $n_i \in E_{inc}(a)$ to every node $n_j \in E_{out}(b)$. Then, the heuristic defines the set P'_s which contains all paths $p \in P_s$ that cross edge e. Finally a priority to edge e is assigned, set to $|P'_s|$.

2.3.4 Other algorithms

There exist other Alternative Route Planning algorithms in the literature which were not considered in this work as hardly fitting the ARP problem as formulated in Definition 2.1.7, but they are briefly reported hereinafter for completeness.

Pareto A classical approach to compute alternatives by means of Pareto optimality [11–13]. The typical procedure is to add to the primary weight function (e.g. edge length) a secondary weight function that is zero for edges that are not part of any recommended path and identical to the primary edge weight for edges that are part of at least one recommended path. A path is now said *Pareto-optimal* if there is no other path which is better with respect to both weight functions. As the experimental study in [3] reported Pareto approach producing low-quality results, it was not investigated in this work.

Plateau An approach that identifies promising segments in the graph, called *plateaus* [14], such that s-t-paths flowing through them are good candidates to be alternative paths. In details, forward and backward Dijkstra searches are performed from s to all nodes and from t to all nodes respectively. Then, the discovered shortest path trees are intersected, discovering a set of simple paths, named plateaus, with a peculiar property: let (u, v) be an edge of a plateau, then $d_s(u) + d_t(u) = d_s(v) + d_t(v)$, meaning that the edge (u, v) can be reached with the same, shortest, traveling distance from both s and t. As the plateaus are usually too many, they are ranked according to $rank = \ell(p_{st}) - \ell(\bar{p})$, where p_{st} is the shortest s-t-path via the plateau $\bar{p}$. Therefore, a plateau that corresponds to a shortest path from s to t has rank zero, which is the best value. The major downside of this approach is that recommended paths are not guaranteed to be sufficiently dissimilar since the similarity only to the shortest path is taken into account.

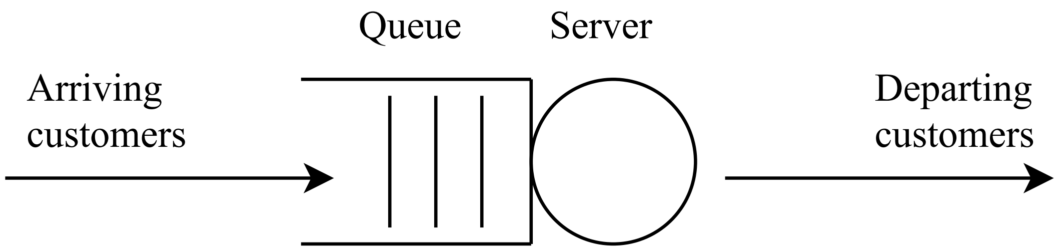


Figure 2.1: *A single service center.*

2.4 Computing Systems Modelling

In this section, we provide some background information about *Queueing Networks* (*QN*) [15] and *Petri Nets* (*PN*) [16] models. This two formal models are widely employed in the context of computer systems modelling, an abstraction distilling those aspects of a system that are essential to study its execution behavior, a matter we are going to leverage in chapter 5.

2.4.1 Queueing Networks

Queueing Network [15] modelling is a particular approach to computer systems modelling in which a computer system is represented as a network of queues. A network of queues is a collection of *service centers*, which represent system resources, and *customers*, which represent users or jobs running. Figure 2.1 pictures the fundamental entity of a QN: the service center.

Customers arrive at a service center, possibly wait in a queue, receive service from the *server* and depart. In the model of Figure 2.1 we identify two parameters. For starters, we must specify the workload intensity, which is the rate at which customers arrive. Moreover, we must specify the service *demand*, which is the average service requirement of a customer.

Given some values of arrival rate and service demand, we can evaluate, if we solve a set of analytic equations, or simulate, if we employ a QN simulator, the model and compute a number of performance indices, which we summarize as follows:

Definition 2.4.1. *Given a queueing network, as shown in Figure 2.1, we introduce the following basic quantities:*

T, *the length of time we observe the system.*

A, *the number of request arrivals we observe.*

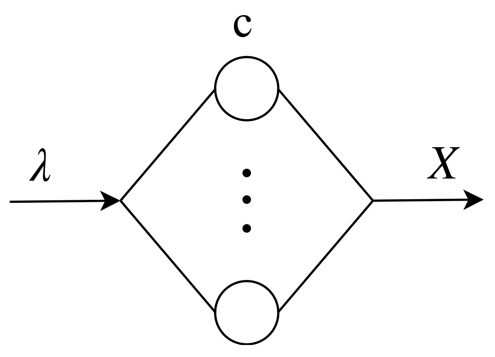
C, *the number of completions arrivals we observe.*

N, *the average number of customers in the system.*

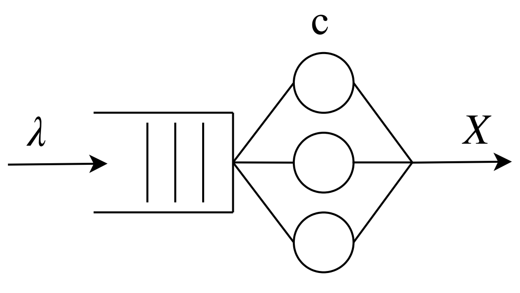
R, *the average system residence time, or the time a customers in a queue.*

B, *the length of time a single resource was observed to be busy.*

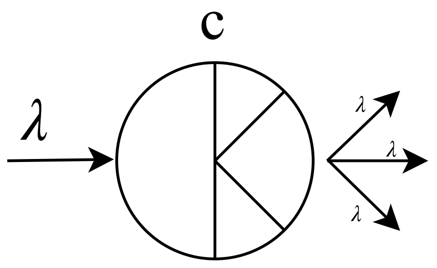
From this measurements, we can define the following additional quantities:



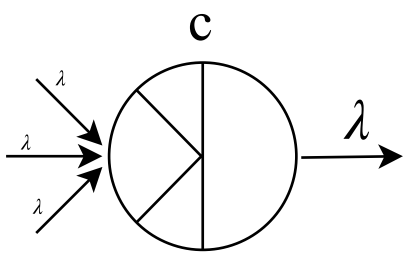
(a) *A Delay station.*



(b) *A G/G/c station.*



(c) *A Fork station.*



(d) *A Join station.*

Figure 2.2: *Queuing Network service stations*

Definition 2.4.2. *Given a queueing network, as shown in Figure 2.1, we introduce the following derived quantities:*

$$\lambda = \frac{A}{T}, \quad \textit{the arrival rate (req/s)} \tag{2.10}$$

$$X = \frac{C}{T}, \quad \textit{the throughput (req/s)} \tag{2.11}$$

$$U = \frac{B}{T}, \quad \textit{the utilization (\%)} \tag{2.12}$$

$$S = \frac{B}{C}, \quad \textit{the service time per request (s)} \tag{2.13}$$

Since it is hard to imagine characterizing a modern computer system with single-resource service centers, we introduce several additional kinds of service centers which will be employed in chapter 5.

Delay Station

A Delay station is a multi-server service center where jobs experience no residence (queueing) time. Let us consider a delay station equipped with $c \in [1, \infty)$ servers. When a customer arrives at the station, there are two possible outcomes:

- There is a free server available and the job is immediately served.
- There are no free servers and the job is dropped.

A delay station is pictured in Figure 2.2a.

G/G/c Station

A G/G/c is a multi-server service center where c servers are available and jobs that cannot be served immediately wait in a queue. When a resource becomes available, waiting jobs are processed from the queue according to some policy (e.g. First-Come, First-Served). A G/G/c station is shown in Figure 2.2b.

In queueing theory, queue stations are named according to three factors, i.e. A/S/c, where A and S are the stochastic processes modelling the arrivals and departures respectively, while c is the number of servers.

A notable queue station in the literature is the M/M/1 queue, where M stands for *Markovian* process. An M/M/1 queue, therefore, is characterized by an arrivals process where inter-arrival times are *exponentially* distributed and jobs are served one at a time and processed in exponentially-distributed times.

The most general formulation of a queue station is the G/G/c queue, where G stands for *general*, so that no assumption on the arrivals and departures distribution is made. Such models are perfectly reasonable in a simulation context, where service times can be loaded from real-world system logs. From an analytical perspective, instead, employing general distributions is highly impracticable, hence typical approaches involve choosing a known theoretical distribution sharing most meaningful statistics with the real system, e.g. mean and coefficient of variation.

Fork Station

A Fork station [17] splits arriving job into several *tasks*, one or more per outgoing link, modelling a typical approach in computer systems to achieve fast processing are minimizing service time. In parallel processing, this concept is also known as *thread*-based (or *task*-based) computing. Generally, no service time is allocated to Fork stations and no limit on the capacity is set. The degree of parallelism is modelled by the number of outgoing links to other network stations, generating c tasks on each link. A representation of a Fork station is drawn in Figure 2.2c.

Join Station

Join stations [17] are complementary to Fork stations and are used to recombine the tasks corresponding to a job that had been previously split and then route the job to the following stations. Like Fork stations, Join stations have no service time and no limit on the capacity is placed. A representation of a Join station is drawn in Figure 2.2d.

Join stations may implement several *join strategies*, modelling many typical behaviors of a parallel application. In chapter 5 we will leverage the following two policies:

- **Standard join**: This is the usual policy adopted by Join stations. All the tasks of a job generated by a Fork station must arrive at the Join station before that the job will be released. In parallel computing, this is also known as a *barrier*.

- **Quorum**: With this strategy, when the Join station has received some fixed number of tasks of a job, then it will release the job. The number of tasks to wait at the Join can be less than the ones generated by the Fork.

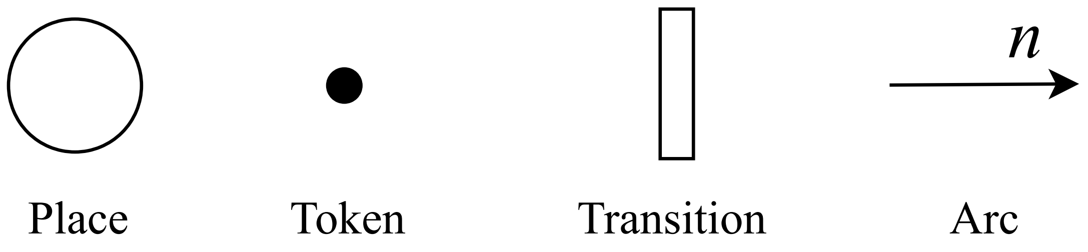


Figure 2.3: *Place-Transition Net notation*

2.4.2 Petri Nets

Petri Nets (*PN*) [16] are a class of modelling formalisms to describe the execution behavior of distributed and parallel systems. PN were originally proposed by Carl Adam Petri in a formulation known today as *Place-Transition* (*PT*) *nets* or *ordinary* Petri nets. PN offer, like other modelling languages, a graphical notation to represent step-wise processes that include choices, iteration and concurrent execution. As an advantage, PN also has an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.

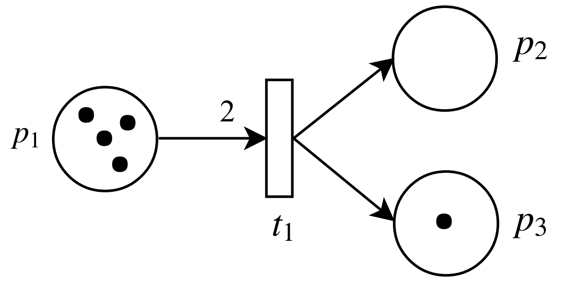
Since the introduction of PT nets, many variations have been developed to extend to expressive power of the modelling language, such as *coloured Petri nets*, *stochastic Petri nets*, *generalized, stochastic Petri nets*, *coloured, generalized stochastic Petri nets* and *Queueing Petri nets* (*QPN*) [18].

Place-Transition Nets

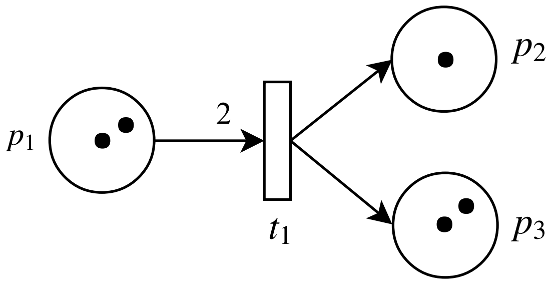
Pictured in Figure 2.3, Place-Transition Net notation comprised the following primitives:

- **Place**: Drawn as a circle, it is used to represent a state, such as a phase of a process or the number of resources available, or an object in the system, e.g. a program variable.
- **Token**: Drawn as a black dot, it is an *identity-less* marker present in places. The number of tokens in a place indicates the value of the state or object represented in that place.
- **Transition**: Drawn as a rectangle, it is used to characterize events or activities that occur in the system. A transition *fires* to indicate the occurrence of the event the transition represents.
- **Arc**: Drawn as an arrow, it is used to express the relationship between states and events. An *input* arc, from a place to a transition, represent the conditions that can cause an event to happen. An *output* arc, from a transition to a place, reflects the result of the event. Each arc is assigned a *weight*. If not specified explicitly, the weight is assumed unitary.

Figure 2.4 shows a sample Place-Transition Net. It involves three places, p_1, p_2 and p_3 with 4, 0 and 1 tokens, respectively. Place p_1 is connected to t_1 transition through



(a) *A simple PT net.*



(b) *The PT net after transition t_1 fired once.*

Figure 2.4: *A sample Place-Transition Net before and after a transition fire.*

an *input* arc with weight 2. Similarly, transition t_1 is connected both to place p_2 and p_3 through two *output* arcs with, implicit, weight 1.

PN transitions are said to fire when they consume tokens from their *input places*, i.e. sources of input arcs, and creates other tokens in their *output places*, i.e. target places of output arcs. This notion is subtle but fundamental in understanding PN. Since tokens have no identity, thinking of them as *flowing* through transitions would lead to erroneous interpretations. As a matter of fact, when a transition fires, tokens are effectively destroyed from the input places and generated, as new instances, in the output places. This process is governed by the following rules:

- **Enabling Rule**: A transition is enabled if all its input places contain *at least* as many tokens as defined by the weight of their input arc. In Figure 2.4a, transition t_1 is enabled as p_1 contains at least 2 tokens. In fact, it contains four of them.
- **Firing Rule**: An enabled transition is ready to fire. The firing process causes the destruction, for each input place, of the number of tokens corresponding to the weight of the respective input arc. Consequently, it creates, in each output place, the number of tokens corresponding to the weight of the respective output arc. In Figure 2.4b, when transition t_1 fires, it consumes 2 tokens from place p_1 and puts 1 token both in p_2 and p_3.

Stochastic Petri Nets

Place-Transition Nets can be used to verify the functional properties of a system. On the other hand, since no timing information can be expressed, it is impossible to employ them in the analysis of computer systems performance. Time-augmented variants of ordinary Petri Nets have been proposed to overcome this limitation. A widely used variation is named *stochastic Petri nets* (*SPN*) [18] which is hereafter introduced.

SPN are developed from PTN by introducing *timed transitions* characterizing events that occur in the system. As opposed to non-timed (or *immediate*) transitions, timed transitions, once enabled, do not fire immediately, but hold until their firing time elapses. By definition, in SPN the firing time of a transition is exponentially distributed, with the following probability density function:

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

where λ is said to be the *firing rate* of the transition. With a notion of time formulated in these terms, the inter-fires times of a transition form a notable type of stochastic process, i.e. a *Markov process*. Consequently, Markovian techniques can be implied to analyze the model behavior and, more specifically, important performance measurements can be evaluated, such as the *mean number of tokens in a place* and the *firing throughput at a transition*.

2.4.3 Java Modelling Tools (JMT)

In chapter 5, the proposed system models are simulated in Java Modelling Tools (JMT) [19–21] in order to analyze the performance indices of our interest. JMT is a free open source suite consisting of six tools for performance evaluation, capacity planning, workload characterization, and modelling of computer and communication systems. The suite implements several state-of-the-art algorithms for the exact, approximate, asymptotic and simulative analysis of queueing network models, either with or without product-form solution. Models can be described either through wizard dialogs or with a graphical user-friendly interface. The workload analysis tool is based on clustering techniques. The suite uses an XML data layer that enables full re-usability of the computational engines [20].

Along the aforementioned features, JMT implements a simulator engine for *Queueing Petri Nets* (*QPN*) [18], a powerful modelling formalism merging Queueing Networks and Generalized Stochastic Petri Nets, extending the notion of performance metrics described in Definition 2.4.1 and Definition 2.4.2 to PN.

Introducing such an effective compatibility between QN and PN increases the available expressive power [22], enabling models of much more complex systems, by means of advanced components obtained by a combination of different stations together.

2.5 Summary

In this chapter, we introduced the fundamental concepts that will be part of this work of thesis. For starters, we presented the Alternative Route Planning (ARP) problem as a key part of a car navigation system design. The previous work investigating on this topic served us as a solid background on which to design and implement *ARLib*, an efficient, configurable alternative route planning C++ library that we are going to present in section 3.3. In fact, proofs-of-concept aside, no implementations of state-of-the-art ARP algorithms were available to carry out a fair comparison of the strengths and weaknesses of each approach, as we are going to do in chapter 3, and eventually, employ them in a real car navigation system implementation to use online.

Moreover, Queueing Network and Petri Nets will be employed in chapter 5 to build a reliable representation of our car navigation system, using those modelling formalisms to study our service from a performance evaluation perspective and furthermore propose a capacity planning analysis methodology to effectively compute the right amount of resources to handle some expected workload.

CHAPTER 3

Alternative Route Planning

In this chapter, we present our approach to an adaptive car navigation system design, involving a pipeline of three stages in which the most time-consuming step is represented by the Alternative Route Planning phase, main focus of this work of thesis.

In the previous chapter, we introduced a variety of heuristics addressing the ARP problem with different approaches to reduce the search space in large graphs representing the road network. Observing the range of available solutions, our first research question led us to investigate whether they all performed equal or some heuristic worked better than the others. Considering that the Alternative Route Planning problem is characterized by number of alternative paths k and similarity threshold θ parameters, we asked ourselves if some algorithm could perform best under some combination of k and θ, while badly behaving for others.

In order to answer this question, a common approach is to carry out a Design Space Exploration (DSE), systematically analyzing problem configurations from multiple perspectives, like execution time and quality of solutions. In particular, in our scenario, this means comparing ARP algorithms in different (k, θ) setups to study their behavior. Although, to deliver a fair study, equivalent implementations should be taken into account to actually compare algorithms performance while keeping language and data-structure issues aside. To our knowledge, aside proofs-of-concept, no publicly-available implementation of those algorithms is available, especially no available library ships a range of ARP heuristics executing on the same graph data structure, making any comparison meaningless. Therefore, we developed and publicly distributed a library offering the flexibility we required to answer our question, eventually distributing an easy-to-use solution for any software developer interested in alternative routing topic. This library is named *ARLib* and it is introduced starting from section 3.3.

Based on ARLib, we carried out a DSE experiment to deeply analyze ARP algorithms behavior under different k and θ combinations, both from latency and quality of results perspectives along with their failure rate. From the experiment results, we propose a policy for our car navigation system introducing a first degree of adaptivity, enabling the service provider to effectively choose the system working point within the design space.

3.1 An Adaptive Car Navigation System

Considering the rising wave of self-driving cars [23], the amount of car navigation requests will increase rapidly [24] together with the need for real-time updates and processing on large graphs representing the urban network. At present also, car, taxi and motorcycle drivers rely on navigation systems to reach their destinations within the city to discover the paths with least traffic. Moreover, mail, packages and food delivery services experience their business rising [25] as a result of the non-stopping trend of online goods purchasing, from books to house furniture to clothing to meals. From the urban transfers perspective, the depicted scenario causes increasingly complex issues in managing traffic jams and air pollution levels in certain areas of the city.

The adaptive car-navigation system proposed hereafter targets a smart city wishing to address the aforementioned problem with effective route planning and traffic optimization processes. In order to do that, the municipality should provide a service significantly improving the navigation experience, either from more reliable and faster paths in terms of expected travel time, either from discounts on other metropolitan expenses like parking tickets, fines or permissions to access limited-traffic areas. Upon a request for a path to some destination, the service would respond with a single path that is optimal according to some quality metric, subject to a number of constraints set by the municipality, for instance minimizing the number of traveling vehicles around the city center. The ideal target of the service would be autonomous-cars that would follow passively any computed route. Although, local drivers may still decide to take detours from the suggested path, either because different from the one they are accustomed to or because they believe to know a better way. Such behavior, if shared among a substantial number of drivers, would undermine the service traffic-splitting effectiveness. As a solution, the city might produce a rewarding plan for well-behaving drivers that follow the service computed paths and obtain discounts on urban taxes.

From the municipality perspective, the service should have a low cost, in terms of used amount of resources, and it should satisfy the whole load of requests, possibly absorbing unexpected requests peaks that might arise in presence of accidents, harsh climate conditions or natural disasters that would affect the viability. As for the end-user viewpoint, a good service would provide high-quality results, i.e. a route with best travel time, within a realistic response time, possibly tuning the path along the way through re-routing requests.

3.2 System Architecture

The proposed adaptive car-navigation system is composed of a three-stage pipeline which can be described as follows:

1. Alternative Route Planning (ARP)

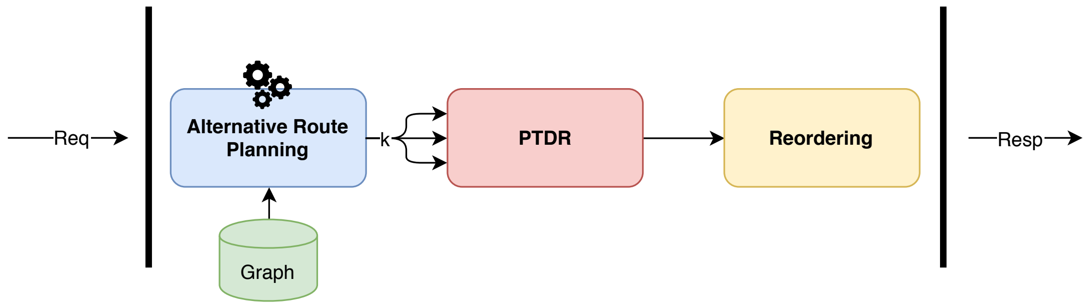


Figure 3.1: *A graphical representation of the proposed car-navigation system pipeline. The Alternative Route Planning module is annotated with a gear to highlight its high degree of configurability.*

2. Probabilistic Time Dependent Routing (PTDR)

3. Reordering phase

The first step, and main focus of this thesis, consists of determining a number k of alternative paths from some source to some destination to be passed to the next step. In the context of navigation, identifying the shortest path is not enough to discover a good solution, but the traffic situation must be taken into account. A shortest path in terms of distance or average traveling time might not be the optimal one in every time frame, therefore an expected arrival time should be computed considering the current speed profile on a given route.

In the second step, for each one of the k alternative paths, the travel time is estimated using the PTDR [4–6] module. While the exact solution to the expected travel time has exponential complexity, a Monte Carlo approach is proposed to efficiently approximate the solution of the problem. This is the first degree of adaptivity of the proposed service. Because the traffic situation can change dramatically the arrival time along a way or another, the more alternative routes we compute the higher the chances are to find the fastest one. Ergo, the system adapts to the varying traffic conditions, suggesting to the end-user possibly different routes for the same source-destination pair, according to an in-time computed expectation of the travel time.

The third and last stage gathers the timing information provided by the k instances of PTDR module for every single request and selects the best path to return to the user. Within this phase, routes are chosen not simply by traveling time, but also taking into account municipality policies, for instance penalizing those paths to pass through the city center or an area under investigation for high pollution level. A reordering policy should reflect the local jurisdiction view on the ideal urban traffic, without completely ignoring those routes that, despite breaking some constraints, have still a much higher quality with respect to the other candidates.

3.3 ARLib

In the process of developing our car navigation system, considering that PTDR step has been deeply investigated in other works [4–6], in this thesis we are going to focus on Alternative Route Planning module. The goal of this module is to accept a request for an ARP solution, given the source and the destination of the request along with the number of desired alternative routes and a threshold on the similarity, compute a set of paths and forward them to the PTDR module to estimate the traveling time.

As illustrated in section 2.3, several algorithms are available in the literature, proposing different approaches to solve the ARP problem. Indeed, because finding the exact solution is not computationally feasible, each algorithm employs some heuristic to reduce the computational complexity and return a solution in a reasonable amount of time. Since heuristics are never guaranteed to be optimal under all circumstances, it might be true that some algorithm performs better under certain conditions while another one works better in other cases. Therefore, we would like a configurable ARP module, enabling us to switch among algorithms, k and θ configurations, in order to use the most suitable set of parameters satisfying our target requirements.

While implementations of ARP algorithms exist [26–29], most of them are nothing more than proofs-of-concept, supporting the claims of researchers proposing them. First of all, most of them are written in non-performance-oriented languages, like Python or Java, which are not suitable for low latency systems like ours. Moreover, all of them are implemented on their own custom graph data structure, which is undesirable for at least two reasons:

- Implementing a graph data structure which is efficient enough for real-world constraints is hard, in fact, different storage strategies might suit best different graph topologies, so it is very unlikely that a single structure might fit all the possible scenarios.
- Switching among algorithms with different graph representations would require to reload in memory the whole graph every time, which is hardly ideal when dealing with road networks of hundreds of thousands of nodes and edges.

Therefore, the first step was to adopt a single, maintained, flexible, high-quality library providing efficient data structures and operations on graphs. We elected Boost Graph Library (BGL) [30] for several motivations:

- It is a well-established solution, born in late 2000 and still maintained.
- It provides three different, very efficient, graph data structures.
- It is *generic*, meaning that data types are customizable at compile time (hence, no run-time overheads), but with a single interface, enabling the developer to run the same algorithm on different graph structure implementations.
- It is released under a permissive license, the Boost Software License [31], enabling both private and commercial usage.

On top of BGL, we developed **ARLib**, a generic [32–34], configurable, C++ **A**lternative **R**outing **Lib**rary, providing an implementation of a set of state-of-the-art algorithms. In the interest of preserving the generality of usage of BGL, ARLib follows the same conventions that official BGL algorithms set, so that ARLib users will find executing alternative routing algorithms on their graphs natural and non-intrusive. In details, ARLib implements `OnePass+` and `ESX` algorithms, presented in [8] and `Penalty` algorithm, presented in [10]. Along with alternative routing solutions, ARLib provides

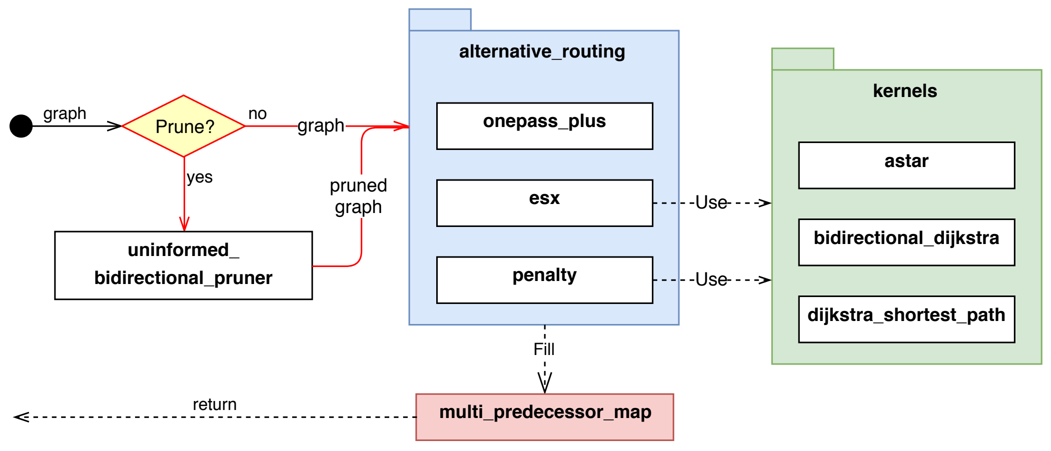


Figure 3.2: *A graphical representation of the user-level ARLib architecture. On the left, the optional pruning step to reduce the search space. In the center, the alternative routing functions along with the data structure containing the problem solution. On the right the set of interchangeable routing kernels.*

an implementation of `Bidirectional Dijkstra` [35], a speed-up variant of Dijkstra's algorithm [7], and `Uniformed Bidirectional Pruning` [9], a graph pre-processing algorithm to reduce the alternative routes search space.

3.3.1 Software Architecture

In this section, we give a description of ARLib software architecture to offer a rationale behind its design choices.

As a library to compute ARP problem solutions, knowing that each algorithm comes with its trade-offs, ARLib forces no default-choice to find *alternative s-t paths*, but it exposes a consistent interface so that the user can exchange the solving algorithm with nearly-none code modifications. Upon algorithm termination, the user is provided with a set of *alternative routes* $P = \{p_0, \ldots, p_{k-1}\}$ of k paths *such that*: p_0 is the shortest path from s to t and $\forall p_i, p_j \in P$ with $i \neq j : Sim(p_i, p_j) \leq \theta$, according to the problem formulation in Definition 2.1.7.

In the interest of preserving the same user experience that BGL offers, ARLib is structured around two main components: *Alternative Routing* component, to find the alternative routes, and a *Property Map* for storing the problem solution, both shown in the center of Figure 3.2.

Property Maps [36] are the devices that BGL already employs to represent properties of vertices and edges of the graph, that are required by an algorithm to find a solution. Common properties are the distance between two nodes, edge weight or vertex color. By relying on *property maps*, BGL decouples graph properties from the actual implementation of the graph. To do so, a common interface is set. Each property map is defined by a `key`, e.g. a vertex or an edge, and a `value`, representing the property of the `key`. ARLib introduces `multi_predecessor_map` (`MPM`) property map to record the alternative paths in the input graph. To each vertex v, `MPM` associates, for each path p, the predecessor vertex of v along p.

Alternative Routing component exposes *generic, free functions* implementing state-of-the-art alternative routing algorithms: (a) `OnePass+` (b) `ESX` (c) `Penalty`. They all adhere to a common interface that expects arguments specified in terms of *compile-time checked concepts* [37] instead of concrete-types. Consequently, ARLib remains open to different implementations of the input graph type or of the property map types. The only requirement is set on the interface they must provide.

As reported on the right of Figure 3.2, `ESX` and `Penalty` are customizable by means of *routing kernels*. While `OnePass+` involves a custom policy to explore the graph, `ESX` and `Penalty` are driven by an iterative search of a shortest path. For the sake of flexibility, ARLib enables the user to choose which shortest path algorithm to use, making additional room for auto-tuning processes [38, 39] to choose the best kernel according to the properties of the input graph. Available kernels are: (a) `Dijkstra's Algorithm` (b) `A* Star`, a heuristic-driven variant of Dijkstra's algorithm (c) `Bidirectional Dijkstra`, a Dijkstra's algorithm variant exploring the search space from the source and the target vertices, simultaneously.

Finally, ARLib provides an implementation of `Uninformed Bidirectional Pruning`. ARLib users can prune the graph in such a way to exclude all those edges that are very unlikely to be part of an *alternative-s-t-path* and therefore reducing the runtime of subsequent alternative routing executions. In fact, since the pre-pruning step depends on a source-destination pair, it is especially beneficial for those routes that are queried more often. Indeed, the pruned graph can be stored so that the pre-pruning pass cost is amortized by future requests that will run faster because of the reduced graph space. On the other hand, for infrequent source-destination pairs, the cost of applying such a pruning overcomes the cost of the alternative routing phase, penalizing the overall execution time.

3.3.2 Software Engineering

As illustrated in previous sections, ARLib provides the developer with a convenient interface to the offered algorithms so that picking up the library and using it would result in a straightforward activity. The main design goal, since the very beginning, was not to develop a software strongly coupled with the whole system it would have been part of, but rather as an independent library to efficiently solve a problem for which no open-source, production-ready solution exists. On top of everything, we aimed for a product that a user would put trust into.

Because of this very motivation, ARLib has been developed according to the *Test-Driven Development* (TDD) methodology [40]. TDD is one of the key points of the Agile development approach and it guides the software development toward a better-quality product through an iterative process based on unit tests. TDD can be summarized as follows:

1. **Add a test**: in Test-Driven Development, each new feature begins with writing a test. This is a differentiating feature of TDD versus writing unit tests after the code is written, indeed it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. **Run all tests and see if the new test fails**: this validates that the testing-suite is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility

that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. **Write the code**: the next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5. At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. **Run tests**: if all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. **Refactor code**: the growing code base must be cleaned up regularly during test-driven development, following all the software engineering practices to manage the complexity and increase the maintainability of the source code. By continually re-running the test cases throughout each refactoring phase, the developer can be confident that the process is not altering any existing functionality.
6. **Repeat**: starting with another new test, the cycle is then repeated to push forward the functionality.

By adopting TDD methodology, we are able to ship ARLib with a set of unit tests to cover the advertised features of the library. This way, we are able to continuously verify that we are compliant with the functional requirements. Such confidence is particularly important from a deployment perspective. Since we are publishing ARLib as an open-source, public repository, potential users require forms of guarantees on the library correctness. Indeed, with algorithms operating on an enormous amount of data, manually checking the output results is quite unfeasible. For this reason, we set up a *Continuous Integration* [41,42] service [43] introducing a managed source code push, testing and documentation production cycle. Whenever a modification is pushed to the repository, it is merged into production code only if all unit tests are still running nicely. Therefore, unit tests act also as regression tests. With this setup in place, potential users can convince themselves about the quality of ARLib software.

3.3.3 API Reference

In this section, we provide a description of ARLib APIs, to cover all the features that compose the user interface.

As previously mentioned, ARLib implements several state-of-the-art algorithms with a keen focus on peformance and usage flexibility. Available algorithms are summarized as follows:

- k-SPwLO OnePass+ [8]
- k-SPwLO ESX [8]
- Penalty algorithm [10]
- Bidirectional Dijkstra [35]
- Uninformed Bidirectional Pruning [9]

Multi Predecessor Map

The first class we introduce is `multi_predecessor_map`. Multi Predecessor Map is a Property Map designed to record alternative paths computed by some ARP algorithm.

Class interface is available in Listing 3.1.

```
template <typename Vertex, typename UnorderedAssociativeContainer>
class multi_predecessor_map {
public:
  multi_predecessor_map();
  multi_predecessor_map(multi_predecessor_map const &other);
  multi_predecessor_map &operator=(multi_predecessor_map const &other);
};
```

Listing 3.1: *Multi Predecessor Map interface*

Implementing `Read Property Map` concept, Multi Predecessor Map exposes the following accessor free function:

```
template <typename Vertex, typename Key>
return_type get(multi_predecessor_map<Vertex> &pmap, Key const &k);
```

whose parameters are summarized by the following table:

Parameter	**Description**
k	The graph `vertex_descriptor` v
$return$	Reference to an *Unordered Associative Container*. That container has an `int` key representing the alternative path number i, whereas its value if the `vertex_descriptor` of the predecessor of v in alternative route i.

OnePass+

OnePass+ algorithm implementation is provided by `onepass_plus` function, whose interface is reported in Listing 3.2.

```
template <typename Graph, typename WeightMap, typename MultiPredecessorMap,
          typename Terminator, typename Vertex>
void onepass_plus(const Graph &G, WeightMap weight,
                  MultiPredecessorMap &predecessors, Vertex s, Vertex t, int k,
                  double theta, Terminator &&terminator)
```

Listing 3.2: *OnePass+ interface*

OnePass+ calling parameters are summarized by the following table:

ESX

ESX algorithm implementation is provided by `esx` function, whose interface is reported in Listing 3.3.

```
template <typename Graph, typename WeightMap, typename MultiPredecessorMap,
          typename Terminator, typename Vertex>
void esx(const Graph &G, WeightMap const &weight,
         MultiPredecessorMap &predecessors, Vertex s, Vertex t, int k,
         double theta, routing_kernels algorithm, Terminator &&terminator)
```

Listing 3.3: *ESX interface*

ESX calling parameters are summarized by the following table:

Parameter	Type	Description
G	In	The input graph on which to compute alternative routes.
$weight$	In	G's weight map, associating to each edge of G a value representing its weight.
$predecessors$	Out	A Multi Predecessor Map to be filled with computed alternative routes.
s	In	The source node.
t	In	The destination node.
k	In	The number of alternative paths to computed.
$theta$	In	The similarity threshold.
$terminator$	In	The terminating policy to apply. By default, *always-continue* policy is used. More details are available in Table 3.3.3.

Parameter	Type	Description
G	In	The input graph on which to compute alternative routes.
$weight$	In	G's weight map, associating to each edge of G a value representing its weight.
$predecessors$	Out	A Multi Predecessor Map to be filled with computed alternative routes.
s	In	The source node.
t	In	The destination node.
k	In	The number of alternative paths to computed.
$theta$	In	The similarity threshold.
$algorithm$	In	The *routing kernel* to employ for shortest path computation. ARLib provides implementations of the following shortest path kernels: Dijkstra's Algorithm [7], A*Star search and Bidirectional Dijkstra [35].
$terminator$	In	The terminating policy to apply. By default, *always-continue* policy is used. More details are available in Table 3.3.3.

Penalty

Penalty algorithm implementation is provided by `penalty` function, whose interface is reported in Listing 3.4.

```
template <typename Graph, typename WeightMap, typename MultiPredecessorMap,
          typename Terminator, typename Vertex>
void penalty(const Graph &G, WeightMap const &original_weight,
             MultiPredecessorMap &predecessors, Vertex s, Vertex t, int k,
             double theta, double p, double r, int max_nb_updates,
             int max_nb_steps, routing_kernels algorithm, Terminator &&terminator)
```

Listing 3.4: *Penalty interface*

Penalty calling parameters are summarized in Table 3.1.

Terminator

Terminator class abstracts a *terminating policy* for an ARP algorithm execution. ARP algorithms promise to verify terminating condition at each main loop iteration. Indeed, searching for a valid set of alternative routes involves expanding a search tree or running shortest path queries multiple times until a valid solution is discovered.

On the other hand, ARP algorithm caller might not be interested anymore in a solu-

Table 3.1: *Penalty calling parameters documentation.*

Parameter	Type	Description
G	In	The input graph on which to compute alternative routes.
$weight$	In	G's weight map, associating to each edge of G a value representing its weight.
$predecessors$	Out	A Multi Predecessor Map to be filled with computed alternative routes.
s	In	The source node.
t	In	The destination node.
k	In	The number of alternative paths to computed.
$theta$	In	The similarity threshold.
p	In	The penalty factor for edges in the candidate path.
r	In	The penalty factor for edges incoming and outgoing to/from vertices of the candidate path.
$max\_nb\_updates$	In	The maximum number of times an edge can be penalized.
max_nb_steps	In	The maximum number of steps of the algorithm.
$algorithm$	In	The *routing kernel* to employ for shortest path computation. ARLib provides implementations of the following shortest path kernels: Dijkstra‘s Algorithm [7], A*Star search and Bidirectional Dijkstra [35].
$terminator$	In	The terminating policy to apply. By default, *always-continue* policy is used. More details are available in Table 3.3.3.

tion if some condition holds, for instance, if too much time has past. By leveraging a Terminator, we can flexibly specify if and when to quit the search.

A Terminator concrete class should expose the interface reported in Listing 3.5.

```
struct terminator {
  bool should_stop() const;
};
```

Listing 3.5: *Terminator interface*

ARLib provides two terminating policies to satisfy two common user needs: *always-continue* and *timer*.

Always-continue is the identity Terminator, that is, it never explicitly stop an ARP algorithm execution. More practically, a call to $should\_stop()$ always returns $false$.

On the other hand, *timer* implements a time-out strategy and its interface is reported in Listing 3.6.

```
class timer {
public:
  explicit timer(std::chrono::milliseconds timeout);
  explicit timer(std::chrono::microseconds timeout);

  bool should_stop() const;
};
```

Listing 3.6: *Timer interface*

A *timer* is constructed with a time-out and it keeps track of the instant t_0 it is instantiated. Upon $should\_stop()$ invocation, another time instant t_1 is collected. If past time between t_1 and t_0 is greater than time-out, the function returns $true$. Otherwise it returns $false$.

Bidirectional Dijkstra

Bidirectional Dijkstra [35] implementation is provided by `bidirectional_dijkstra` function. Bidirectional Dijkstra is a speed-up variant of well-known Dijkstra‘s algorithm particularly suitable for large graphs to discover a shortest path more efficiently. Just like other shortest path algorithms officially supported by Boost.Graph, ARLib‘s Bidirectional Dijkstra exposes a rich overload-set to support ease-of-usage and flexibility when required.

The most basic function interface is reported in Listing 3.7.

```
template <typename Graph, typename PredecessorMap, typename DistanceMap,
          typename WeightMap, typename BackGraph, typename BackWeightMap,
          typename BackIndexMap, typename Vertex>
void bidirectional_dijkstra(const Graph &G, Vertex s, Vertex t,
                            PredecessorMap predecessor, DistanceMap distance,
                            WeightMap weight, const BackGraph &G_b,
                            BackWeightMap weight_b, BackIndexMap index_map_b)
```

Listing 3.7: *Bidirectional Dijkstra interface*

Bidirectional Dijkstra calling parameters are summarized by the following table:

Parameter	Type	Description
G	In	The input graph on which to compute shortest path.
s	In	The source node.
t	In	The destination node.
$predecessor$	Out	The predecessor map records the edges in the shortest path tree, the tree computed by the traversal of the graph. Upon completion of the algorithm, the edges $(p[u], u)$ for all u in V are in the tree. The shortest path from vertex s to each vertex v in the graph consists of the vertices v, $p[v]$, $p[p[v]]$, and so on until s is reached, in reverse order. It must be a Read/Write Property Map whose key and value types are the same as the vertex descriptor type of the graph.
$distance$	In	The shortest path weight from the source vertex s to each vertex in the graph G is recorded in this property map. The shortest path weight is the sum of the edge weights along the shortest path.
$weight$	In	G's weight map, associating to each edge of G a value representing its weight.
G_b	In	The reverse graph of G, i.e. a graph equal to G with all the edge orientations reversed.
$weight\_b$	In	Weight map for G_b
$index\_map\_b$	In	This maps each vertex of G_b to an integer in the range $[0, num\_vertices(G\_b))$

Several additional overloadings of `bidirectional_dijkstra` function, which we are omitting here for the sake of brevity, are provided by ARLib in case additional information about reverse graph is required.

Uninformed Bidirectional Pruning

Uninformed Bidirectional Pruning [35] implementation is provided by `uninformed_bidirectional_pruner` function. Uninformed Bidirectional Pruning leverage

Bidirectional Dijkstra‘s search tree to filter all those vertices in the graph that are too far from both *source* and *destination* nodes according to some pruning factor τ.

Uninformed Bidirectional Pruning interface is reported in Listing 3.8.

```
template <typename Graph, typename WeightMap, typename RevWeightMap,
          typename Vertex>
PrunedGraph<Graph>
uninformed_bidirectional_pruner(const Graph &G, WeightMap const &weight_f,
                                boost::reverse_graph<Graph> const &rev_G,
                                RevWeightMap const &weight_b, Vertex s,
                                Vertex t, double tau)
```

Listing 3.8: *Uninformed Bidirectional Pruning interface*

Uninformed Bidirectional Pruning calling parameters are summarized by the following table:

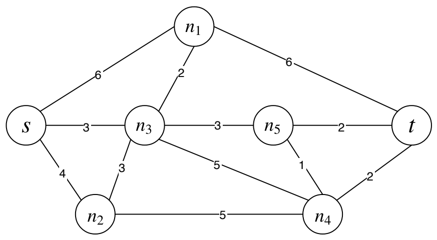
Parameter	**Type**	**Description**
G	In	The input graph to prune.
$weight\_f$	In	G's weight map, associating to each edge of G a value representing its weight.
rev_G	In	The reverse graph of G, i.e. a graph equal to G with all the edge orientations reversed.
$weight\_b$	In	Weight map for G_b
s	In	The source node.
t	In	The destination node.
tau	In	The pruning factor.

`uninformed_bidirectional_pruner` function returns a *Pruned Graph*, that is, an adaptor wrapping input graph G filtering out pruned vertices and edges.

3.3.4 Usage Examples

In this section, we illustrate the functionalities of ARLib by means of a running example that shows how the library gracefully integrates with BGL. The use-case presented hereafter considers a case where we interested in knowing **3** alternative paths from node s to node t that are dissimilar to each other by at least **50%**. Because of the simple topology of the target graph, we do not perform any *pre-pruning*, but we are interested in exploring the, possibly different, solutions that `OnePass+`, `ESX` and `Penalty` compute. Figure 3.3a shows the target problem graph. Even though the graph is still human-manageable, the edge degree of every node and the relatively similar weights make it possible to find non-trivial alternative routes. In Listing 3.9 we report the code required to accommodate this use-case.

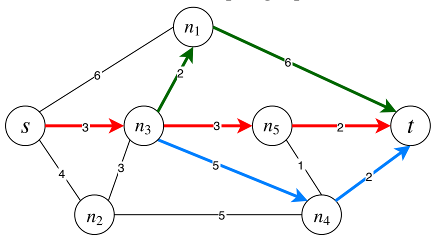
Line 5 defines the graph structure and properties types, the same way the user is customary in BGL applications. Thanks to the generic definition of ARLib functions, any type that satisfies the required graph *concept* will work. Between Line 13 and Line 24 the graph shown in Figure 3.3a is constructed. Remainder lines report the typical snippet of code that an ARLib user would write. Line 27 instantiates a `multi_predecessor_map` object to store the alternative routing solution, as described in Section 3.3.1. Line 31 collects the *weight* property map from input graph, mapping,



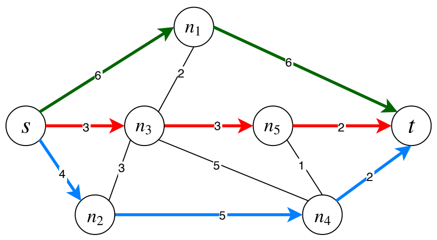
(a) *The sample graph*

Alg.	Alternative paths
`OnePass+` `ESX`	$\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$ $\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$ $\langle(s,n_3),(n_3,n_1),(n_1,t)\rangle$
`Penalty`	$\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$ $\langle(s,n_2),(n_2,n_4),(n_4,t)\rangle$ $\langle(s,n_1),(n_1,t)\rangle$

(b) *Discovered solutions*



(c) *OnePass+ and ESX (same) solutions*



(d) *Penalty solution*

Figure 3.3: *Alternative routing solutions on a sample graph, still human-manageable, but complex enough to admit non-trivial alternative paths. Results are obtained by running the code illustrated in Section 3.3.4.*

to each edge, a weight value. Line 37 effectively runs the alternative routing algorithm. For the sake of clarity, Listing 3.9 reports only an example of `Penalty` execution. Since in the use-case we were interested in exploring the solution of all the alternative routing algorithms, it would simply require to instantiate a new `multi_predecessor_map` for each other algorithm and call `arlib::onepass_plus` and `arlib::esx` functions. Line 39, most notably, highlights the possibility to change the internal *routing kernel* of `Penalty`. Line 41 uses `arlib::to_paths` function to pack the solutions embedded in the `multi_predecessor_map`, into a vector of *views* (`arlib::Path`) on the input graph. Such *view* avoids any unnecessary copies of the discovered alternative paths. In fact it simply exposes only those vertices and edges of the input graph that make up a found alternative route. Finally, discovered solutions are reported in Table 3.3b and pictured in Figures 3.3c and 3.3d.

A tutorial providing step-by-step documentation of the library usage is included on the ARLib repository[1].

3.4 An ARP Design Space Exploration

In previous sections, requirements for the end-user specified that our ideal car navigation service should provide a solution with the best quality possible within a fixed response time that is considered acceptable by the user. If such constraints are not met, the user is likely to stop waiting for a results, labeling the service as not working effectively and switching to other competitors application. As previously stated, the quality of a path is perceived by the user in terms of least travel time. To compute it, tough, a number of alternative paths must be discovered beforehand. The more

[1]`https://github.com/leonardoarcari/arlib`

```cpp
#include <arlib/penalty.hpp>
#include <arlib/graph_utils.hpp>

// Create type-aliases for the Graph type
using Graph = boost::adjacency_list<boost::vecS, boost::vecS,
                                    boost::bidirectionalS, boost::no_property,
                                    boost::property<boost::edge_weight_t, int>>;
using Vertex = typename boost::graph_traits<Graph>::vertex_descriptor;
using Edge = typename boost::graph_traits<Graph>::edge_descriptor;

int main() {
  // Classic BGL graph creation
  enum { S, N1, N2, N3, N4, N5, T };
  const long unsigned num_vertices = T;

  const auto edges = std::vector<std::pair<int, int>>{
      {S, N1},  {N1, S},  {S, N2},  {N2, S},  {S, N3},  {N3, S},
      {N1, T},  {T, N1},  {N3, N1}, {N1, N3}, {N3, N5}, {N5, N3},
      {N3, N2}, {N2, N3}, {N3, N4}, {N4, N3}, {N2, N4}, {N4, N2},
      {N5, T},  {T, N5},  {N5, N4}, {N4, N5}, {N4, T},  {T, N4}};

  const auto weights = std::vector<int>{6, 6, 4, 4, 3, 3, 6, 6, 2, 2, 3, 3,
                                        3, 3, 5, 5, 5, 5, 2, 2, 1, 1, 2, 2};
  auto G = Graph{edges.begin(), edges.end(), weights.begin(), num_vertices};

  // Alternative routing section
  auto predecessors = arlib::multi_predecessor_map<Vertex>{};

  int k = 3;                                      // Nb alternative routes
  double theta = 0.5;                             // Overlapping threshold
  auto weight = boost::get(boost::edge_weight, G); // Get Edge WeightMap


  double p = 0.1, r = 0.1;                        // Penalty algorithm
  int max_nb_updates = 10, max_nb_steps = 100000; // own parameters

  arlib::penalty(G, weight, predecessors, s, t, k, theta, p, r,
                 max_nb_updates, max_nb_steps,
                 arlib::routing_kernels::bidirectional_dijkstra);

  auto alt_paths = arlib::to_paths(G, predecessors, weight, s, t);
}
```

Listing 3.9: *Example of an ARLib application, using the* `Penalty` *algorithm.*

paths discovered, higher the chances to find the fastest. On the other hand, among the stages composing our navigation system pipeline, alternative route planning is the step that takes the longest and searching for a too high number of alternative paths might quickly exhaust the available time slot. Moreover, alternative routes should not overlap too much, otherwise a PTDR step would predict equivalent traveling times, which is hardly desirable from the traffic optimization perspective.

This view shows the presence of a trade-off in the alternative routing step concerning the quality of the solutions and the time to compute them, making room for an analysis on the impact of the goodness and execution time requirements on the system configuration. In this work, three ARP algorithms from the literature, and implemented in ARLib, have been taken into account for a performance comparison:

- k-SPwLO OnePass+
- k-SPwLO ESX (A* Star routing kernel)
- Penalty (BiDirectional Dijkstra routing kernel)

Considering the ARP problem formulation, each algorithm is then parameterized to manage the solution quality and hence the execution time:

- The number of alternative paths, k
- The overlap ratio threshold, θ

The main interest of this research is to gather information about the behavior of each algorithm, subject to parameters and input variation, to understand whether there exists one dominating all the others or if one algorithm is better under certain circumstances and worse under others, enabling the selection of one or another to carry out the best solution depending on the environment requirements. Hereinafter the conducted experiment is reported. First, we introduce the experiment setup.

Datasets The analysis is performed on Milan and New York City areas maps. The Milan map was retrieved from the OpenStreetMap [44] database, whereas the New York City area map is part of the 9th DIMACS Implementation Challenge dataset [45]. We chose these two locations in order to explore the behavior of ARP algorithms on cities with very different topologies. Indeed, New York City, especially the Manhattan borough, shows a grid structure while Milan exposes a structure made of sectors with a central hub. Since we would like to obtain a view as wide as possible, sticking with a single urban model would have biased the experiment conclusions. For instance, in the frame of discovering several alternative routes with limited overlap, a grid structure, like Manhattan, would make the task easier since at any crossroad the algorithm could take a detour in order to satisfy the overlap constraints. This might be harder to do in case of topologies exposing a limited number of fast lines that connect the city sectors to the central hub.

Query design A relevant design choice is how to build the *source-destination matrix*, i.e. the set of source-destination pairs on which to execution an ARP query. In some other works [3,8] those points are chosen completely at random, by uniformly sampling from 100 to 1000 source-destination pairs out of the node set N. In our opinion, this approach lacks of robustness, because it hardly mimics the way a user would stimulate the car-navigation service. Indeed, we are interested in gathering the strengths and weaknesses of the ARP algorithms for non-trivial queries, i.e. for which the source and destination points are at significant distance. Running just random queries leaves no control on which nodes are selected, possibly executing ARP for points very close to each other, hence leading to uninteresting results. Therefore, we followed a strategy proposed by the 9th DIMACS Implementation Challenge [45], formally named *Point-to-Point problem*, which generates source-destination pairs as follows:

1. Select the source node s at random from the nodes set N.
2. Run Dijkstra's algorithm from s to compute the *rank* of all the nodes in the graph, where a node has a rank of t if it is the t-th node scanned by the algorithm.
3. For a set of values of i, select a random vertex v_i from vertices with ranks in range $[2^i, 2^{i+1})$ and output an $s - v_i$ pair.

The power of this strategy resides in the configurability that it offers to build the source-destination matrix. Indeed, given some value of i, we can generate as many source-destination pairs as we need and they would all be set at a comparable distance. This way, instead of generating query points at unknown distance, we are able to control the number of pairs equivalently far, in order to build a meaningful statistical value for this experiment. Both from the Milan and the New York City maps, we chose $i \in [9, 17]$, as suggested in [45], and generated 50 source-destination pair for each range $[2^i, 2^{i+1})$, summing up to 450 samples.

Concerning New York City, the maximum flight distance between two nodes in the source-destination matrix is 100Km, with a 25th quantile of 3.83Km and a 75th quantile of 31.68Km. Regarding Milan, the maximum flight distance between two nodes in the source-destination matrix is 26Km, with a 25th quantile of 1.67Km and a 75th quantile of 9.2Km.

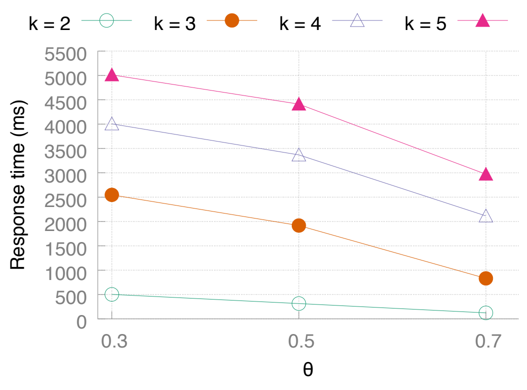
Problem constraints Recalling that the system is supposed to return a single route to the end-user, parameters k and θ have a relevant impact on the solution quality and execution time, as mentioned above. Therefore, values for k are chosen in the set $\{2, 3, 4, 5\}$, while for θ in the set $\{0.3, 0.5, 0.7\}$. Concerning the former, having too many alternative paths resulted unworthy according to some initial experiments, leading to a small increase in the solution quality at cost of an unacceptable execution time. About the latter, finer-grained similarity thresholds, say to the second decimal digit, showed irrelevant changes both in the execution time and in the quality.

3.5 Experimental Results

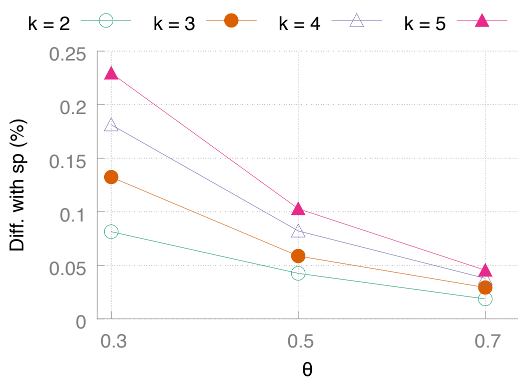
The experimental results presented hereinafter were collected on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (16 cores, 20MB Cache), running Ubuntu 16.04.5 LTS and exploring 8 configurations in parallel.

3.5.1 Performance

In this section, we present the obtained results from the experiment previously described. As already mentioned in section 3.4, we are interested in understanding the

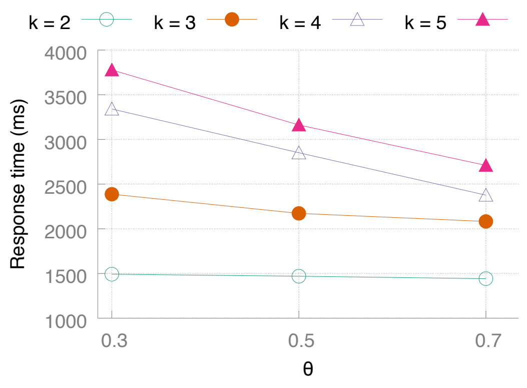


(a) *Execution time*

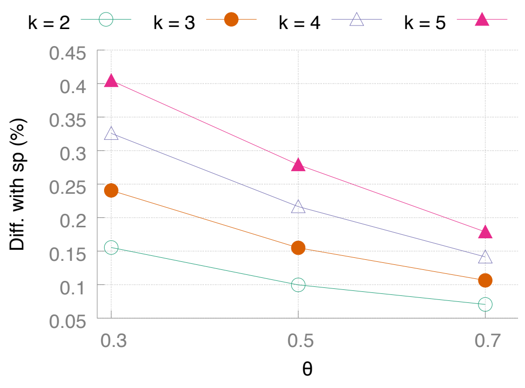


(b) *Quality*

Figure 3.4: *OnePass+ performance on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better).*



(a) *Execution time*



(b) *Quality*

Figure 3.5: *ESX performance on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*

AR algorithms behavior under different (k, θ) configurations. Our question, indeed, is whether there exists an algorithm that dominates all the others or if one is the best only under certain conditions.

In Figure 3.4a, Figure 3.5a and Figure 3.6a we plot the gathered time measurements of OnePass+, ESX and Penalty respectively. On the x-axis we report the similarity threshold θ, on the y-axis the average execution time over all the requests and we draw a line for each number of alternative paths k. In Figure 3.4b, Figure 3.5b and Figure 3.6b we plot the quality measurements of OnePass+, ESX and Penalty respectively. On the x-axis we report the similarity threshold θ, on the y-axis the average $\mathbf{spDifference}$ quality metric and we draw a line for each number of alternative paths k.

At a very first look, to also simplify our analysis, we observe that, from the pictured point of view, ESX algorithm is always dominated by Penalty both from response time and quality perspective. Therefore, we are not going to consider it in the following discussion.

To begin the analysis of the results, let us restrict to the case where $\theta = 0.7$, reminding that high values of similarity threshold θ mean that solutions are more similar. By

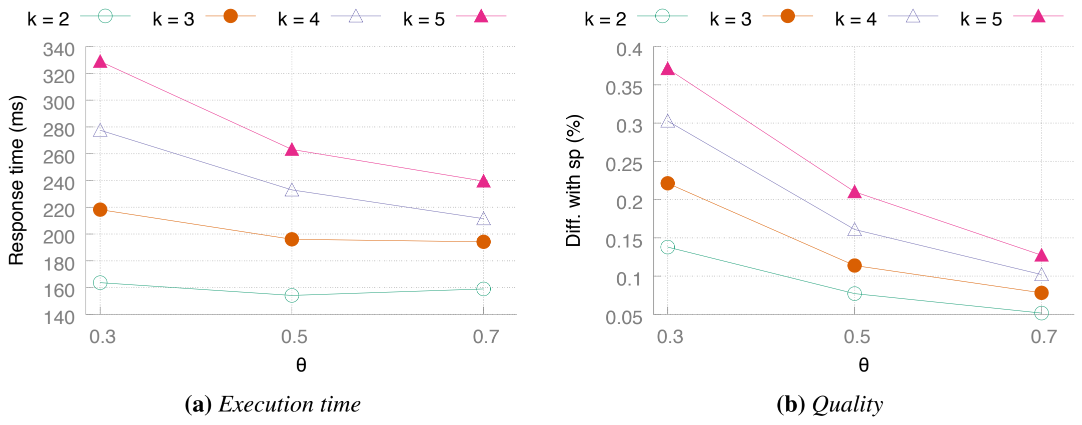


Figure 3.6: *Penalty performance on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better).*

comparing OnePass+ and Penalty, we notice that for a number of alternative paths equal to 2, OnePass+ performs better than Penalty both in terms of execution time, though slightly, and of quality, keeping the discovered paths under 5% of length difference with respect to the shortest path. Assuming that the provider of a car-navigation service is fine with providing the shortest path along another possible alternative, guaranteed to be at least 30% different from the shortest path, then OnePass+ represents the optimal algorithm with no better options.

The picture changes if we consider a high number of alternative paths. As we already mentioned in section 3.4, the more paths we compute, the higher the chances PTDR module has to discover a faster road. If we look for higher values of k we notice that OnePass+ execution time raises up to 5 seconds on average, while Penalty never takes more than 250ms. On the other hand, whilst Penalty quality decreases reaching a value of almost 15%, OnePass+ consistently maintains an $\mathbf{spDifference}$ under 5%.

This comparison supports our previous claim on the existing trade-off between quality and execution time among Alternative Routing algorithms. In particular, from this analysis, we understand that on average OnePass+ offers the best quality of results, never exceeding 25% of length difference with respect to the shortest path. At the same time, Penalty offers the fastest execution time trading some quality in the solutions, reaching more than 35% of $\mathbf{spDifference}$ for a number of paths equal to 5 and a similarity threshold equal to 0.3.

3.5.2 Visual Comparison

Comparisons between algorithms are also visually appreciable in Figure 3.7, Figure 3.8 and Figure 3.9, where we draw the discovered paths for each algorithm between a pair of source-destination nodes in Milan and New York City. As illustrated, OnePass+ tends to find solutions that are more compact around the shortest path, trying to keep Equation (2.3) at minimum. ESX and Penalty, on the other hand, discover paths by penalizing edges on the graph, leading to more disperse solutions. While this might looks as a downside, because computed paths are generally much longer than the shortest one, this may lead to solutions more robust to potential viability reductions. Indeed, if alternative paths share the least number of segments, an event that would make some paths very slow to travel (e.g. car accident, roadworks) would make other, non-overlapping, paths faster and therefore still leading to a satisfactory solution. Whether to prefer non-overlapping or closer-to-shortest-path solutions is therefore more a *service policy* than a clear aspect of the ARP problem. The strength of ARLib is the flexibility that it offers to the developer, leaving the freedom to explore the possible approaches to the matter and letting the system designer decide which policy to adopt.

3.5.3 Failure Rate

In subsection 3.5.1, we showed a comparison among algorithms under varying configurations to understand the existing trade-offs between execution time and quality of results. In this section, we extend the comparison from a different point of view: the completion rate.

The reason why we are interested in this comparison is clear if we recall what a valid ARP solution is. As we mentioned in Definition 2.1.7, a query to an AR algorithm is expected to lead to a set of k-alternative paths, non-overlapping among each other more than a factor θ. Moreover, since we are interested in a usable car-navigation system, we can only afford to wait for a solution within a t^* time frame. Accordingly, a query to an algorithm might fail for the following reasons:

1. A request for k alternative paths was issued, but no k paths exist between source and target nodes.
2. A request for k alternative paths was issued, but no k paths exist between source and target nodes that do not overlap less than a factor θ.
3. A request for k alternative paths was issued, a solution exists, but the algorithm pruned, during the search, a path that would have been part of the solution, falling back to cases 1 or 2.
4. A request for k alternative paths was issued, a solution exists, but t time units have passed and the procedure times out.

Consequently, a relevant deciding factor to which algorithm to adopt under certain (k, θ) configuration should be to what extent the algorithm fails in providing a valid solution.

In Figure 3.10, we report the experimental results on the completion rate for all the algorithms. On the left column, we illustrate the experiment run on the New York City map, while on the left the experiment executed on the Milan map.

On each x-axis, we indicate the similarity threshold θ values, on the y-axis the average completeness ratio across all the queries. Then we plot one bar for each value of k. In this experiment, we consider a solution to an ARP query valid if:

1. Exactly k alternative paths are discovered.
2. For each pair of computed alternative paths, the similarity of the two is less than or equal to θ.
3. A solution is computed in less than 20 seconds.

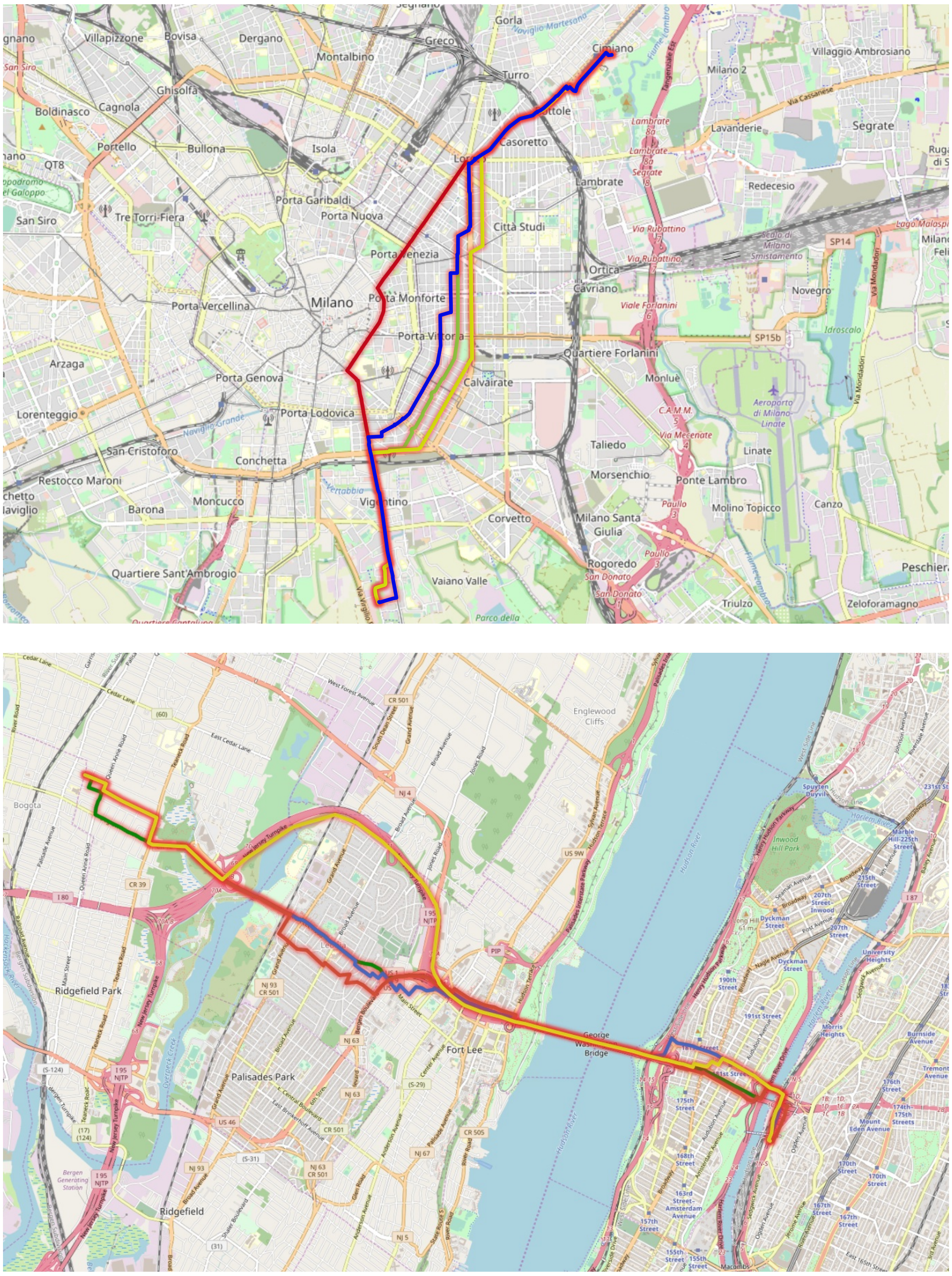


Figure 3.7: *OnePass+ 4-alternative-paths on Milan area (above) and New York City area (below) with 50% similarity threshold*

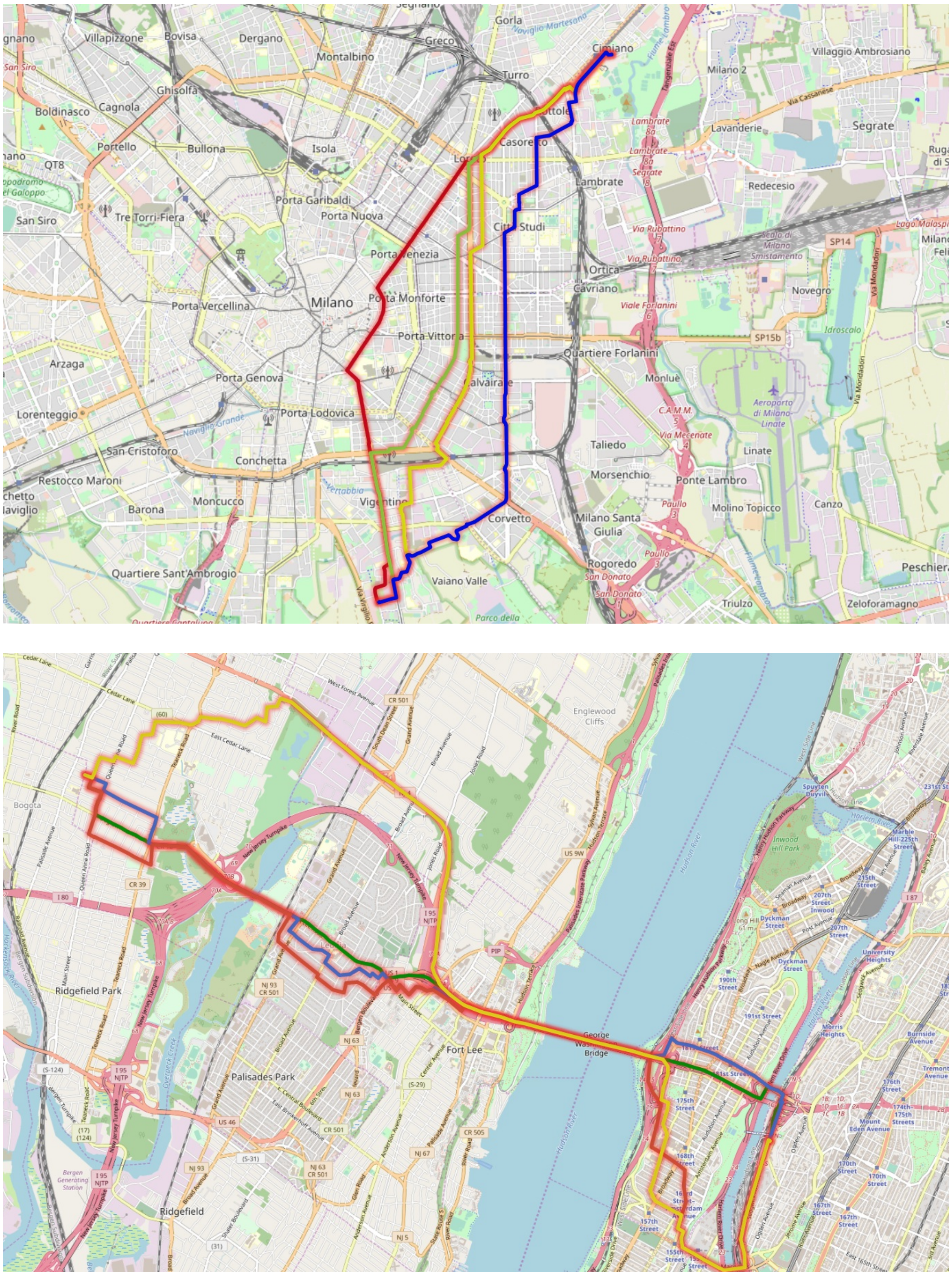


Figure 3.8: *ESX 4-alternative-paths on Milan area (above) and New York City area (below) with 50% similarity threshold*

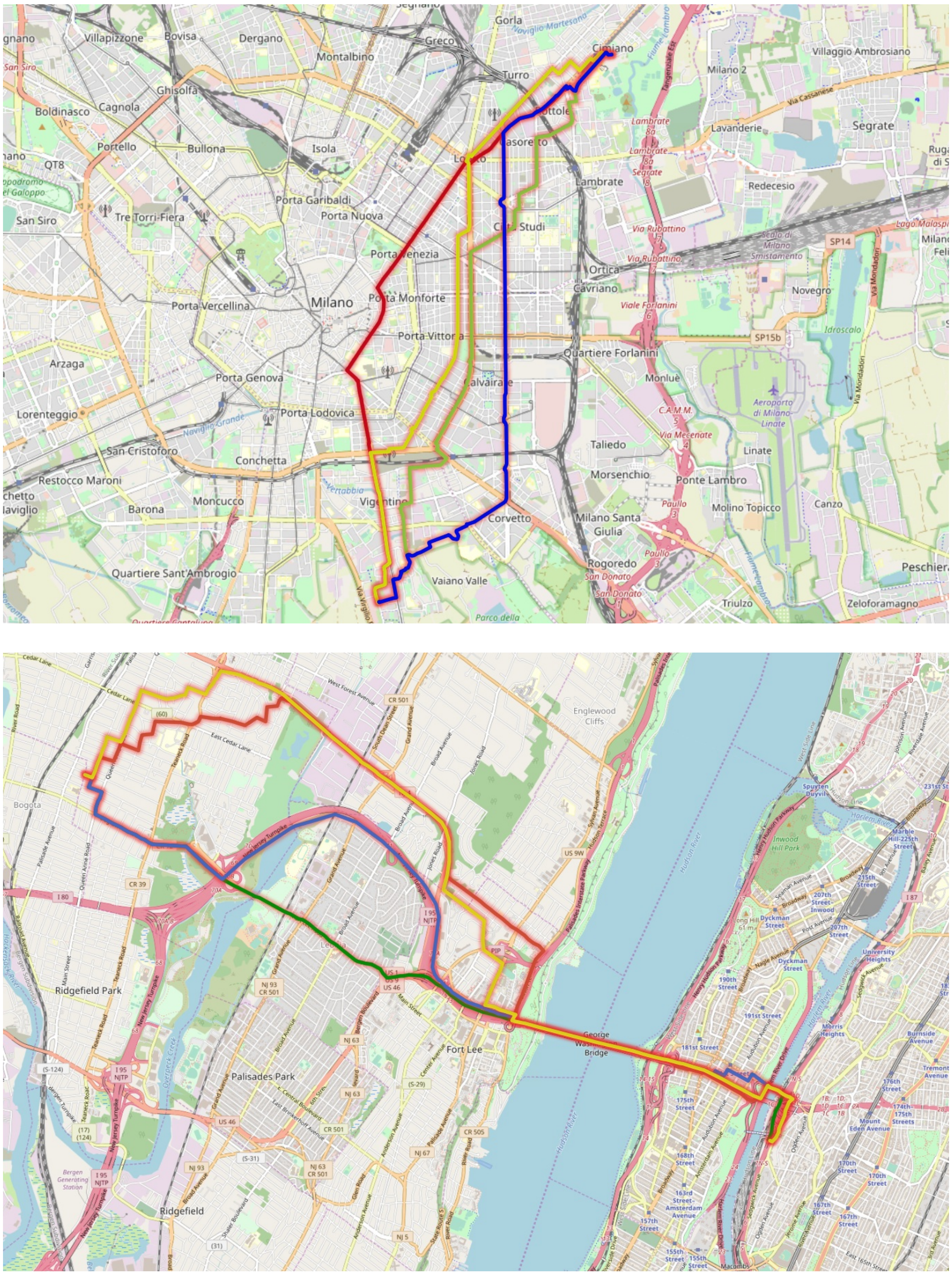


Figure 3.9: *Penalty 4-alternative-paths on Milan area (above) and New York City area (below) with 50% similarity threshold*

In Figure 3.10a and 3.10b we illustrate the completeness ratio of OnePass+. For an increasing number of alternative routes requested, we notice that OnePass+ fails more often to discover a valid solution, dropping to less than 80% of successful queries when requesting very dissimilar paths.

In Figure 3.10e and 3.10f we report the completeness ratio of Penalty. It is straightforward to conclude that the algorithm always finds the requested number of alternative routes, with similarity constraints satisfied, within a 20 seconds time frame.

3.6 An Adaptive Policy

In sections 3.1 and 3.2 we outlined use-cases and requirements that we would like our proposed car-navigation system to have, in order to guide our design towards a solution that both the service provider and users would enjoy. In subsection 3.5.1 we reported our experimental results that justify our interest in an adaptive solution, showing that no optimal algorithm exists leading to both low execution time and best quality of results. In subsection 3.5.3 we pictured another experiment to quantify the number of times each algorithm is able to return a valid solution, varying the (k, θ) configuration, within a time frame of 20 seconds.

In this section we leverage the aforementioned concepts to describe a first *adaptive policy* that a car-navigation service provider, e.g. the municipality, could implement to maximize its Quality of Service (QoS). According to our proposed policy, it can be done by concurrently:

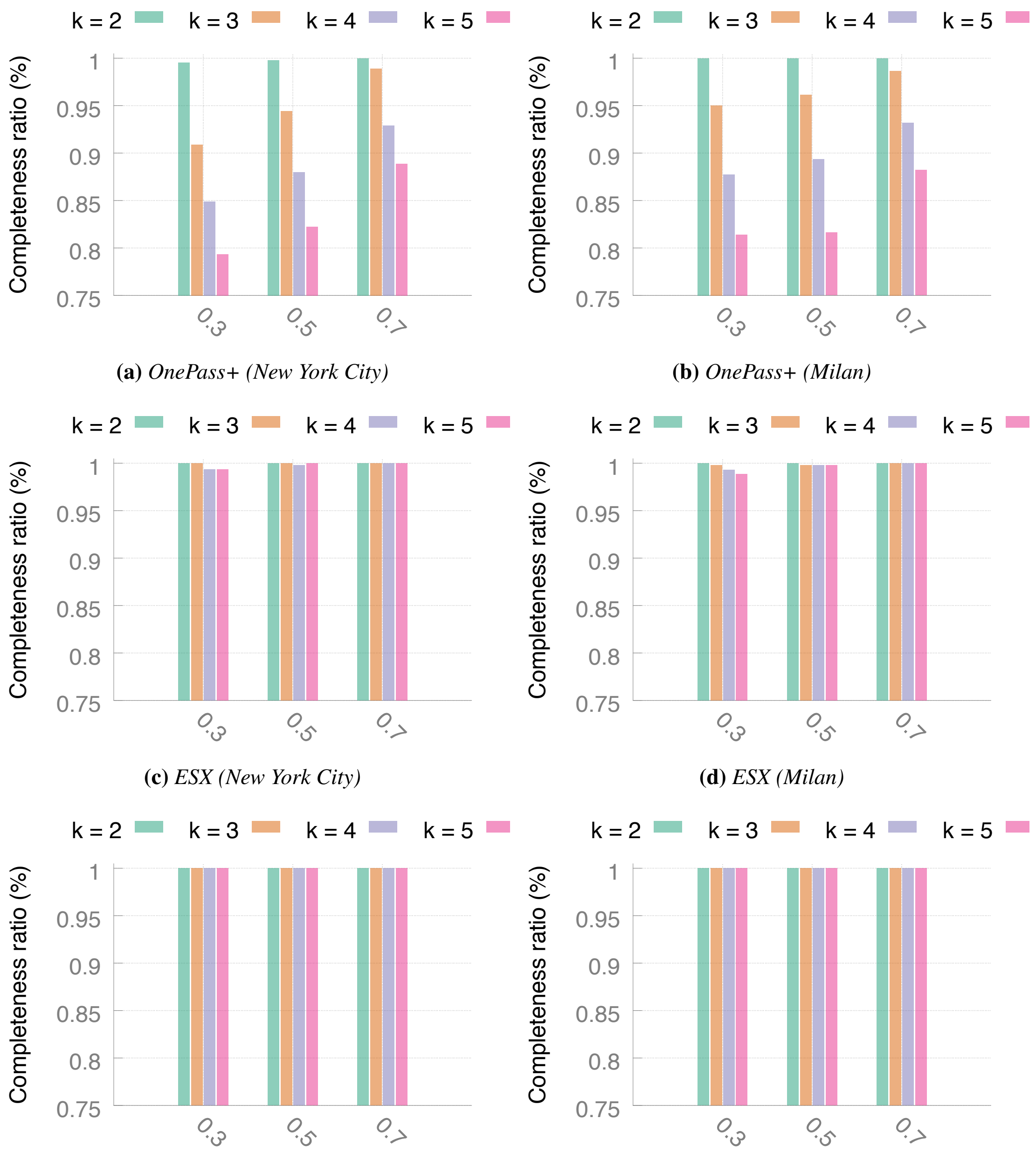
- Keeping the latency between the first routing request and a solution as low as possible.
- Keeping the results quality as high as possible.

In order to achieve our goal, we rely on the following observation. In a car-navigation system there are three types of requests:

1. **First routing request**: the user turns up the navigation client (e.g. a mobile application) and asks for the fastest route to some destination
2. **Re-rerouting request (type-A)**: the user misses a turn suggested by the navigator which now should recompute the fastest route. This generally is a very easy task because it simply requires a shortest-path-query to the previously suggested fastest route. For this reason, we are not considering this type of request in our analysis.
3. **Re-rerouting request (type-B)**: the navigator client periodically checks for the best route, which might have changed. Indeed, since the fastest path strongly depends on the traffic conditions, the user could be redirected to another path that would lower the expected arrival time.

Consequently, we elaborate the following adaptive strategy:

- The service provider fixes the number of alternative paths k to compute and the similarity threshold θ, depending on the available computing power or service level agreement.



(a) *OnePass+ (New York City)*

(b) *OnePass+ (Milan)*

(c) *ESX (New York City)*

(d) *ESX (Milan)*

(e) *Penalty (New York City)*

(f) *Penalty (Milan)*

Figure 3.10: *Average completeness ratio (%) per query, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$, for all the algorithms with a timeout of 20 seconds.*

- Because the user needs the first solution as soon as possible, we employ Penalty algorithm, which is the fastest in most configurations. This way, the user is ready to drive towards destination even though the suggested route might not be the best possible.

- As the user drives along the suggested path, the car navigator will periodically issue re-routing requests of type-B, checking whether faster paths exist. Because the request is performed in background while the user is driving, we can afford to spend some more time in the alternative routing phase, to potentially discover higher-quality solutions. As shown in Figure 3.4b, OnePass+ always leads to best quality solutions, making it the optimal algorithm for this task.

According to the proposed policy then, the system would provide a fast response to the user to keep the time-to-navigation to the minimum, whilst periodically refining the suggested path. Therefore, we can effectively employ better quality algorithms by hiding their latencies from the user perspective, behind a background activity.

There is one last issue to address, though: the failure rate. Indeed, OnePass+ is the best algorithm in terms of **spDifference** but it also fails to discover a valid result in 20% of the queries for some configurations, as reported in Figure 3.10a and 3.10b. Hereinafter, we propose two actions to mitigate this problem:

- Upon timeout, use the alternative routes computed so far, which will be less than k. The PTDR module will work on a reduced number of paths, potentially missing the best alternative, but the service would still work seamlessly.

- Use a *hybrid approach*. Let t^* be the Alternative Routing module request time-out. Upon a re-routing request of type-B, first compute a solution running Penalty, which will take some time $\bar{t}$. Then, launch a OnePass+ query setting a time-out of $t^* - \bar{t}$. If OnePass+ returns in time, use its high-quality solutions. Otherwise, forward to PTDR module those solutions computed by Penalty.

3.7 Summary

In this chapter, we presented our car navigation system, introducing the three main stages composing it, namely ARP, PTDR and Reordering modules. As the main focus of this thesis, we directed our efforts on ARP step and we introduced ARLib, an efficient, configurable C++ library implementing state-of-the-art alternative routing algorithms.

From a software engineering perspective, ARLib is a stand-alone library that can be imported in every C++ project using Boost.Graph as graph data structure framework, seamlessly integrating with existing BGL code-bases. Thanks to its flexibility, varying user requirements in terms of execution time and quality of results can be accommodated through the range of parameters that ARLib exposes.

From our research point of view, having a performance-oriented implementation of most promising ARP algorithms enabled us to pursue an investigation on the behavior of ARP heuristics from different perspectives. Indeed, sharing the same programming language and graph data structure, ARLib allowed us to explore the available design space, first analyzing if there exists a single algorithm which provides both fast execution time and top solution quality, or, on the other hand, if some heuristic work best only under some configuration while badly behaving under others.

Achieved results highlighted the existing tradeoffs between algorithms, changing from a (k, θ) configuration to another, therefore answering our first research question. The presence of those tradeoffs, thus, represents a promising opportunity to introduce a first level of adaptivity in our car navigation system and leverage those performance differences to raise the overall service quality. As a result, in section 3.6 we proposed a methodology to tailor our navigation system to some service provider needs, depending on the available time budget. In fact, by choosing the right values for k and θ parameters and algorithms to employ for different types of request we can bring the system to run at some working point satisfying some given constraints.

In the next chapter, we are going to take the analysis one step further. In fact, while the experiment carried out in this chapter showed encouraging results, it only displayed values in an *average case* scenario. Our next research question, then, will focus on a finer-grained study, to understand whether and which properties of a routing query influence an ARP algorithm from latency and quality perspectives.

CHAPTER 4

Proactive System Auto-Tuning

In the previous chapter, we conducted an *average case* exploration on the ARP problem design space, systematically analyzing ARP algorithms behavior under varying number of alternative paths k and similarity threshold θ values. Achieved results highlighted the existence of tradeoffs between ARP heuristics, for instance having some algorithm to perform best for small values of k, while demonstrating terrible execution times when searching for high number of alternative paths. In this context, we presented a first adaptive policy for a car navigation system to statically identify a good working point for the service, given some constraints e.g. its response time or required computational resources.

As a consequence of our experiment, we asked ourselves how different the reality could be from an average case analysis. How are execution times distributed for different queries? How much disperse are they with respect to the mean value?

Figure 4.1 depicts ARP algorithms latency for several queries with $k = 4$ and $\theta = 0.5$, by means of box plots. For visualization purposes, we considered only those queries that completed in less than 3 seconds. Still, we are able to draw an immediate conclusion: while both Interquartile Range and box whiskers show consistent results for OnePass+ and Penalty algorithms, executing in less than 500 ms, outliers, together with ESX box plot, exhibit a significant variance in the achieved results, with an important number of samples doubling or tripling the average latency value.

This variable behavior made us question whether this variance should be accounted to some ARP query characteristics and, if so, whether we could use those characteristics to predict heuristics execution time in order to proactively choose the optimal algorithm to employ for each incoming request.

In this chapter, we present a proactive auto-tuning process to raise the level of adaptivity of our car-navigation. In particular, we propose a methodology on implementing

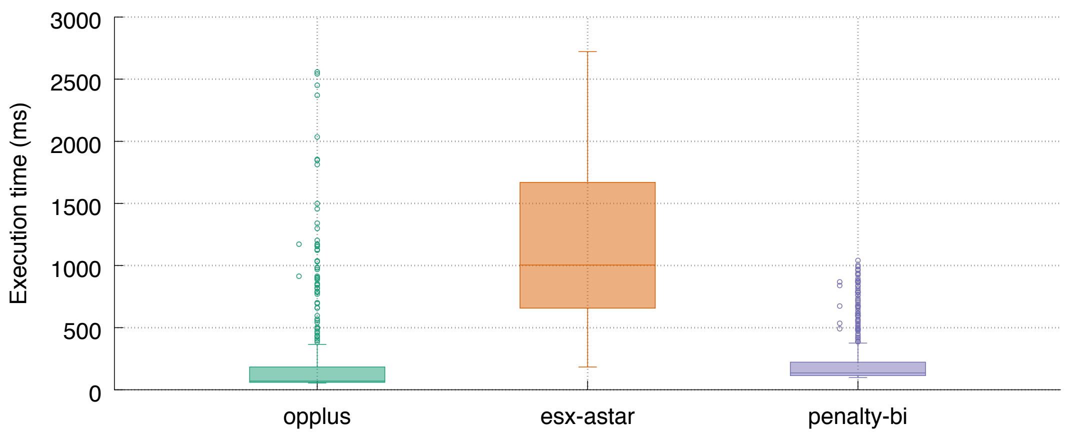


Figure 4.1: *Execution time of ARLib's algorithms search for $k = 4$ alternative paths with $\theta = 0.5$ similarity threshold (lower is better)*

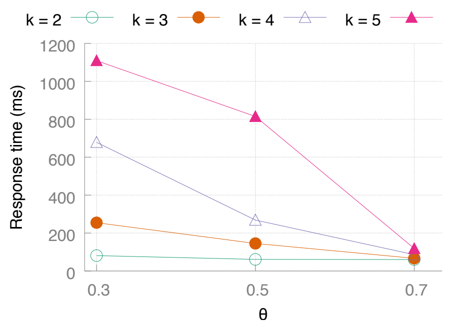
and training a machine learning model for best algorithm prediction depending on requests characteristics. Finally, we extended the adaptive policy presented in the previous chapter to include the auto-tuner recommendations and bring the navigation system adaptivity to a dynamic level.

4.1 A Finer-Grained Analysis

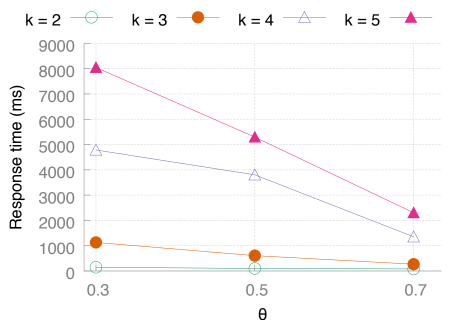
In this section, we are going to investigate our first research question: is there any features we can extract from a routing request influencing ARP heuristics execution time? We begin from the following hypothesis: the higher the number of segments in the shortest path between two nodes, the higher the chances that finding a number of alternative paths is going to take longer. We can easily justify this hypothesis with the following example. Let p_{st} and q_{uv} be two shortest paths between $s - t$ and $u - v$ respectively in a graph G and $|p_{st}|$ and $|q_{uv}|$ be the number of segments in p_{st} and q_{uv} respectively, such that $|p_{st}| < |q_{uv}|$. Let us assume that for each node in p_{st} and q_{uv} there exist 3 out-edges in G. This is reasonable as, in general, nodes represent crossroads in the road network. Let $out(p) = \sum_{v \in p} |outEdges(v)|$ be the number of out-edges for each node in a path, then $out(p_{st}) < out(q_{uv})$. Moreover, since each of the out-edges target nodes has other 3 out-edges, this search space complexity inequality grows exponentially. While it is true that AR algorithms try to prune the search space as much as possible, it is still true that higher the number of segments, the higher the number of candidate alternative routes.

Based on this hypothesis, we conducted the experiment illustrated in Figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7. We follow the same design of experiment as reported in section 3.4, collecting measurements on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (16 cores, 20MB Cache), running Ubuntu 16.04.5 LTS and exploring 8 configurations in parallel.

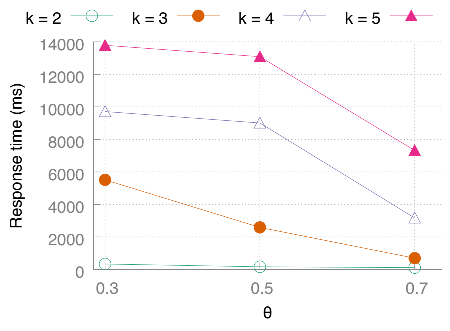
In every figure, we draw four charts. In each of them, we show the obtained results for those queries where the number of segments in the shortest path (*SPS*) falls in a certain range. We consider the following four ranges: $[0, 200)$, $[200, 300$, $[300, 400)$ and $[400, \infty)$.



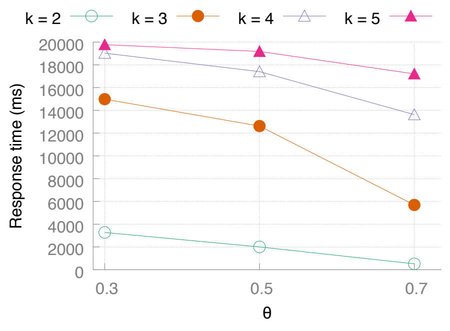
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*

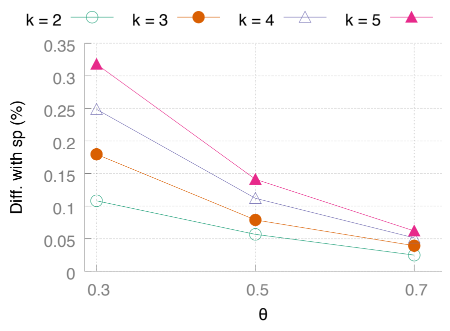


(c) *Segments in the shortest-path: [300, 400)*

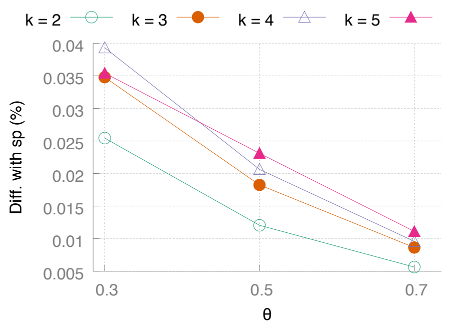


(d) *Segments in the shortest-path: [400, ∞)*

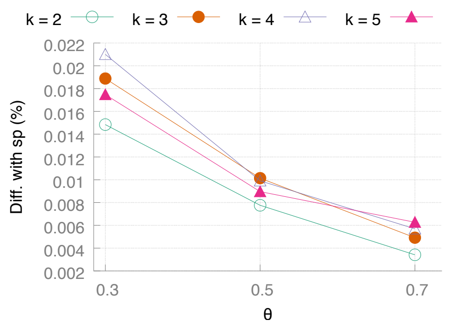
Figure 4.2: *OnePass+ execution time on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*



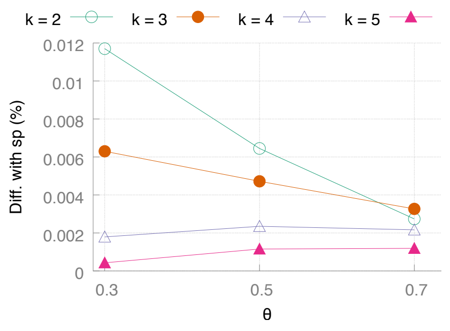
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*

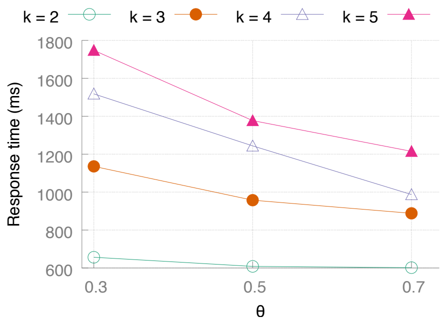


(c) *Segments in the shortest-path: [300, 400)*

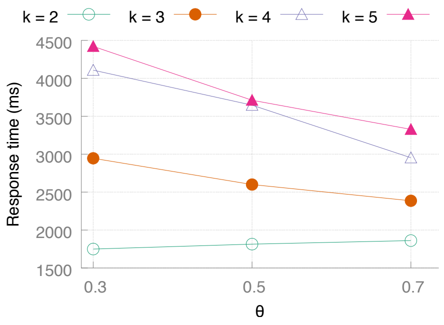


(d) *Segments in the shortest-path: [400, ∞)*

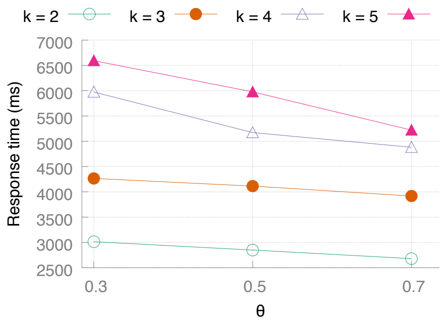
Figure 4.3: *OnePass+ quality on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*



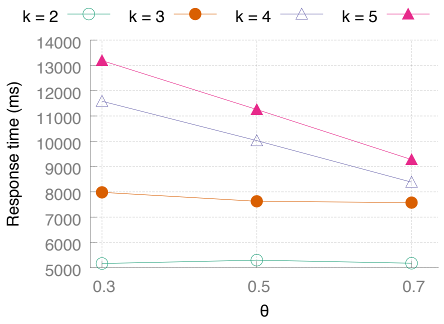
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*

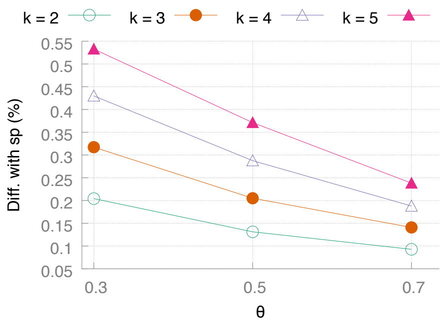


(c) *Segments in the shortest-path: [300, 400)*

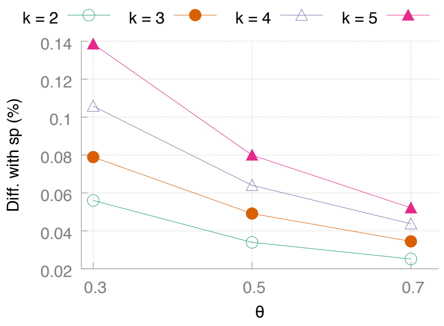


(d) *Segments in the shortest-path: [400, ∞)*

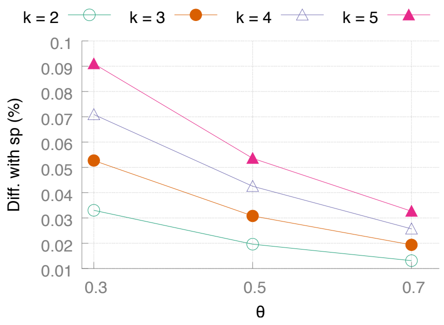
Figure 4.4: *ESX execution time on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*



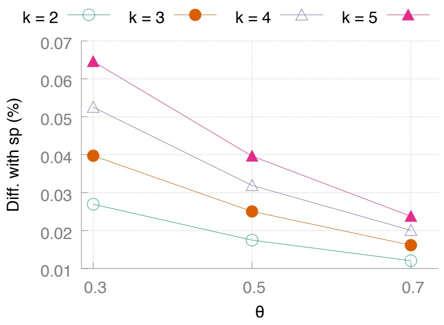
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*

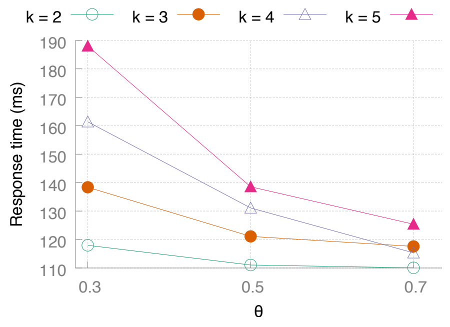


(c) *Segments in the shortest-path: [300, 400)*

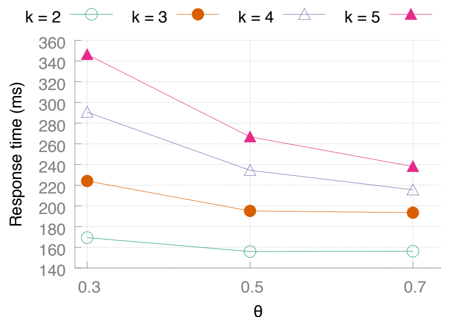


(d) *Segments in the shortest-path: [400, ∞)*

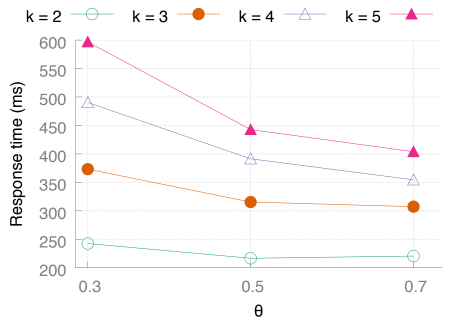
Figure 4.5: *ESX quality on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*



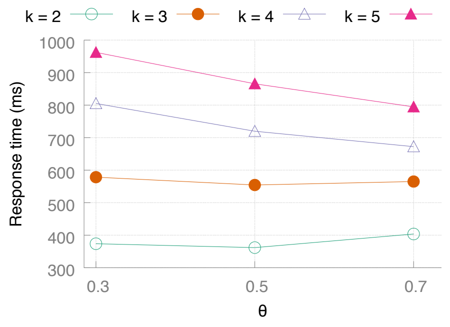
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*

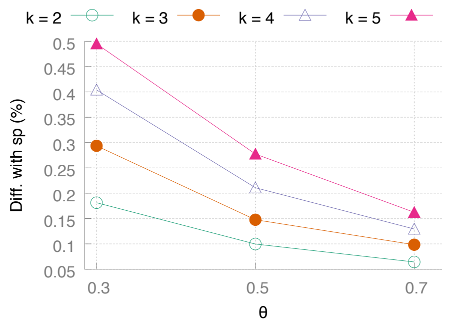


(c) *Segments in the shortest-path: [300, 400)*

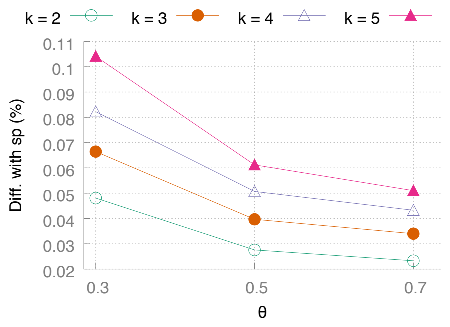


(d) *Segments in the shortest-path: [400, ∞)*

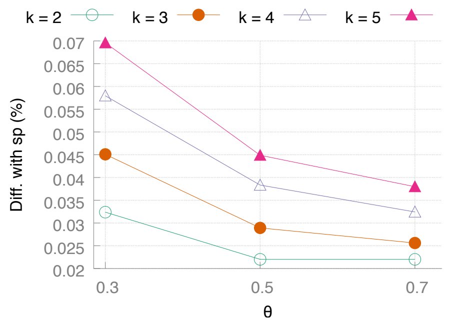
Figure 4.6: *Penalty execution time on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*



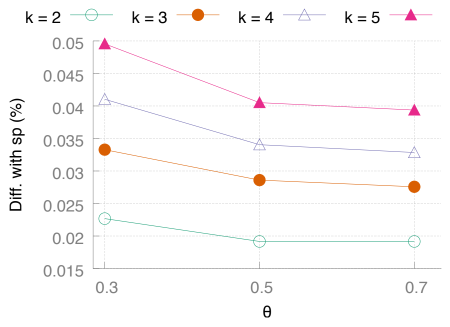
(a) *Segments in the shortest-path: [0, 200)*



(b) *Segments in the shortest-path: [200, 300)*



(c) *Segments in the shortest-path: [300, 400)*



(d) *Segments in the shortest-path: [400, ∞)*

Figure 4.7: *Penalty quality on New York City, varying alternative paths $k \in \{2, 3, 4, 5\}$ and similarity threshold $\theta \in \{0.3, 0.5, 0.7\}$ (lower is better)*

On every x-axis we report the similarity threshold θ values. In Figures 4.2, 4.4 and 4.6, on the y-axis, we plot the average execution time for every range of number of segments in the shortest path. In Figures 4.3, 4.5 and 4.7 on the y-axis, we plot the average **spDifference** for every range of number of segments in the shortest path.

To begin our analysis, we remind the reader that we are comparing algorithms for two use-cases:

1. First-routing request, which should be sufficiently fast to avoid annoying the service user with long waiting times.
2. Re-routing requests of type-B, which can take longer because their latency is hidden behind a background activity of the navigator client. This does not represent a problem since the navigator is already guiding the user along some path.

In both the scenarios, we are interest in choosing the feasible configuration leading to the highest expected quality. With this picture in mind, we move to the illustrated data.

At a very first look, also to simplify our analysis, we observe that, from the average point of view, ESX algorithm is always dominated by either Penalty or OnePass+ both in terms of response time and quality. Therefore, we are not going to consider it in the following discussion. Then, we notice that OnePass+ is confirmed as the best algorithm in terms of average **spDifference** under all configurations and range of SPS, like previously stated in subsection 3.5.1. As opposed to subsection 3.5.1 conclusions, though, the situation concerning the execution time in this experiment is different.

In Figures 4.2a and 4.6a we plot the average execution time for $[0, 200)$ segments range. If we fix our attention on $\theta = 0.7$, we notice that OnePass+ runs faster than Penalty for any number of alternative paths k. Consequently, if the service provider is satisfied with a maximum overlap ration of 70%, for requests whose number of SPS is less than 200, OnePass+ represents the optimal candidate. If we consider $\theta = 0.5$ instead, OnePass+ executes faster than Penalty only for $k = 2$, while runs slower for higher values of k. On the other hand, OnePass+ might still be sufficiently fast. Indeed, if we set a maximum time frame of 500 milliseconds for first-routing requests, the algorithm leads to valid solutions is time for all test values of k but $k = 5$. The same applies for $\theta = 0.3$ and k up to 3. In any case, for SPS in $[0, 200)$ range, OnePass+ is fast enough for re-routing queries of type-B.

Similar conclusions can be drawn on Figure 4.2b and 4.6b, except that OnePass+ becomes unreasonably slow even for type-B re-routing requests for $k = 5$ and $\theta \leq 0.5$. For $k > 2$, then, Penalty represents the only feasible choice for a number of SPS greater than 300.

The first consequence of this experiment is that the hypothesis that we postulated above is actually supported by empirical evidence. Moreover, and most importantly, this experiment shows that, given some (k, θ) configuration, the optimal algorithm changes depending on some routing-request features.

4.2 A Proactive System

In section 4.1, we concluded that at least one feature of the input request, i.e. the number of SPS, exists influencing the choice of the optimal ARP algorithm to employ,

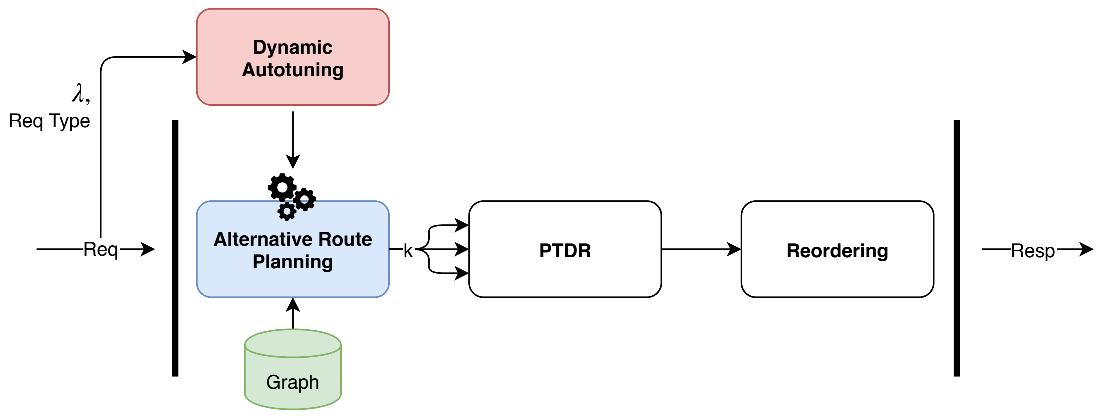


Figure 4.8: *A graphical representation of the proposed car-navigation system pipeline. In red, a Dynamic Auto-tuner selects an ARLib's algorithm and its parameters to maximize the solutions quality, while keeping the system response time within the fixed constraints.*

pairing the already mentioned (k, θ) configuration.

In this section, then, we propose an extension of the adaptive car-navigation system presented in chapter 3 to bring its adaptivity to a finer-grained level. Our idea is to add an intelligent module, a *dynamic auto-tuner*, that, relying on trained machine learning models, chooses the best algorithm depending not only on the routing type of request, but also on upcoming requests properties. The extended system is pictured in Figure 4.8, while an extensive description of the predictive model starts from section 4.4.

Along with an improvement to solutions quality, a dynamic auto-tuner would also bring opportunities from workload management perspectives. Let us consider a system configured and sized to work under some (k, θ) setup and some expected workload. Let us imagine that, suddenly, the system experiences an unforeseen requests spike that would be impossible to handle without introducing strong delays in the overall response time. Having a model of the application behavior enables an auto-tuning process to select another $(k', \theta', algorithm)$ configuration which will degrade the solution quality only that much that the system needs to reduce the AR search time and restore the desired service response time.

Indeed, a predictive model would introduce a number of enhancements to our car navigation system, but are we able to build it in the first place? For instance, previous experiment confirmed that the number of SPS is a good candidate feature, but can we extract it from an upcoming request *before* running any alternative routing algorithm?

We discovered that it is possible. In fact, we notice that all the AR algorithms have a first, fundamental step in common: they all compute the shortest path between the query source-destination pair. Therefore, we propose the following steps to efficiently implement a *dynamic auto-tuning process*:

1. Let s and t be the source and target nodes respectively. Compute the shortest path p_{st} between s and t with any shortest path algorithm (e.g. ARLib's fast Bidirectional Dijkstra).
2. From the input query and the shortest path p_{st}, extract a number of features $\boldsymbol{x}$ characterizing the request (e.g. the number of SPS).
3. Let $g_{k,\theta,\alpha}$ be a trained model for the system (k, θ) configuration and algorithm

$\alpha \in \{\text{OnePass+, ESX, Penalty}\}$. For each algorithm α, let $\hat{t_\alpha} = g_{k,\theta,\alpha}(\boldsymbol{x})$ be the expected execution time for α for $\boldsymbol{x}$. Let t^* be the fixed search timeout. Compute the set $A = \{\alpha \,|\, \hat{t_\alpha} \leq t^*\}$ of the algorithms that terminate in the available time frame.

4. Let $\alpha^* = arg\,max_\alpha \overline{\mathbf{Quality}}(\alpha)$ be the algorithm with the best average quality. For instance, if we employ $\mathbf{spDifference}$ as quality metrics, we pick the α with minimum value.

5. Pass p_{st} to the optimal algorithm α^* as the first path in the ARP solution and continue the search.

In the following sections, we present the input features, the learning model and the training pipeline that we employed to implement a valuable proactive dynamic auto-tuner for a car-navigation system.

4.3 Feature Engineering

In this section, we present the set of input features we extract from a request in order to provide a better characterization.

In the process of engineering the input features, we should keep the following list of properties in mind:

- Since we do not own any historical data about the user performing some query, the only data we can extract features from are the source and destination nodes, s and t, and the shortest path between the two, p_{st}.
- Since we are building a model to predict the search time of an algorithm, a good feature should communicate the complexity of the search space.
- Since features are computed online, the operation should take a negligible amount of time with respect to the available time frame.

In the following paragraphs we introduce the features we employ and how to extract them from the available input data.

Shortest path segments The first feature we consider is the shortest path number of segments (SPS) that we already introduced in section 4.1. As we deeply motivated, the purpose of SPS is to capture the complexity of the graph search space. The more segments the shortest path contains, the higher the search space is likely to be, growing exponentially for non-trivial road networks. Computing the number SPS is trivial, as it simply requires to count the number of edges in p_{st}, which is an $\mathcal{O}(1)$ operation if we store the length of a path in a graph. Otherwise it has linear time complexity.

Flight distance With this feature, we express the geographical (great-circle) distance between s and t nodes one the Earth surface. This is efficiently computed with the Haversine formula [46]. Let r be the radius of the Earth (approximately), φ_s and φ_t the latitudes of s and t respectively and λ_s and λ_t the longitudes of s and t respectively, we compute the great-circle (or flight) distance as:

$$d = 2r\ \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_t - \varphi_s}{2}\right) + \cos(\varphi_s)\cos(\varphi_t)\sin^2\left(\frac{\lambda_t - \lambda_s}{2}\right)}\right) \tag{4.1}$$

While this flight distance alone might say a little on the graph search complexity, it might be a discriminative factor, in conjunction with SPS, in case of uncrossable natural places in the road network. In fact, high SPS and high flight distance could mean that s and t are far part, but searching might be fast because many different alternatives exist. On the other hand, a high SPS and a low flight distance might mean that a lake or a hill exist between s and t and therefore maybe few paths connect the two nodes, making the search for fairly different alternatives hard.

Betweenness centrality Another feature we employ is the Betweenness Centrality (BC) [47, 48] measure. BC is a very well-known metric which relies on shortest paths. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices, such that the sum of the wights is minimized. The Betweenness Centrality for each vertex is the number of these shortest paths that pass through the vertex. Formally, the Betweenness Centrality of a node v is given by the expression:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}, \quad \forall\,(s,t) \in V \times V \tag{4.2}$$

where σ_{st} is the total number of shortest paths from node s and node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

BC is widely applied in network theory. It represents the degree of which nodes stand between each other. More practically, a node with high BC would have more control over the network because more information passes through it. In the case of a road network, nodes with high BC represent *hot spots*, “mandatory” places to pass through to reach many other points in the map. For instance, let us consider a city with a river flowing through it. In this scenario, a bridge represent a place with high BC, being the only way to connect the two sides of the river.

We propose to employ a Top-N BC average over p_{st} nodes to capture the complexity of finding good alternatives. Indeed, if a path contains many hot spots, most alternative will be forced to flow through those points, making discovering different paths more complex.

Moreover, we point out that, even if computing BC is a time-consuming activity, it can be calculated just once, offline, and then used as a look-up table online, making the process an $\mathcal{O}(V \log V)$ operation, where V is the number of nodes in the shortest path p_{st}. Indeed, we need to look the BC up for each node in p_{st}, which is an $\mathcal{O}(V)$ operation, sort them, which is $\mathcal{O}(V \log V)$ and average the top-N results, which takes $\mathcal{O}(1)$ time. The total complexity is $\mathcal{O}(V \log V)$.

In our experiments, we perform a *Top-3* Betweenness Centrality average.

Clustering coefficient This metric provides a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, nodes tend to create tightly knit groups characterised by a relatively high density of ties. In particular, we employ the clustering coefficient [49, 50] variant known as *local clustering coefficient*, which quantifies how close its neighbours are to being a clique (complete graph). Formally, let $G = (V, E)$ be a graph and e_{uv} be an edge connecting u to v. We define the neighborhood N_v of a node v as its immediately connected neighbours, as follows:

$$N_v = \{u : e_{uv} \in E \vee e_{vu} \in E\}. \tag{4.3}$$

The local clustering coefficient $C(v)$ for a vertex v is then given by the proportion of links between the vertices within its neighbourhood divided by the number of links that could possibly exist between them. Formally, let k_v be the number of vertices, $|N_v|$, of the neighborhood of v, then the local clustering coefficient for directed graphs is given as

$$C(v) = \frac{|\{e_{uw} : u, w \in N_v, e_{uw} \in E\}|}{k_v (k_v - 1)}. \tag{4.4}$$

We propose to employ the *sum of the local clustering coefficient* over each node in p_{st} as a feature. Like Betweenness Centrality, we can compute the local clustering coefficient of every node in the road network offline and then look them up online. The complexity of the operation is $\mathcal{O}(V)$ operation, where V is the number of nodes in the shortest path p_{st}.

Shortest path out degree The last feature we propose provides a local measure of the search space complexity. We simply sum the number of outgoing edges for each node in p_{st}. Formally,

$$Out(p) = \sum_{v \in p} outDegree(v). \tag{4.5}$$

We introduce this metric as a support to the number of SPS to capture the actual number of outgoing edges, which we previously assumed to be uniform across all vertices. The complexity of the operation is $\mathcal{O}(V)$ operation, where V is the number of nodes in the shortest path p_{st}.

Finally, we notice that to compute all the aforementioned features, a single pass over the shortest path p_{st} is necessary, which is a $\mathcal{O}(V)$ operation. The overall feature extraction complexity is then dominated by the Top-N BC average calculation, taking a reasonable $\mathcal{O}(V \log V)$ time. Better complexity can be achieved by employing a fixed-size priority queue, leading to an $\mathcal{O}(V \log N)$ overall time complexity.

4.4 Regression Model

In section 4.3, we proposed a number of features that we can efficiently extract from an input request to characterize it and extract as much information as possible about a solution search complexity. In this section we present the learning task we want to apply to our use-case, along with a rationale our design choices, supported by experimental evidence.

As previously introduced in section 4.2, we want to extend the proposed car navigation system with an intelligent module to perform dynamic auto-tuning. To achieve our goal, the module should be able to predict the execution time of processing an incoming input request with some $(k, \theta, algorithm)$ configuration.

In machine learning and statistics domains, the problem of learning a function that maps an input to a real continuous output, based on sample input-output pairs, is called *Regression*, a subset of *Supervised Learning* task. Formally, given a set of N training samples of the form $\{(\boldsymbol{\phi}(\boldsymbol{x}_1), t_1), \dots, (\boldsymbol{\phi}(\boldsymbol{x}_n), t_n)\}$ such that $\boldsymbol{\phi}(\boldsymbol{x}_i)$ is the feature vector of the i-th sample and t_i is its label (i.e. the real value), a learning algorithms seeks for a function $g : X \rightarrow Y$, where X is the input space and Y is the output space. The function g is an element of some space of possible functions G, usually called *hypothesis space*. In order to measure how well a function fits the training data, a loss function $\mathcal{L} : Y \times Y \rightarrow \mathbb{R}^{\geq 0}$ is defined. For training sample $(\boldsymbol{\phi}(\boldsymbol{x}_i), t_i)$, the loss of predicting the value $\hat{t}_i$ is $\mathcal{L}(t_i, \hat{t}_i)$. The task of a learning algorithm is to find g such that it minimizes the expected loss, that is $\mathbb{E}[\mathcal{L}] = 1/n \sum_{i=1}^{n} \mathcal{L}(t_i, \hat{t}_i)$.

In our use-case, the set of training samples is obtained from the experiment run in section 3.4, such that $\boldsymbol{\phi}(\boldsymbol{x}_i) = [$ shortestPathSegments$(\boldsymbol{x}_i)$, flightDistance$(\boldsymbol{x}_i)$, betweennessCentrality$(\boldsymbol{x}_i)$, clusteringCoefficient$(\boldsymbol{x}_i)$, shortestPathOutDegree$(\boldsymbol{x}_i)$ $]$, while t_i is the execution time of i-th sample. As generally postulated, we assume that the target variable t is given by a deterministic function $g(\boldsymbol{\phi}(\boldsymbol{x}_i))$, with an additive Gaussian noise, so that

$$t = g(\boldsymbol{\phi}(\boldsymbol{x}_i)) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2).$$

An important corollary of this assumption is that training samples are supposed to be *homoscedastic*. Homoscedasticity is the property of a sequence of random variables to have the same finite variance, which, in our training set, is the σ_ϵ^2 variance of the $(t_1, \dots, t_n)$ target variable sequence.

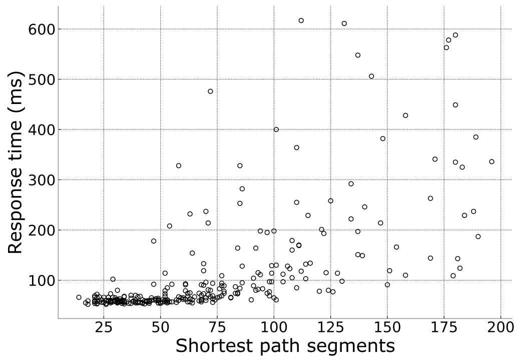
From a very early analysis, we noticed that our dataset is *heterogeneous* in many features, providing very few samples for a number of SPS greater than 200, resulting in a very sparse, noisy samples distribution. For this reason, hereinafter we restrict ourselves to the $[0, 200)$ range of SPS, which, anyway, we observe accounting for more than 70% of the experiment input queries. For all the inputs having a number of SPS greater than 200, we refrain from building a predictive model, backing up to the adaptive policy described in section 3.6.

In Figure 4.9, we plot some of the input features against the target variable, the execution time, for OnePass+ and Penalty algorithms, $k = 5$ and $\theta = 0.5$. This figure is a representative example that shows a behavior common to most configurations: *heteroscedasticity* of the samples.

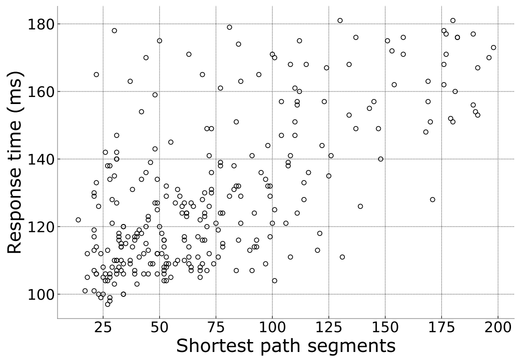
Indeed, if we take a look to Figures 4.9a and 4.9c, for instance, we notice that surely there exists some trend in the data, but, while for small values of the features we see the same dispersion around an imaginary trend line, for higher feature values the target variable explodes, leading to very disperse samples.

Multiple reasons might concur in creating such phenomenon. It can be an intrinsic unpredictability of the time-to-solution in too complex spaces, or maybe the features we engineered are not able to fully capture the response time behavior, just to name a few. Nevertheless, heteroscedasticity is the problem we must deal with.

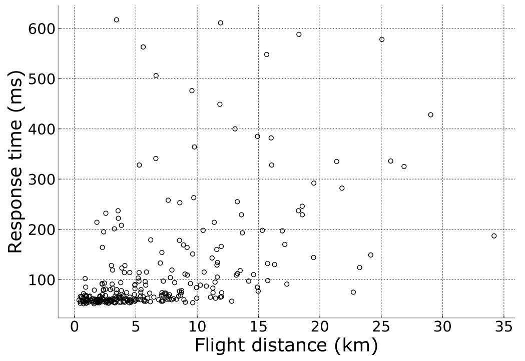
In the literature, the typical approach to address this issue is to apply a *weighing* process to the training samples. In fact, the purpose of this operation is to reduce the



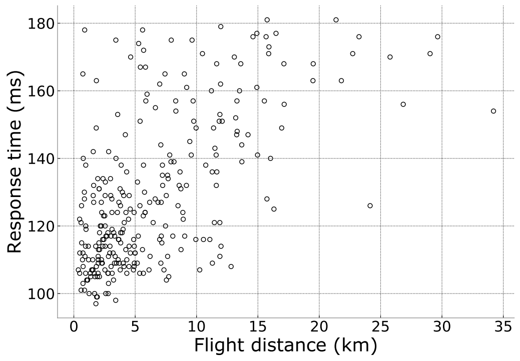
(a) *OnePass+ response time wrt. number of SPS*



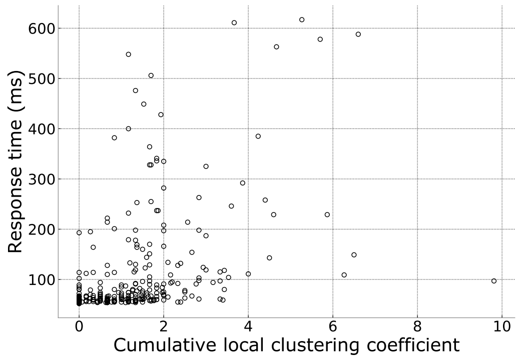
(b) *Penalty response time wrt. number of SPS*



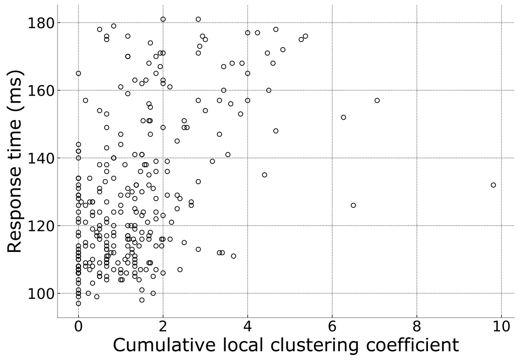
(c) *OnePass+ response time wrt. flight distance*



(d) *Penalty response time wrt. flight distance*



(e) *OnePass+ response time wrt. clustering coefficient*



(f) *Penalty response time wrt. clustering coefficient*

Figure 4.9: *OnePass+ and Penalty response time on New York City, with respect to some input features, with $k = 5$ and $\theta = 0.5$.*

importance, on the model fitting, of those samples that are less reliable, or formally, that are instances of highly-disperse random variables. From a theoretical perspective [51], the optimal weight for a sample x generated by a random variable X is given as $1/\mathrm{Var}[X]$. Since we do not know anything about X distribution, we could estimate $\mathrm{Var}[X]$ with the sample variance. To do so, we should run the experiment described in section 3.4 a sufficient number of times in order to obtain a meaningful population from X.

While hypothetically possible, this process is extremely time consuming, considering that a single experiment execution takes about 10 hours to complete. In such cases, a typical approach is to use proxies that would weigh less those samples that are expected to have a larger variance. Typical approaches are to use the inverse of the target value (i.e $1/\boldsymbol{t}$) or the inverse of some feature with respect to which the target value shows an exponential increase in the variability (e.g. $1/\text{shortestPathSegments}(\boldsymbol{x}_i)$).

In our research to build the most reliable predictive model, we considered the following machine learning methods: Lasso [52], Ridge Regression [53], XGBoost [54] and ExtraTrees [55]. All these models were trained on the same dataset, we tuned their hyper-parameters using a grid-search approach and shuffled 10-fold cross-validation (or Leave-One-Out for Lasso and Ridge) and evaluated them with shuffled 10-fold cross-validation measuring the Root Mean Squared Error (RMSE).

The outcome from this analysis was that none of the trained models was able to perform nicely enough on the entire dataset, always scoring a too high RMSE. This phenomenon was highly anticipated by the heteroscedasticity of our data. Indeed, since the variance is not homogeneous in the target variables, the model commits worse errors for those samples we were less confident in. Aforementioned techniques to correct it have been tried, unfortunately leading to no significant improvement. As a consequence, we moved our research from a model capable of predicting the exact execution time to an *upper-bound* prediction. Such approach is beneficial for the following two reasons:

- It mitigates the heteroscedasticity, because instead of predicting a mean value, we predict an upper bound of it, that accounts also for the target variable variance.
- Because execution time is hardly deterministic, and it is influenced by a large number of factors, like system workload, concurrently running processes, scheduler, temperature and so on, estimating an upper bound makes the model more robust to unpredictable variations.

In order to have a statistically meaningful estimation, we seek for a model which results in normally-distributed residuals. Indeed, if residuals have Gaussian distribution, we can provide an upper-bounded prediction $\hat{t}^+ = (1 + n\sigma_r)g(\boldsymbol{\phi}(\boldsymbol{x}_i))$, where σ_r is the standard deviation of the normalized residuals on the training set. Moreover, we obtain a statistical confidence on the percentage of samples we are under-estimating. For instance, let $n = 3$, so that $\hat{t}^+ = (1 + 3\sigma_r)g(\boldsymbol{\phi}(\boldsymbol{x}_i))$, the probability of under-estimating the target variable t is given as:

$$\begin{aligned} P(t > \hat{t}^+) &= 1 - P(t < \hat{t}^+) \\ &\approx 1 - \Phi(3\sigma_r) \\ &= 0.13\%. \end{aligned} \tag{4.6}$$

While more complex models, like XGBoost and ExtraTrees, achieved better RMSE in our experiments, their normalized residuals were not normally-distributed, making them unfeasible for our learning task.

Moreover, while sample weights proxies might work in some circumstances, in our experiments they led to a decrease in the predictive performance of the trained models. Hence, we decided to abandon their employment.

From our analysis, the best performing model was Ridge Regression, an L_2 regularized least squares regression method, which minimizes the following loss function:

$$\mathcal{L}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \{t_n - \boldsymbol{w}^\intercal \boldsymbol{\phi}(\boldsymbol{x}_n)\}^2 + \frac{\lambda}{2}\boldsymbol{w}^\intercal \boldsymbol{w}, \tag{4.7}$$

where $\boldsymbol{w}$ is the model weights vector to tune to minimize the loss function $\mathcal{L}$ and λ is the regularization coefficient to manage the bias-variance trade-off.

In order to assess the normal distribution of the model residuals, we employ a numerical hypothesis test along with a graphical method. We leverage Kolmogorov-Smirnov hypothesis test [56, 57] for the former and Q-Q Plot [58] for the latter.

The Kolmogorov-Smirnov test is a non-parametric goodness of fit statistical test, which quantifies a distance between empirical distribution function of the samples and the cumulative distribution function of a reference distribution. Let be the set of normalized residuals, given as

$$\boldsymbol{r} = \left\{ \left(\frac{t_1 - \hat{t_1}}{t_1} \right), \ldots, \left(\frac{t_n - \hat{t_n}}{t_n} \right) \right\},$$

and μ_r and σ_r^2 be the sample mean and sample variance respectively.

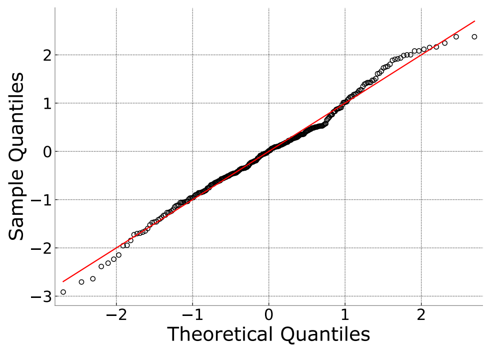
We perform the Kolmogorov-Smirnov test for goodness of fit of the empirical cumulative distribution $G(x)$ of the observed residuals against a normal distribution with μ_r mean and σ_r^2 variance, namely $\Phi(\frac{x-\mu_r}{\sigma_r})$. Formally, we consider the following hypothesis test:

$$\begin{aligned} H_0 &: G(x) \leq \Phi\left(\frac{x-\mu_r}{\sigma_r}\right) \\ H_1 &: G(x) > \Phi\left(\frac{x-\mu_r}{\sigma_r}\right), \text{ for at least some } x. \end{aligned} \tag{4.8}$$

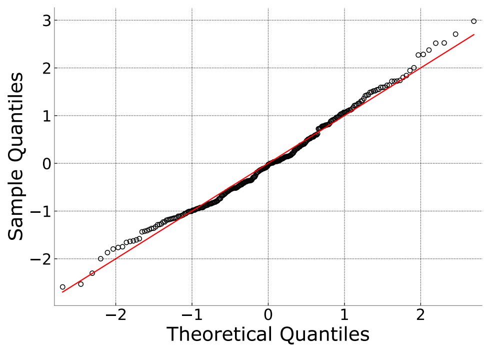
We reject the null-hypothesis H_0 for any p-value < 0.05.

Along to the Kolmogorov-Smirnov test of normality, we employ a visual method known as Q-Q Plot. A Q-Q (Quantile-Quantile) plot is a probability plot, a method for comparing two probability distributions by drawing their quantiles against each other. If two distributions being compared are similar, point in the Q-Q Plot will approximately lie on the line $y = x$. In our scenario, we plot the normalized residuals quantiles against a population generated by a random variable $X \sim \mathcal{N}(\mu_r, \sigma_r^2)$.

In Figure 4.10 we draw the Q-Q Plots of OnePass+ (left) and Penalty (right) of the normalized residuals for $k = 5$ and $\theta = 0.5$ on the New York City map. On the x-axis we report the theoretical normal quantiles while on the y-axis the empirical quantiles. The red line represents the $y = x$ line. We notice that the residuals points lie almost completely on the red line, showing that the model residuals are normally distributed.



(a) *OnePass+ (KS test p-value = 0.694255)*



(b) *Penalty (KS test p-value = 0.586956)*

Figure 4.10: *OnePass+ and Penalty Q-Q Plots of the normalized residuals on New York City, with $k = 5$ and $\theta = 0.5$*

Moreover, we emphasize the Kolmogorov-Smirnov test p-value scoring 0.69 and 0.59 for OnePass+ and Penalty respectively, supporting our claim on the normality of residuals.

4.5 Training Process

In section 4.4, we presented the regression model we chose to predict ARLib algorithms execution time. We elected Ridge Regression method as the one with best fitting of its residuals with respect to normal distribution.

In this section, we describe the training process we employ for the model learning to obtain the results shown in Figure 4.10. The training pipeline we are depicting hereinafter is schematized in the UML Activity Diagram of Figure 4.11.

4.5.1 Data Preparation

The first step is task learning is data preparation, where the training set is cleaned and prepared for model fitting. The initial dataset originates from results of the experiment described in section 3.4. To briefly recap, for 450 source-destination pairs we performed a query to each ARLib algorithm for $k \in \{2, 3, 4, 5\}$ and $\theta \in \{0.3, 0.5, 0.7\}$, we measured the execution time and computed the input features $\boldsymbol{\phi}(\boldsymbol{x})$, deeply illustrated in section 4.3.

Our objective is to train a model for the adaptive navigation system proposed in section 4.2, given some (k, θ) configuration fixed by the service provider. Moreover, recalling that we are training a model for each ARLib algorithm, we underline that the procedure we are describing applies to the subset of the training set corresponding to a given k, θ and algorithm α.

Trimming First of all, we apply a 10% trimming step, dropping those data entries whose execution time is above the 90th percentile. We remind the reader about the heteroscedasticity of the data. By applying a trimming phase, we exclude those outliers that were potentially caused by those highly-dispersed target variables.

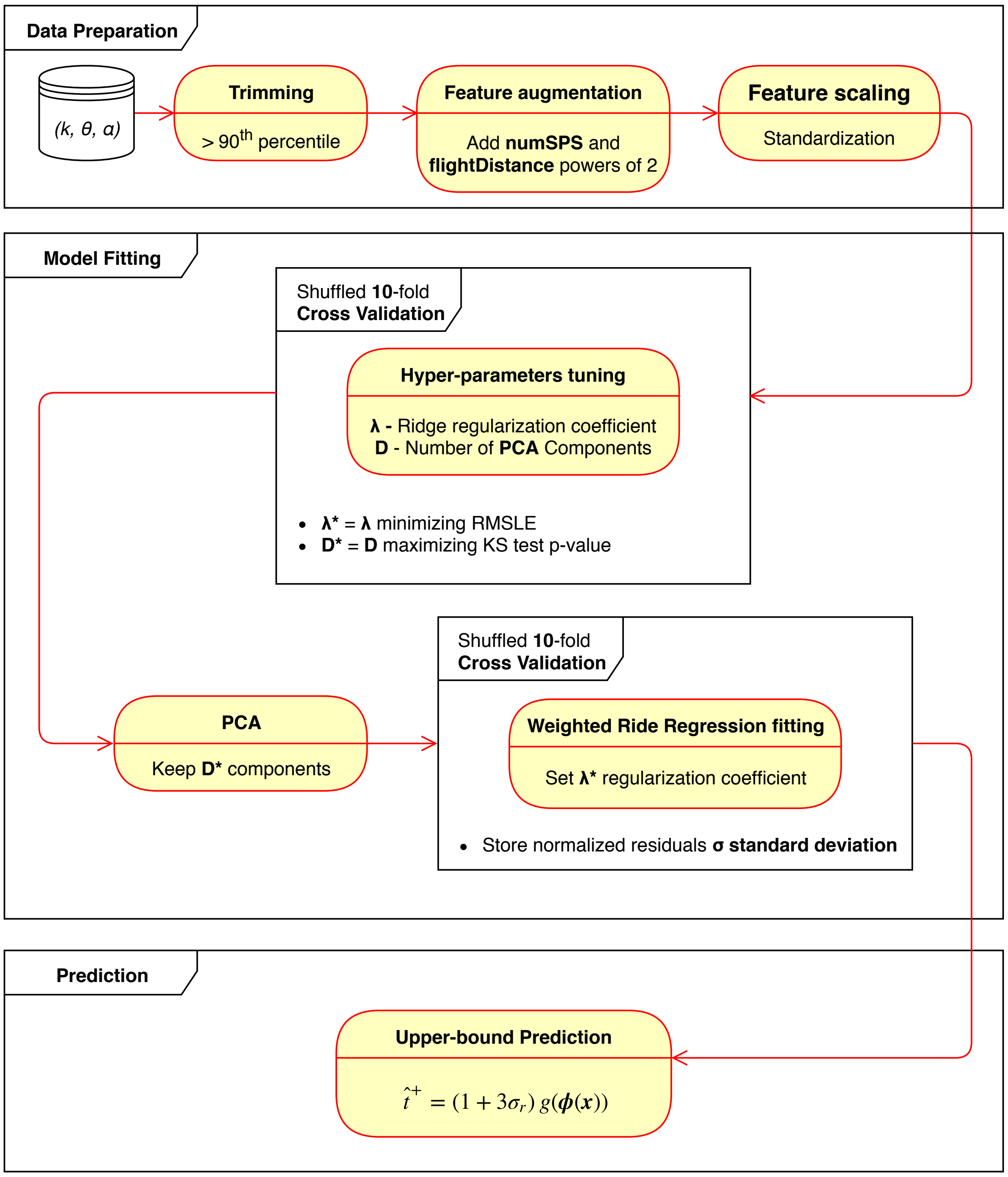

Data Preparation
(k, θ, α)
Trimming
> 90th percentile
Feature augmentation
Add numSPS and flightDistance powers of 2
Feature scaling
Standardization
Model Fitting
Shuffled 10-fold Cross Validation
Hyper-parameters tuning
λ - Ridge regularization coefficient
D - Number of PCA Components
λ* = λ minimizing RMSLE
D* = D maximizing KS test p-value
PCA
Keep D* components
Shuffled 10-fold Cross Validation
Weighted Ride Regression fitting
Set λ* regularization coefficient
Store normalized residuals σ standard deviation
Prediction
Upper-bound Prediction
$\hat{t}^+ = (1 + 3\sigma_r)\, g(\boldsymbol{\phi}(\boldsymbol{x}))$

Figure 4.11: *UML Activity Diagram of the training process employed for predictive model learning. On top the data preparation steps are reported, showing how the training set is cleaned and enriched. In the center, the model fitting part is listed describing how an effective model is obtained from the training data. Finally, on bottom, the execution time upper-bound prediction is given from the model fitted function* $g(\boldsymbol{\phi}(\boldsymbol{x}))$.

Feature augmentation Subsequently, we augment the feature vector $\boldsymbol{\phi}(\boldsymbol{x})$ with the power of 2 of number of SPS and flight distance features. In fact, we noticed that, for some algorithms, as perceivable from Figures 4.9a and 4.9c, the execution time shows a non-linear trend with respect to those two features. Therefore, we project them in the power-of-2 space in order to establish a more linear relationship with the target variable.

Feature scaling The last step in data preparation is the feature standardization. This means that, for each feature j, we compute μ_j and σ_j, the feature mean and standard deviation respectively. Then, we subtract the mean from the feature and divide the mean-centered values of the feature by its standard deviation. Formally:

$$\phi'_{ij} = \frac{\phi_{ij} - \mu_j}{\sigma_j},$$

where ϕ_{ij} is the value of feature j for the i-th sample.

This procedure is fundamental for machine learning algorithms, like Ridge Regression, that employ a Euclidean distance in their error function. Without normalization, features with broader ranges would indeed govern such distance, implicitly weighing more on the final prediction value. By applying normalization, we scale values for each feature to have zero mean and unit-variance, keeping them approximately equally important to the learned model.

4.5.2 Model Fitting

In this second macro-stage, we perform the actual model fitting, applying Ridge Regression method to learn a function $g(\boldsymbol{\phi}(\boldsymbol{x}))$ minimizing the loss function in Equation (4.7).

Instead of directly feeding the augmented feature vector $\boldsymbol{\phi}(\boldsymbol{x})$ to the training algorithm, we first apply PCA [59, 60] in order to project the feature space into a more informative coordinate system and, eventually, to reduce the number of features keeping only those that explain the target variable the most.

Since PCA provides us with a valuable feature selecting tool, the number of principle components to keep becomes an hyper-parameter of the training process to tune, which we name D.

Along to D parameter, Equation (4.7) exposes another hyper-parameter, the regularization coefficient λ, which we remember managing the bias-variance trade-off of the learned model, to limit an over-fitting phenomenon.

Heteroscedasticity As previously mentioned, we abandoned the sample weights employment to fix the data heteroscedasticity, since estimating the variance of each sample was unfeasible and no good proxy was found. Consequently, we resorted to another way to deal with non-linear growth of the target variable. In the literature, a typical way to measure a model performance on a dataset exposing such behavior is to employ the Root Mean Squared *Logarithmic* Error (RMSLE), given as:

$$\begin{aligned} MSLE &= \frac{1}{N}\sum_{i=1}^{N}(\ln(1+t_i) - \ln(1+\hat{t}_i))^2 \\ RMSLE &= \sqrt{MSLE} \end{aligned} \tag{4.9}$$

To help the reader intuition, we point out that RMSLE penalizes an under-predicted estimate greater than an over-predicted estimate. From our perspective, this behavior is highly appreciated. Indeed, considering that we always set a timeout on each request, having a model that optimistically predicts lower execution time values would make the system choose to run some algorithm that is unlike to terminate in time, leading to none or partial solutions. On the other hand, following a conservative approach, if the model suggests a over-estimated execution time, but still lower than the timeout, we are more likely to obtain high quality solutions within the available time budget.

Therefore, in our training process, we always evaluate models performance according to their RMSLE instead of the more common RMSE.

Hyper-parameters tuning The first step we perform, then, is hyper-parameter tuning, that is, we search for the optimal (λ, D) pair that minimizes the RMSLE metric while keeping the residuals normally distributed. The process we employ is the following:

1. Let F be the set of features in $\boldsymbol{\phi}(\boldsymbol{x})$. For D in $\{1, 2, \ldots, |F|\}$ set, we compute the first D principle components from the feature matrix $\boldsymbol{\Phi}(\boldsymbol{X})$ of the training set.
2. We search for λ^* minimizing the RMSLE in the 200-elements logarithmic space from 10^{-8} to 1, using Leave-One-Out validation technique on the training set.
3. Using 10-fold cross validation, we train a Ridge Regression model to cross-predict the target vector $\hat{\boldsymbol{t}}$.
4. We compute the Kolmogorov-Smirnov normality test p-value, as described in Equation (4.8), of the normalized residuals $(\boldsymbol{t} - \hat{\boldsymbol{t}})/\boldsymbol{t}$.
5. We select the number of components D^* maximizing the Kolmogorov-Smirnov test p-value.

PCA Having discovered the optimal D^* number of components, we apply PCA to the feature matrix $\boldsymbol{\Phi}(\boldsymbol{X})$ to obtain a more informative projection of the training data.

σ_r estimation Recalling that we are ultimately interest in predicting an upper bound of the execution time, we still lack of the residuals standard deviation to build our estimate from. In order to compute it, we perform a 10-fold cross-validation to predict the target vector $\hat{\boldsymbol{t}}$ using Ridge Regression with $\lambda = \lambda^*$ on the projected feature matrix $\boldsymbol{\Phi}(\boldsymbol{X})$. We calculate the normalized residuals and, subsequently, the sample standard deviation σ_r.

For the final model, we retrain a Ridge Regression model over the whole dataset.

4.5.3 Prediction

The final stage is, of course, the prediction step. In the fitting phase, we learned a function $g(\boldsymbol{\phi}(\boldsymbol{x}))$ that minimizes the loss function of Equation (4.7). Moreover, we also estimated the standard deviation σ_r in the normalized residual errors.

Ergo, we are ready to provide an upper bound of the execution time $\hat{t}^+$, given the system (k, θ) configuration, for an α ARLib algorithm and some sample $\boldsymbol{x}$:

Table 4.1: *Ridge Regression models prediction performance on New York City, for OnePass+ and Penalty, $\theta = 0.5$ and varying $k \in \{2,3,4,5\}$. In third column the Root Mean Squared Logarithmic Error on the training set is reported. In column 4 and 5, the percentage of underestimated training samples with a predicted upper bound of $2\sigma_r$ and $3\sigma_r$, respectively.*

Algorithm	**k**	**Training RMSLE**	**Miss. $2\sigma_r$**	**Miss. $3\sigma_r$**
OnePass+	2	0.041	0.02	0.01
	3	0.11	0.07	0.02
	4	0.27	0.07	0.03
	5	0.47	0.06	0.03
Penalty	2	0.044	0.05	0.01
	3	0.07	0.05	0.02
	4	0.09	0.04	0.01
	5	0.12	0.04	0.02

$$\hat{t}^{+} = (1 + 3\sigma_r)\, g(\boldsymbol{\phi}(\boldsymbol{x}))\big|_{k,\theta,\alpha} \,. \tag{4.10}$$

In Table 4.1, we report the prediction performance of the trained Ridge Regression models on the training set for a similarity threshold $\theta = 0.5$.

4.6 An Improved Adaptive Policy

In section 4.2, we outlined the characteristics of an improved, adaptive, car-navigation system capable of proactively select the best algorithm for each incoming routing request.

The proposed extension introduces an intelligent module that extracts some features from a user query and, based on that, elects the best-quality algorithm that is expected to complete the search within a fixed time frame. In section 4.3 we described in details which features we extract and in Sections 4.4 and 4.5 we deeply illustrated the machine learning model we employ to estimate the routing execution time, along with the process to train it.

In this section, we propose another adaptive policy, improving the one presented in section 3.6, leveraging the proactive capabilities of the auto-tuning module. The adaptive policy we are presenting hereinafter is summarized by the UML Activity Diagram in Figure 4.12.

Let (k, θ) be the configuration set by the service provider and s and t the source and destination nodes of an incoming routing request. The first step is to compute the shortest path p_{st} with some shortest path algorithm (like ARLib's fast Bidirectional Dijkstra). Then, from $\boldsymbol{x} = (s, t, p_{st})$ we extract the feature vector $\boldsymbol{\phi}(\boldsymbol{x})$, as described in section 4.3.

As previously discussed in section 4.4, if the number of shortest path segments is greater than 200, we are not able to deliver a meaningful model of the execution time behavior. Therefore, we back up to the simpler adaptive policy illustrated in section 3.6, selecting Penalty or OnePass+ depending on the system (k, θ) configuration and the type of routing request. Otherwise, if the number of shortest path segments is less than

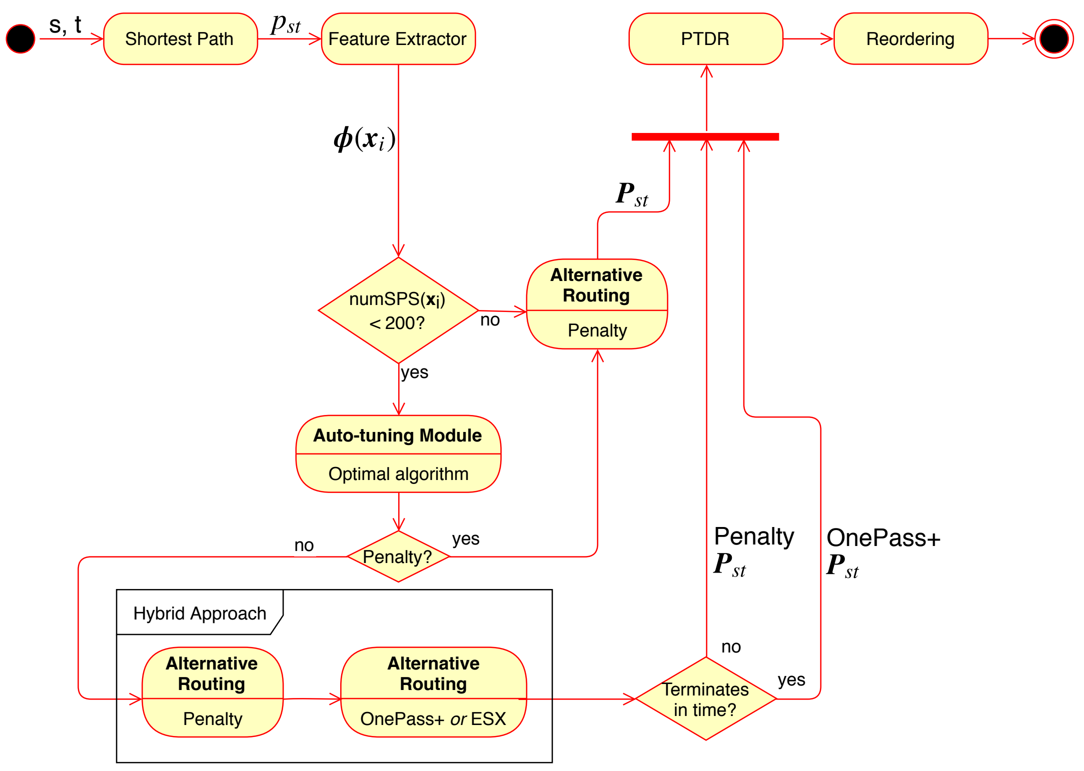


Figure 4.12: *UML Activity Diagram of the adaptive policy leveraging the auto-tuning module. Based on query features, the auto-tuner elects the highest-quality algorithm leading to a solution within the available time budget.*

200, which we remind happening in more than 70% of the experiment queries, we are able to achieve a finer-grained adaptivity.

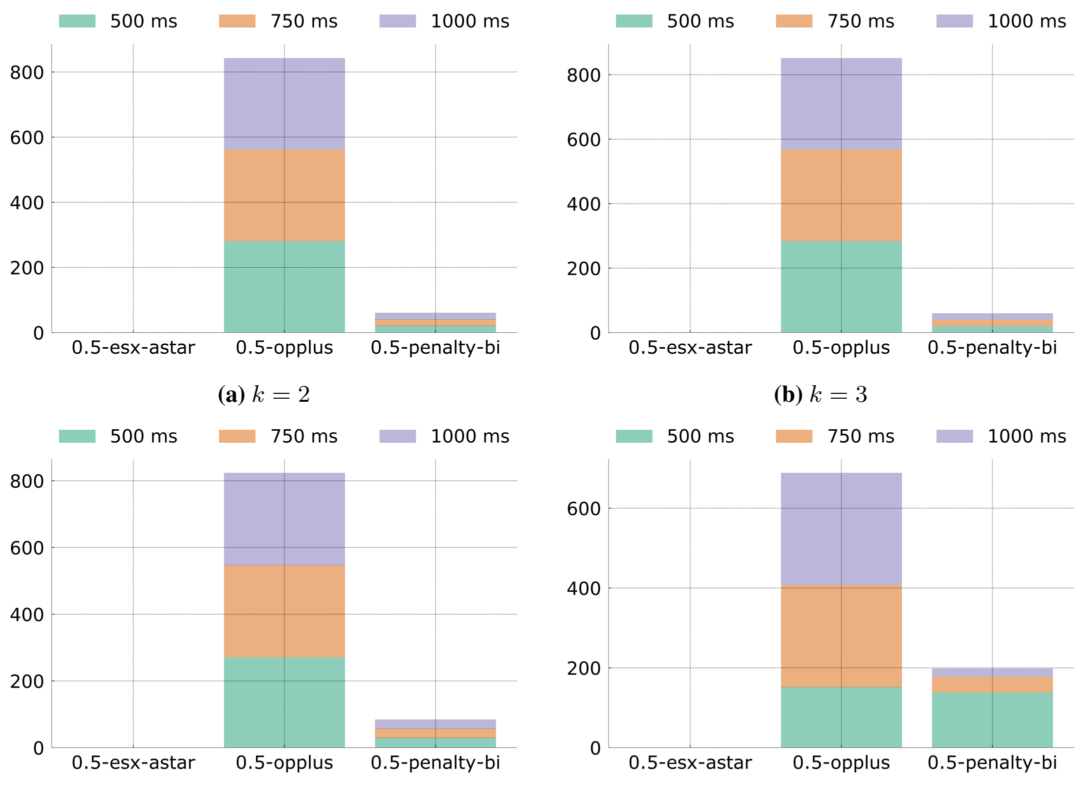
From $\phi(x)$, the auto-tuning module selects the optimal algorithm α^* as described in the *dynamic auto-tuning process* of section 4.2. If $\alpha^* =$ Penalty, we compute a set of k alternative paths P_{st} and forward it to the PTDR module. On the contrary, if α^* is either OnePass+ or ESX we must consider a safer approach.

As we illustrated in subsection 3.5.3, OnePass+ and ESX do not ensure 100% success rate, failing to always find valid solutions within a limited time frame. While the the predictive model provides an upper bound on the algorithms execution time, underestimating some rare samples is a possibility and outliers might take more time than expected, causing strong delays in the system. For this reason, we rely on a hybrid approach similar to what we described in section 3.6.

Let T^* be the available time budget. The auto-tuning module searches for the optimal algorithm to employ, as the following:

- When considering Penalty, it predicts an upper bound $\hat{t}^+_{Penalty}$ as reported in Equation (4.10).
- When considering OnePass+ (or ESX), it predicts the execution time of the entire hybrid approach, that is, the expected Penalty time $\hat{t}^+_{Penalty}$ and the expected OnePass+ (or ESX) time $\hat{t}^+_{OnePass+}$, which is given as $\hat{t}^+_{Hybrid} = \hat{t}^+_{Penalty} + \hat{t}^+_{OnePass+}$.

Consequently, if $\hat{t}^+_{Hybrid} < T^*$, hybrid approach is selected, as leading to beast qual-



(a) $k = 2$ **(b)** $k = 3$ **(c)** $k = 4$ **(d)** $k = 5$

Figure 4.13: *Auto-tuner optimal algorithm selection frequencies on New York City, with* $\theta = 0.5$*, varying* $k \in \{2, 3, 4, 5\}$ *and time budget* $T^* \in \{500, 750, 1000\}$ *milliseconds.*

ity solutions. Executing hybrid approach requires the following steps:

1. Compute the set of k alternative paths with Penalty. The search is expected to terminate in a time $T_{Penalty} \ll T^*$.

2. Compute the set of k alternative paths with OnePass+ (or ESX) setting a time-out of $T = T^* - T_{Penalty}$.

3. If OnePass+ (or ESX) terminates before the time-out, forward its solutions $\boldsymbol{P}_{st}$ to PTDR module and continue the pipeline processing.

4. Otherwise, forward the Penalty solutions.

The adaptive policy we just proposed tries to fully exploit the available time budget to concurrently lead to highest quality and in-time solutions, avoiding system delays but still resulting in valid alternative routes is case execution time expectations turn out to be wrong.

In Figure 4.13 we picture the frequency of selection for each algorithm by the auto-tuning module, varying the number of paths k and the search time budget T^*. As expected, hybrid OnePass+ is the most selected configuration in general, as we aim at the maximum result quality. On the other hand, by lowering T^* from 1 second to 500 ms and raising the number of paths k, we notice the frequency of Penalty increasing since hybrid OnePass+ is not expected to make it anymore in such a limited time frame.

4.7 Summary

In this chapter we presented a methodology for implementing and training a machine learning model to online predict the optimal ARP algorithm to employ for each query depending on its characteristics. From the analysis carried out in section 4.1, we discovered that ARP heuristics latency is strongly influenced by the number of segments composing the shortest path connecting the source and destination nodes of the query. This study led us to question whether there were other request features that could be used to proactively decide which algorithm to employ to obtain the highest quality possible with the available time budget.

In section 4.3, we presented a set of features engineered to be quickly extracted from the sole query information, along with its shortest path computation which we would have computed anyway as the base solution for every ARP algorithm. Then we described our model selection process, along with the actual training pipeline from data preparation, to learning, to final prediction.

Finally, in section 4.6 we further extended our car navigation system adaptive policy, first introduced in section 3.6, to include a proactive step, leveraging the auto-tuning module for computing the optimal ARP heuristic to employ for each incoming request.

From a functional perspective, this chapter exhausts the contributions of this thesis on the design of an adaptive car navigation system. In fact, along with previous work [5] on PTDR module, our system exhibits three levels of adaptivity. The first level builds on traffic-monitoring data: relying on daily travelling time distributions, PTDR module adaptively elects the fastest route to follow, according to current traffic conditions. The second level is supported by our analysis on ARP problem design space, enabling the service provider to choose the configuration of the system, by means of number of alternative paths k and similarity threshold θ, according to arbitrary constraints on latency or on number of computational resources. The third and final level of adaptivity is reached by introducing a dynamic auto-tuning module, proactively recommending an optimal heuristic for each incoming request and raising the overall solutions quality by exploiting slower, but more accurate, algorithms when sufficient time budget is available.

In the following chapter, we will shift from a functional to an extra-functional point of view, introducing a Queuing Petri Nets model of our car navigation system. In fact, in the context of a smart city conducting a feasibility study on delivering a navigation service, the ability to evaluate the system from a performance perspective is fundamental. Most notably, having a system model enables a capacity planning analysis in order to determine the necessary computational power to put in place such a service and therefore the amount of financial resources that should be allocated.

CHAPTER 5

System Modelling

In this chapter, we will shift from a functional to an extra-functional point of view, introducing a Queuing Petri Nets model of our car navigation system.

In the context of a smart city conducting a feasibility study on delivering a navigation service, the ability to evaluate the system from a performance perspective is fundamental. Properties like scalability and tolerance to unexpected workload spikes are just two of the important studies that can be carried out on our car navigation system. Moreover, having a system model enables a capacity planning analysis in order to determine the necessary computational power to put in place such a service and therefore the amount of financial resources that should be allocated.

Starting from the theoretical background on performance evaluation introduced in the second part of chapter 2, we present a model of the system with a combination of Queueing Networks and Petri Nets to effectively represent the complex behavior of our car navigation system. Moreover, we propose a strategy to efficiently carry out a capacity planning analysis to optimally size the number of resources given some expected workload. Furthermore, an extension on the model is explored to introduce an overload-tolerance capability to the system, by introducing a fast lane enabled when the system is overloaded, producing low-quality, but very fast responses in order to cope with unexpected workload spikes.

5.1 A Car Navigation Service in HPC context

In chapter 4, we presented our approach to a car-navigation system design, relying on several adaptivity layers. We identified a number of customization points enabling an intelligent tuning of the system, from the traffic perspective, through the PTDR module, from the quality perspective, through a variety of software knobs such as number of

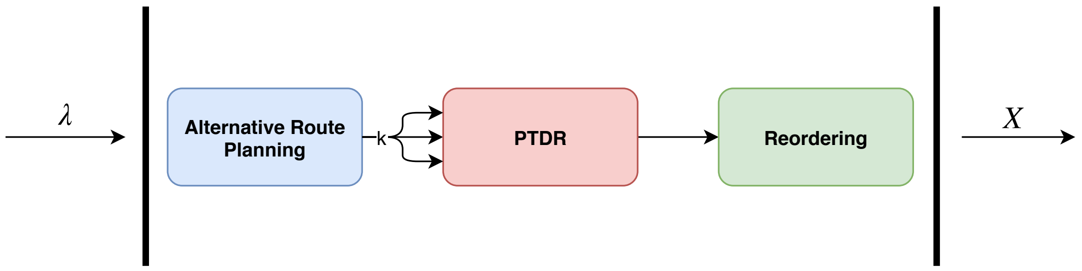


Figure 5.1: *A schematic representation of the system software pipeline to be modelled.*

alternative routes and similarity threshold, and finally, from an optimal configuration perspective, through a proactive auto-tuning module, selecting the optimal algorithm for each upcoming routing request.

Such a complex system raises a number of questions. For instance, one might wonder whether the system, when deployed, would scale and, if so, what would be the number of resources required to effectively process a given expected workload.

A common approach for non-performance-critique online services is to delegate the scalability problem to an IaaS provider. Cloud infrastructures [61], typically, allow their customers to rent some number of cores of a virtual machine. Then an auto-scaling option will automatically spawn more cores if incoming requests rate raises to a level such that it is no longer possible to process them within a given amount of time. To put the read back in our frame, we remind that our goal is to provide a traffic-optimization service for a smart-city populated by autonomous cars. Autonomous cars would be constantly connected to our service, requiring the system to be deployed on a much more powerful infrastructure with respect to a Cloud one.

To manage such a workload, supercomputing centers are viable option. In HPC context, typically, scalability issues are addressed by customers themselves. Indeed, customers actually rent a number of *real*, *non-virtualized* resources for a given amount of time, paying whether exploiting them or not. Moreover, in the circumstance of an unexpected workload, requesting for more resource is not as smooth as in Cloud scenario. As a matter of fact, requesting for more compute power issues a request the could take up to tens of minutes to be processed, causing, in the meantime, strong delays in the service response time.

Even worse, if, while waiting for more cores to be assigned, system workload decreases back to normal, HPC customers would be required to pay for more resources than they would be using. And, again, asking for resources reduction would issue another request to be processed in the following ten minutes, causing a vicious circle.

As it is clear in this description, correctly sizing the number of required resources is a key aspect of deploying a software system in a HPC environment.

To accomplish this, a sound model of the system is required, in order to simulate request arrivals, full pipeline execution and effectively discover the optimal amount of compute units to keep the system response time within a due service time.

In this work, Queueing Networks and Petri Nets, two system modelling formalisms, widely studied in the literature, are employed to develop a reliable model [62] of the system we proposed in chapter 4. In the following sections we will provide a detailed description of the model along with a rationale on how it effectively abstracts the real

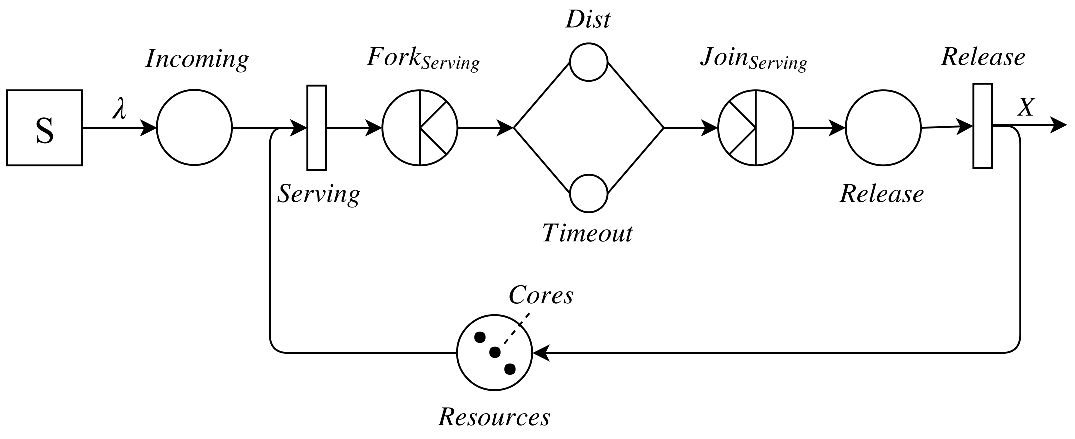


Figure 5.2: *QPN modelling Alternative Route Planning module.*

software system we developed.

To briefly recap, we are considering a software system implementing a pipeline of three stages, as illustrated in Figure 5.1. For starters, *ARP* stage computes a number of alternative paths with limited overlapping. In the second stage, *PTDR* module estimates the expected travelling time for each alternative, according to the current traffic conditions. Finally, *Reordering* stage elects the best path to follow according to arrival time and other arbitrary policies set by the navigation service provider.

5.2 Alternative Route Planning Stage

As we already mentioned in section 3.4, ARP stage is the most critical part of our car-navigation pipeline. Indeed, compared to following stages, ARP accounts the most for response time and, above all, it shows the most variable behavior, especially for routing requests between two long-distant nodes.

For this reason, in chapter 4, we proposed an original approach which we are summarizing as follows:

- Compute the shortest path between source and destination nodes and extract some features from it.
- If the number of shortest-path-segments (SPS) is greater than 200, select the algorithm that, on average, shows the optimal execution time and quality trade-off (*static* adaptivity level).
- If the number of SPS is less than 200, use trained machine learning models to predict the optimal configuration (*dynamic* adaptivity level).
- In any case, set a time-out on the ARP execution both for first-routing and re-routing requests.

In order to reliably model such a complex behavior, a combination Queueing Networks and Petri Nets is employed. In Figure 5.2 we illustrate our QPN model for ARP stage.

Customers of our car-navigation service perform routing requests through some client application according to an aggregated Poisson process of rate λ_f, modelled with the $Source$. The parameter λ_f represents the workload generated during a burst of first-routing requests, corresponding to peak hours of a working day.

For each first-routing request, clients are expected to issue periodic re-routing requests, seeking for better alternative routes, according to an aggregated Poisson process of rate $\lambda_r = 10\lambda_f$. This introduces a second class of requests in our model, which, in the literature, is dealt through *multi-class* Queueing Networks and *coloured*, generalized stochastic Petri Nets formalisms.

An important place in our QPN model is named $Resources$. This place models a *reserve* of available *servers* for the ARP stage. Therefore, $Resources$ place is preloaded with some number of tokens of another closed-class called $Nodes$. This third class is necessary to keep track, in this stage, of the number of busy servers, so that incoming requests must wait for a $Node$ to become available.

Routing requests arriving at the system are queued in $Incoming$ place and are served whenever a $Node$ is available. This is modelled by immediate $Serving$ transition, enabled by 1 token of either first-routing or re-routing class and 1 token of $Core$ class. As a result of $Serving$ transition firing, a new token of first or re-routing class, respectively, is generated in $Fork_{Serving}$ station.

Fork-Delay-Join component is a key part of our model. As previously mentioned, routing requests show highly variable behavior in their service time. Moreover, a timeout is set in any case to put a hard limit on the ARP service time.

Modelling such a functional requirements requires the following clever approach. Routing jobs arriving at station $Fork_{Serving}$ are split into 2 *fictitious*[1]tasks and sent to $Dist$ and $Timeout$ delay servers.

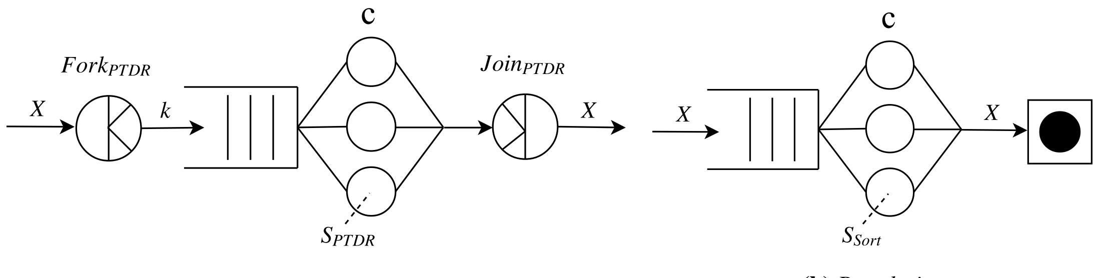
$Dist$ servers process each task according to some stochastic process, while $Timeout$ servers process tasks deterministically. To model *execution with timeout* behavior, we introduce a $Join_{Serving}$ station with *Quorum* strategy, as described in Figure 2.4.1. In particular, $Join$ station waits just for the *first* of the two tasks to complete, therefore dropping the other one. Indeed, a job arriving at $Fork$ station should depart from $Join$ station either according to $Dist$ stochastic process, or, if a too long time was sampled from the stochastic distribution, according to $Timeout$ server, which deterministically completes in a fixed amount of time. Necessarily, delay station is equipped with an infinite amount of servers, which is perfectly realistic since the number of physical processors is managed by $Core$ tokens from $Resources$ place.

Jobs departing from $Join$ station are sent in $Release$ place, enabling $Release$ transition which moves the routing request to the next pipeline stage and restores a $Core$ token in $Resources$ place.

5.3 PTDR Stage

Alternative routes discovered in ARP stage are forwarded to PTDR module to compute a travel time estimate on each of them. Since this process can be performed in parallel, for each ARP solution, producing k alternative routes, we spawn k parallel tasks

[1]We call them fictitious because no task is generated at that point in the real software pipeline. They are introduced in the model simply to implement the timeout mechanism, which otherwise would not be possible to model by means of a single QPN elementary object.



(a) *PTDR* **(b)** *Reordering*

Figure 5.3: *QPN models for PTDR (on the left) and Reordering (on the right) modules*

running PTDR on each of them.

In our model, illustrated in Figure 5.3a, we implement this behavior by means of a Fork-Join station generating k tasks for each job arriving at $Fork_{PTDR}$ station. Generated tasks are all sent to an M/M/c queue, serving tasks according to a Poisson stochastic process of rate $1/S_{PTDR}$. $Join_{PTDR}$ station, then, applies a *Standard Join* strategy, waiting for all k-tasks and forwarding jobs to the Reordering stage.

5.4 Reordering Stage

Reordering stage is, at the same time, the fastest and the easiest stage to model. At this step, arbitrary sorting rules are applied according to service provider policies. For instance, paths can be penalized if flowing across the city center, or nearby some city event. In this work, we assume fr this stage to take a negligible amount of time, following a Poisson process of rate $1/S_{Sort}$. Therefore, we employ a simple M/M/c station, processing jobs arriving from PTDR module and returning a final response to service customers. In Figure 5.3b we illustrate our QPN model for Reordering stage.

5.5 Capacity Planning

In previous section we presented our approach to our car-navigation system modelling, proposing a QPN model for each stage of the software pipeline. As we previously mentioned, we resorted to a system model to understand whether our car-navigation system would scale nicely and, if so, to discover the optimal amount of servers to handle a given expected workload.

In the literature, this is known as capacity planning problem and in this section we present a methodology for efficient plans exploration along with our experimental results.

We start by stating what we are looking for. From a system point of view, we consider each pipeline module to be deployed on a different *server rack*. Each rack is connected to the following, running the subsequent pipeline module, through some network we can assume fast enough to be considered negligible from an execution time perspective.

Each rack contains a number of servers, shipping some number of cores each. More specifically, there are c_{ARP} cores dedicated to ARP module, c_{PTDR} cores dedicated to PTDR module and c_{Sort} cores dedicated to Reordering module. Given some expected workload of rate λ_r, we perform a capacity planning analysis to discover the optimal values for c_{ARP}, c_{PTDR} and c_{Sort}.

Then, by assigning some values to λ_r, c_{ARP}, c_{PTDR} and c_{Sort}, we leverage JMT simulation engine to compute a number of performance indices, like system response time and system throughput. If the simulation reports stable response time and average throughput equal to the incoming requests rate, we known that we have found a valid capacity plan.

A naive, yet impractical, approach would be to explore every combination of c_{ARP}, c_{PTDR} and c_{Sort} values from 0 up to some limit providing a valid capacity plan. Considering that a single simulation can take from tens of seconds to several minutes, this approach appears clearly unfeasible.

In the following section we will illustrate our approach for an efficient resource space exploration.

5.5.1 Resource Exploration Design

In this section we provide a detailed description on how we explore the number of cores per each pipeline stage in our capacity planning analysis.

Let λ_r be the arrival rate of re-routing requests to the system. Let $\boldsymbol{S}$ be the vector of ARP, PTDR and Reordering mean service times, such that

$$\boldsymbol{S} = [S_{ARP}, S_{PTDR}, S_{Sort}]\,,$$

and let $S_{TOT} = \sum_i S_i$ be the total pipeline service time.

Then, let $\boldsymbol{X}$ be the vector of maximum throughput achievable by a single core of each pipeline stage, such that:

$$\boldsymbol{X} = \left[\frac{1}{S_{ARP}}, \frac{1}{S_{PTDR}}, \frac{1}{S_{Sort}}\right].$$

Since we are looking for the optimal number of cores per stage and since each stage must keep the same throughput in order to avoid undesired growing queues, it is reasonable to assume that higher number of resources should be allocated to stages that take longer to serve a single job. Therefore, we compute the vector $\boldsymbol{S}_{\%}$ of the fraction of time spent in each stage, as follows:

$$\boldsymbol{S}_{\%} = \left[\frac{S_{ARP}}{S_{TOT}}, \frac{S_{PTDR}}{S_{TOT}}, \frac{S_{Sort}}{S_{TOT}}\right]$$

.

Moreover, since we have to process jobs at least as fast as they arrive, again in order to avoid growing queues, we need at least $\boldsymbol{c}^{-}$ cores per stage, such that:

$$\boldsymbol{c}^{-} = \lceil \lambda_r \cdot \boldsymbol{S} \rceil.$$

Indeed, if a system receives 10 jobs/sec and it can process a job in 1 second, it would need, at least, 10 parallel workers to avoid increasing delays in the system response time.

While this works in the ideal scenario of constant arrival rate, it is very likely not to be enough under realistic arrival rate conditions. In fact, following the usual assumption considering request arrivals following a Poisson process, it might happen to experience inter-arrival times less than $1/\lambda_r$. Similarly, as service time follows some stochastic process as well, it is likely to take longer than average value S, again, introducing undesired delays and growing waiting queues.

Therefore, in our exploration, we are going to span from $\boldsymbol{c}^-$ up to some maximum number of cores $\boldsymbol{c}^+$, such that:

$$\boldsymbol{c}^+ = n \cdot \boldsymbol{c}^-.$$

Let $c^-_{TOT} = \sum_i c^-_i$ and $c^+_{TOT} = \sum_i c^+_i$ be respectively the minimum and maximum total amount of cores over the whole pipeline. We explore capacity plans as follows:

1. Let $c^i_{TOT} = c^-_{TOT}$.
2. Compute the number of cores per stage $\boldsymbol{c}^i$ according to the fraction of time spent in each stage, such that:

$$\begin{aligned}\boldsymbol{c}^i &= c^i_{TOT} \cdot \boldsymbol{S}_\% \\ &= \left[c^i_{ARP}, c^i_{PTDR}, c^i_{Sort}\right].\end{aligned}$$

3. Simulate QPN model for a re-routing requests arrival rate of λ_r and $\boldsymbol{c}^i$ capacity plan.
4. Compute next iteration total amount of cores c^{i+1}_{TOT}, such that:

$$c^{i+1}_{TOT} = \left(1 + \frac{1}{3}\right) c^i_{TOT}$$

5. If c^{i+1}_{TOT} is less than c^+_{TOT}, go to 2.
6. End exploration otherwise.

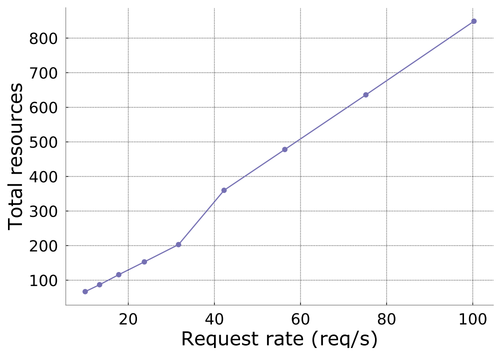
5.5.2 Experimental Results

In this section, we report our experimental results from a capacity planning analysis on our system under several configurations.

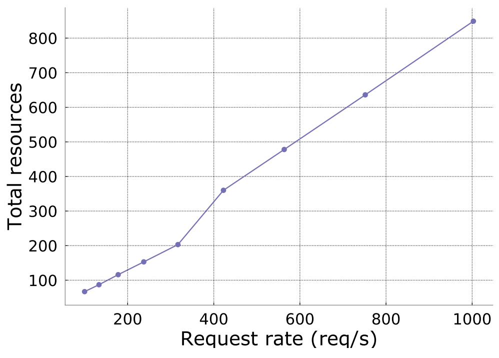
In this experiment we consider a model of a car-navigation system with number of alternative paths $k = 5$ and similarity threshold $\theta = 0.5$. ARP service time process have

Table 5.1: *Service time values of our model service stations for first-routing and re-routing requests for a car-navigation system with $k = 5$ and $\theta = 0.5$ on New York City map.*

Request class	Station	Service time distribution	Mean	CV
First-Routing	Dist	Pareto($\alpha = 2.289, k = 0.144$)	0.257	1.230
	Timeout	Deterministic($k = 1$)	1.000	0.000
	PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
	Reordering	Exponential($\lambda = 1000$)	0.001	1.000
Re-Routing	Dist	Pareto($\alpha = 2, k = 0.5$)	0.300	3023
	Timeout	Deterministic($k = 3$)	3.000	0.000
	PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
	Reordering	Exponential($\lambda = 1000$)	0.001	1.000



(a) *First-routing*



(b) *Re-routing*

Figure 5.4: *Optimal total amount of resources for varying* λ_f *and* λ_r *in* $[10, 100]$ *reqs/s and* $[100, 1000]$ *reqs/s respectively.*

been modelled according to ARLib's Penalty timing results, as described in section 3.4, on New York City map. Table 5.1 summarizes service time of each component.

Figures 5.4 and 5.5 depict results obtained from a capacity planning analysis using JMT simulator (v1.0.4-Beta4) employing a 99% confidence interval.

Given some arrival rate λ_r, we explored several capacity plans as described in subsection 5.5.1. We considered a capacity plan to be *valid* if:

- We measured a throughput equal to the arrivals rate.
- We measured a CPU utilization less than or equal to 70%.
- We measured a system response time at 95th percentile for first-routing requests less than 1.3 seconds.
- We measured a system response time at 95th percentile for re-routing requests less than 3.9 seconds.

Then, among all valid capacity plans, we elected as optimal the one requiring the least total amount of resources.

In Figures 5.4a and 5.4b we illustrate optimal total amount of cores to manage arrivals rates spanning from $\lambda_f = 10$ reqs/sec and $\lambda_r = 100$ reqs/sec to $\lambda_f = 100$ reqs/sec and $\lambda_r = 1000$ reqs/sec respectively.

Obtained results show that our car-navigation system scales linearly in the number of necessary cores with respect to the arrival rate for $k = 5$ and $\theta = 0.5$.

In Figures 5.5a and 5.5b we depict measured system response time for each value of λ_f, λ_r and optimal capacity plan.

5.6 An Overload-Tolerant Model Extension

In the previous section, we provided experimental evidences that our car-navigation system is capable of scaling, making it a feasible option for a car-navigation service. Moreover, we presented a working approach to perform an efficient capacity planning analysis in order to reliably size the system and accommodate some expected workload.

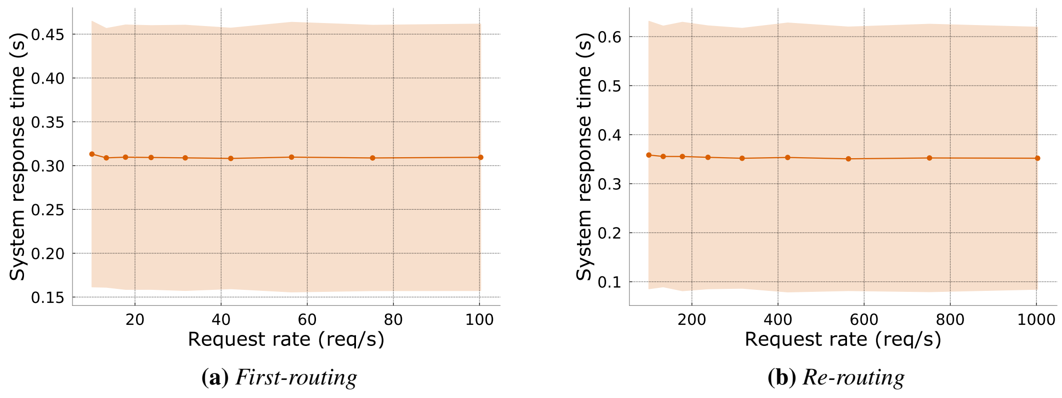


Figure 5.5: *95th percentile system response time under optimal capacity plan for varying λ_f and λ_r in* $[10, 100]$ *reqs/s and* $[100, 1000]$ *reqs/s respectively.*

In this section, we want to make another step forward. In section 5.1 we illustrated a relevant issued in HPC context, i.e. a certain rigidity in resource acquisition and release, which may result in both slow resource scaling in presence of unexpected workload increase and resource, and money, wasting in presence of a workload significantly lower than expected. Sure, a navigation service provider would employ historical data to characterize its workload, for example categorizing it with respect to daytime time slots. On the other hand, let us consider the scenario of a serious car-accident in some city hot-spot or of a natural disaster. Events like those would cause an immediate, sustained increase in service workload to route cars in some area away and avoid undesired congestions in such a critical zone.

Indeed, while scaling policies should be in place to issue resource acquisition requests to HPC infrastructure provider in case of excessive arrivals rate, within the timeframe between request and actual resource allocation, incoming jobs would saturate all the available service stations raising system response time over any acceptable threshold.

In order to overcome this issue, we propose an extension to the QPN model presented in section 5.2 at ARP stage.

Our idea is simple yet effective. We introduce a fast lane, parallel to the main pipeline. According to Little's Law [63–65], the number of customers waiting at some station is equal to the arrival rate at the station times the service time of the station itself. Consequently, if the system is working in flow balance condition, i.e. number of arrivals equals number of completions, the number of waiting jobs is zero. On the other hand, in case the arrivals rate increases, yet the number of resource stays the same, according to Little's Law, the number of waiting requests will start growing.

In order for our fast lane to be effective, it should process jobs considerably faster than the main one, to cope with the workload increase. Nonetheless, we want to keep serving valid responses to navigation users, maybe degrading solution quality, but we do not consider dropping requests a valid option. Indeed, request drop might be interpreted by users, or clients, as momentary service unavailability, causing recurrent, successive queries to the system, viciously further increasing the arrivals rate.

Therefore, we propose a fast lane computing just a single shortest path and return it

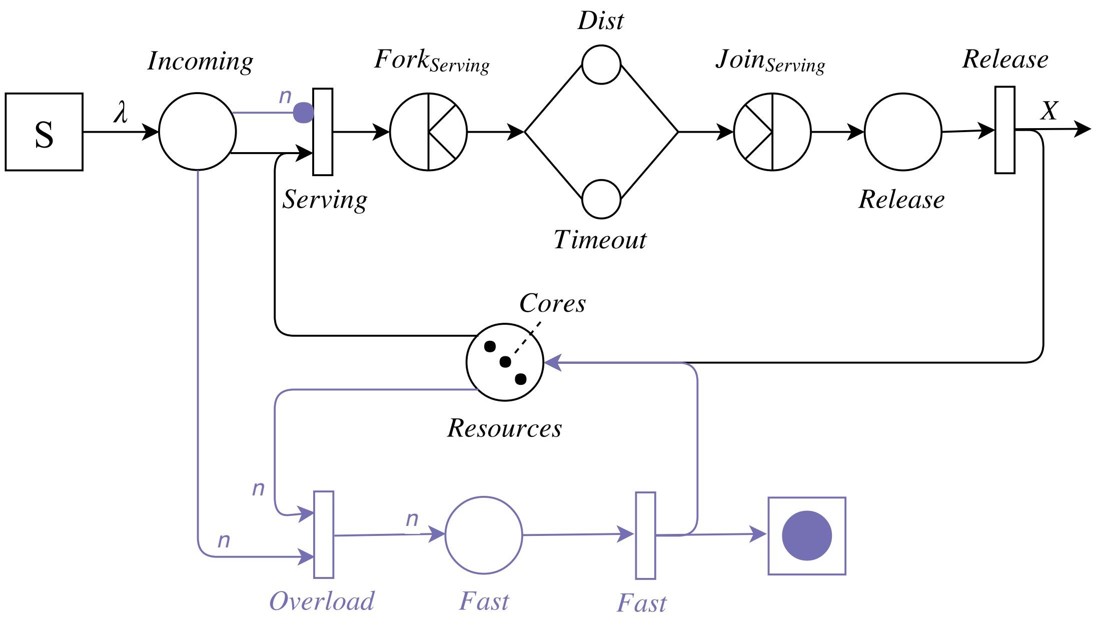


Figure 5.6: *QPN extended Alternative Route Planning model with fast lane. Differences with base model from Figure 5.2 are highlighted in purple.*

immediately to the user. That lane should only be open in case a given number of customers is detected in ARP waiting queue, while main pipeline should be preferred when flow-balance condition is restored. Figure 5.6 illustrates our ARP model extension.

For starters, we introduce a $Overload$ transition routing jobs to flow in the fast lane. $Overload$ transition is enabled when n jobs are waiting in $Incoming$ place, where n is a parameter to size depending on arrivals rate and service times. Clearly, we also need n cores in $Resources$ to process those n jobs.

Moreover, since having n tokens in $Incoming$ place concurrently enables $Serving$ and $Overload$ transitions, we introduce an *inhibitor arc* in $Serving$ transition in order to deterministically prefer fast lane.

Then, jobs are forwarded to $Fast$ place to be served by $Fast$ timed transition. When job processing completes, the job exits the system, responding to waiting user, and a $Core$ token is restored in $Resources$ place.

5.6.1 Experimental Results

In this section, we report our experimental results on the overload-tolerance capability of our extended model.

In this experiment, we want to verify that a car-navigation system, sized to effectively run under a given workload, is capable of handling a significantly higher requests rate by leveraging additional fast lane, while still serving some jobs through the main pipeline.

We designed the experiment as follows:

1. Let λ_f and λ_r be the expected arrivals rates for first-routing and re-routing requests, respectively.

Table 5.2: *Service time values of our extended model service stations for first-routing and re-routing requests for a car-navigation system with $k = 5$ and $\theta = 0.5$ on New York City map.*

Request class	Station	Service time distribution	Mean	CV
First-Routing	Dist	Pareto($\alpha = 2.289, k = 0.144$)	0.257	1.230
	Timeout	Deterministic($k = 1$)	1.000	0.000
	Fast	Exponential($\lambda = 6.293$)	0.160	1.000
	PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
	Reordering	Exponential($\lambda = 1000$)	0.001	1.000
Re-Routing	Dist	Pareto($\alpha = 2, k = 0.5$)	0.300	3023
	Timeout	Deterministic($k = 3$)	3.000	0.000
	Fast	Exponential($\lambda = 6.293$)	0.160	1.000
	PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
	Reordering	Exponential($\lambda = 1000$)	0.001	1.000

2. Perform a capacity planning analysis for λ_f and λ_r on the *base* model as described in subsection 5.5.1 and discover the optimal number of resources $\boldsymbol{c}^*$.

3. Using capacity plan $\boldsymbol{c}^*$, run several simulations on the *extended* model, increasing λ_f and λ_r up to $n\lambda_f$ and $n\lambda_r$.

4. Measure system response time and rate of jobs served through the fast lane.

In Figure 5.7 we depict our results from a JMT (v1.0.4-Beta4) simulation with base $\lambda_f = 10$ reqs/sec and $\lambda_r = 100$ reqs/sec of a car-navigation system with $k = 5$ and $\theta = 0.5$ on New York City. Table 5.2 summarizes service time of each component.

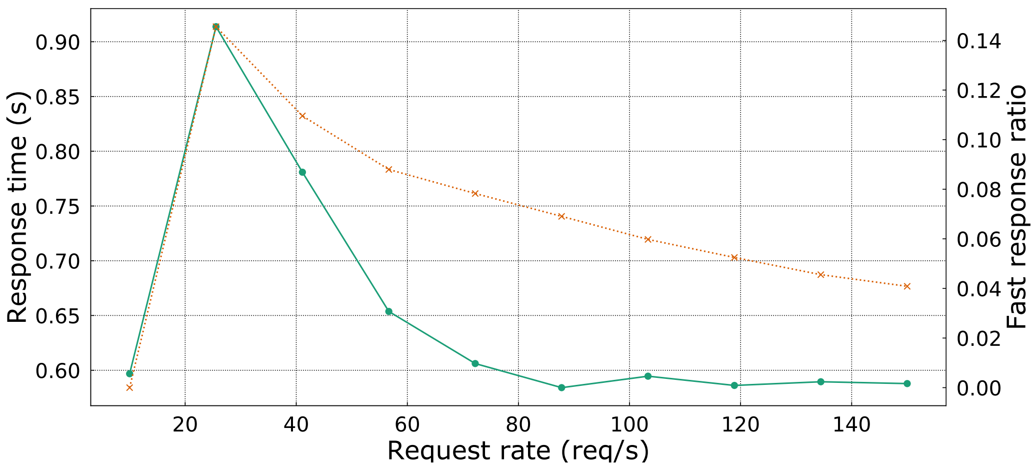
Let us consider Figure 5.7a. On the x-axis we report λ_f arrivals rate, on the left y-axis, in green, solid line, we plot 95th percentile system response time for first-routing requests, while on the right y-axis, in orange, dotted line we plot the rate of requests served through the next fast lane.

For starters, we notice that for $\lambda_f = 10$, the same rate according to which the base system is sized, the number of jobs served by the fast lane is zero. This is fundamental in order to guarantee a degree of consistency with the base model.

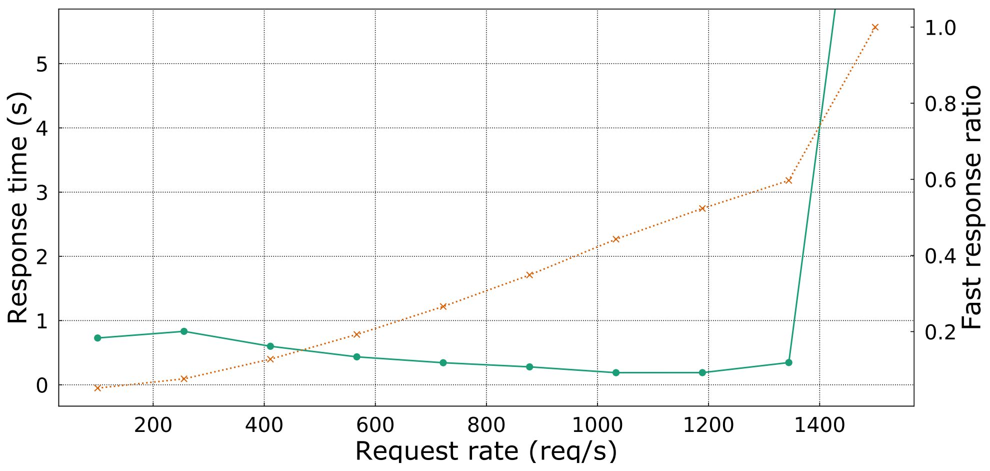
In second place, we observe that even for higher values of λ_f, no job is served by fast lane. We should remember that first-routing requests are both the fewest and the fastest to process, so it is reasonable than they never queue. On the other hand, it might surprise the reader to see the response time decreasing even if no job gets routed to fast lane. In fact, this is a typical behavior of multi-class systems. As we are going to show in a moment, since several re-routing jobs are directed to fast lane, the immediate result is that the main pipeline turns out to be less loaded, enabling jobs to experience faster waiting time.

Let us move our attention to Figure 5.7b. Again, on the x-axis we report λ_f arrivals rate, on the left y-axis, in green, solid line, we plot 95th percentile system response time for re-routing requests, while on the right y-axis, in orange, dotted line we plot the rate of requests served through the next fast lane.

Just like for first-routing requests, we notice that for base $\lambda_r = 100$ reqs/sec the number of jobs served through the fast lane is consistently zero. On the contrary, by overloading the system with a higher number of requests we observe that the rate of requests processed by fast lane monotonically increases. As a consequence of that,



(a) *First-routing*



(b) *Re-routing*

Figure 5.7: *Overloading scenario of a system running $\lambda_f = 10$ and $\lambda_r = 100$ optimal capacity plan. System is overloaded by varying λ_f and λ_r in $[10, 150]$ reqs/s and $[100, 1500]$ reqs/s respectively. In green, solid lines we plot 95th percentile system response time. In orange, dotted lines we plot the rate of requests routed to fast lane to cope with the excessive workload.*

the system response time decreases, because more and more jobs skip the full pipeline to cope with the excessive workload. In this experiment, we observe our overload-tolerance extension hitting its limit for $\lambda_r = 1500$ reqs/sec. Under that condition, 100% of incoming requests are forwarded to fast lane but the number of available resource is still not enough to manage such a workload, leading the system response time to diverge.

Finally, we notice that even for a number of requests ten times bigger than expected, the system is still able to serve more than 50% of requests through the main pipeline, granting top quality of results to a relevant number of users.

5.7 Summary

In this chapter, we presented a Queuing Petri Nets model, abstracting the car navigation system introduced in chapters 3 and 4. Resorting to System Modelling techniques was fundamental to efficiently analyze our navigation service from extra-functional perspectives using a computer simulation tool like JMT.

Together with the system model, we presented a methodology to carry out a capacity planning analysis, efficiently exploring the number of necessary resources for each pipeline stage, given some expected arrivals rate. Achieved results exhibited scalability property of our system, with a number of required computational resources linear with the incoming workload. Moreover, we further extended the proposed QPN model to introduce overload-tolerance capabilities by means of a fast lane, degrading the solution quality in order to process requests faster. This way the system would be able to handle sustained workload spikes and keep the system utilization within reasonable limits while more resources are being allocated. Conducted experiments showed that, considering a number of cores sized for some arrivals rate, our model extension allowed to manage more than ten times that rate by directing more and more routing jobs to the fast processing lane.

In the next chapter, we are going to conduct a validation experiment for our QPN model by comparing simulation and actual system performance indices by carrying out a case study on the city of Milan.

CHAPTER 6

Milan: A Case Study

In chapter 5, we presented our approach to a capacity planning analysis for our car-navigation system, driven by a simulated Queueing Petri Nets model. Employing a system model turned out to be an effective choice, enabling us to experiment on system variations in order to extend it with overloading-tolerance capabilities.

Experimental results, moreover, proved us that a car-navigation service, so designed, is able to scale to effectively accommodate an increasing number of requests. In this chapter, we want to do the last step to verify our proposed methodology, by designing a validation test to compare simulation and real system results starting from a case study on Milan urban area.

6.1 A Service For A Smart City

In the context of providing a traffic optimization service for a smart city, we are interested in applying our approach to a car-navigation system to real data, describing the traffic behavior of a municipality. Indeed, the whole methodology we presented in this thesis work focuses on optimization strategies to tailor such a system of some municipality needs in order to reduce the number of resources required to grant a fast, satisfactory service with the highest possible quality.

Therefore, we begin our case study on Milan urban area by analyzing the requirements of such a smart city and apply, step by step, our methodology to package a realistic service that this municipality could adopt.

In this area, the population is composed of approximately 4 Million people. Every day, local agencies estimate to have more than 5 Million trips, considering that only less than 50% are done using public transportation [66, 67]. For the sake of simplicity, let us consider to have 2.5 Million trips, by car, per day.

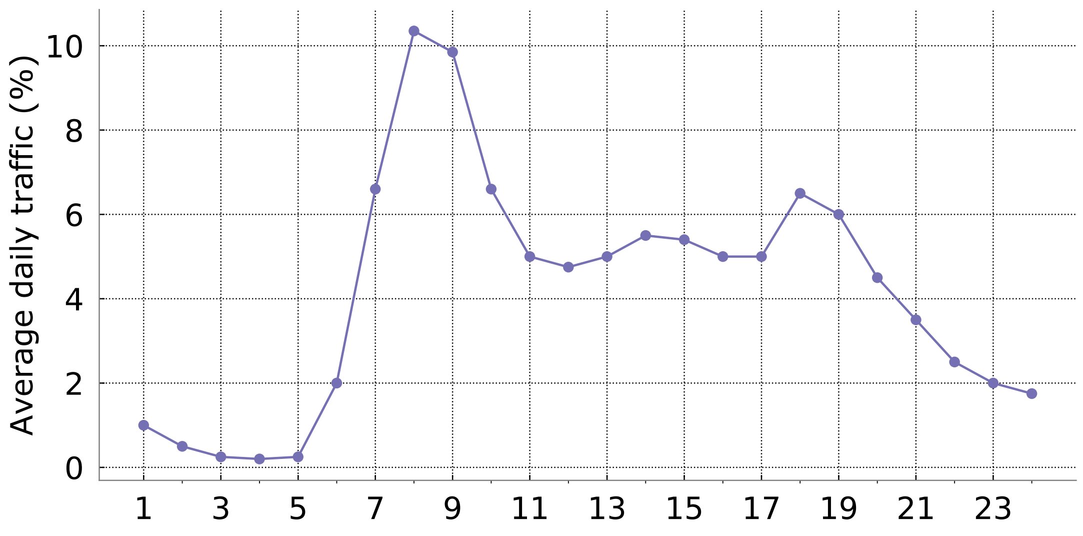


Figure 6.1: *Average daily traffic distribution per-hour of a working day in Milan urban area.*

In Figure 6.1 we depict the average daily traffic distribution according to a study commissioned by Milan municipality [68]. On the x-axis, we fix the daily time-slots whereas on the y-axis we report the percentage of car trips measured within a hour range.

As expected, we notice minimum traffic during the first hours of the day, exploding to a maximum peak between 7 and 9 am, when people drive to work, carry children to school and so on. Then again, after some hours of stable traffic, we observe another peak between 5 and 7 pm, when most workers drive back home.

In the interest of roughly characterizing the daily workload of our car-navigation system, and conduct an effective capacity planning analysis, we could start by dividing the traffic profile in correspondence of severe trend changes.

A possible splitting could define the following ranges: 0-6, 7-10, 11-16, 17-19 and 20-24. Each split accounts for some percentage of daily traffic. For each of them, we should carry out a capacity planning analysis, depending of the time slot workload.

Let us focus, for a while, on the most intense time slot, from 7am to 10am, as depicted in Figure 6.2. This peak range accounts for 33.4% of the daily traffic, resulting in 853'000 first-routing requests within a 4-hours slot.

Making a naive assumption, we are going to consider those requests as uniformly distributed across the 4-hours time span. We understand that is might be far from reality, but in this case study we shall focus on applying our methodology to deliver a traffic optimization system for a smart city, rather than developing the most reliable characterization of the expected workload for the Milan area.

As a result, we obtain 208'750 requests per hour, i.e. 57.99 first-routing requests per second. On top of that, considering an average of 10 periodic re-routing requests for each first-routing one, we add 579.86 re-routing requests per second. Similarly, we follow the same approach to estimate the workload for each other time slot throughout the day.

For the case study, we introduce two possible service plans that Milan municipality

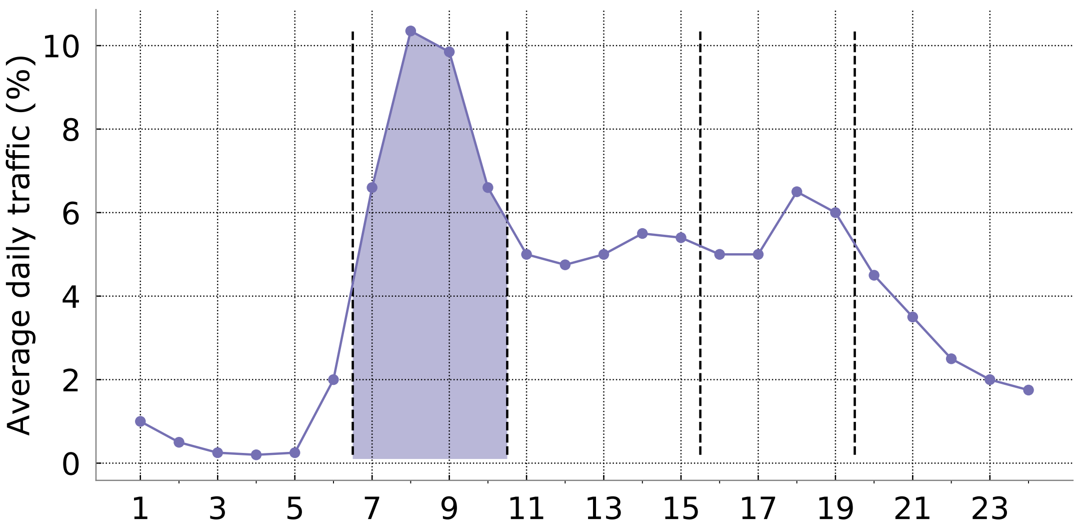


Figure 6.2: *Average daily traffic distribution of a working day in Milan urban area, divided in time slots according to severe changes in traffic trend. Colored area highlights maximum peak hours.*

could provide to its citizens: *Basic* and *Premium* plans. The two services differentiate by quality of service and deployment costs and are summarized in Table 6.1.

Basic plan would be characterized by $k = 2$ and $\theta = 0.5$ configuration. Moreover, in order to process requests as fast as possible, and therefore requiring less resources, Basic plan would employ, for alternative route planning step, Penalty algorithm only, which we showed in subsection 3.5.1 to be, on average, the fastest one.

On the other hand, Premium plan offers the full-optional service, delivering all the features we presented so far. It would be characterized by $k = 5$ and $\theta = 0.5$, hence increasing the probability of finding the fastest route to travel at present time. Alternative route planning would be performed by Auto-tuner suggested, optimal algorithm, maximizing the quality of service within a satisfactory response time. As a consequence of that, requests would be processed with a higher service time, thus raising the required number of resources.

From the characteristics of aforementioned service plans we apply our capacity planning analysis from subsection 5.5.1. To run this analysis we set ARP service times according to real-data statistics from ARP executions, running ARLib's algorithms on 10'000 P2P source destination pairs generated as described in section 3.4, within Milan urban area. Service time measurements are obtained on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (16 cores, 20MB Cache), running Ubuntu 18.04 LTS.

Then, we perform a capacity planning analysis through a JMT (v1.0.4-Beta4) simulation with station timing parameters summarized in Table 6.2 for both Basic and

Table 6.1: *Overview of possible service plans offered by a smart city municipality.*

Service Plan	k	θ	Auto-tuner	ARP Algorithms	QoS	Deployment Cost
Basic	2	0.5	No	Penalty	Low	\$
Premium	5	0.5	Yes	OnePass+, ESX, Penalty	High	\$\$\$

Table 6.2: *Service time JMT simulation parameters summary for Basic and Premium plans.*

Service Plan	Request Class	Station	Service Time	Mean	CV
Basic	First-Routing	Source	Exponential($\lambda = 57.99$)	0.020	1.000
		Dist	Pareto($\alpha = 3.07, k = 0.13$)	0.190	0.550
		Timeout	Deterministic($k = 1$)	1.000	0.000
		PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
		Reordering	Exponential($\lambda = 1000$)	0.001	1.000
	Re-Routing	Source	Exponential($\lambda = 579.86$)	0.002	1.000
		Dist	Pareto($\alpha = 3.07, k = 0.13$)	0.190	0.550
		Timeout	Deterministic($k = 3$)	3.000	0.000
		Fast	Exponential($\lambda = 6.293$)	0.160	1.000
		PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
		Reordering	Exponential($\lambda = 1000$)	0.001	1.000
Premium	First-Routing	Source	Exponential($\lambda = 57.99$)	0.020	1.000
		Dist	Pareto($\alpha = 2.99, k = 0.27$)	0.410	0.585
		Timeout	Deterministic($k = 1$)	1.000	0.000
		PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
		Reordering	Exponential($\lambda = 1000$)	0.001	1.000
	Re-Routing	Source	Exponential($\lambda = 6.293$)	0.160	1.000
		Dist	Pareto($\alpha = 2.02, k = 0.34$)	0.680	5.240
		Timeout	Deterministic($k = 3$)	3.000	0.000
		Fast	Exponential($\lambda = 6.293$)	0.160	1.000
		PTDR	Exponential($\lambda = 38.462$)	0.026	1.000
		Reordering	Exponential($\lambda = 1000$)	0.001	1.000

Premium service plans.

Table 6.3 summarizes achieved results. Each row describes the optimal infrastructure sizing for each daily time slot, both for Basic and Premium plans. According to traffic-monitoring data, early hours in the morning and late hours in the night register the least traffic levels and, consistently, our analysis reports the least amount of resources required to run the navigation service, with just 22 cores for Basic and 62 core Premium plans. On the other hand, in correspondence to the maximum traffic peak, from 7 to 10 am, the amount of resources grows to reach the maximum value of 290 cores for Basic and 696 for Premium plans.

This simulation reveals an interesting opportunity from a financial perspective. Without loss of generality, let us consider Premium plan scenario. While it is true that 696 cores are required to deliver the navigation service with no slowdowns throughout the day, our model showed that such amount of resources is necessary during the 4-hours maximum peak only. This means that for the rest of the day, a significantly lower amount of cores could be rented from supercomputing centers, dramatically reducing the municipality expenses. In fact, a system running the whole day on 696 cores would cost the provider 16’704 core-hours. By resorting to our system model, we achieve the lower value of 8’452 core-hours, granting a 51% saving on infrastructure rental.

Concerning the decision to adopt Basic or Premium plan, the usual factors apply. As we already mentioned, raising the number of alternative paths k and resorting to slower but more accurate algorithms causes the system to trade some speed in response computation in order to achieve a higher quality in the final result. Higher quality of

Table 6.3: *Capacity plannning analysis for Basic and Premium plans considering different time slots during the day.*

	Basic				Premium			
Time Slot	**ARP**	**PTDR**	**Reordering**	**Total**	**ARP**	**PTDR**	**Reordering**	**Total**
01 - 06	16	5	1	**22**	51	10	1	**62**
07 - 10	228	60	2	**290**	584	111	1	**696**
11 - 15	147	39	1	**187**	373	71	1	**445**
16 - 19	154	41	1	**196**	393	75	1	**469**
20 - 24	61	16	1	**78**	200	38	1	**239**

solutions, indeed, will make service users happier, increasing their level of fidelization and building a stronger user base that will hardly choose other similar, competitor services. On the other hand, some municipality may decide not to allocate enough financial resources to a traffic-optimization service shipping all the features presented in this work, while it might settle with reasonable subset of them.

In any case, the degree of adaptivity offered by our car-navigation system enables a variety of customizations to effectively tailor such a service on the provider needs and finances.

In the following section, we will present our validation experiments, applying the methodology illustrated in previous chapters on the Milan urban area and validate the conclusion we drew against real system execution data.

6.2 Validation Test Design

In the previous section we described a simple step-by-step customization process to deliver our car navigation service to a smart city and its citizens. To do that, we considered a concrete example, focusing our attention on Milan. Starting from some historical data illustrating the average daily traffic, we isolated the most critical time slot, which generates the biggest volume of requests and we performed a capacity planning analysis to find the optimal amount of resources to effectively deliver the service, with respect to two different possible plans.

In this section we aim at validating the simulation results we obtained in previous section and show that the real system performs comparably to what the model says.

We performed our validation tests on Milan urban area, considering the area ranging in $[45.3743, 45.5509]$ latitude and $[9.0519, 9.3507]$ longitude (EPSG:4326/WGS84 reference system), pictured in Figure 6.3 and retrieved by OpenStreetMap [44] database.

In this validation test, we are going to focus on ARP module only for several reasons:

- It is the pipeline stage accounting the most for service time, hence representing the most critical section for an efficient navigation service.
- It is the stage this work of thesis mostly focused on.
- It is the stage for which we developed a high-performance software implementation that we can employ for real-world evaluation.

Therefore, we are going to validate the following two systems: for Basic plan, we are considering a straightforward system accepting first-routing and re-routing requests and

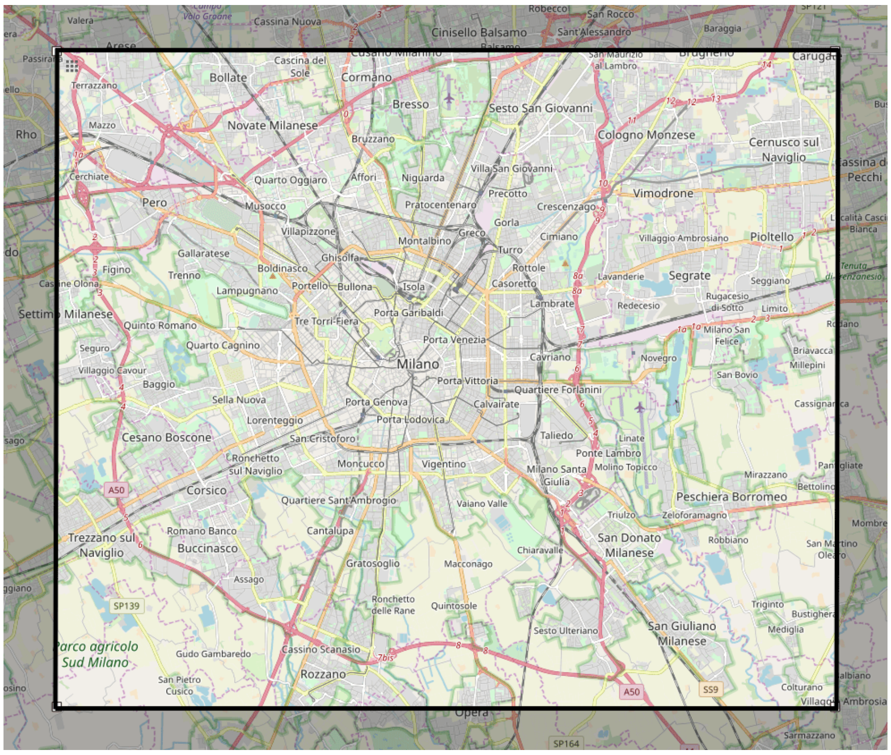


Figure 6.3: *Milan map considered in this validation experiment. Latitude and longitude limits are marked by black, solid bounding box.*

computing 2 alternative routes with a similarity threshold of 50% employing Penalty algorithm.

Conversely, for the Premium plan, we take into account a system accepting first-routing and re-routing requests, computing 5 alternative routes with a similarity threshold of 50%, but leveraging the optimal algorithm for each request, suggested by an auto-tuning module making decisions based on per-request features.

Concerning the auto-tuner, we are training its predictive models following the same pipeline we presented in section 4.5. We build an ARP algorithms behavior dataset, as described in section 3.4, to collect real ARLib algorithms execution time values on sampled source-destination pairs from Milan area. We extract several features from gathered data and train a machine-learning model for each $(k, \theta, \text{algorithm})$ configuration.

From a resources perspective, we are employing an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (16 cores, 20MB Cache), limiting the maximum number of concurrent ARP executions to 16. Therefore, we resort to the system modelling approach from chapter 5 to do the reverse: discovering how many first-routing and re-routing requests per second, λ_f and λ_r, we can handle with the available compute power. Indeed, while in previous section we concluded that 128 and 411 cores would be necessary to execute ARP module for Basic and Premium plan respectively, we do not have such a compute power available. Therefore, in our validation experiment we are going to scale the number of incoming requests so that, according to our system model, we should be able to handle it. If real system execution confirms our model's behavior, we can consider the model trustworthy and reliably employ it for any mix of incoming requests rates.

The last step we need to make is to build a driver application simulating request arrivals and dispatching them to ARP module. Such a simulator would take λ_f and λ_r rates as input parameters and issue first-routing and re-routing requests according to a Poisson process. A number of workers implementing proactive ARP service would accept those requests. By leveraging trained predictive models, an auto-tuning module would recommend the optimal algorithm for each request depending on system's (k, θ) configuration. Finally, each request is processed by some worker according the optimal algorithm suggested by the auto-tuner.

In the following section, we introduce our car-navigation system simulator implementing all the steps we have just described.

6.3 Car Navigation Service Simulator

The car-navigation simulator we built is a concurrent, client-server application written in C++17 and Python3.6, putting in place the proactive variant of our navigation service.

The server application is written in Python and it implements the auto-tuner module to suggest the optimal algorithm for a given input request. We decided to leverage Python for such a task because of the very mature and widely adopted software stack for machine learning development. Indeed, we employed Numpy, Scipy, Scikit-learn and StatsModels frameworks in order to explore, implement and validate the model training pipeline we described in chapter 4.

On the other hand, the C++ application implements the real navigation service simulator, generating routing requests, performing features extraction and processing ARP queries. Its main dependencies are Boost.Graph [30] for efficient graph representation and, of course, ARLib for Alternative Route Planning step.

In Figure 6.4 we depict a UML diagram of our simulator top-level architecture.

From a high-level perspective, the simulator is based on three main components: *CarNavigationSystem*, *PipelineWorker* and *Auto-tuner*.

The *CarNavigationSystem* component is instantiated with a rich set of arguments fully specifying the simulation scenario, such as:

- (k, θ) configuration.
- The map graph description.
- A set of source-destination pairs to employ for generated requests.
- λ_f and λ_r arrivals rates.
- t_f and t_r service time timeouts.
- Number of parallel ARP workers c_{ARP}.

Moreover, *CarNavigationSystem* starts a pool of c_{ARP} workers, communicating with them through a synchronized single-producer, multi-consumer queue for ARP jobs dispatch.

PipelineWorker component implements an independent worker processing ARP requests. Under the hood, it leverages *Pipeline* class to: compute shortest path between

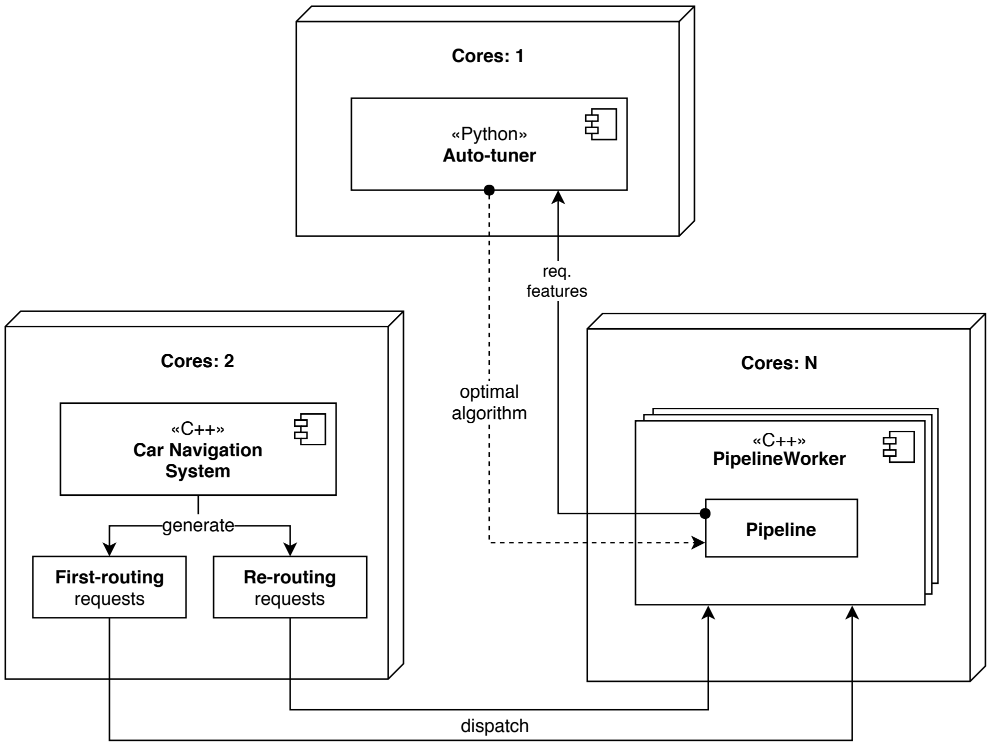


Figure 6.4: *Car-navigation simulator architecture diagram. Node boxes show the required number of cores.*

source and destination nodes, extract features from it, issue a request to the *Auto-tuner* for the optimal algorithm to run and, finally, process the ARP query.

Auto-tuner component, on the other hand, implements a TCP server predicting the optimal ARLib's algorithm to run depending of some request's features. *Auto-tuner* is instantiated with a set of pre-trained predictive models, learned according to the process described in section 4.5.

In the UML diagram we also reported the number of cores we suggest to allocate to each component. This specification will be necessary in order to assess if the available compute power can handle the number of requests generated by the simulator.

6.4 Experimental Results

In this section we report our results for the experiment we described in section 6.2, validating the system model for Milan municipality against real measured data from the navigation service illustrated in section 6.3.

QPN system model service times are characterized from real data collected from our navigation service implementation on 10'000 P2P source-destination pairs within Milan urban area.

On the contrary, validation measurements are obtained on a different set of 10'000 P2P source-destination pairs, generated from a different random-numbers-generator *seed*.

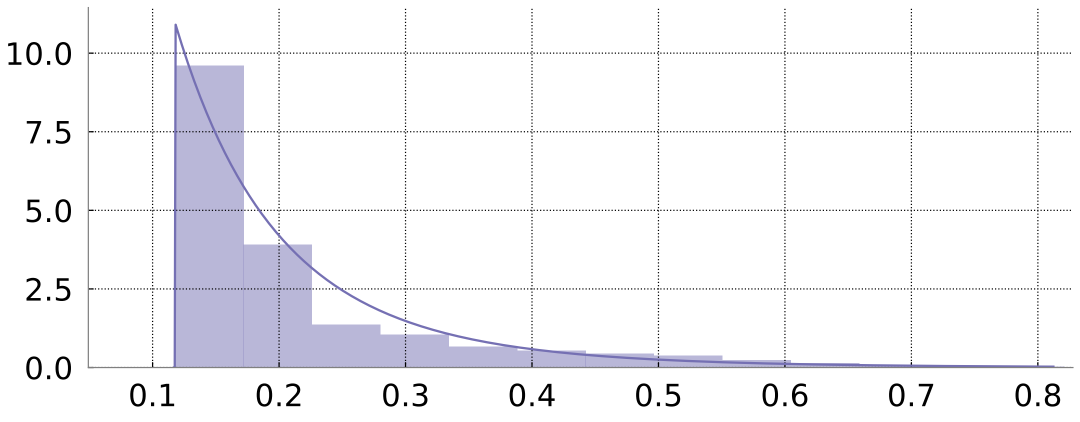


Figure 6.5: *Basic-plan service time empirical distribution (histogram) vs. theoretical Pareto distribution (line plot).*

Table 6.4: *Comparison of Simulation vs. Actual performance indices for Basic-plan navigation service.*

Performance Index	**Simulation** (Average)	**Simulation** (Std. Dev.)	**Actual** (Average)	**Actual** (Std. Dev)	**Percentage Error**
First-routing Throughput	3.98	N/A	3.97	N/A	0.2%
Re-routing Throughput	40.03	N/A	39.26	N/A	2%
First-routing Service Time	0.21	0.10	0.22	0.12	4.5%
Re-routing Service Time	0.22	0.12	0.22	0.13	0%
First-routing Response Time	0.22	0.11	0.25	0.13	12%
Re-routing Response Time	0.23	0.14	0.25	0.14	8%

6.4.1 Basic Plan

In this experiment we are validating a model for a system accepting first-routing and re-routing requests and computing $k = 2$ alternative routes with a similarity threshold θ of 50% employing Penalty algorithm, as reported in Table 6.1.

As we mentioned, our hardware available for measurements ships 16 physical cores. To generate first-routing and re-routing requests we allocate 2 cores to the *car navigation system* component, while another core must be allocated to the *auto-tuner* module. As a result, we are left with 13 free cores to dedicate to ARP module.

According to our QPN system model, by setting first-routing requests arrivals rate $\lambda_f = 4$ reqs/sec and re-routing requests arrivals rate $\lambda_r = 40$ reqs/sec, the average servers utilization is around 70%, which is ideal according to the performance evaluation rule of thumb mentioned in chapter 5.

From ARP service time perspective, we use a Pareto distribution for both first-routing and re-routing requests, choosing their parameters with an optimization process based on Maximum Likelihood Estimation method [69]. In Figure 6.5 we depict the empirical distribution of Basic-plan service times against a theoretical Pareto distribution fitted from real data.

In Table 6.4 we report our measurements comparing simulation results against real system values.

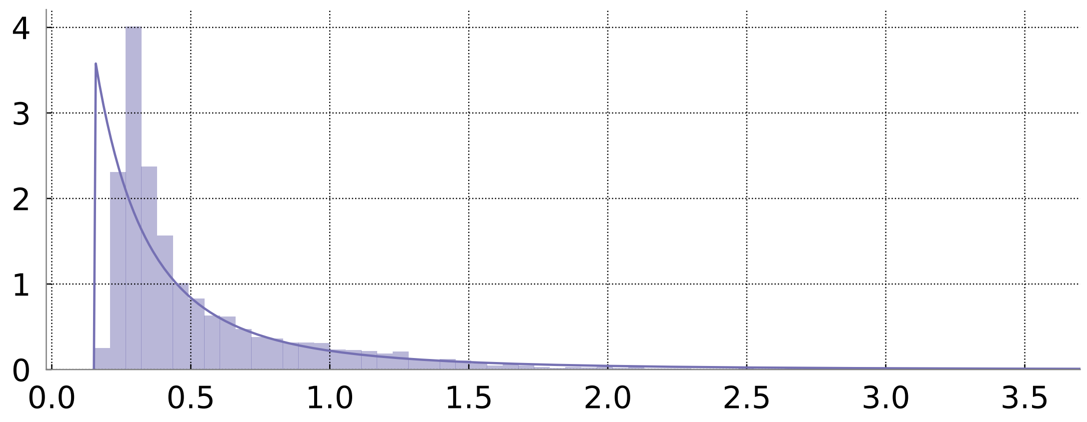


Figure 6.6: *Premium-plan service time empirical distribution (histogram) vs. theoretical Pareto distribution (line plot).*

Achieved results show a general agreement between simulated and actual values. Indeed, we notice that most of the comparisons report a percentage error on average values under 10%. The only error reaching the percentage of 12% is related to first-routing requests response time, which is relatively critical considering that first-routing requests contribute only in a small portion to the overall system workload.

Finally, we mention that simulated and measured system throughput values agree, with a maximum error of 2%, and they match the generated request arrival rates which confirms that our capacity planning analysis suggested a correct number of resources to effectively handle the expected workload.

Summing up, achieved results support the claim that our QPN system model is a good abstraction of the real system, making it a trustworthy tool to analyze Basic-plan service behavior.

6.4.2 Premium Plan

In this experiment, we validate a model for a system accepting first-routing and re-routing requests, computing $k = 5$ alternative routes with a similarity threshold θ of 50%, as reported in Table 6.1, and processing ARP requests using the optimal algorithm for each request, suggested by an auto-tuning module taking decisions based on per-request features.

Hence, this model abstracts the proactive-variant of our car navigation system, distinguishing from previous Basic-plan model by means of different service time characterization. Indeed, as we already mentioned, our Premium service would trade some speed in response computation in order to raise solutions quality.

Again, due to limited hardware availability, we can allocate 13 cores to ARP module maximum. Consequently, we resort to our QPN system model to discover the maximum number of requests we can handle, while keeping a CPU utilization under 70% max.

According to our model, we can effectively manage $\lambda_f = 1.1$ first-routing requests per second and $\lambda_r = 11$ re-routing requests per second.

Again, from ARP service time perspective, we employ two Pareto distributions, for

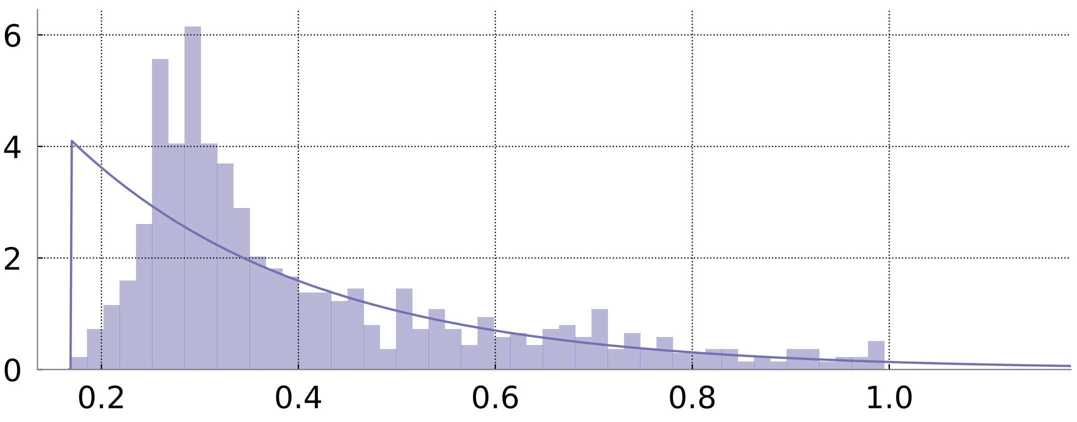


Figure 6.7: *Premium-plan first-routing service time empirical distribution (histogram) vs. fitted, theoretical Pareto distribution (line plot).*

first-routing and re-routing requests, fitted from real data. A picture comparing re-routing service times empirical distribution against fitted, theoretical Pareto is drawn in Figure 6.6.

In Table 6.5 we report our measurements comparing simulation results against real system values.

Let us begin from measured throughput. As in previous experiment, both simulation and actual values agree with an error of 2.8% maximum, with a measured throughput matching the generated arrivals rate, which allows us to conclude that our capacity planning analysis suggested a sufficient number of resource for the expected workload. Furthermore, re-routing requests service and response time agree nicely, with a negligible error of 1.5% max.

On the other hand, first-routing requests timing results show, apparently, some discouraging values, with a percentage error of 23% on response time index. Such a relevant error in simulation, according to our investigation, must be accounted to the employed theoretical distribution for service time sampling, which poorly describes the real service time distribution. As a matter of fact, Premium-plan first-time requests are processed via some suggested algorithm that is expected to find a solution within 1 second timeout. Moreover, some algorithms are also executed with a hybrid-approach,

Table 6.5: *Comparison of Simulation vs. Actual performance indices for Basic-plan navigation service. Simulation service time values are drawn from two theoretical Pareto distributions.*

Performance Index	**Simulation** (Average)	**Simulation** (Std. Dev.)	**Actual** (Average)	**Actual** (Std. Dev)	**Percentage Error**
First-routing Throughput	1.10	N/A	1.07	N/A	2.8%
Re-routing Throughput	10.93	N/A	10.82	N/A	1%
First-routing Service Time	0.40	0.16	0.50	0.28	19%
Re-routing Service Time	0.64	0.45	0.64	0.62	0%
First-routing Response Time	0.40	0.18	0.52	0.30	23%
Re-routing Response Time	0.65	0.48	0.66	0.63	1.5%

pairing to them a Penalty run in order to ensure a valid solution within the available time budget. In Figure 6.7 we draw the empirical first-routing requests distribution versus the fitted theoretical one. As we can notice, the empirical distribution shows an important right tail, possibly caused by the strong differences in terms of execution time among ARLib's algorithms. To our knowledge, no simple distribution models such behavior.

To support this claim, we executed another simulation, this time directly employing real service times for first-routing requests, instead of resorting to some theoretical distribution to draw service times from. Such a simulation, in fact, is possible in JMT tool by means of *Replayer* feature [17], which allows us to explicitly provide a service time series.

As a result, we obtain 0.50 sec service time average value, with 0.27 sec standard deviation, performing a 0% error on the average actual values. Moreover, we achieve 0.50 sec average response time in simulation, with 0.29 sec standard deviation, committing a 3.8% error on the actual value, as reported in Table 6.6.

Summing up, in this experiment we successfully validated our QPN system model for Premium-plan as well. Indeed, we observed that modelling service time for auto-tuner suggested algorithms was poorly achieved by means of a single, theoretical distribution, especially for short available time budgets. On the other hand, employing real data series let us obtain satisfactory results, further supporting our QPN system model as a trustworthy tool for our navigation service behavior analysis.

Finally, in Table 6.7 we report pick rate and failure rate for ARLib's algorithms.

Pick rate is the percentage of times some algorithm was selected as optimal by the auto-tuner, subject to the constraint of 1 second time budget for first-routing requests and 3 seconds for re-routing requests.

As we already observed in chapter 4, ESX algorithm is always dominated by either OnePass+ or Penalty, so the auto-tuner never picks it. Conversely, we notice that Hybrid OnePass+ is picked more than 50% of the times, both for first-routing and re-routing requests, making the introduction of an auto-tuner module a winning point of our service, which will serve best quality of results for more than half of the queries. On the other hand, failure rate represent the number of times a *complete* solution was not discovered within the available time frame. We consider a solution to be complete if a ARP query asking $k = 5$ alternative paths returns exactly 5 routes.

Table 6.6: *Comparison of Simulation vs. Actual performance indices for Basic-plan navigation service. Simulation first-routing service time values are obtained replaying real measurements, while re-routing service time ones are drawn from theoretical Pareto distribution.*

Performance Index	**Simulation** (Average)	**Simulation** (Std. Dev.)	**Actual** (Average)	**Actual** (Std. Dev)	**Percentage Error**
First-routing Throughput	1.10	N/A	1.07	N/A	2.8%
Re-routing Throughput	10.93	N/A	10.82	N/A	1%
First-routing Service Time	0.50	0.27	0.50	0.28	0%
Re-routing Service Time	0.64	0.45	0.64	0.62	0%
First-routing Response Time	0.50	0.29	0.52	0.30	3.8%
Re-routing Response Time	0.65	0.48	0.66	0.63	1.5%

Table 6.7: *ARLib's algorithms auto-tuner pick rate and failure rate statistics.*

Request Class	Algorithm	Pick rate	Failure Rate
First-routing	OnePass+ (*hybrid*)	54.93%	0.37%
	ESX (*hybrid*)	0.00%	N/A
	Penalty	45.07%	8.66%
Re-routing	OnePass+ (*hybrid*)	59.24%	0.00%
	ESX (*hybrid*)	0.00%	N/A
	Penalty	40.76%	0.05%

Considering Hybrid OnePass+ algorithm, we notice that the algorithm failed to compute a complete solution only less than 1% of the times, while always completing rerouting requests processing. On the contrary, Penalty algorithm scores more than 8% of failure rate on first-routing requests. Considering that Penalty always converges to a complete solution, as shown in subsection 3.5.3, this result means that computing 5 alternative routes with maximum overlapping of 0.5 in less than a second is not feasible for 8.66% of validation data.

In order to cope with this situation, two options are available: either we increase the available time budget for first-routing requests or we simply settle with the alternative routes computed at the timeout moment, hence forwarding less than five paths to the PTDR module following in the system.

6.5 Summary

In this chapter, we conducted a validation experiment to evaluate the goodness of our car navigation system model in simulation with respect to real system execution. We started from a case study on the city of Milan, roughly characterizing the service workload based on real traffic-monitoring data collected by local authorities. Hence, we applied our capacity planning methodology to discover the optimal number of resources needed to deliver two types of service, named *Basic* and *Premium* plans, identified by different quality of service and amount of financial resources required. Moreover, due to lack of computational power, we scaled those values to fit our hardware and run validation tests both for Basic and Premium plans.

Achieved results successfully validated our QPN system model for Basic and Premium plans, showing it to be a trustworthy tool for performance evaluation studies on our car navigation system. Finally, concerning Premium plan, pick and failure rates values confirm the goodness of our auto-tuning module, which exploits the available time budget to deliver top quality results in more than half of service queries, while minimizing the chances of request failure.

CHAPTER 7

Conclusions

In this thesis, we addressed the problem of designing a car navigation system for upcoming market scenarios. The rising wave of self-driving cars and a forecast exponential growth of GPS navigation systems demand in the next eight years [1] anticipate the urgent need of smart services ready to react to a massive flood of routing requests. The main outcome is a methodology for an adaptive alternative route planning module, leveraging machine learning models to combine state-of-the-art algorithms and selecting the best one on a per-request basis. Moreover, we designed and implemented an efficient, configurable alternative route planning C++ library, named *ARLib*, offering the first publicly-available, production ready solution to search for alternative paths in a real scenario. Furthermore, we experimentally validated our methodology on New York City and Milan urban areas to shows the applicability of our presented approaches. The remainder of this chapter summarises the finding and limitation of the proposed approach and provides recommendations for future works.

Main contributions

The main results of the work carried out in this thesis might be summarised as follows:

- While proof-of-concept implementations exist for state-of-the-art algorithms, no realistic implementation was publicly available. As a matter of fact, implementers resort to custom data structures or slowly-performing languages, for the sole purpose of presenting the algorithm, making the interoperability between ARP heuristics impossible. In the context of performing a fair comparison between those algorithms and actually employ them in a real world car navigation system, we developed an efficient, configurable C++ library, named *ARLib*. Given its flexibility, ARLib proved itself to be a fundamental component of our car navigation system, enabling us to experiment several route planning configurations, switching kernels for single shortest path computation and tuning terminating conditions. Considering its adherence to Boost.Graph conventions and ease of integration, ARLib could be employed by any software developer interested in alternative routing as a drop-in solution in existing code-bases using Boost.Graph.

- Starting from ARLib algorithms implementations, we performed a Design Space Exploration of ARP problem parameters ranging from the number of alternative paths k to compute, to the overlapping threshold θ, from the ARP algorithm to the single shortest path kernel employed by those algorithms. We analyzed the design space from execution time and quality perspective along with search failure rate. This experiment showed that, on average, OnePass+ algorithms achieve best quality of results while Penalty is both the fastest and the most reliable, never failing to discover a valid solution.

- We presented and evaluated a methodology to train a machine learning module to recommend the optimal algorithm for each incoming routing query. Indeed, considering a per-request perspective, instead of an average case scenario, we discovered that each algorithm performance changed depending on query characteristics. Experimental results showed that our machine learning model underestimated algorithms execution time a negligible number of times, therefore suggesting best quality algorithm for a relevant fraction of queries.

- In the context of providing a realistic navigation service to a smart city, being able to perform feasibility studies and capacity planning analysis is a key aspect for a municipality government in its process of deciding whether to allocate financial resource to such a service or not. Experimental results achieve by simulating our proposed car navigation system model demonstrate the scalability of our system design, showing a linear relation between the expected workload and the required number of resources to handle it. Furthermore, our proposed extended model exhibited *overload-tolerance* capabilities, allowing the system to absorb unexpected workload peaks while gently transitioning to a higher number of HPC resources.

- From applicability perspective, we carried out a case study on the Milan urban area, applying our methodology to a concrete example, starting from local authorities historical traffic data. We employed our car navigation system model to perform a capacity planning analysis for two type of service plans, different from quality of results and financial resources required. In both cases, a validation experiment supported our claim on the goodness of both system model and auto-tuning module, showing, for the former, minimal percentage error between simulated performance indices and actual ones. For the latter, measured statistics showed a relevant percentage of queries processed by OnePass+ with minimal failure rate, therefore granting best quality of results to a proper amount of queries, while still resorting to Penalty for those complex requests that require the fastest algorithm to be successfully processed.

Recommendation for future works

Experimental evaluations of the proposed methodology show promising results; however, there are still open questions to investigate. In our opinion, the most challenging point to solve are the following:

- In this work, three state-of-the-art algorithms were considered for execution time and quality comparisons and implemented in ARLib. In the literature, several other heuristics are available, most notably Pareto [11] and Plateau [14] algorithms. Therefore, providing high-quality implementations of those in ARLib would enable a much wider analysis on solution time and quality trade-offs, possibly identifying new optimal heuristics for particular routing queries.
- In our alternative route planning algorithms analysis, to focused our attention on **spDifference** quality metric, arbitrarily choosing it as our reference goodness measurement. However, no investigation was carried out to study whether this measurement matches human understanding of a good routing solution or whether new quality metrics should be developed to better summarize desirable result features.
- While proposed machine learning model for optimal algorithm recommendation achieved satisfactory results, no good punctual prediction could be achieve, leading us to resort to an upper bound prediction of the execution time. The main problem we faced was the so-called *heteroscedasticity* of our data, possibly meaning that engineered features do not completely explain the variance in our measurements. Therefore, a deeper study on execution-time-prediction task learnability should be carried out, in order to identify better features or more advanced learning techniques and improve model performance.
- Validation results showed that proposed QPN model of our car navigation system is a trustworthy representation of it, leading to minimal differences between simulation and actual values when considering the service time of a single algorithm. On the other hand, using a simple service time theoretical distribution proved itself to be a poor model of real values when considering the complex scenario of an auto-tuning module electing the optimal algorithm to employ. Therefore, an extension of our QPN model could be carried out to better abstract the auto-tuner behavior from an extra-functional perspective.

Bibliography

[1] I. Research, “Global automotive navigation system market forcast 2019-2027.” `https://www.inkwoodresearch.com/reports/global-automotive-navigation-system-market/`, 2018.

[2] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser, “Alternative routing: K-shortest paths with limited overlap,” in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL ’15, (New York, NY, USA), pp. 68:1–68:4, ACM, 2015.

[3] R. Bader, J. Dees, R. Geisberger, and P. Sanders, “Alternative route graphs in road networks,” in *Theory and Practice of Algorithms in (Computer) Systems*, pp. 21–32, Springer, 2011.

[4] R. Tomis, L. Rapant, J. Martinovič, K. Slaninová, and I. Vondrák, “Probabilistic time-dependent travel time computation using monte carlo simulation,” in *International Conference on High Performance Computing in Science and Engineering*, pp. 161–170, Springer, 2015.

[5] M. Golasowski, R. Tomis, J. Martinovič, K. Slaninová, and L. Rapant, “Performance evaluation of probabilistic time-dependent travel time computation,” in *IFIP International Conference on Computer Information Systems and Industrial Management*, pp. 377–388, Springer, 2016.

[6] E. Vitali, D. Gadioli, G. Palermo, M. Golasowski, J. Bispo, P. Pinto, J. Martinovic, K. Slaninova, J. M. Cardoso, and C. Silvano, “An efficient monte carlo-based probabilistic time-dependent routing calculation targeting a server-side car navigation system,” *IEEE Transactions on Emerging Topics in Computing*, 2019.

[7] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser, “Exact and approximate algorithms for finding k-shortest paths with limited overlap,” in *20th International Conference on Extending Database Technology: EDBT 2017*, pp. 414–425, 2017.

[9] A. Paraskevopoulos and C. Zaroliagis, “Improved alternative route planning,” in *OASIcs-OpenAccess Series in Informatics*, vol. 33, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[10] Y. Chen, M. G. Bell, and K. Bogenberger, “Reliable pre-trip multi-path planning and dynamic adaptation for a centralized road navigation system,” in *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pp. 257–262, IEEE, 2005.

[11] D. Delling and D. Wagner, “Pareto paths with sharc,” in *International Symposium on Experimental Algorithms*, pp. 125–136, Springer, 2009.

[12] P. Hansen, “Bicriterion path problems,” in *Multiple criteria decision making theory and application*, pp. 109–127, Springer, 1980.

[13] E. Q. V. Martins, “On a multicriteria shortest path problem,” *European Journal of Operational Research*, vol. 16, no. 2, pp. 236–245, 1984.

[14] CAMVIT, “Choice routing.” `http://camvit.com/`, 2009.

[15] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queuing network models*, vol. 22. Prentice Hall Upper Saddle River, 1984.

[16] C. A. Petri, "Kommunikation mit automaten," *Universitat Hamburg*, vol. PhD. Thesis, 1962.

[17] M. Bertoli, G. Casale, and G. Serazzi, "An overview of the jmt queueing network simulator," *Politecnico di Milano-DEI, Tech. Rep. TR*, vol. 2007, 2007.

[18] F. Bause *et al.*, "Stochastic petri nets–an introduction to the theory," 2002.

[19] M. Bertoli, G. Casale, and G. Serazzi, "The jmt simulator for performance evaluation of non-product-form queueing networks," in *40th Annual Simulation Symposium (ANSS'07)*, pp. 3–10, IEEE, 2007.

[20] M. Bertoli, G. Casale, and G. Serazzi, "Jmt: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[21] G. Casale and G. Serazzi, "Quantitative system evaluation with java modeling tools," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pp. 449–454, ACM, 2011.

[22] M. Gribaudo, *Theory and application of multi-formalism modeling*. IGI Global, 2013.

[23] Grand View Research, "Self driving Cars and Trucks Market Size, Share & Trends Analysis Report By Application (Transportation, Defense), By Region (NA, Europe, APAC, South America, MEA), And Segment Forecasts, 2020 - 2030." `https://www.grandviewresearch.com/industry-analysis/driverless-cars-market`, 2018.

[24] Grand View Research, "Global Positioning Systems (GPS) Market Size, Share & Trends Analysis Report By Deployment, By Application (Aviation, Marine, Surveying, Location-Based Services, Road), And Segment Forecasts, 2018 - 2025." `https://www.grandviewresearch.com/industry-analysis/gps-market`, 2018.

[25] Market Realist, "A Look at the Courier Service Industry in the United States." `https://articles.marketrealist.com/2015/07/look-courier-service-industry-united-states/`, 2015.

[26] GraphHopper, "GraphHopper Routing Engine." `https://github.com/graphhopper/graphhopper`.

[27] Singapore's Open Data Portal, "K shortest path algorithms for networkx." `https://github.com/datagovsg/k-shortest-path`.

[28] Qi, Yan, "K-Shortest Paths - Yen's algorithm." `https://github.com/yan-qi`.

[29] Sh., Meral, "K-Shortest Path - Yen's algorithm." `https://mathworks.com/matlabcentral/fileexchange/32513-k-shortest-path-yen-s-algorithm`.

[30] J. Siek, L.-Q. Lee, and A. Lumsdaine, "Boost graph library (bgl)." `https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/index.html`, 2000.

[31] Boost.org, "Boost Software Licence." `https://www.boost.org/users/license.html`.

[32] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in c++," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 291–310, 2006.

[33] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[34] D. R. Musser and A. A. Stepanov, "Generic programming," in *International Symposium on Symbolic and Algebraic Computation*, pp. 13–25, Springer, 1988.

[35] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *The computer journal*, vol. 9, no. 3, pp. 275–280, 1966.

[36] J. Siek, L.-Q. Lee, and A. Lumsdaine, "Property maps." `https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/using_property_maps.html`, 2000.

[37] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in c++," in *First Workshop on C++ Template Programming*, 2000.

[38] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicore architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 173–180, IEEE, 2015.

[39] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *ACM SIGPLAN Notices*, vol. 50, pp. 379–390, ACM, 2015.

[40] "Test-driven development." `https://en.wikipedia.org/wiki/Test-driven_development`.

[41] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.

[42] "Continuous integration." https://en.wikipedia.org/wiki/Continuous_integration.

[43] "Travis CI." https://travis-ci.org/.

[44] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org ." https://www.openstreetmap.org, 2017.

[45] C. Demetrescu, A. Goldberg, and D. Johnson, "9th dimacs implementation challenge - shortest paths." http://www.diag.uniroma1.it/challenge9/index.shtml, 2006.

[46] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957.

[47] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[48] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[49] P. W. Holland and S. Leinhardt, "Transitivity in structural models of small groups," *Comparative group studies*, vol. 2, no. 2, pp. 107–124, 1971.

[50] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[51] A. C. Aitken, "On least squares and linear combination of observations," *Proceedings of the Royal Society of Edinburgh*, vol. 55, pp. 42–48, 1936.

[52] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[53] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[54] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, ACM, 2016.

[55] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[56] A. Kolmogorov, "Sulla determinazione empirica di una legge di distribuzione," *Inst. Ital. Attuari, Giorn.*, vol. 4, pp. 83–91, 1933.

[57] N. Smirnov *et al.*, "Table for estimating the goodness of fit of empirical distributions," *Annals of Mathematical Statistics*, vol. 19, no. 2, pp. 279–281, 1948.

[58] M. Biometrika19685511Wilk and R. Gnanadesikan, "Probability plotting methods for the analysis of data," *Biometrika*, vol. 55, pp. 1–17, 1968.

[59] K. Pearson, "On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[60] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.

[61] P. Mell, T. Grance, *et al.*, "The nist definition of cloud computing," 2011.

[62] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.

[63] J. D. Little, "A proof for the queuing formula," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[64] W. S. Jewell, "A simple proof of: L= λ w," *Operations Research*, vol. 15, no. 6, pp. 1109–1116, 1967.

[65] S. Eilon, "Letter to the editor—a simpler proof of l= λ w," *Operations Research*, vol. 17, no. 5, pp. 915–917, 1969.

[66] Milano Agenzia Mobilita' Ambiente e Territorio, "Annual mobility report." https://www.amat-mi.it/it/documenti/.

[67] Marco Bedogni, Milano Agenzia Mobilita' Ambiente e Territorio, "Road traffic measures in the city of milan." http://www3.gdos.gov.pl/Documents/Wizyty/W%C5%82ochy/Road%20Traffic%20Measures%20in%20the%20city%20of%20Milan.pdf.

[68] Lavecchia, C. and Pilati, S. and Angelino, E. and Fossati, G., "Analisi dei dati di traffico esistenti per la definizione dei profili temporali: metodologia ed esempio di applicazione.." http://groupware.sinanet.isprambiente.it/expert_panel/library/ept13/fossati-angelino-pilati-_1/download/1/Fossati-Angelino-Pilati-Lavecchia-ARPAL_GALILEO-dati%20traffico.pdf.

[69] S. S. Wilks, “The large-sample distribution of the likelihood ratio for testing composite hypotheses,” *The Annals of Mathematical Statistics*, vol. 9, no. 1, pp. 60–62, 1938.