# ABSTRACT

BHAT, SHIREESH. Network Service Orchestration within the ChoiceNet Architecture. (Under the direction of Dr. George N. Rouskas and Dr. Rudra Dutta.)

In this research we present Network Service Orchestration algorithms for Open Marketplaces which allow for various Data Plane Services in the routing domain to be advertised, queried, composed, purchased and provisioned. We use ChoiceNet as an example of an Open Marketplace in our work. Orchestration of services allows for constructing a ''composed service'' using the various compatible services participating in the Marketplace in response to a ''composed service'' request by the User. The Orchestration algorithm presents the User with not just ''a composed service'' but a list of ''composed service(s)'' to choose from. Our contribution can be classified into two main categories. First, we enable Orchestration by solving three key problems: a) Identify compatibility of adjacent services in a composed service; b) Provide the ability to compare service offerings from different providers and c) Develop a Planner (Orchestration Algorithm) module with request/response automation. Second, we develop three complementary algorithms which perform service Orchestration: a) Find optimal $k$ composed services in a Marketplace, which allows combining multiple service functionalities into one service; b) Find optimal time-dependent, time-constrained composed services which supports in-advance path reservation and c) Find a optimal composed tour of services. We address the key problems for enabling Orchestration by first defining the Semantics Language for advertising the Data Plane Services to be compatible with other services which are a logical choice. In addition, we define the Protocol for interaction between the entities of ChoiceNet to achieve complete automation of the Planner. Later, we present three flavors of Planners which perform service orchestration on three different graph models which correspond to three different Network Applications.

Network Service Orchestration within the ChoiceNet Architecture

by
Shireesh Bhat

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

| | |
|---|---|
| Dr. George N. Rouskas<br>Co-chair of Advisory Committee | Dr. Rudra Dutta<br>Co-chair of Advisory Committee |
| Dr. David Thuente | Dr. Khaled Harfoush |

Dr. Ilya Baldin

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# Chapter 1

# Introduction

In the journey from the ARPANET world in the 1960's to the Internet of the 21st century the network has evolved significantly. One of the major factors which has contributed to the popularity and success of the Internet of today is the plethora of services which are now available at the edge of the network. The invisible barrier which seems to be preventing the evolution of the core network (the backbone of the current Internet) at the same rate as the edge network is the lack of sustained innovation. This brings up the question ''What does it take for both the core network and the edge network which are integral to the functioning of the Internet, to evolve at a faster rate?".

Unfortunately, there is no silver bullet to the problem of asymmetrical growth in the building blocks of the Internet. One of the ways we can narrow the gap in the rate of innovation in the core and the edge network is through an ''Open Marketplace" [1] which allows for various stakeholders of the Internet infrastructure to be compensated for the innovation/reliability by the users of this infrastructure. This would have a positive impact in leveling the playing field for the application providers who use the underlying Internet backbone, Content Delivery Network (CDN) service providers, infrastructure providers whose resources support the application and CDN service providers, and the end users who rely on these service providers. The keyword ''Open" in the Open Marketplace is used to differentiate this from a Marketplace which is designed for a setting where the syntax and semantics of the advertised service follows a proprietary notation and the advertised services represent a small and centrally controlled network environment.

We use the term ''network services" to broadly classify any service which is offered by a provider in the ''Open Marketplace" for a potential user. The ''Marketplace" is not just a single entity but it represents all the entities which fulfill the functionality of allowing the network service providers to advertise their services and the users to choose from the set of offerings and compensate the providers for the prorated service usage. The implicit objective of the

‘‘Marketplace” is to establish trust between the service providers and the users. Once the users have used the service(s) provided at the Marketplace, the users verify if the service performed as advertised with the help of measurement tools and provide feedback, building a sense of trust between the users and the providers.

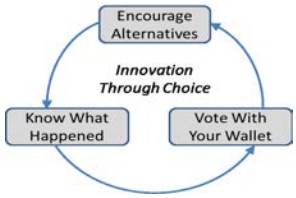


Figure 1.1: Foundation principles and their dependencies

Three principles originally proposed in [1] which hold the key for bringing in sustained innovation at the core network through an ‘‘Open Marketplace” are

- *‘‘Encourage Alternatives”* which is realized through the Marketplace by allowing multiple service providers to compete for the business from the users
- *‘‘Know What Happened”* allows the users to know the services which performed well and the services which did not
- *‘‘Vote With Your Wallet”* is the compensation to the service providers from the users for the services which delivered as advertised.

These three principles and their dependency on each other is illustrated in Figure 1.1.


Figure 1.2: Feature dependency (reflects the principles)

As we move from inception to application [76, 77], the features and their dependency which is a reflection of the above principles are shown in Figure 1.2. The three features which we relate to when describing the realization of ChoiceNet are

- *''Level Playing Field* which talks about the Open Marketplace were network services can compete on level terms and users can search and compare network services
- *''Verification and Accountability"* which talks about segregating services which performed well from those which did not
- *''Reward Selectively"* is the compensation which is handed out selectively to providers of the network service which delivered as promised.

These features are responsible for driving innovation at the core network by enabling choice and gaining acceptance among the service providers and the service users.

Applying the principles described above in the current Internet would require the understanding and approval of all the service providers and the users who would want to be a part of the innovation cycle. Figure 1.3 first proposed in [1] shows how the above principles shape the underlying network and the interaction between the various entities which are part of this network. The ''ChoiceNet" architecture [1] is a means to realize the three fundamental principles in the current Internet and is described in Figure 1.3. The current ''ChoiceNet" architecture has evolved from the one envisioned in Figure 1.3 and is described in Figure 1.4. Using the evolved architectural diagram we briefly describe the modules and their role in the Architecture:

- *Marketplace*: is a vital cog in the functioning of ChoiceNet and is the central entity which interacts with all the entities in ChoiceNet and influences in their decision making.
- *Verification Infrastructure* [2]: supports in the realization of the second principle ''Verification and Accountability". Only by effectively separating the roles and responsibilities of a service being measured and verified can we achieve accountability of providers of the service. It is also responsible for building the reputation of the service providers in the Marketplace by providing feedback.
- *Planner*: can churn out sophisticated plans using the advertisements in the ''Marketplace". The ''Planner" is responsible for presenting the users with a choice of alternate service offerings. The ''Planner" presents a list of composed service(s) to the user to choose from. A composed service is a meta service which consists of multiple services with a prespecified ordering rule. The functionality of a ''Planner" is described using the term ''service composition". ''Service Composition" can be realized at various levels i.e., horizontal and vertical layers of the network and can be broadly classified into *''Protocol Stack Composition"*, *''Path (Route) Composition"*, *''In-network service Composition"* or a combination of all three. When we use the term ''service composition" we imply a combination of all of them. The role of the ''Marketplace" in the functioning of the ''Planner" now becomes

clear as only by expressing a network service through a semantic language in the Marketplace can we hope to separate the ownership/responsibility/boundary of one network service from another, thereby constructing a wide range of "composed service(s)" which meet(s) the user requirements.

- *Provisioning Infrastructure*: a "composed service" can be provisioned in the network in one of two ways. *"Full Delegation"* [1], the onus of rendering a service on the user request is transferred sequentially from the first service to the next service, the services which we refer to in this context are the ones which make up the composed service, this is analogous to the current Internet approach where the next hop is responsible for routing the packet based on destination address. *"Transparent"* [1], this model is analogous to the strict/loose source routing where the service which needs to be applied next on the user request is part of the user request.

- *In-force Contracts*: is a repository of all the active service contracts.

- *Service Infrastructure*: represents the services which are made available by the providers who are compensated in the Economy Plane. We will discuss about provisioning in the service infrastructure later in this work to highlight how a service first seen in the "Marketplace" is realized in the network, bringing to fruition the network service life cycle.

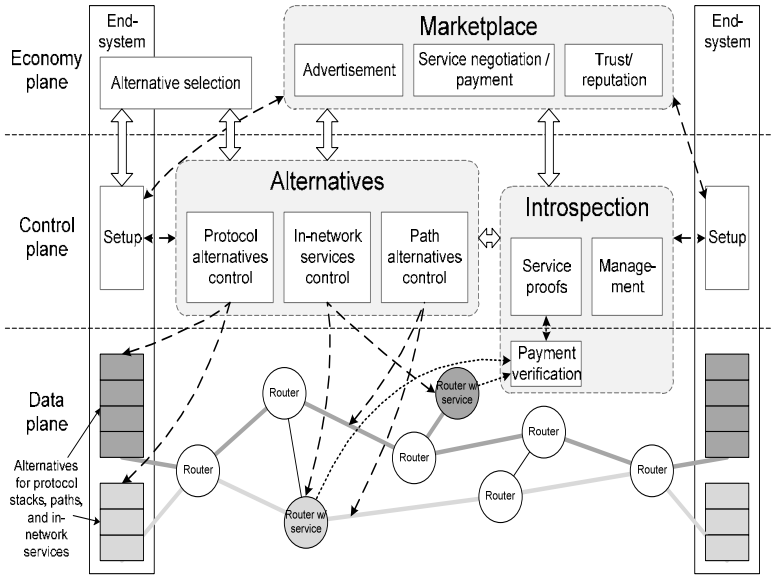


Figure 1.3: Initial ChoiceNet Architecture

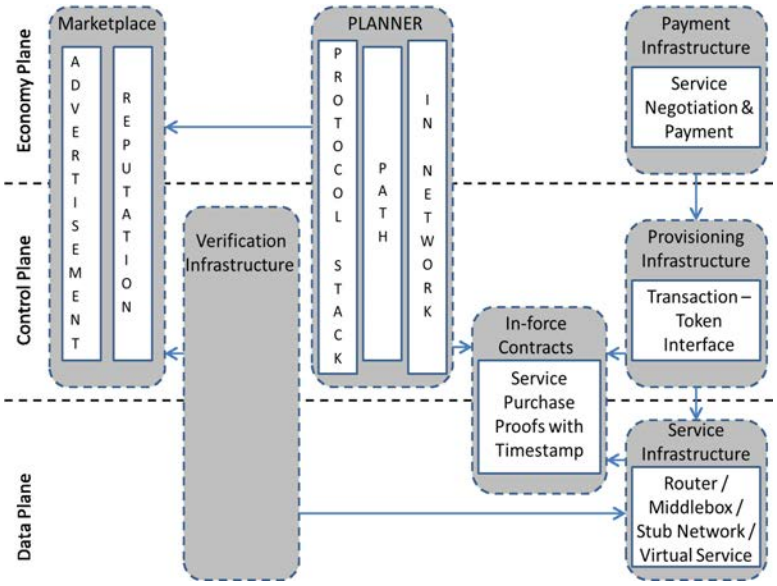


Figure 1.4: Evolved ChoiceNet Architecture

"Service Orchestration", "Service Composition", "Service Choreography", "Service Stitching", "Planning", are some of the terms which have been associated with integrating services to provide a value added service which provides an end-to-end functionality, which cannot be provided by the individual services on their own. We use the term "Planner" which performs service orchestration and "Orchestration Algorithm" interchangeably in this work. In the context of Network services, Orchestration refers to the integration of services across heterogeneous networks using an Open Marketplace where the services advertised by the providers in different domains can be purchased for short term or long term time scales. We envision that resellers would make up the bulk of the sellers in the Marketplace by providing value added services. Our contribution is in solving two key problems:

- A semantics language which forms the basis of the interaction between the entities within ChoiceNet and the Economy Plane Protocol which defines the interactions.
- A Planner which performs orchestration by
  - Extracting the service advertisements selectively from the Marketplace based on the user request
  - Building one or more composed service(s) from services which could be individual or composed service.

In this work we present several novel orchestration algorithms which are designed for an Open Marketplace of Data Plane Services. Here "Data Plane Services" is used to denote the set

of services which transport, modify, store or analyze user traffic in the routing domain. Open Marketplaces enable users to select from a set of data plane services offered by multiple competing network providers so as to construct customized end-to-end paths for their applications. This is analogous to online travel marketplaces that allow users to explore travel options and book their travel.
In this work we discuss the key problems in the context of ChoiceNet:

1. Designing a semantic language for developing a common understanding between the entities which interact with the Marketplace (Chapter 3).

2. Identifying and ordering the services to construct a composed service through the Planner (Chapter 3).

3. Automating the Planner request and response (Chapter 3).

4. Providing the user with a list of optimal ‘‘composed services” in a Marketplace consisting of services which store, modify and transport data (Chapter 3).

5. Providing the user with a list of optimal time-dependent and time-constrained ‘‘composed path services” (Chapter 4).

6. Providing the user with a optimal ‘‘composed service route” which can be modeled as a shortest tour problem (Chapter 5).

## 1.1 Thesis Organization

Chapter 2 provides an overview of ChoiceNet and describes the main components, the semantic language, the economy and use plane protocol, which are essential for realizing the principles of ChoiceNet. Chapter 3 presents a complete prototype, building on the pieces introduced in Chapter 2 and describes in detail a Planner which constructs $K$ composed network services. Chapter 4 and Chapter 5 describe two complementary Planners ‘‘Enhanced Path Planner” and ‘‘Service Routing Planner”. Finally, Chapter 6 summarizes our contribution and discusses future work.

# Chapter 2

# Introduction to ChoiceNet: An Open Marketplace Realization

## 2.1 Overview

We provide an overview on the ‘‘ChoiceNet” architecture in the following sections.

### 2.1.1 Motivation

The ability of users to create and deploy a wide range of devices, applications, and services at the edge of the Internet has created a vibrant environment for innovation at the application layer. Such a model is not currently supported in the network core, and it has been argued in [1,3,4] that the lack of mechanisms to enable market forces to play out within the network act as an impediment for the innovation. To overcome this limitation, as part of the ChoiceNet project [1], we have proposed supporting choice as a design principle for enabling sustained innovation in the core of the network. Choice implies that users can choose from alternative services that can be deployed dynamically into the network; in this context, we use the term service as a general term that denotes any functionality that can be realized within the network. In ChoiceNet, choices are exposed through a new economy plane that complements the well-known data and control planes. Whereas business relationships between network service providers and users currently take place out-of-band, off-line, and over long time scales, the economy plane facilitates the establishment of such relationships in-band, in real time, and for short time scales. The marketplace is a key component of the economy plane that automates the process of offering (advertising) and selecting services, and the establishment of contracts for the purchase and use of services.

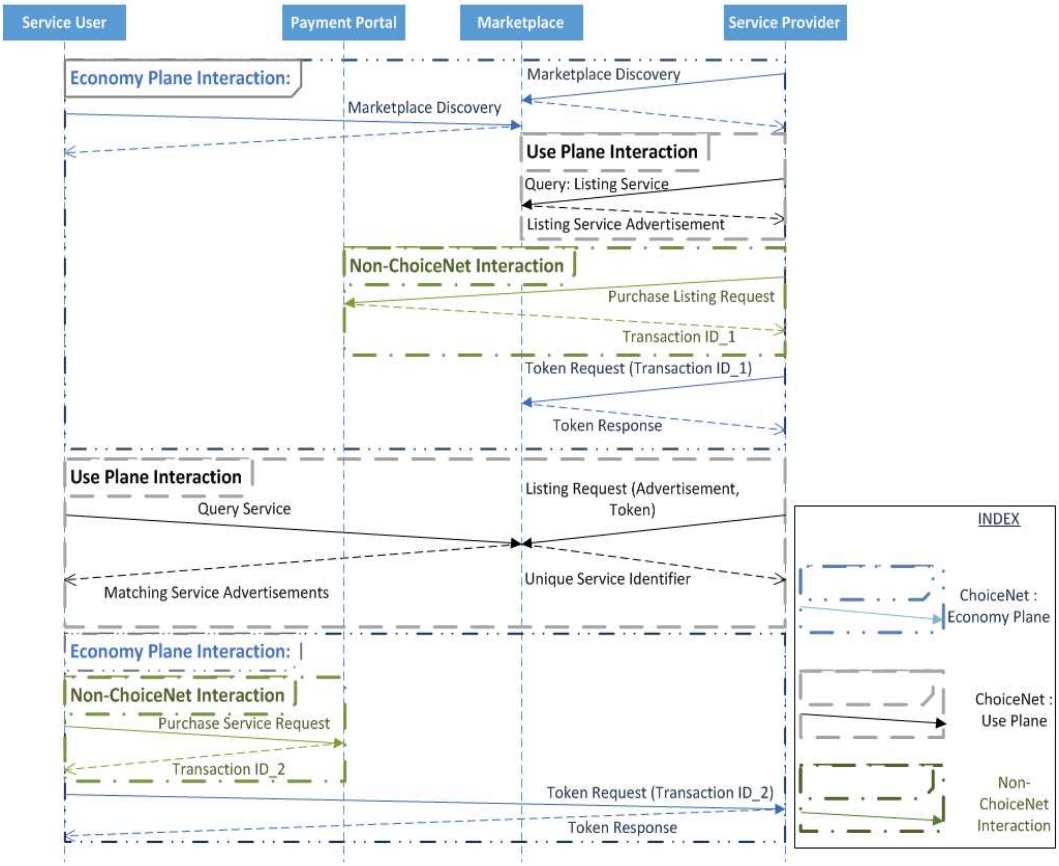


Figure 2.1: Economy Plane and Use Plane Interaction

### 2.1.2 Bootstrapping

For the Marketplace, the service providers and the users to function in ChoiceNet they first need to discover each other and be able to establish an economic contract. All this needs to happen over the Economy Plane which uses the routing services connecting these entities. The natural question is who pays for these routing services in the Economy Plane. There are several ways of addressing this bootstrapping problem and some of them are mentioned below:

1. We assume the Economy Plane continues to use the current existing Internet infrastructure with no strict guarantees on the performance.

2. The Marketplace pays for the routing services to and from itself in the hope of offsetting this cost by the revenue it generates from the services sold through it. To take the Netflix analogy, Netflix pays for the routing services for providing streaming service to the users.

3. The routing services allow traffic to and from the Marketplace(s) where they have a stake

free of cost, in the hope of reaping benefit by having their services purchased by the users transiting over their network.

### 2.1.3 Economy Plane and Use Plane Entities

Three main entities participate in the economy plane and use plane of ChoiceNet: the *marketplace*, *service providers*, and *service users*. The ChoiceNet entities and the typical interactions between them are described below in the context of the Figure 2.1.

A *marketplace* provides the framework in which network services are advertised and queried. Service providers purchase listing authorization with a marketplace and advertise new service offerings, while service users may query the marketplace to discover available services. The marketplace entity provides templates for service advertisements, based on a common vocabulary that can be used as reference by the service providers. The templates are extensible in that service providers are allowed to extend the vocabulary to further characterize the services they offer. Our design allows for multiple distinct marketplaces to co-exist so as to promote competition and to support implementation diversity. Within a given marketplace, we assume that a service provider and advertised service are uniquely identified. A *service provider* purchases listing request in the Marketplace and advertises services within the Marketplace; we consider such advertisements as claims made by the service providers about the services they provide. We assume that each service advertisement is assigned a unique service ID by the marketplace. The marketplace stores service advertisements in a service repository, and allows providers to withdraw existing advertisements using the service ID assigned earlier. A service provider may modify an existing service only by withdrawing the existing service and advertising it as a new service that is assigned a new ID; this approach avoids ambiguity and ensures a valid association between an economic contract and a given service. A *service user* interacts with the marketplace to discover services and negotiate contracts with service providers; in this context, the term user is broadly defined to include either a human or an agent acting on a humans behalf. To discover services, a user queries the marketplace by specifying a filter. The marketplace applies the user-specified filter against the list of services in its repository, and returns the service advertisements that match. The user selects one of the services in the list received from the marketplace, and contacts the service provider to negotiate an economic contract for using the service. We use the concept of tokens to access and authenticate the use of service. A token is a form of authorization to the service user for using the service, and is issued by the provider of the service. Figure 2.1 shows the entities discussed above and a typical sequence of messages involved in finding the marketplace, advertising/listing services, querying the marketplace, purchasing a service by making the payment at the portal, and using the transaction id to receive the authorization token for using the service. We classify all the

interactions leading to and signing of the economic contract as being part of the economy plane.

### 2.1.4 Service Composition Requirement

The need for service composition stems from the fact that it is not possible to advertise all theoretical possible service combinations in the Marketplace in the expectation of it being required by one of the users. Service composition provides the ability to assemble selected services in custom combinations to satisfy specific user requirements from the existing services offered in the Marketplace. Further, ‘‘Service composition" can be offered as a service by the Marketplace allowing it to be discovered, purchased and used by the service users. To use an airline analogy, a composition service provider is akin to a travel website that acts as an intermediary between passengers and airlines.

## 2.2 Related Work

### 2.2.1 Marketplace(s) for Network Services

Routing algorithms are at the core of network design and operation, and their functionality has evolved over the last sixty years from finding single shortest paths [5] to encompassing a wide range of considerations, including multiple paths [6], quality-of-service (QoS) constraints [7], and various modes of communication beyond point-to-point [8]. Nevertheless, for the most part, these routing algorithms have been designed for use by network providers/operators who have complete control over all aspects of the network. Users of the network typically have no visibility into the network topology or access to the routing function, and their traffic usually follows paths assigned by the network provider - although, using service level agreements (SLAs) they may request paths that satisfy certain properties.

Due to the evolving nature of network applications, requirements of routing functionality are also likely to evolve over time. However, at a time when network customers demand more flexibility in path selection, changes in routing-level components in the Internet require broad consensus among a diverse set of stakeholders and, hence, are increasingly difficult to implement. Accordingly, there has been some work in providing users with options over the routing path [9--11] in a manner that separates the data plane (the paths that packets follow) from the control plane (routing decisions) and allows the two to evolve separately.

A natural next step in realizing ‘‘routing-as-a-service’’ (RaaS) is the creation of open marketplaces of path services that will enable customers to select from a set of path services offered by multiple competing network providers, and *stitch* them together to construct customized end-to-end paths for their applications. This is analogous to online travel marketplaces, including Travelocity, Orbitz, and Expedia, among others, that allow users to explore travel options,

make plans, and book their travel. Similarly we can extend the concept of "routing" to other data plane services.

In the virtual domain economic contracts play an important role in defining the quantity (time) of resource (service) which has been mutually agreed upon by the seller and the buyer. Mechanisms for contracts to be signed either implicitly or explicitly, and for proof to be obtained regarding resource ownership have been described in [12] [13] [14] [15] These systems have limited scope in terms of the type of resources that can be auctioned, claimed or leased; in particular, these frameworks have been designed for a specific resource like online ads, peer-to-peer storage or computing resources.

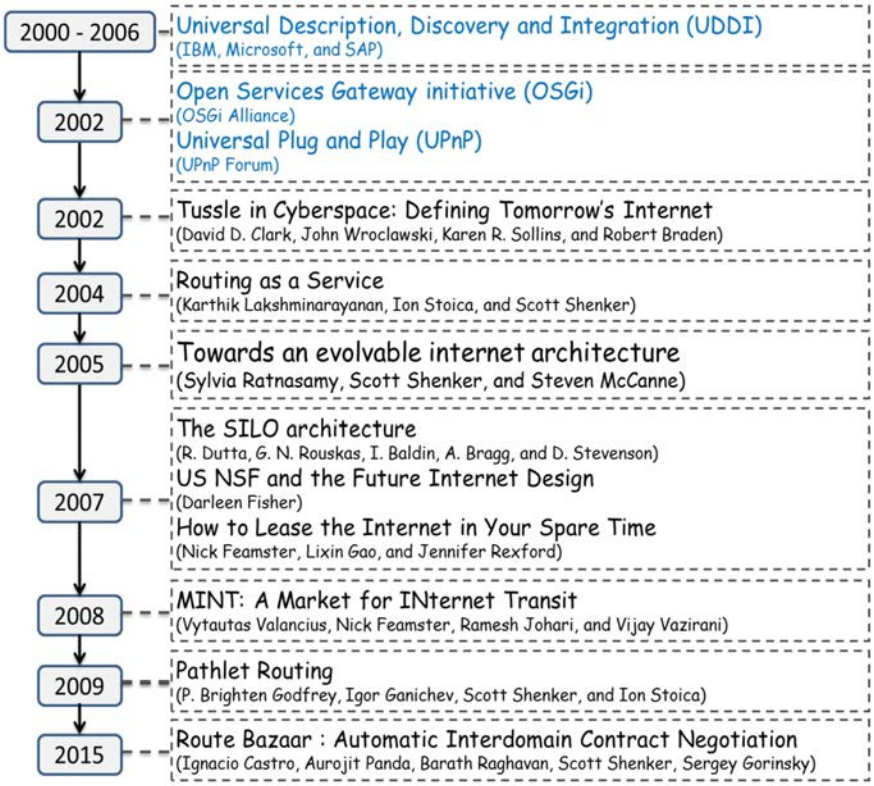


Figure 2.2: Timelines for work on Marketplace for Network/Web Services

#### 2.2.1.1 Semantic Web Services

The Semantic Web services [16, 17] have a well defined language and protocol. ChoiceNet uses some of the design principles of the Semantic Web Service in defining the language and Protocol. The key components which are essential in realizing the Semantic Web and how they have helped in shaping the ChoiceNet language and protocol is summarized below:

**Web Services Description Language (WSDL)** enables web service definitions to be exposed to the world. For a sender and receiver to have an interaction they need to share the common WSDL file. WSDL has a clear separation of data types, operations and service bindings making it modular. The service advertisements in ChoiceNet in principle play the role of WSDL files.

**Simple Object Access Protocol (SOAP)** defines a common format for XML messages over HTTP and other transport protocols. The sender and receiver exchange SOAP messages conforming to the WSDL document published by the sender and shared by the receiver(s). SOAP, a one-way asynchronous message passing mechanism, with its lightweight form, allows applications to build on top of this. The ChoiceNet Use Plane Protocol uses some of the concepts in SOAP.

**Resource Description Framework (RDF)** is a standard for encoding meta-data. It is described as statements and each statement comprises of subject, predicate, object in this order. There are several RDF serialization formats like RDF/XML, Notation-3, Turtle and N-Triples which allows machines to interpret the resource information. RDF has influenced in how we define the various services (resources) in ChoiceNet so it can be interpreted by machines with minimal effort.

**RDF Schema (RDFS) and Web Ontology Language (OWL)** help in creating a vocabulary which describes the ontology used in writing RDF documents. The vocabulary brings uniformity among service advertisements and helps the user in comparing and selection.

**SPARQL Protocol and RDF Query Language** To complete the picture of the semantic web we need one last piece, SPARQL, a query language specification that allows the RDF triples to be constructed, queried and updated. This piece is abstracted in the Use Plane Protocol in ChoiceNet to allow any database to be used at the Marketplace.

**Universal Description Discovery and Integration (UDDI)** is a public service entity which is hosted by a select group of companies. These companies are responsible for maintaining the database containing information on business registry data. SOAP is used for publishing and querying for information. When this document was being written, UDDI was being offered free for basic service. Business data can be registered with one of the vendors and it is the responsibility of the UDDI to replicate this information across other vendors. UDDI provides separate WSDL files for registration and discovery services. The ChoiceNet Marketplace enables

formation of contracts which is lacking in UDDI because of the trust deficient model. ChoiceNet Marketplace will likely draw comparisons with the UDDI since the underlying principle of both remain the same.

**Universal Plug and Play (UPnP)** offers a real time picture of the service and its state. UPnP [18] allows a device to advertise its services to the control points in the network using the Simple Service Discovery Protocol (SSDP). Similarly when a new control point is added to the network, SSDP allows device discovery. Notification events about the changes in the service is carried to the control points using General Notification Event Architecture (GENA). Every service maintains three URLs that provide the information necessary for control points to communicate with services.

- The ControlURL for control points to post requests to control this service.
- The EventSubURL for control points to post requests to subscribe to events.
- The DescriptionURL tells control points the location from which they can retrieve the service description document.

Some of the concepts from UPnP can be applied to the service composition algorithm when we want a more robust control over the services being advertised in the market place. This allows the market place to provide a real time update on the state of the services being offered.

**Open Services Gateway Initiative (OSGi)** architecture provides a flexible model for maintaining services. The basic building blocks of OSGi framework are bundles and the service registry. Bundles provide well defined services [19] and they publish the services in the service registry. The services published in the registry can be found by other bundles who want to use this service. OSGi allows for dynamic addition, update and removal of bundles. The simple yet flexible model offered by OSGi is what makes it interesting and these concepts are part of the ChoiceNet framework.

### 2.2.2 Service Composition

#### 2.2.2.1 Web Service Composition

An overview of recent research efforts in automatic Web service composition is provided in [20]. Most of the approaches can be classified as work flow or AI planning. The service providers propose Web services for use. The service requesters consume services on offer. The translation engine translates between the external languages used by the participants and the internal languages used by the planner. For each request, the planner tries to generate a plan that composes the available services in the service repository to fulfill the request. If more than one plan is found, the evaluation engine evaluates all plans and gives the result to the service

user. The composed service recipe is given to the orchestration engine which is responsible for identifying the individual service which is part of the recipe. As part of orchestration the interaction between various services is formalized. The orchestration engine then interacts with the instantiation engine for provisioning the service.
A composite service is similar to a work flow as this includes a set of services with the control and data flow among the services keeping them together. Similarly, a work flow has to specify the flow of work among these services. Two approaches in the work flow model are summarized below:

- *Static work flow :* A set of tasks and their interdependencies are abstracted into a process model at configuration time. But the binding of these services is done dynamically.
- *Dynamic work flow :* The creation of the abstract process model and selection of the web services are done automatically.

The composition problem can also be dealt via AI planning. The AI planning methods can be broadly divided into five categories, namely, the situation calculus, the Planning Domain Definition Language (PDDL), rule based planning, the theorem proving and others.

In [21], a greedy algorithm is used to solve the web service composition problem. Based on the set of input data they determine the services which can be invoked. These services can be invoked one after the other or in parallel. They keep linking services based on the output produced of the already invoked services. They form directed acyclic graphs with multiple root nodes and terminating with the leaf node producing one or more of the desired output parameters or the leaf node leading to a result which is infeasible corresponding to the query. The key idea is to select the service with the best accumulated QoS. A priority queue is defined to record all the satisfied services. A service becomes satisfied only when all of its inputs are satisfied. The priority of a service is determined by its accumulated QoS. A smaller accumulated response time gives a higher priority. A larger accumulated throughput also gives a higher priority. A trace back function is used at the end of the main process to generate Business Process Execution Language (BPEL) format solution.

#### 2.2.2.2 Semantic Matching Model

A framework for performing dynamic service composition by exploiting the semantic match-making between service parameters (i.e., outputs and inputs) to enable their interconnection and interaction is introduced in [22]. They focus on a framework for service composition based on functional aspects, in which services are chained according to their functional description i.e., inputs, outputs, preconditions and effects (IOPEs). The suggested framework uses the Causal

Link Matrix (CLM) formalism [23] in order to facilitate the computation of the final service composition as a semantic graph. They have also developed a composition algorithm that follows a semantic graph-based approach, in which a graph represents service compositions and the nodes of this graph represent semantic connections between services. Moreover, functional and non-functional properties of services are considered, to enable the computation of relevant and most suitable service compositions for some service request. Once this is done a trace is done starting from the output parameters and stopping at the input parameters.

#### 2.2.2.3 Services Integration controL and Optimization (SILO)

In [24] the application requests SILO Management Agent (SMA) for a composed service. SMA in turn contacts the SILO Construction Agent (SCA) to compose the service. The SCA composes the service if possible and it contacts the SMA with the composed service. The SMA contacts the application with the result of the service composition. The job of the SCA is to map the service name supplied by the application to match with the service names maintained as part of SILOs Resource Description Framework file and compose the service within the boundary of the constraints. The job of the SMA is to make sure these services are loadable. In this approach a dictionary is created using the semantic web notation of RDF and RDFS. All the required Services are found by querying the SPARQL Helper. The mandatory opening and closing services are identified and inserted in the ordered list of services. As part of the First Phase of the algorithm for every service in the required services list a constrained insertion is done as part of the tentative service recipe. Constrained insert involves both strict and loose constraints. As part of the Second Phase of the algorithm the loose constraints ordering is verified. As part of the work being presented in this paper we propose to make this composition algorithm domain independent.

## 2.3 Contribution

We briefly describe the solutions being put forth to solve the problems mentioned in Chapter 1 in the following sections.

### 2.3.1 Semantics Language

To know if a service can be combined with another service we first need to describe the service with sufficient information for a user of this service to understand, compare and combine this with other service(s). We define a semantics language which allows for various Data Plane services to be advertised, queried and composed. By defining a semantics language we move a step closer in standardizing the syntax and the semantics of the services being advertised in

the Marketplace. The language provides uniformity which allows for competing services from different network providers to be compared. The language has also provisions for describing the scope/boundary of the service which allows for clear separation of responsibility allowing for service composition.
The language also allows the use of logical operators to define a service with all the features. This is advantageous to not just the Marketplace, which has to deal with fewer number of services but also to the service provider, which is now not required to split the original service to create multiple smaller services to express the various features of the original service. A detailed explanation on the design of a semantics language is presented in Chapter 2.

### 2.3.2 Economy Plane and Use Plane Protocol

For the sustenance of an Open Marketplace two things are essential. First, the services advertised in the Marketplace needs to be purchased generating revenue for the Marketplace. Second, the Marketplace should drive more traffic into it in the hope of making it attractive for Service Providers and Users. The messages which are exchanged between the various entities participating in the Marketplace over physical/virtual network where consideration/money changes hands is classified as being part of the Economy Plane. The messages which are exchanged post service purchase is classified as being part of the Use Plane. We associate the Planner with providing a composition service which needs to be purchased by the user before using the Planner. Its also possible that the composition service can be offered free of cost at the Marketplace to drive the traffic and indirectly increase the revenue. In this work we describe the interaction between the Planner and the various entities in the context of Use Plane. Defining the Use Plane Protocol for an Open Marketplace is essential in automating the Planner request and response. A detailed explanation on the economy plane interaction is presented in Chapter 3.

### 2.3.3 Planner: Choice

One of the main requirements of Service Composition is providing options for the User to choose from. The input to service composition is the user request and the list of service advertisements in the Marketplace at the time of the user request. The output from service composition is the list of ‘‘composed services” sorted in non-decreasing order of the accumulated cost of the ‘‘composed service”. We have developed three Planners which are complementary and are designed for three different applications of network services. The three planners are presented in Chapters 3, 4,and 5.

# Chapter 3

# ChoiceNet Prototype

In this section, we describe a ChoiceNet Prototype from the perspective of the Planner

## 3.1 ChoiceNet Semantics Language (CSL)

CSL helps represent a service definition, a service advertisement and a service requirement which are essential pieces in the working of the Planner. CSL helps define an extensible schema/vocabulary for building a consensus between the entities interacting with the Marketplace. This schema/vocabulary may be managed by a regulated and widely accepted authority such as the Internet Assigned Numbers Authority (IANA), that enforces the *vocabulary's* syntax and semantics. The attributes used in the description of a service definition or a service advertisement or a service requirement and later in the economy plane and use plane interactions are fully specified using the triple: (attribute name, attribute value, *vocabulary* location), the attribute name and value are defined in the context of the *vocabulary* whose location is specified in the last part of the triple. The service advertisement and the service requirement are illustrated in Figure 3.1.

### 3.1.1 Layer Abstraction

Since we are dealing with network services it becomes important to state the layer at which a particular service is being provided. We use a layering abstraction which is realized using the address type and format type fields of the service advertisement. In the framework we classify all the path services as being realized at layer 2 of the TCP/IP protocol architecture and all the other services as being realized at layer 2 or above of the TCP/IP protocol architecture. The layering abstraction enables us to extend this framework to realize services at layer 1 of the TCP/IP protocol architecture.

### 3.1.2 Addressing Schema

The addressing schema should support subnetting if the goal is for a wider adoption of this Marketplace by the data plane services. We use IPv4 as the addressing schema in our framework but this can be extended to any other addressing schema which supports subnetting. While interpreting a service advertisement if the ‘‘from” and ‘‘to” address are different and the format fields are identical, we interpret that this service is a ‘‘path service”. If the address fields are identical and the format fields are different, or if both the address fields and the format fields are different then we interpret that this service is not a ‘‘path service”.
The interpretation is similar in the case of a service requirement and also in the case of a ‘‘composed service”.

### 3.1.3 Format Schema

The format schema is used to specify the functionality of the data plane service with respect to how it treats the user data. To define pure transit services which are responsible for routing data, we support wild card formats to represent them. While interpreting a service advertisement if the ‘‘from” and ‘‘to” formats are different we interpret that this service either modifies/stores/analyzes the data, if they are identical then we interpret that this may be a path service. If the ‘‘from” and ‘‘to” formats have wild card formats then we are dealing with a routing service which transports any data, else we are dealing with a routing service which transports data selectively. In our framework we use formats to refer to the data at the application layer of the TCP/IP architecture.
The interpretation is similar in the case of a service requirement and also in the case of a ‘‘composed service”.

### 3.1.4 Logical Operators

We support the Logical ‘‘OR” operator in all the address and format fields. The ‘‘OR” operator in the service advertisement indicates that all possible combinations of the operands make up this service advertisement. The ‘‘OR” operator in the service requirement indicates that any possible combination of the operands in the ‘‘composed service” satisfies the user request. We plan to support the Logical ”AND” operator in our future work and we will dicuss the challenges and the flexibility it offers in Chapter 6.

### 3.1.5 Consideration

To allow multiple ways of being compensated for the service, we provide a way to specify the consideration type and the amount of consideration in the service advertisement. The

consideration type and the value in the service requirement indicates the cost the user is willing to pay for a composed service. We assume all the services which are part of the ‘‘composed service” have the same consideration type and the value is the accumulated sum of the values of the services which make up the ‘‘composed service”. We can extend the framework to have services with different consideration type being part of the ‘‘composed service” but we would need a service which can convert values from different consideration types i.e., some kind of a consideration exchange service.

### 3.1.6 $k$-composed service

The user request has the option of specifying the number of ‘‘composed services” which should be returned in nondecreasing order of the accumulated cost of the ‘‘composed services” which is below the threshold consideration value specified in the user request. The other fields in the Service advertisement including the Service Name, Service Description, Provisioning Detail, and Purchase Portal do not influence the service composition and hence are not relevant to the discussion.

## 3.2 ChoiceNet Economy Plane and Use Plane Protocol

### 3.2.1 Message Syntax and ChoiceNet Port

We use CSL to define the message syntax between the various entities in ChoiceNet. These messages are exchanged over the standard ChoiceNet TCP/UDP Port unless stated otherwise. Some of the messages which are relevant from the perspective of the planner include:

- *Planner Listing Request* is sent from the Planner to the Marketplace to advertise the Planner service in the Marketplace. Generally this needs to be done post purchasing the slot for advertisement in the Marketplace. For the ChoiceNet framework we assume that the Planner service is provided by the entity which manages the Marketplace.
- *Query Request* is sent by the user to the Marketplace to search for services matching one or more fields in the service advertisement. In response the Marketplace sends a list of matching service advertisements.
- *Planner Request* is sent by the user to the Planner found from the Marketplace. In response the Planner responds back with a list of ‘‘composed services”

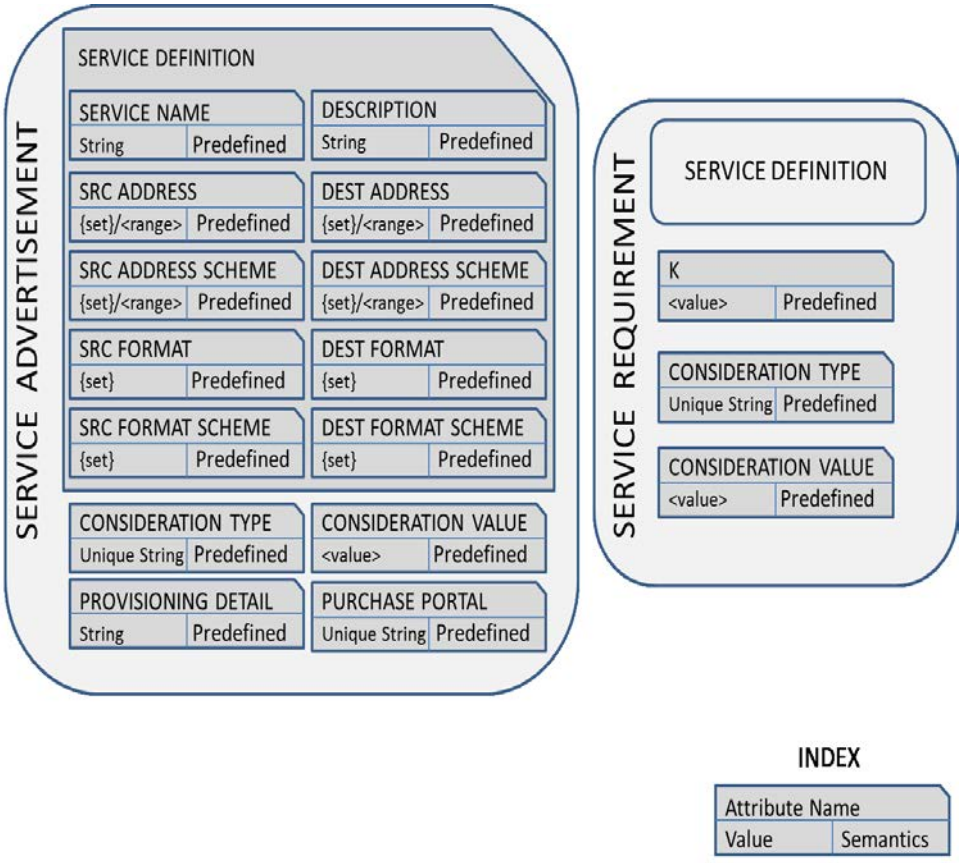


Figure 3.1: Service Advertisement and Requirement Schema

### 3.2.2 User, Planner, and Marketplace Interaction

For the user to start using the planner, the user needs to first discover the planner in the Marketplace. If the Planner is part of the Marketplace then the Planner request can be sent to the Marketplace directly else the user needs to search for the Planning service(s) and choose one of them. The user can search for the Planner either based on the service description or based on the source and destination format. As part of the service advertisement the Planner needs to state the source format as being the requirement schema and the destination format as being the list of ‘‘composed services”. This can be done using CSL and be made part of the schema/vocabulary.

Once the Planner is found the user can send the request in the requirement schema illustrated in Figure 3.1. There are two approaches for the Planner to get the service from the Marketplace for constructing a list of ‘‘composed services” to satisfy the user request.

- *Pull Method:* In the first approach the Planner requests the Marketplace for advertise-

ments on a need basis and builds a partial graph locally based on the advertisements received from the Marketplace. This approach explores the advertisements in the Marketplace systematically looking for matching services which can be composed minimizing the cost.

- *Push Method:* In the second approach the Planner subscribes to the Marketplace to receive advertisements which match the subscription filter registered at the Marketplace by the Planner.

In this framework we have used the pull method and we would like to employ subscription based model in our future work. Further, all our messages are sent over UDP but this can be extended to use TCP but with a rider. Its possible to send arbitrarily large amount of data to the Marketplace or the Planner as part of the listing request or planner request respectively and overwhelm the module. To prevent Denial of Service (DOS) attack to either the Marketplace or the Planner, we tend to limit the payload size.

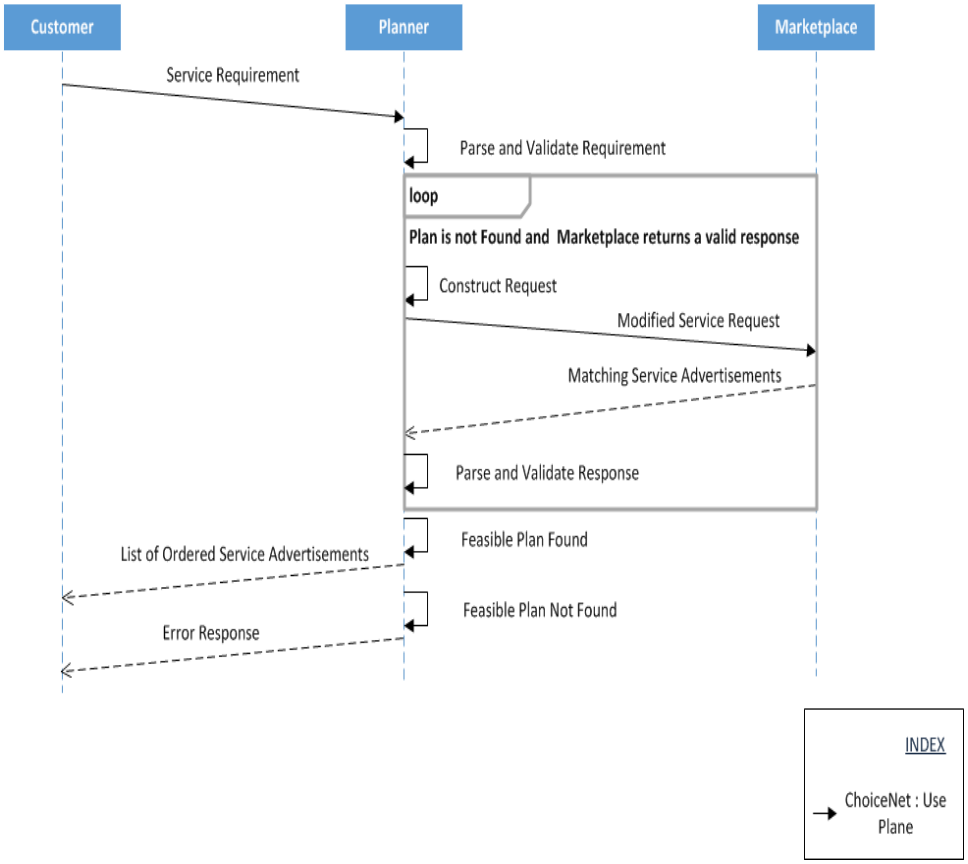


Figure 3.2: Planner Interaction

## 3.3 ChoiceNet Planner

Two flavors of the Planner have been developed, both providing Choice based on Yen's K-Shortest Path Algorithm [25]. The first Planner is part of the ChoiceNet framework while the second Planner described in Chapter 5 is a standalone module which is designed for composing and reserving path services which are time sensitive. The second Planner also has an option of finding Pareto Optimal paths based on bi-criteria constraints of cost and delay. Integrating the second Planner with the ChoiceNet framework is part of the future direction and will be discussed in Chapter 6.

The main problem involved in composing services within the ChoiceNet framework described above is identifying services which can be combined to satisfy user requirements and presenting the user with options to choose from a number of composed services(s) arranged in non-decreasing order of cost.

### 3.3.1 Input

The input to the Planner is a service request from the user which is illustrated in Figure 3.1 and the interaction between the user and the Planner is described in Figure 3.2. The request is sent from the user as a ChoiceNet message over a transport layer protocol. The Planner is listening on the ChoiceNet TCP/UDP port for ChoiceNet messages. The Planner receives the message and parses the request which is based on CSL referred to in Section 3.1

### 3.3.2 Output

The output of the Planner is a list of ‘‘composed service(s)" which is structured as shown in Figure 3.3 and is sent back to the user as a ChoiceNet message over the same transport layer protocol. The user or an agent on behalf of the user receives the message and parses the response which is based on CSL referred to in Section 3.1 and makes a Choice based on the options presented by the Planner. There are subtle differences between a Service Advertisement and the ‘‘composed service" which uses the service advertisements to construct a ‘‘composed service". *K-Composed Service*, the ‘‘$K$" stands for the number of ‘‘composed service(s)" which is returned by the Planner in non-decreasing order of the accumulated cost of each ‘‘composed service". The number of ‘‘composed service(s)" returned may be less than or equal to the number of ‘‘composed services(s)" requested by the user. Each composed service consists of the consideration type which is uniform across all the services in the ‘‘composed service". It also consists of the accumulated cost of all the services which are part of the ‘‘composed service". This is followed by the service advertisement instances arranged sequentially in the order the services need to be executed. Each service advertisement instance consists of the Service Advertisement

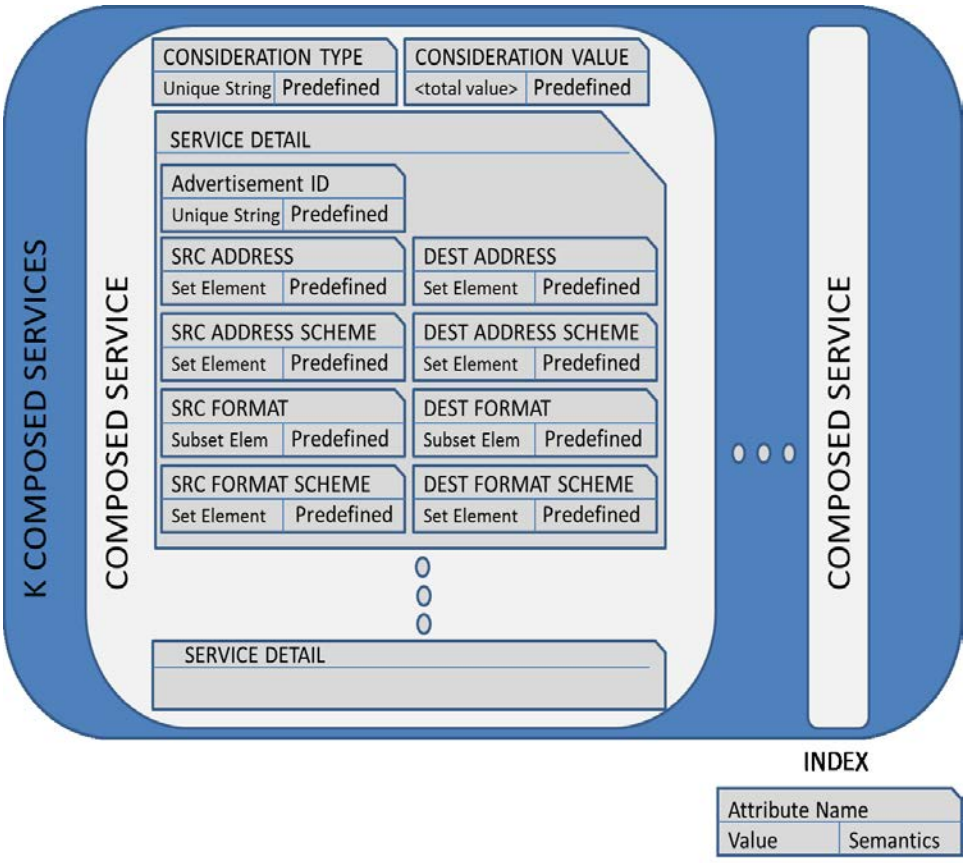


Figure 3.3: Composed Service Schema

Identifier which is used for creating the instance followed by the source and destination address information which contains one of the set elements from the original service advertisement. The source and destination format information contains one of the set elements from the original service advertisement. The format values in the instance needs to take into account wild card formats which the universal set containing all formats supported by CSL. This can be achieved by replacing the wild card formats, if present in a service advertisement with a specific format value and type being requested by the user. So, it is possible for two composed service(s) to have the same advertisement identifiers but what sets them apart is the service instance which is returned by the Planner. If the Planner cannot find a ‘‘composed service” which matches the user request it returns a empty list of ‘‘composed service(s)”.

### 3.3.3 Algorithm

The subscript and superscript notations used in Section 3.3.3.1 is represented as array indices and structure data members in Algorithm 2 and Algorithm 1.

| Notation | Definition |
|---|---|
| $N$ | The number of nodes in the graph, each node represents one set element each from the source address and source format information of a service advertisement or one set element each from the destination address and destination format information of a service advertisement. |
| $(i)$ | $i = 1, 2, ..., N$, represent the nodes where (1) is the starting node and $(N)$ is the final node |
| $d_{ij}$ | $i \neq j$, is the cost of the direct arc from $(i)$ to $(j)$ |
| $A^k$ | $A^k$ = (1)-($2^k$)-($3^k$)-...-($Q_k^k$)-$N$, $k = 1,2, ..., K$, denotes the $k^{th}$ shortest composed service from (1) to $(N)$ where ($2^k$), ($3^k$) represent the $2^{nd}$ and $3^{rd}$ node respectively and so on. |
| $A_i^k$ | Path which deviates from $A^{k-1}$ at $(i)$, $i = 1, 2, ..., Q_k$ |
| $R_i^k$ | Root path of $A_i^k$ coincides till the ($i^{th}$) node of $A^{k-1}$ |
| $S_i^k$ | Spur path of $A_i^k$ has only one node coinciding with $A^{k-1}$ |

Table 3.1: Notation and Definition

#### 3.3.3.1 Notation and Definitions

We use the notations from [25] to bring out the similarity and differences between the algorithm presented here and the original Yen's algorithm. The notation and their meaning is described in the Table 3.1

#### 3.3.3.2 Pseudocode

The algorithm assumes that a virtual source node exists which can reach all nodes which represent one set element of the source address and source format in the user request at zero cost. Similarly, there exists a virtual destination node which can be reached from all nodes which represent one set element of the dst address and dst format in the user request at zero cost. A variation of Dijkstra's algorithm [5] is used to find the $K$ shortest cost composed service(s) between this virtual source and virtual destination node and is represented using Algorithm 2 and Algorithm 1.

#### 3.3.3.3 Description

In Algorithm 1 we use a variation of Dijkstra's [5] shortest path algorithm to compute a shortest cost composed service. The input to this algorithm is the virtual source node and the virtual destination node along with the user constraints which must be satisfied. We list below the variations to the original Dijkstra's algorithm:

- In this variation we query the Marketplace repeatedly till we find a end-to-end composed service or we are certain that no such end-to-end composed service exists which satisfies

the user constraints. We assume that the Marketplace is just a repository of services and doesn't have the intelligence of inferring the functionality associated with the service advertisements. So, the intelligence of making sense of the advertisements needs to be built outside the Marketplace. We build this intelligence in the planner module and this is executed as part of the service pruning functionality in the planner. This transforms the Planner from being a simple service concatenation tool to a service composition platform.

- To make this work with Yen's algorithm we need to avoid edges which are part of the previously calculated composed service instance(s). One of the approaches for achieving this is by maintaining a open chain hash table which keeps track of all service instances associated with an advertisement which have been taken. Also, we need to avoid nodes which are part of the rootPath node. One of the approaches for achieving this is by maintaining a tree which supports the compact representation of an address which allows subnetting. Each node should also store the format instances which are used to represent these rootPath nodes.

In Algorithm 2 we use a variation of Yen's [25] $k$-shortest loopless paths algorithm to compute the $K$ shortest cost composed service(s). The input to this algorithm is the user request. We list below the variations to the original Yen's algorithm:

- In this variation we represent a node by the tuple ($Addr$, $Fmt$) and we transform the user request by constructing virtual source and virtual destination nodes.
- To avoid visiting nodes which are part of the rootNode or edges which share the same rootPath of the previously computed composed service instances we store them in a patricia tree and open chain hash table respectively.

#### 3.3.3.4 Service Pruning

Service pruning mentioned in Section 3.3.3.3 and in the Algorithm 2 refers to the feature in the Planner which is needed to discern the functionality of a service advertisement which uses the layering abstraction mentioned in Section 3.1.1 to describe the service. The Planner queries the Marketplace for services matching the address schema. The Marketplace returns all services which match the search criteria. The Planner is then tasked with sifting through these service advertisements to identify service advertisements which can be part of a composed service. If a composed path service is the end goal, the Planner considers only path service advertisements. If a composed non-path service is the end goal, the Planner considers non-path service advertisements which match the tuple ($Addr$, $Fmt$) and it considers all path service advertisements since they match the tuple ($Addr$, $Fmt$) as they support all possible formats in the Marketplace. Two tuples ($Addr_1$, $Fmt_1$) and ($Addr_2$, $Fmt_2$) are a match if an element

from the "$Addr_1$" set matches with an element from the "$Addr_2$" set and if an element from the "$Fmt_1$" set matches with an element from the "$Fmt_2$" set.
We also perform service pruning when we need to split the service advertisement in Algorithm 1. We define splitting a service advertisement as decomposing a service advertisement to represent all possible combinations of the logical operands individually i.e., if a service advertisement has two possible source address and source formats each and three possible destination address and destination formats each, then post splitting a service advertisement we will be left with 2 * 2 * 3 * 3 i.e., 36 individual service advertisements which have been derived from one advertisement. All of these individual service advertisements do not match the search filter which was used in getting the original service advertisement. We prune out all the individual service advertisements which do not match the search criteria.

#### 3.3.3.5 Backtracking

Once we find a service advertisement whose destination tuple matches with the destination tuple of the service requirement, we backtrack to construct the composed service. For every edge traversal which leads to the discovery of at least one new tuple we store the service advertisement information which represents this edge. A new tuple ($Addr_n$, $Fmt_n$) is defined as a tuple which contains at least one element from the "$Addr_n$" set or one element from the "$Fmt_n$" set which hasn't been found. We use the service advertisement information which is associated with the discovered tuples to backtrack and find the composed service.

#### 3.3.3.6 Example

We present two examples, the first example describes constructing a composed path service while the second example describes constructing a non-composed path service.

- **Round Trip**: Figure 3.4 shows a set of path and non-path service advertisements for the first example. Figure 3.5 shows the path service and non-path service advertisements represented in a network topology diagram with each node represented using the tuple (Addr, Fmt) and each edge represented using the advertisement ID. Figure 3.6 shows the input and output to and from the Planner respectively and we obtain two composed non-path services in this example.

- **Routing**: Figure 3.7 shows a set of path service advertisements for the second example. Figure 3.8 shows the path service advertisements represented in a network topology diagram with each node represented using the tuple (Addr, Fmt) and each edge represented using the advertisement ID. Figure 3.9 shows the input and output to and from the Planner respectively and we obtain eight composed path services in this example.

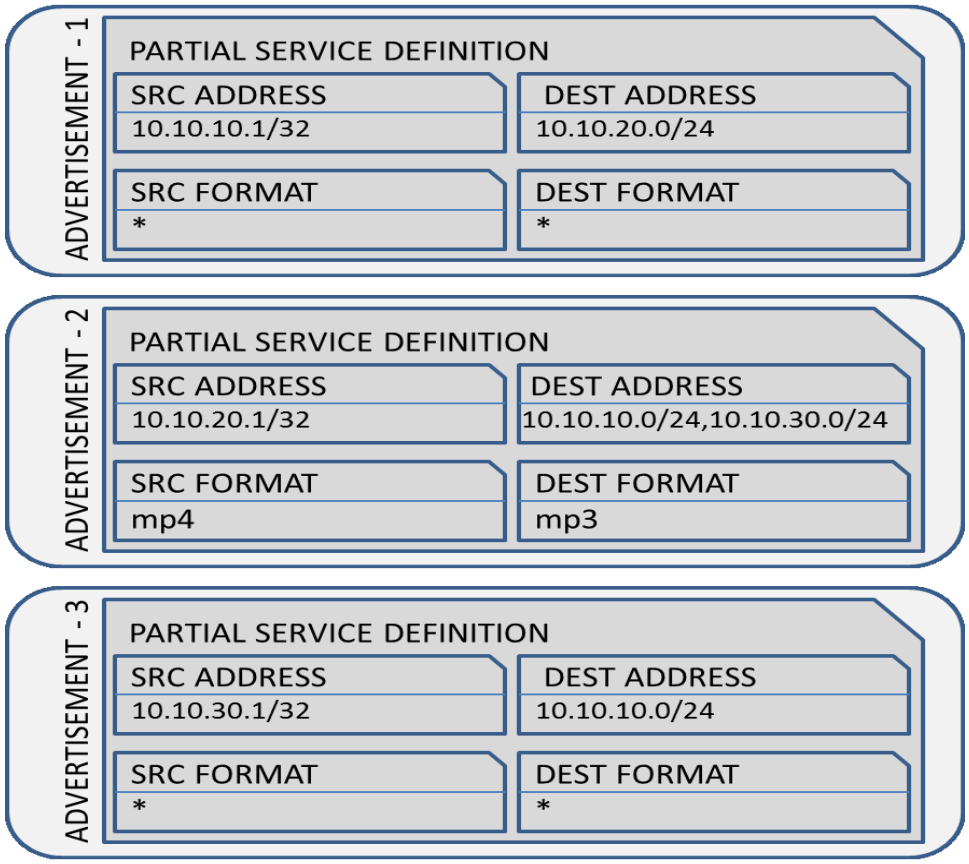


Figure 3.4: Round Trip Example: Service Advertisements

#### 3.3.3.7 Correctness

The proof of correctness of this algorithm follows from Yen's algorithm.

### 3.3.4 Complexity

The time complexity of the modified Yen's algorithm is O($KN(M+NlogN)$) which is identical to Yen's complexity where $K$ is the number of shortest cost loopless composed services, $M$ is the number of edges and $N$ is the number of nodes. In the modified algorithm, $N$ and $M$ are functions of the set of advertisements $ADV$ in the Marketplace, the number of supported formats in the Marketplace $F$, and the number of shortest loopless composed services required $K$. The set of advertisements $ADV$ are classified into $ADV\_PATH$ and $ADV\_OTHER$ which denote path and non-path service advertisements respectively. The path service advertisements include all services which can carry any user traffic without modifying the payload of the packet. The non-path service advertisements includes all services which cannot be classified as being

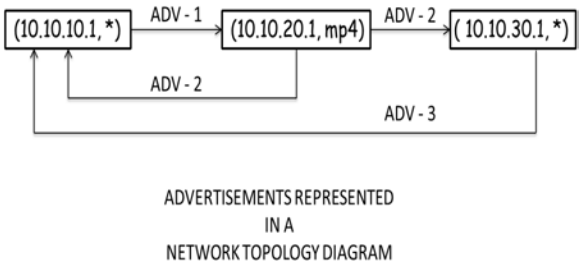


Figure 3.5: Round Trip Example: Network Topology

path services. The algorithm for finding $K$ composed path services and $K$ composed non-path services is the same. The distinction between the advertisements is made only to highlight the time complexity of the algorithm. While a composed path service is derived using only path service advertisements, a composed non-path service is derived using one or more non-path service advertisements and zero or more path service advertisements.

#### 3.3.4.1 Bounds for $K$ composed Path Services

When finding $K$ composed path services, $N$ is given by (3.1), and $M$ is given by (3.2), where the notations $ADV_{n_{k-1}}^{SRC\_ADDR}$ and $ADV_{n_{k-1}}^{DEST\_ADDR}$ are used to denote the "$SRC\_ADDR$" and "$DEST\_ADDR$" address sets (refer Figure 3.1) of the advertisement whose "$SRC\_ADDR$" address set is used for deriving the $n^{th}$ node of the $(k-1)^{th}$ composed service, $n_{k-1}$ is used to denote the $n^{th}$ node in the $(k-1)^{th}$ composed service and $N_{k-1}$ is used to denote the number of nodes in the $(k-1)^{th}$ composed service.

$$N = (2{*}ADV\_PATH){+}\sum_{k=2}^{K}\sum_{n_{k-1}=1}^{N_{k-1}}(\left|ADV\_PATH_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_PATH_{n_{k-1}}^{DEST\_ADDR}\right| - 2) \tag{3.1}$$

$$M = ADV\_PATH + \sum_{k=2}^{K}\sum_{n_{k-1}=1}^{N_{k-1}}(\left|ADV\_PATH_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_PATH_{n_{k-1}}^{DEST\_ADDR}\right| - 1) \tag{3.2}$$

#### 3.3.4.2 Analysis for $K$ composed Path Services

Since each path service advertisement consists of "$SRC\_ADDR$" and "$DEST\_ADDR$" address sets, its possible for all the advertisements to have "$SRC\_ADDR$" and "$DEST\_ADDR$"

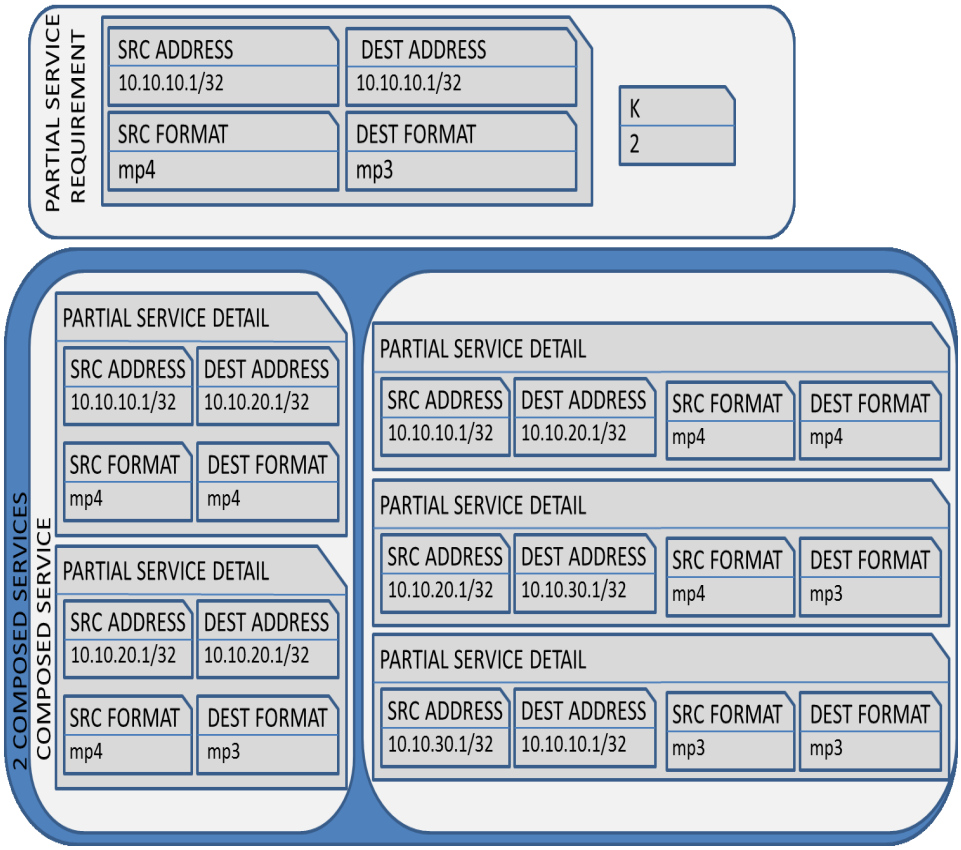


Figure 3.6: Round Trip Example: Input and Output

address sets to be different from each other, which would make a advertisement contribute two unique nodes to $N$. Although each node is represented by the tuple ($Addr$, $Fmt$) where each element in the tuple is a set, in case of Path services, the format field is considered to be a wild card. Hence, the theoretical maximum number of nodes which can be present when finding the first composed service is given by (3.1). Finding subsequent composed services with Yen's algorithm involves the previous composed service. We split the compact service notation used in the advertisement which leads to a possible increase in the number of nodes $N$. We calculate the increase in $N$, by multiplying the size of the "$SRC\_ADDR$" and "$DEST\_ADDR$" address sets of an advertisement whose "$SRC\_ADDR$" address is used for finding the previous composed service and subtracting two, the contribution of the advertisement to the number of nodes in the compact notation. We do this over all the nodes which are part of all the previous composed services to find the theoretical maximum number of nodes over $K$. Similarly its possible that each advertisement contributes an unique edge which would lead to the theoretical maximum given by (3.2). Since a non-path service advertisement isn't considered when finding

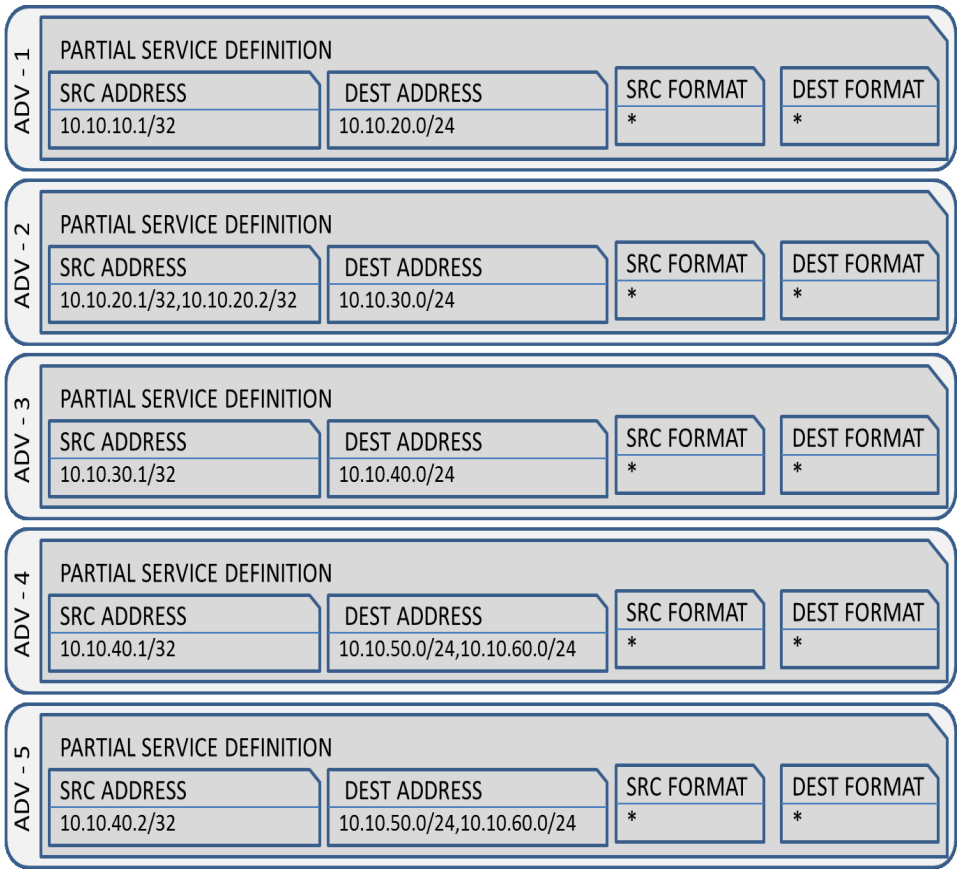


Figure 3.7: Routing Example: Service Advertisements

a composed path service, we do not factor them in.

#### 3.3.4.3 Bounds for $K$ composed Non-Path Services

When finding $K$ composed non-path services, $N$ is given by (3.3), and $M$ is given by (3.4), where the notations $ADV_{n_{k-1}}^{SRC\_ADDR}$ and $ADV_{n_{k-1}}^{DEST\_ADDR}$ are used to denote the "$SRC\_ADDR$" and "$DEST\_ADDR$" address sets, $ADV_{n_{k-1}}^{SRC\_FMT}$ and $ADV_{n_{k-1}}^{DEST\_FMT}$ are used to denote the "$SRC\_FMT$" and "$DEST\_FMT$" format sets (refer Figure 3.1) of the advertisement whose "$SRC\_ADDR$" and "$SRC\_FMT$" sets are used for deriving the $n^{th}$ node of the $(k-1)^{th}$ composed service, $n_{k-1}$ is used to denote the $n^{th}$ node in the $(k-1)^{th}$ composed service and $N_{k-1}$ is used to denote the number of nodes in the $(k-1)^{th}$ composed service.

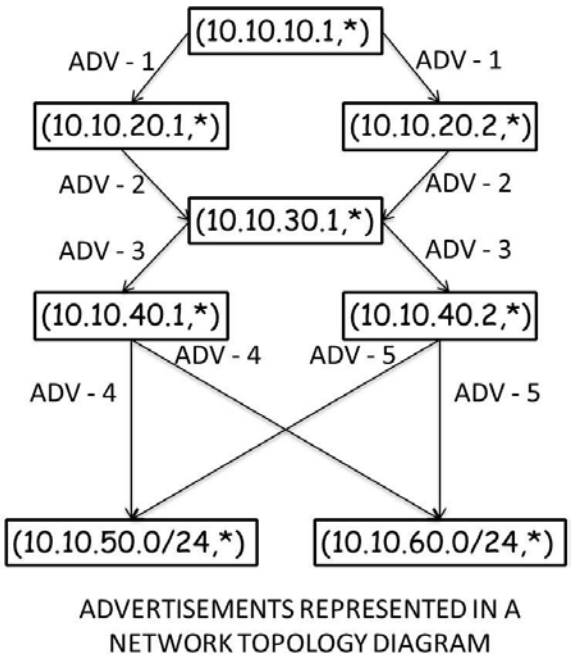


Figure 3.8: Routing Example: Network Topology

$$N = (2 * ADV\_PATH * F) + \sum_{k=2}^{K} \sum_{n_{k-1}=1}^{N_{k-1}} (\left|ADV\_PATH_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_PATH_{n_{k-1}}^{DEST\_ADDR}\right| - 2)$$
$$+ (2 * ADV\_OTHER) + \sum_{k=2}^{K} \sum_{n_{k-1}=1}^{N_{k-1}} (\left|ADV\_OTHER_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_OTHER_{n_{k-1}}^{DEST\_ADDR}\right|$$
$$* \left|ADV\_OTHER_{n_{k-1}}^{SRC\_FMT}\right| * \left|ADV\_OTHER_{n_{k-1}}^{DEST\_FMT}\right| - 2) \tag{3.3}$$

$$M = (ADV\_PATH * F) + \sum_{k=2}^{K} \sum_{n_{k-1}=1}^{N_{k-1}} (\left|ADV\_PATH_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_PATH_{n_{k-1}}^{DEST\_ADDR}\right| - 1)$$
$$+ (ADV\_OTHER) + \sum_{k=2}^{K} \sum_{n_{k-1}=1}^{N_{k-1}} (\left|ADV\_OTHER_{n_{k-1}}^{SRC\_ADDR}\right| * \left|ADV\_OTHER_{n_{k-1}}^{DEST\_ADDR}\right|$$
$$* \left|ADV\_OTHER_{n_{k-1}}^{SRC\_FMT}\right| * \left|ADV\_OTHER_{n_{k-1}}^{DEST\_FMT}\right| - 1) \tag{3.4}$$

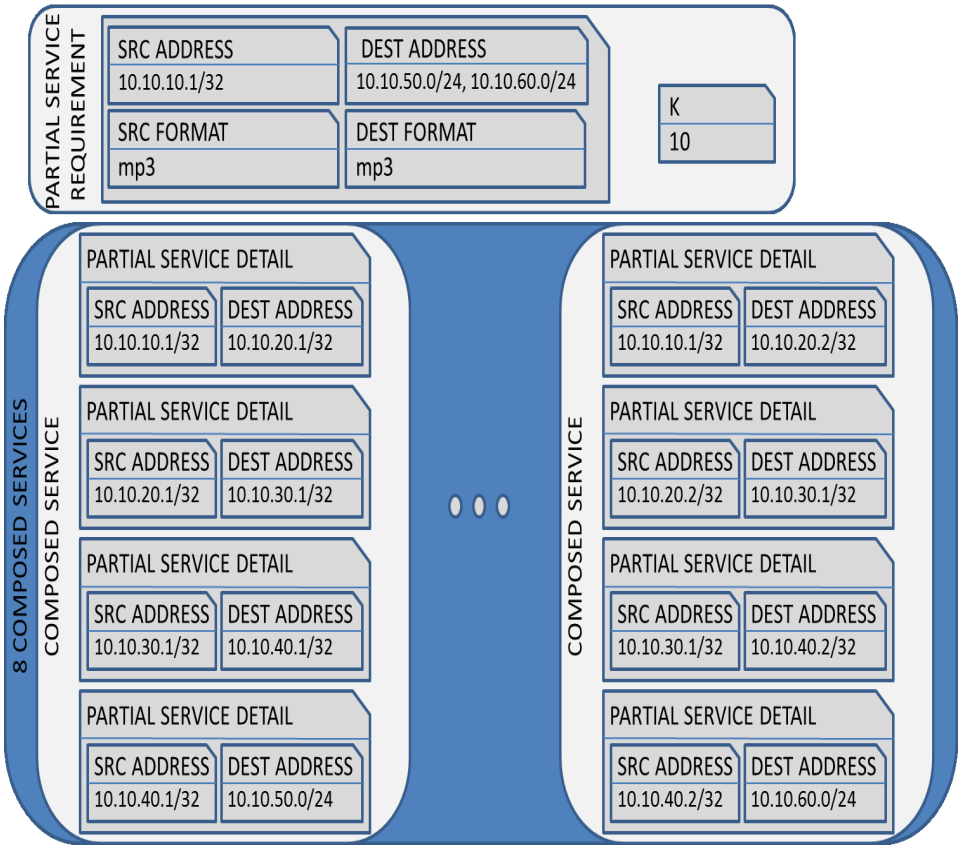


Figure 3.9: Routing Example: Input and Output

#### 3.3.4.4 Analysis for $K$ composed Non-Path Services

In the analysis for composed non-path services, a path service advertisement's possible contribution of two unique Address sets needs to be combined with all supported formats in the Marketplace to obtain the theoretical maximum number of nodes which can be present when finding the first composed service when considering only path service advertisements. The non-path service advertisement contributes two unique nodes to N, the two unique nodes are represented by the tuple (($SRC\_ADDR$), ($SRC\_FMT$)) and (($DEST\_ADDR$), ($DEST\_FMT$)). For finding subsequent composed services we split the compact service notation used in the advertisement which leads to a possible increase in the number of nodes N. For path service advertisement, we calculate the increase in $N$, by multiplying the size of the "$SRC\_ADDR$" and "$DEST\_ADDR$" address sets of a path advertisement whose "$SRC\_ADDR$" address is used for finding the previous composed service and subtracting two, the contribution of the advertisement to the number of nodes in the compact notation. We do this over all the nodes which are part of all the previous composed services to find the theoretical maximum number of nodes

over $K$. For non-path service advertisement, we calculate the increase in $N$, by multiplying the size of the "$SRC\_ADDR$", "$SRC\_FMT$", "$DEST\_FMT$" and "$DEST\_ADDR$" address sets of a path advertisement whose "$SRC\_ADDR$" and "$SRC\_FMT$" sets are used for finding the previous composed service and subtracting two, the contribution of the advertisement to the number of nodes in the compact notation. We do this over all the nodes which are part of all the previous composed services to find the theoretical maximum number of nodes over $K$. We do a similar analysis while finding the theoretical maximum number of edges $M$ given by (3.4).

```
Data: virSrc, virDst, User Constraints
Result: Shortest cost Composed Service Instance satisfying user constraints
price_from_src = 0
while (!find_in_explored_patricia_tree(virDst)) do
    advertisements = searchMarketplace(virSrc.Addr)
    forall the (service ∈ advertisements) do
        /*
         * Infer the functionality based on the format values of
         * advertisements returned by the Marketplace and prune
         * the service advertisements based on the search filter
         */
        prune(service)
        if
        (check_in_hash(service)||find_in_avoid_patricia_tree(service.srcAddr, service.srcFmt))
        then
            Split Service advertisement
            forall the (split_service ∈ service) do
                if split_service.srcAddr == virSrc.Addr then
                    insert_in_dij_minHeap(split_service, price_from_src + split_service.cost)
                end
            end
        else
            insert_in_dij_minHeap(service, price_from_src + service.cost)
        end
    end
    while (1) do
        if (newService = heap_del_dij_minHeap()) then
            if
            (!find_in_explored_patricia_tree(newService.dstAddr, newService.dstFmt))
            then
                insert_in_explored_patricia_tree(newService.dstAddr, newService.dstFmt))
                virSrc = newService.dst
            end
        else
            break
        end
    end
end
```

**Algorithm 1:** Modified Dijkstra's Algorithm

```
Data: User Request
Result: Upto K Shortest cost Composed Service Instances
/*Store shortest cost composed service instance from virSrc to virDst*/
A[0] = Dijkstra_variation(virSrc, virDst, userConstraint)
for k ∈ [1, K] do
    for i ∈ [0, (|A[k − 1]| − 1)] do
        A[k-1].explored_patricia_tree = new patricia_tree
        A[k-1].avoid_patricia_tree = new patricia_tree
        hash_table = new open_chain_hash_table
        spurNode = A[k-1].node(i)
        rootPath = NULL
        for j ∈ [0, i] do
            rootPath = rootPath + A[k-1].node(j)
        end
        for p ∈ [0, (k − 1)] do
            for j ∈ [0, i] do
                if rootPath.node(j)! = A[p].node(j) then
                    match = 0
                    break
                else
                    match = 1
                end
            end
            if match == 1 then
                insert_in_hash(A[p].node(i).advertisement_id, A[p].edge(i, i+1))
            end
        end
        for j ∈ [0, (i − 1)] do
            insert_in_avoid_patricia_tree(A[p].node(i).srcAddr, A[p].node(i).srcFmt))
        end
        if rootPath == NULL then
            spurPath = Dijkstra_variation(virSrc, virDst, userConstraint)
        else
            virSrc = spurNode
            spurPath = Dijkstra_variation(virSrc, virDst, userConstraint)
        end
        totalPath = rootPath + spurPath
        insert_in_yens_minHeap(total_path, total_path.cost)
        free patricia tree(s) and hash table
    end
    while (tentative_path = heap_del_yens_minHeap()) do
        if tentative_path is not a duplicate path then
            insert_in_hash(A[p].node(i).advertisement_id, A[p].edge(i, i+1))
            break
        end
    end
end
```

**Algorithm 2:** Modified Yen's Algorithm

# Chapter 4

# Enhanced Path Planner

## 4.1 Related Work

### 4.1.1 Shortest Path Algorithms

The planner must present the user with several options (i.e., viable alternative end-to-end solutions) that meet multiple criteria, including price, bandwidth capacity, delay, the inclusion or exclusion of sub-paths from certain providers, etc. We envision that planners will differentiate from the competition by deploying sophisticated and specialized algorithms for constructing a list of ‘‘composed services”.

To simplify the explanation, we take the example of path services. Introducing one additional resource constraint (e.g., a delay constraint along with a cost constraint), makes the shortest path problem NP-Complete [26, Problem ND30]. Consequently, a wide range of heuristics and approximation algorithms have been developed for a diverse set of constrained shortest path problem variants [27, 28]. Also, while efficient algorithms exist for constructing $k$-shortest elementary (i.e., acyclic) [25] and non-elementary [29] paths, the $k$-constrained shortest path problem is significantly harder and has received little attention [30].

Just as the planner of a travel site takes into consideration flights from multiple airlines, many of which offer competing flights between the same pairs of cities, a path planner must consider advertisements from multiple providers, including virtual operators who may lease capacity from the same physical infrastructure. Consequently, the path planner takes as input a topology that is a superset of the topologies representing the networks of individual providers, and that is likely to include parallel edges between nodes for which there exist competing path services. Such a topology is expected to be much larger than each of its constituent individual provider topologies.

### 4.1.2 In-advance service reservation

We extend the example of path services to discuss the In-advance service reservation. Planners must allow users to reserve end-to-end paths during specific continuous time intervals in the future; this feature is analogous to booking a hotel for a set of consecutive days long before travel takes place. On the other hand, support for time constraints allows users to explore additional options whenever their communication plans are flexible, in the same manner that travel planners allow users to provide a range of acceptable start and end dates for their travel. In-advance path reservations involve reserving resources along an end-to-end path for a continuous interval of time that has a specific duration and starts at a specific instant, either in the present or in the future. Algorithms for finding and reserving paths with sufficient bandwidth resources well in advance of the start of communication [31--33] have generally been designed for small, centrally controlled connection-oriented networks in which only a relatively small fraction of connections require such advance reservations. These algorithms may be extended to account for cost and delay constraints, but do not directly support time constraints.

### 4.1.3 Problem Classification

The shortest path problems and network reservation algorithms can be classified based on the objective function(s) as shown in Table 4.1 and Table 4.2 respectively. In this classification we use the notation $\alpha/\beta/\gamma/\pi/\phi$, $\alpha = \{G, T\}$ where $G$ denotes problems which are not time sensitive and $T$ denotes problems which are time sensitive, $\beta \in \mathbb{N}$ denotes the number of resource constrained objective functions, $\gamma = \{P, F\}$ where $P$ denotes pareto or non-dominated paths and $F$ denotes all feasible paths, $\pi = \{1, K\}$ which indicates if the problem finds 1 optimal path or $K$ paths in non-decreasing order of cost, $\phi = \{O, A\}$ where $O$ denotes optimal paths and $A$ denotes approximate or subset paths, to categorize problems and this classification differs from the one in [34] as we extend the classification to problems which are time driven, non-dominated and those which consider $K$ non elementary shortest paths.

Dijkstra's algorithm [5] is designed to find a single source shortest path and runs in polynomial time. The rest of the algorithms shown in Table 4.1 either run in pseudopolynomial or exponential time. The concept of non-dominated or pareto solution(s) was first defined on multiplicative lattices. It was then extended to network graph models [35--37] using the label setting/correction approach. Climaco et.al [38,39] use the $K$-shortest [40,41] paths approach to find the pareto solution(s). Joksch [42] introduced the notion of shortest paths with constraints. The shortest path problems with resource constraints was solved using Lagrangian relaxation by Handler et.al [43] and using $K$-shortest paths algorithm by Martins et.al [41]. Henig [44], Warburton [45] and Hassin [46] introducted the concept of approximate solutions to pareto and constrained shortest path problems. The concept of shortest path problem with time windows

| Notation | Original Algorithm | Variations |
|---|---|---|
| G/0/F/1/O | Dijkstra [5] | |
| G/0/P/1/O | Brown et.al [53] | Thuente [35], Hansen [36], Martins [37] |
| G/0/P/K/O | Climaco et.al [38] | Climaco et.al [39] |
| G/1/F/1/O | Joksch [42] | |
| G/1/F/K/O | Handler et.al [43], Martins et.al [41] | Ning [54] |
| G/0/P/1/A | Henig [44], Warburton [45] | |
| G/1/F/1/A | Hassin [46] | |
| T/0/F/1/O | Desrochers et.al [47], Desaulniers et.al [55] | |
| T/0/P/1/O | Hamacher et.al [48] | |
| T/0/F/K/O | Rouskas et.al [49] | |
| T/1/P/1/O | Rouskas et.al [49] | |

Table 4.1: Classification of Shortest Path Problems

| Notation | Original Algorithm | Variations |
|---|---|---|
| T/1/F/1/O | Guerin et.al [56] | Balman et.al [57] |
| T/0/F/K/O | Rouskas et.al [49] | |
| T/1/P/1/O | Roukas et.al [49] | |

Table 4.2: Classification of Network Reservation Algorithms

was introduced by Desrochers et.al [47] and later it was defined for non-dominated shortest path problems with time windows by Hamacher et.al [48] for vehicle routing problems. The concept of time windows was extended to network graph models and the concept of non-dominated and constrained shortest path problems with time windows is defined by Rouskas et.al [49]. This work connects the domain of shortest path problem with advance resource reservation in network graph models. Some of the notable network reservation algorithms without preemption are shown in Table 4.2

Aneja et.al [50], Beasley et.al [51] and Dumitrescu et.al [52] employ preprocessing to improve the running time of several constrained shortest path algorithms.

### 4.1.4 Pros and Cons of finding Pareto paths using $k$-shortest paths

There are two main challenges when finding pareto paths.

- How quickly can you find the pareto solutions
- How do you choose the optimal/sub-optimal solution among these pareto solutions

There are basically two ways of finding pareto solutions.

- Label setting/correction approach

- $K$-shortest cost paths

Since finding pareto solution(s) is NP-Hard, both the approaches take exponential running time. For certain graph problems one of the approaches might be better suited than the other. In Figure 4.1 we have three examples where the number and the gap between the pareto solutions is different. Suppose every path from source to destination has a unique pair of (cost, time) solution and each corresponds to one of the of the $K$-shortest cost paths. In that case, for all the three examples we can quickly find all the pareto solutions relatively quickly compared with the labeling approach.

The gap between the pareto solutions and the number of pareto solutions which can be found using the $K$-shortest paths plays an important role in determining which approach is the best. In Figure 4.2 we have an example which is not suitable for the $K$-shortest cost/time paths as we have multiple paths which have the same cost or time and we are not guaranteed to find the shortest cost/time path which minimized both cost and time for a small value of $K$ when we are searching for $K$-shortest cost paths or the $K$-shortest time paths. So, we have to find $K$-shortest cost/time paths for a large value of $K$ before we find pareto solutions. This applies for the bi-criteria and multi-criteria pareto solutions.

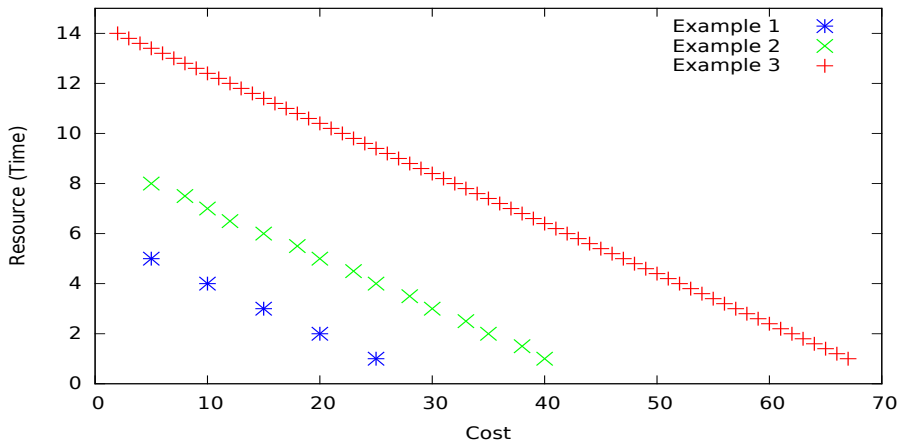


Figure 4.1: The gap between pareto solutions in the bi-criteria case

### 4.1.5 Pros and Cons of Pareto paths as a measure of Utility function

Once we find the pareto solutions, we still have the problem of choosing one or more among these. A utility function is defined as a mapping from the set of pareto solutions to a combined solution. The utility function [58] can be designed in several ways and it is also possible that

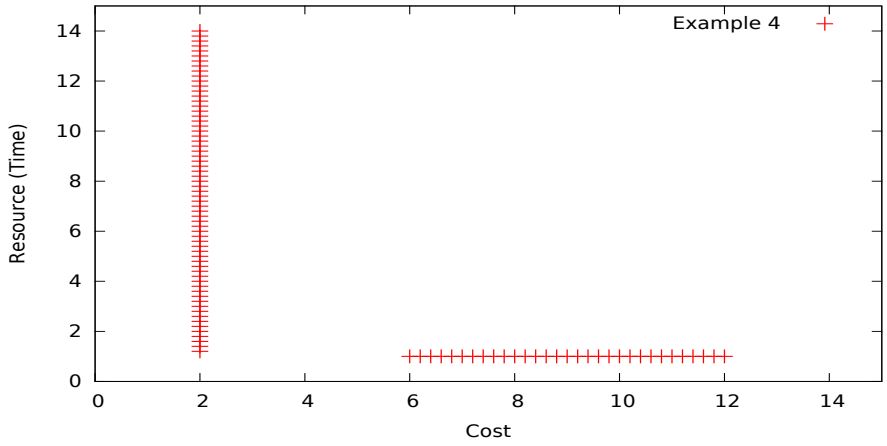


Figure 4.2: Uniqueness of $K$ shortest paths in the bi-criteria case

there might be several utility functions which can be applied to these pareto solutions. Some of the sample utility functions are illustrated in Figures 4.3, 4.3 and 4.3. In Figure 4.3 we have utility function which is linear and in Figure 4.4 and Figure 4.5 we have non-linear utility functions.

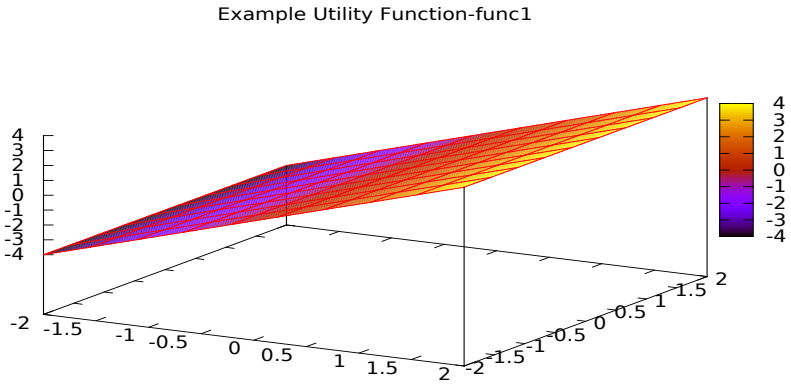


Figure 4.3: Example 1

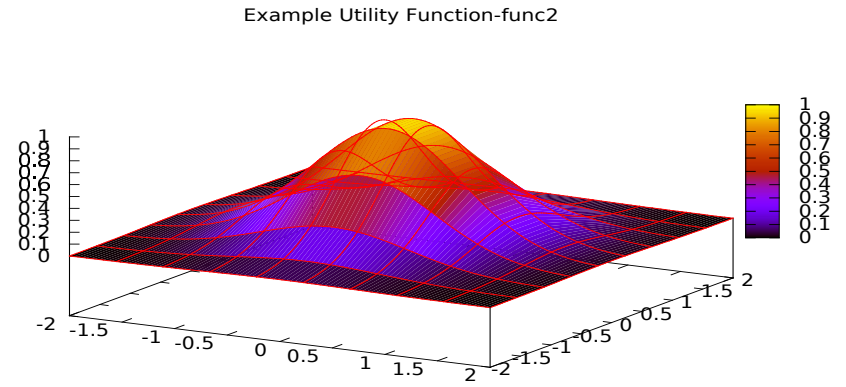


Figure 4.4: Example 2

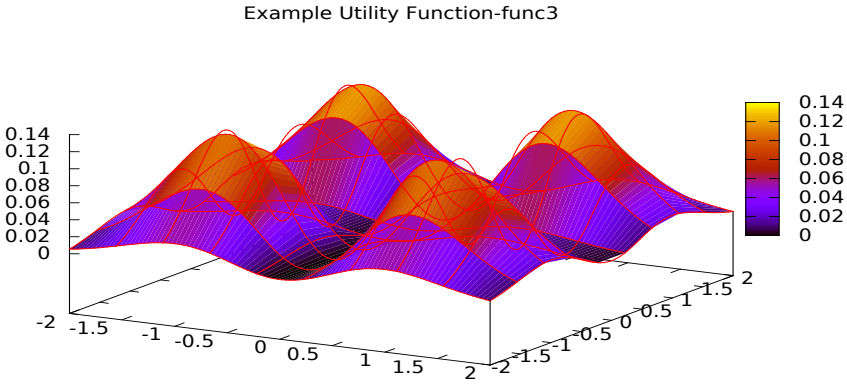


Figure 4.5: Example 3

### 4.1.6 Conjecture

We state the following conjecture and provide counter examples to disprove it.

**Conjecture 1** *Small number of Pareto paths* $\iff$ *Lower running time.*

We provide two counter examples to disprove the conjecture from both directions of the equality. In Figure 4.6 we have shown a 'Source' node and a 'Destination' node along with two intermediary nodes 'A' and 'B' and two sub-graphs connecting them. Suppose we have two pareto paths from 'Source' to 'B' and hundred pareto paths from 'Source' to 'A'. If the large connected sub-graph is dense we end up spending a lot of time traversing the edges and finding the pareto paths. Suppose the pareto paths from 'Source' to 'Destination' via 'B' dominates all the pareto paths from 'Source' to 'Destination' via 'A', we end up having just two pareto paths from 'Source' to 'Destination'. So a small number of pareto paths does not imply a lower running time. In Figure 4.7 we have shown a Directed Acyclic Graph with (cost, delay) attributes mentioned corresponding to every edge. Each path from 'Source' to 'Destination' forms a pareto path, so we end up with four pareto paths on a very small graph with a lower running time. So a small running time does not imply a lower number of pareto paths. The number of pareto paths and the running time depends not only on $N$, the size of the graph but also on the graph structure.

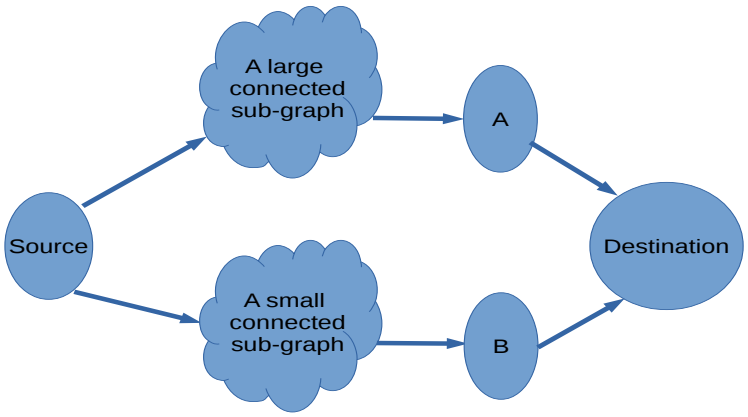


Figure 4.6: Counter Example 1

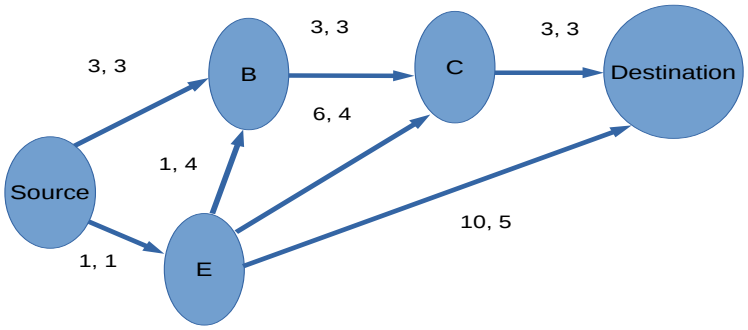


Figure 4.7: Counter Example 2

## 4.2 Overview

The main problem involved in route planning is to combine available services into end-to-end paths that meet user requirements. From an algorithmic point of view, path planning shares a number of challenges with online travel planning:

- *Large network topologies with parallel edges.* Just as the planner of a travel site takes into consideration flights from multiple airlines, many of which offer competing flights between the same pairs of cities, a path planner must consider advertisements from multiple providers, including virtual operators who may lease capacity from the same physical infrastructure. Consequently, the path planner takes as input a topology that is a superset of the topologies representing the networks of individual providers, and that is likely to include parallel edges between nodes for which there exist competing path services. Such a topology is expected to be much larger than each of its constituent individual provider topologies.

- *Support for in-advance reservations and time constraints.* Planners must allow users to reserve end-to-end paths during specific continuous time intervals in the future; this feature is analogous to booking a hotel for a set of consecutive days long before travel takes place. On the other hand, support for time constraints allows users to explore additional options whenever their communication plans are flexible, in the same manner that travel planners allow users to provide a range of acceptable start and end dates for their travel.

- *Multiple alternatives selected using multiple criteria.* The planner must present the user with several options (i.e., viable alternative end-to-end paths) that meet multiple criteria, including price, bandwidth capacity, delay, the inclusion or exclusion of sub-paths from certain providers, etc. We envision that path planning services will differentiate from the competition by deploying sophisticated and specialized algorithms for selecting paths.

Each of the above considerations significantly complicate the path finding process. For instance, introducing one additional resource constraint (e.g., a delay constraint along with a cost constraint), makes the shortest path problem NP-Complete [26, Problem ND30]. Consequently, a wide range of heuristics and approximation algorithms have been developed for a diverse set of constrained shortest path problem variants [27, 28]. Also, while efficient algorithms exist for constructing $k$-shortest elementary (i.e., acyclic) [25] and non-elementary [29] paths, the $k$-constrained shortest path problem is significantly harder and has received little attention [30].

In-advance path reservations involve reserving resources along an end-to-end path for a continuous interval of time that has a specific duration and starts at a specific instant, either

in the present or in the future. Algorithms for finding and reserving paths with sufficient bandwidth resources well in advance of the start of communication [31--33] have generally been designed for small, centrally controlled connection-oriented networks in which only a relatively small fraction of connections require such advance reservations. These algorithms may be extended to account for cost and delay constraints, but do not directly support time constraints.

The general shortest path problem with time constraints involves finding the least cost path from source to destination in a graph whose nodes can be visited within a specified time interval [59]. Similar time-constrained path problems have been studied in the context of vehicle routing [59, 60] and travel planning [61]. The problem is NP-Complete regardless of whether the shortest path is required to be elementary or is allowed to contain cycles.

## 4.3 Marketplace and Graph Model

### 4.3.1 The Marketplace

We consider a marketplace that includes a repository of *path services* as advertised by network services providers. Each path service is represented by the tuple:

$$(L_s, L_d, LID, L_{attr}, T_{start}, T_{end}),\ T_{start} < T_{end}$$

where $L_s$ and $L_d$ are the source and destination nodes, respectively, of a (physical or virtual) link with unique ID $LID$ and attributes $L_{attr}$, and $[T_{start}, T_{end}]$ is the time interval during which this path service is valid. For this work, we assume that the attributes include the available bandwidth, delay, cost, and energy consumption of the link, here cost is expressed as price per unit bandwidth; the link attribute notation is give below:

$$L_{attr} \ = \ (L_{bw}, L_{delay}, L_{cost}, L_{energy}).$$

This representation allows multiple distinct providers, including virtual providers who do not own any physical infrastructure, to advertise path services between the same $(L_s, L_d)$ pairs, that can be distinguished using the unique link ID field.

Users submit to the path planner requests of the form

$$(R_s, R_d, R_{req}, \tau_e, \tau_l),\ \tau_e \leq \tau_l$$

where $R_s$ and $R_d$ are the source and destination node, respectively, of the requested communication service and $R_{req}$ are user requirements that the service must meet, and $[\tau_e, \tau_l]$ is a time interval that specifies the earliest and latest start times for the service; if $\tau_e = \tau_l$, then the service must start at exactly time $\tau_e$. We assume that user requirements include a minimum

bandwidth along the path, an acceptable end-to-end delay, the time duration (length) of the communication, and a maximum cost that the user is willing to pay, i.e.,

$$R_{req} \;=\; (R_{bw}, R_{delay}, R_{len}, R_{cost}).$$

### 4.3.2 Graph of Path Services

The planner uses the path service descriptions stored in the marketplace repository to construct a graph $G = (V, E)$, where $V$ is the set of nodes that are part of at least one service description, and $E$ is the set of unique links defined by the service descriptions. As we mentioned earlier, the graph $G$ will generally include parallel edges representing competing services or virtual links. Each edge includes all information associated with the corresponding link, i.e., $LID$, link attributes (bandwidth, delay, cost and energy), and the interval of time $[T_{start}, T_{end}]$ during which the edge is valid.

We assume that the planner updates the graph of path services in real time whenever each of these four events takes place: (a) when a new path service is advertised, a new edge is added to the graph; (b) when an advertisement updates an existing path service, the attributes of the corresponding edge are updated accordingly (or the edge is removed if the update cancels the service); (c) when a new reservation is established, the attributes (e.g., available bandwidth) of the path services in the end-to-end path are updated accordingly; and (d) when an existing reservation terminates, the attributes of its constituent path services are also updated.

We define a time step [31] as a continuous period of time during which the state of the network does not change; in other words, the graph of path services and their attributes remain the same throughout a time step. The planner updates the sequence of time steps whenever an advertisement creates a new path service or modifies an existing one, and when reservations are set up or terminate. Consider Figure 4.8(a), where three time steps are shown, representing the changes in network state before the arrival of the new path service. As seen in the figure, the time duration of the new path service overlaps with two of the time steps. Therefore the addition of this path service causes changes in the state of the network within each of the two time steps, resulting in the five time steps shown in Figure 4.8(b). Time steps must be similarly updated for new and departing reservations.

We have the following two results.

**Lemma 4.3.1** *For any set of path services that have $m$ unique sets of $[T_{start}, T_{end}]$ time intervals, there can be at most $2m - 1$ time steps.*

**Proof 1** *In geometry, it is known that the number of non-overlapping segments formed by $k$ distinct collinear points is $k - 1$. Since $m$ unique sets of $[T_{start}, T_{end}]$ time intervals include*

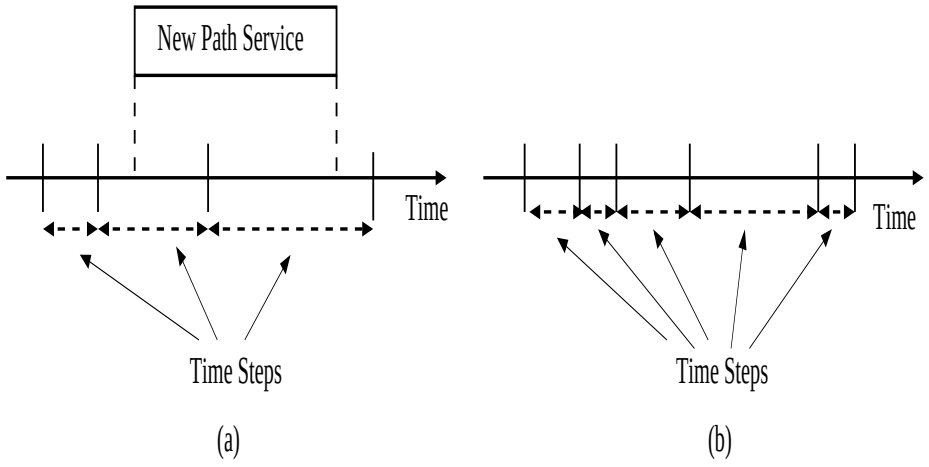


Figure 4.8: The concept of time steps

*at most* $2m$ *distinct time instants at which a path service starts or ends, the number of non-overlapping time segments created by these instants is at most* $2m - 1$. *Since a path service starts or ends at the boundary between two time segments, the state of the network (graph) does not change during any of the time segment. Therefore, there are at most* $2m - 1$ *time steps.*

**Lemma 4.3.2** *Consider a user request for a communication service that may start anywhere in the interval* $[\tau_e, \tau_l]$. *If the time interval* $[\tau_e, \tau_l]$ *overlaps with* $n$ *time steps, then, in order to satisfy this request, it is sufficient to run a path finding algorithm at most* $n$ *times, each time with a start time equal to the beginning of one of the time steps.*

**Proof 2** *Consider time step* $x = [t_1, t_2]$ *that overlaps with the interval* $[\tau_e, \tau_l]$ *of the user request. Let* $\mathcal{P}$ *be the set of paths that a specific path finding algorithm returns under the assumption that the communication service requested by the user starts at time* $t_1$. *Since the state of the network does not change for the duration of time step* $x$, *the same algorithm will not be able to find better paths than the ones in* $\mathcal{P}$ *for any start time* $t$ *of the request such that* $t_1 < t \leq t_2$. *On the other hand, the algorithm may find worse paths when* $t_1 < t \leq t_2$; *this may occur if the later starting time causes the service to end within a later time step in which the network state may not be able to accommodate the quality of features of the paths in* $\mathcal{P}$.

The above two results impose strict bounds on the search space that the planner has to explore to satisfy a user request. These bounds make path computations more efficient than the method used in [31] to divide the search space; the latter method becomes inefficient even for networks of moderate size with a relatively small number of path services.

## 4.4 Multi-Criteria Time Constrained Paths

Our objective is to present each user requesting service with a set of time constrained paths that satisfy multiple user-specified constraints. More formally, the problems we address are variations of the time constrained shortest path (TCSP) problem defined as follows.

**Problem 1 (Non-dominated $k$-TCSP with resource constraints (ND-k-TCSPRC))** *Let $G = (V, E)$ be a graph with path services as edges such that each edge $e$ is valid only during the time interval $[T^e_{start}, T^e_{end}]$. Let $U$ be a utility function defined by the user which maps the set of pareto solutions to the set $\mathbb{R}$. Consider the user request*

$(R_s, R_d, R_{req}, \tau_e, \tau_l), R_{req} = (R_{bw}, R_{delay}, R_{len}, R_{cost})$

*and an integer $k$. Find the top $k$ Pareto-optimal paths from $R_s$ to $R_d$ which provide the maximum utility, such that each path:*

1. *is a concatenation of one or more path services (edges),*
2. *has bandwidth at least $R_{bw}$,*
3. *has end-to-end delay at most $R_{delay}$, and*
4. *is valid throughout the interval $[t, t + R_{len}]$, for any $t \in [\tau_e, \tau_l]$,*

*where a path is considered valid in a given time interval if and only if all path services comprising the path are valid in the same interval.*

**Problem 2 (Non-dominated $k$-TCSP (ND-k-TCSP))** *Let $G = (V, E)$ be a graph with path services as edges such that each edge $e$ is valid only during the time interval $[T^e_{start}, T^e_{end}]$. Let $U$ be a utility function defined by the user which maps the set of pareto solutions to the set $\mathbb{R}$. Consider the user request*

$(R_s, R_d, R_{req}, \tau_e, \tau_l), R_{req} = (R_{bw}, R_{delay}, R_{len}, R_{cost})$

*and an integer $k$. Find the top $k$ Pareto-optimal paths from $R_s$ to $R_d$ which provide the maximum utility, such that each path:*

1. *is a concatenation of one or more path services (edges),*
2. *has bandwidth at least $R_{bw}$,*
3. *is valid throughout the interval $[t, t + R_{len}]$, for any $t \in [\tau_e, \tau_l]$,*

*where a path is considered valid in a given time interval if and only if all path services comprising the path are valid in the same interval.*

We note that both NDTCSP and NDTCSPRC are in the class NPC [62] even for one time instance. To keep the notations uniform for solving the problem we set $R_{delay}$ to $\infty$.

**Problem 3 ($k$-TCSP with resource constraints (k-TCSPRC))** *Let $G = (V, E)$ be a graph with path services as edges such that each edge e is valid only during the time interval $[T^e_{start}, T^e_{end}]$. Consider the user request*

$(R_s, R_d, R_{req}, \tau_e, \tau_l), R_{req} = (R_{bw}, R_{delay}, R_{len}, R_{cost})$

*and an integer k. Find k least cost paths from $R_s$ to $R_d$, such that each path:*

1. *is a concatenation of one or more path services (edges),*
2. *has bandwidth at least $R_{bw}$,*
3. *has end-to-end delay at most $R_{delay}$, and*
4. *is valid throughout the interval $[t, t + R_{len}]$, for any $t \in [\tau_e, \tau_l]$,*

*where a path is considered valid in a given time interval if and only if all path services comprising the path are valid in the same interval.*

This reduces to the $k$-CSP problem [30] which is known to be in NPC even for one time instance.

**Problem 4 ($k$-TCSP)** *Let $G = (V, E)$ be a graph with path services as edges such that each edge e is valid only during the time interval $[T^e_{start}, T^e_{end}]$. Consider the user request*

$(R_s, R_d, R_{req}, \tau_e, \tau_l), R_{req} = (R_{bw}, R_{delay}, R_{len}, R_{cost})$

*and an integer k. Find k least cost paths from $R_s$ to $R_d$, such that each path:*

1. *is a concatenation of one or more path services (edges),*
2. *has bandwidth at least $R_{bw}$,*
3. *is valid throughout the interval $[t, t + R_{len}]$, for any $t \in [\tau_e, \tau_l]$,*

*where a path is considered valid in a given time interval if and only if all path services comprising the path are valid in the same interval.*

We note that this is pseudo-polynomial algorithm [63] even for one time instance. To keep the notations uniform for solving the problem we set $R_{delay}$ to $\infty$.

### 4.4.1 Dynamic Programming Algorithm for Problems 1 and 2

Let $G = (V, E)$ be the graph of path services at the time the user request

$(R_s, R_d, R_{req}, \tau_e, \tau_l), R_{req} = (R_{bw}, R_{delay}, R_{len}, R_{cost})$

arrives. We set the utility function

$U \propto (1/\sum_{L \in \, Pareto \; Path} L_{cost})$

We now present a dynamic programming algorithm that can be used to find Pareto-optimal paths from node $R_s$ to node $R_d$ that are valid in the interval $[\tau_e, \tau_l + R_{len}]$.

Define $F(i, t, R_{delay})$ as the minimum cost of any path from source $R_s$ to the node $i$, $i \in V$, that starts at time $t$, has available bandwidth at least equal to $R_{bw}$, and its cumulative delay (i.e., the total delay along the path services from $R_s$ to $i$) is at most $R_{delay}$. If no such path exists at time $t$, then $F(i, t, R_{delay}) = \infty$.

$F(i, t, R_{delay})$ can be calculated using the following recursion:

$$F(i, t, D) = \begin{cases} 0, & i = R_s \text{ and } D \geq 0 \\ \infty, & D < 0 \end{cases} \tag{4.1}$$

$$\begin{aligned} F(j, t, D) &= \min_{(i,j) \in E} \left\{ F\left(i, t, D - L_{delay}^{(i,j)}\right) + L_{cost}^{(i,j)} \right\}, \\ &\forall (i, j) \in E \;\; , \;\; D \leq R_{delay}, R_{bw} \leq L_{bw}^{(i,j)} \end{aligned} \tag{4.2}$$

The base case (4.1) simply states that (i) the cost of getting from the source node $R_s$ to itself is zero, and (ii) the cost of going from $R_s$ to any node $i$ with a negative delay is infinity since no such path exists. The recursive expression (4.2) can be explained by noting that the minimum cost of getting from $R_s$ to node $j$ with a total delay of at most $D$, is equal to the minimum cost, over all path services $(i, j), i \neq j$, of getting from $R_s$ to node $i$ with a total delay of at most $D - L_{delay}^{(i,j)}$, plus the cost $L_{cost}^{(i,j)}$ of going from $i$ to $j$. Note also that the minimum is taken only over edges (path services) $(i, j)$ that have sufficient bandwidth for the user request.

The optimal solution at time $t$, i.e., the minimum cost of a path that starts at time $t$ and can accommodate the user request, can be computed as:

$$F(R_d, t, R_{delay}). \tag{4.3}$$

Recall now that, according to Lemma 4.3.2, it is sufficient to run the path finding algorithm once for each time step that overlaps with the interval $[\tau_e, \tau_l]$ that represents the allowable start times for the user request. Let $n$ be the number of such time steps and $t_1, \ldots, t_n$ be the time instants when the path finding algorithm must be run; according to Lemma 4.3.2, $t_1 = \tau_e$, while $t_2, \ldots, t_n$ coincide with the start of the following $n - 1$ time steps. Therefore, the overall

optimal solution, i.e., the cost of the minimum-cost path for the user request starting anywhere in $[\tau_e, \tau_l]$, can be obtained as:

$$\min_{t_1,\ldots,t_n} F(R_d, t_i, R_{delay}). \tag{4.4}$$

We note that computing expression (4.3) may require the evaluation of an exponential number of paths. Furthermore, the recursion returns the cost of a minimum-cost, feasible path, if one exists, but it does not directly provide the path services (edges) comprising this path. Importantly, this expression does not compute multiple shortest paths, and hence it does not provide a solution to Problems 1 and 2.

In the following subsection, we show that it is possible to maintain labels at the nodes of graph $G$ during the execution of recursion (4.2), so as to (i) construct Pareto-optimal paths, and (ii) speed up the recursion by eliminating paths (i.e., terminating the recursion early) that will not lead to Pareto-optimal solutions.

#### 4.4.1.1 Tracking Pareto-Optimal Paths

Consider an execution of the recursive algorithm (4.3) for a given start time $t$. At each node $i$ visited by the recursion, we maintain labels to keep track of Pareto-optimal paths passing through that node. Specifically, for each path through node $i$, we maintain the tuple $(C, D)$, where $C$ (respectively, $D$) is the cost (respectively, delay) of the path from the source node $R_s$ to node $i$[1].

Consider two paths through node $i$ with labels $(C_1, D_1)$ and $(C_2, D_2)$, respectively. We say that the first path *dominates* the second, denoted by $(C_1, D_1) \prec (C_2, D_2)$, if $C_1 \leq C_2$ *and* $D_1 \leq D_2$. In other words, the dominating path is better than the dominated one in terms of both cost and delay. When we add a third criteria, energy, the dominating path is better than the dominated one for all three attributes i.e., cost, delay, and energy. Note that, all paths entering node $i$ have the exact same options as path services to continue towards the destination $R_d$. Therefore, it is certain that the dominated path will result in an end-to-end solution that cannot be superior to that resulting from the dominating path in terms of either cost and delay. Consequently, we eliminate the dominated path at node $i$ by terminating the recursion at that point, which also speeds up the overall running time.

At the end of the recursion (4.3), we obtain Pareto-optimal paths that start at time $t$. We execute the recursion $n$ times, once for each time step, as indicated in (4.4), and obtain Pareto-optimal paths that start in $[\tau_e, \tau_l]$. We then extract (up to) $k$ least-cost Pareto-optimal paths from this list, and return them to the user, allowing the latter to make an informed selection.

[1]The label includes two additional parameters: the previous node $j$ towards the source $R_s$ and the unique link ID, LID, of the path service that leads from $j$ to $i$. These parameters make it possible to reconstruct the path starting at the destination node, $R_d$, but are not essential for determining Pareto-optimal paths.

### 4.4.2 $k$-shortest cost paths algorithm for Problem 4

We will now consider the TCSP problem variation where the user does not request non dominant paths but instead requests $k$ least cost paths without resource constraints. Problem 4 may be solved in pseudopolynomial time [59] if the number of time instants is finite using the following steps at each of the $n$ time instants $t_i$ discussed in Section 4.4.1 above: (1) remove from the graph all edges which, at time $t_i$, have available bandwidth less than $R_{bw}$; (2) run Yen's algorithm [25] to construct the $k$ shortest paths between $R_s$ and $R_d$ at time $t_i$. These steps will determine up to $nk$ shortest paths, of which we present the $k$ shortest to the user. Since Yen's algorithm is polynomial, assuming $k$ and the number of time instants are bounded, this algorithm will produce the $k$ shortest paths starting anywhere in $[\tau_e, \tau_l]$ in polynomial time.

### 4.4.3 $k$-shortest cost paths algorithm for Problem 3

We will now consider the TCSP problem variation where the user does not request non dominant paths but instead requests $k$ least cost paths with resource constraints. This is identical to the problem defined in [54] but now in the context of time windows. The algorithmic approach to this problem is similar to the one above but with an additional checking of the resource constraint done at every step when we relax the edge i.e. add a node to the list of explored nodes.

## 4.5 Numerical Results

We now present simulation results to evaluate the algorithms for Problems 1, 2, 3, and 4.

We used BRITE [64] to generate graphs for running the simulation because it is a universal topology generator and offers more than just network connectivity at the AS level. We obtained undirected graphs by configuring BRITE to generate AS-Level Barabasi models. We set the size of the outer and inner planes to 1000 and 100 respectively, for placement of the nodes in a heavy tailed distribution. We set the growth type of the graph to be incremental in nature, we disabled the preferential connection property, and we set the average nodal degree to between 2 and 4. We used a uniform bandwidth distribution with a maximum and minimum bandwidth values of 2500 Mbps and 100 Mbps, respectively, with the additional restriction that bandwidth values be multiples of 100 Mbps. $L_{delay}$, the link delay was set proportional to the Euclidean distance between the two points in the plane representing the endpoints of the edge. $L_{energy}$, the energy consumption of an edge was set independent of both the link delay and link cost. $L_{energy}$ is uniformly distributed between [1 - 1000].

We use two cost models. In the first model, the cost of using a link as a function of the product of bandwidth times duration of the connection. Specifically, we let the cost, $L_{cost}$, per

unit bandwidth (i.e., 1 Mbps) to \$0.06, a value that is approximately one-tenth of the current market cost [65]. Hence, the price that a user has to pay for a connection can be expressed as $\$0.06 \times R_{bw} \times R_{len}$. In the second model. the link cost is negatively correlated to the delay. Finally, we let the start and end times of an edge (path service) to be in the range [0, 15 days].

We generate user requests using the following model:

- The bandwidth $R_{bw}$ requested is uniformly distributed in the range [10, 100 Mbps] with probability 0.6, and in the range (100 Mbps, 500 Mbps] with probability 0.4.
- The duration $R_{len}$ of the request is uniformly distributed in the ranges: [1, 30 min] (probability 0.1), [31 min, 60 min] (probability 0.1), (1 hr, 3 hr] (probability 0.6), and (3 hr, 12 hr] (probability 0.2).
- The earliest start time $\tau_e$ is between [0, 1 day] with probability 0.8, and between (1, 15 days] with probability 0.2.
- The latest start time $\tau_l$ is set to either equal to $\tau_e$ (with probability 0.5) or is uniformly distributed in the range ($\tau_e$, $\tau_e$+ 60 min] (with probability 0.5).
- The end-to-end delay $R_{delay}$ is set to $\sqrt{2}$ times the delay along the Euclidean distance of the diameter in the outer plane of the topology graph for Problems 1 and 3 and is set to $\infty$ for Problems 2 and 4.

We further assume that user requests arrive as a Poisson process with mean equal to 1 minute. We use five different graph models to evaluate the algorithms used for solving the problems defined earlier and the mapping of the graph models to the corresponding problems is captured in Table 4.3.

We have implemented the routing algorithms in C, and we run the simulation experiments on a Linux cluster, each node in the cluster consisting of two Xeon processors (representing a mix of 1, 2, 4, 6, or 8 cores) and 2-4 GB of memory per core. In the figures we present in this section, each data point corresponds to the average of 30 randomly generated problem instances. The figures also plot confidence intervals around the mean, estimated using the method of batch means.

| Graph Model | Link Cost | Link Delay | $R_{delay}$ | Link Energy | Problem |
|---|---|---|---|---|---|
| 1 | Fixed | $\propto$ Euclidean Distance | Finite | NA | 1 |
| 2 | Fixed | $\propto$ Euclidean Distance | $\infty$ | NA | 2 and 4 |
| 3 | $\propto$(1/Link Delay) | $\propto$ Euclidean Distance | $\infty$ | NA | 2 and 4 |
| 4 | $\propto$(1/Link Delay) | $\propto$ Euclidean Distance | Finite | NA | 3 |
| 5 | $\propto$(1/Link Delay) | $\propto$ Euclidean Distance | $\infty$ | Uniformly distributed | 1 |

Table 4.3: Mapping of Problems to Graph Models

### 4.5.1 Model 1: Fixed Cost and finite threshold Delay

Figure 4.9 plots the running time of the dynamic programming algorithm as a function of the number $N$ of nodes in the graph; for these experiments, the average nodal degree was set to 2. For each problem instance, we generated 100 user requests and, hence, run the algorithm 100 times to find paths for each request. The running time shown in the figure is an average over these 100 executions. As we can see, the running time increases faster than linearly with the size of the network, but remains reasonable even for large topologies; for $N = 400$ nodes, it takes about 7-8 seconds, an amount of time comparable to what users experience in online travel sites. We also plot the running time of a $O(N^3)$ function to compare the running time of the dynamic programming algorithm with 2-criteria. We use the reference time for N = 100 to plot the extrapolated running times for N = 200 to N = 500. We expect a running time which is between $\Omega(N^3)$ and an exponential running time since we use a variation of Label Correction Algorithm and also use an adjacency matrix. Since the running time of the algorithm is very close to the cubic function we can claim that the algorithm has been efficiently implemented in the context of Model 1.

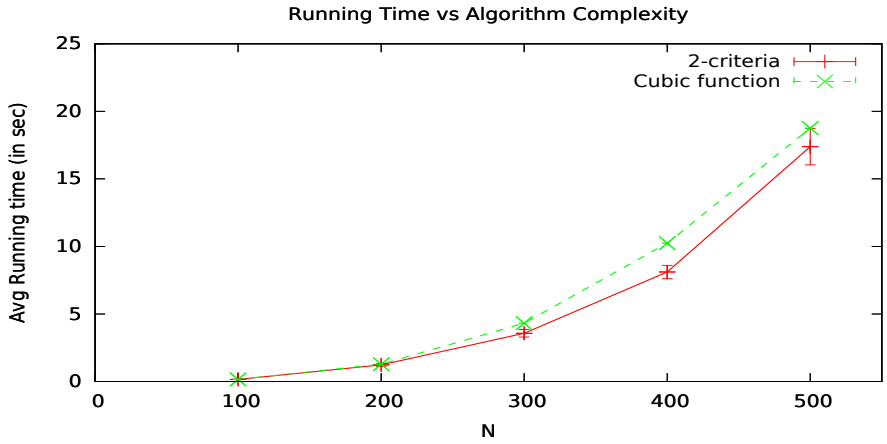


Figure 4.9: Running time of the dynamic programming algorithm for Problem 1

### 4.5.2 Model 2: Fixed Cost and no Threshold Delay

The second model compares the impact of having no resource constraints on Problems 2 and 4

Figure 4.10 presents the average running time of the algorithm for solving Problem 4, as a function of the number $k$ of shortest paths; for these experiments, we generated 1,000 user requests and the average was taken over the 1,000 executions of the algorithm. We can see that the running time increases linearly with $k$, and also with the network size, as expected. Overall, this algorithm runs more than one order of magnitude faster than the dynamic programming algorithm for the same network size, implying that relaxing the delay and dominance constraints makes it possible to scale to very large networks.

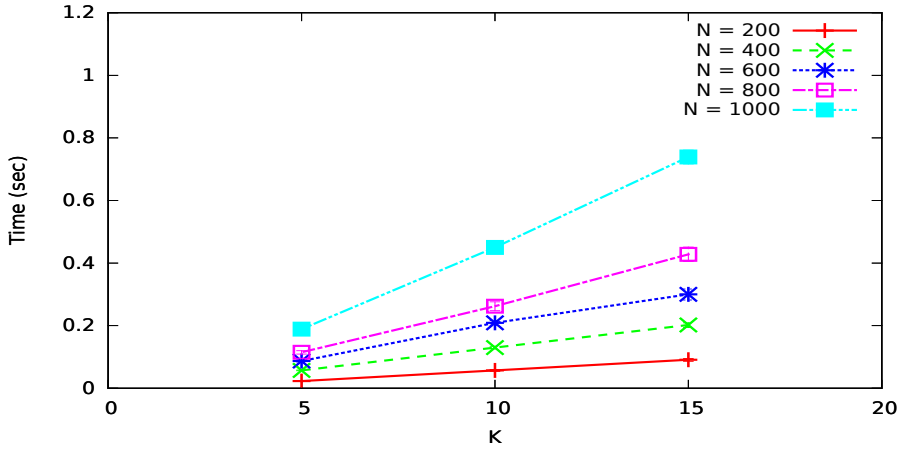


Figure 4.10: Running time of the $k$-shortest cost paths algorithm for Problem 4 with no delay constraints

For Problem 2 to get the fine grain information regarding the impact of pareto paths, average hop count and the number of time instances searched corresponding to a user request we plot their distribution in the following sections and explain their role in the running time of the algorithm. The distribution plots are drawn for 30 simulation runs with each simulation run consisting of 100 user requests or simulation instances. We use the same experimental setup as in Model 1 for evaluating Problem 2.

Figures 4.11 to 4.16 plot the distribution of the number of pareto paths over the simulation instance for N = 100 to N = 600 respectively. Initially, before the first user request arrives all the links have attributes such as bandwidth identical throughout the whole time window. As user requests arrive and bandwidth is carved out from the link, the time window gets fragmented and the probability that the earliest and latest start times of a new user request overlaps with the previous fragmented time windows increases with every new incoming user request. The dynamic programming algorithm doesn't compare path dominance for paths which have different start time. This explains the peaks observed in the Figures 4.11 to 4.16 which is caused by more pareto paths being found for fragmented time windows. This observation is consistent for Figures 4.17 to 4.22 which plots the distribution of the number of time instances i.e., the number of times the dynamic programming algorithm is run to find the pareto paths which have a starting time coinciding with the time instance value.

Figures 4.23 to 4.28 plot the distribution of average hop count of the pareto paths vs the running time (in seconds) of the algorithm. Figures 4.29 to 4.34 plot the distribution of number of time instances searched to find the pareto paths vs the running time (in seconds) of the algorithm. Figures 4.35 to 4.40 plot the distribution of number of pareto paths vs the running time (in seconds) of the algorithm. We observe that we do not see peaks for the running time of the algorithm as the pareto paths or the hop count increases unless it is accompanied by a increase in the number of time instances.

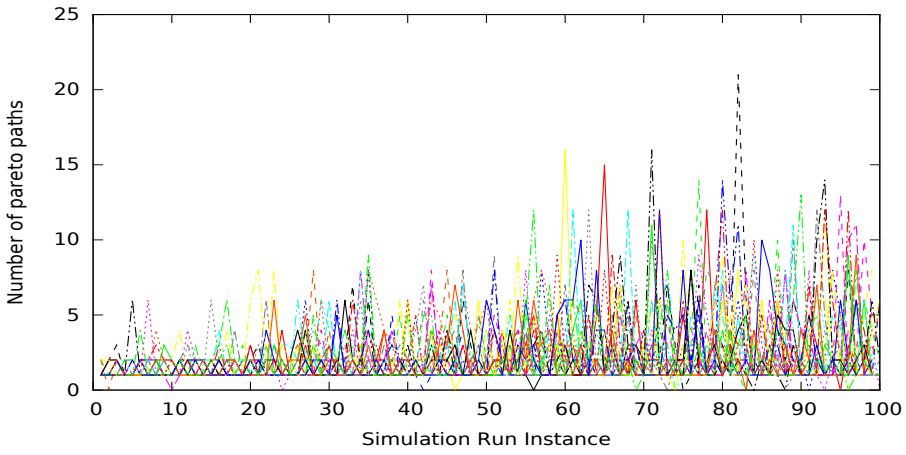


Figure 4.11: Pareto Paths Distribution for N = 100, Fixed Link Cost, Infinite Threshold Delay

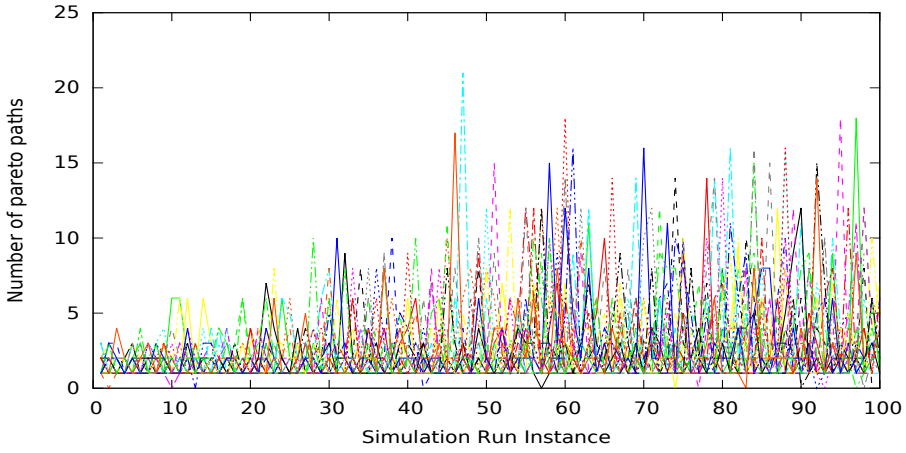


Figure 4.12: Pareto Paths Distribution for N = 200, Fixed Link Cost, Infinite Threshold Delay

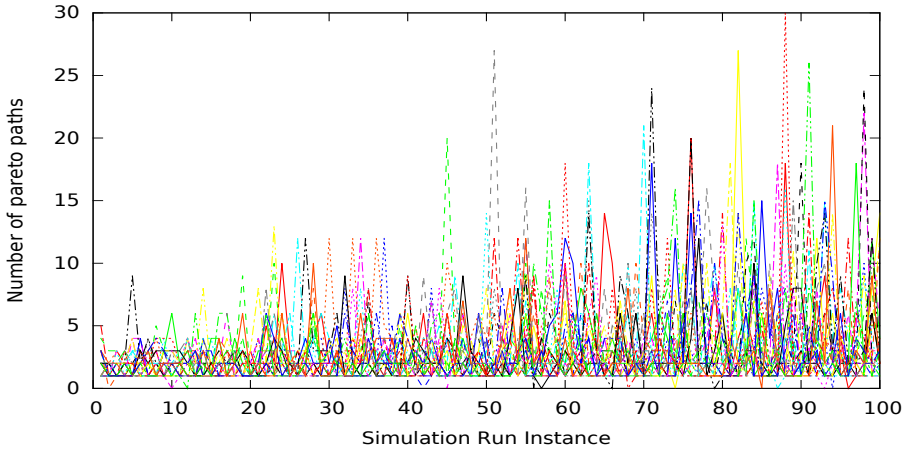


Figure 4.13: Pareto Paths Distribution for N = 300, Fixed Link Cost, Infinite Threshold Delay

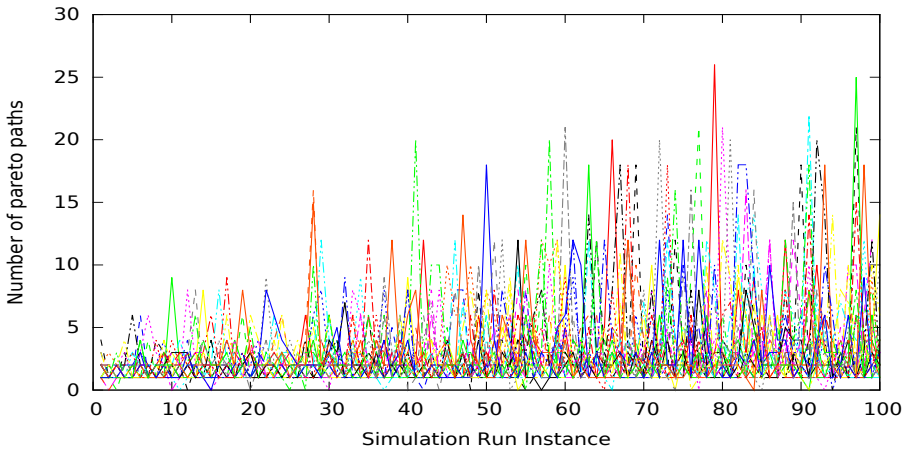


Figure 4.14: Pareto Paths Distribution for N = 400, Fixed Link Cost, Infinite Threshold Delay

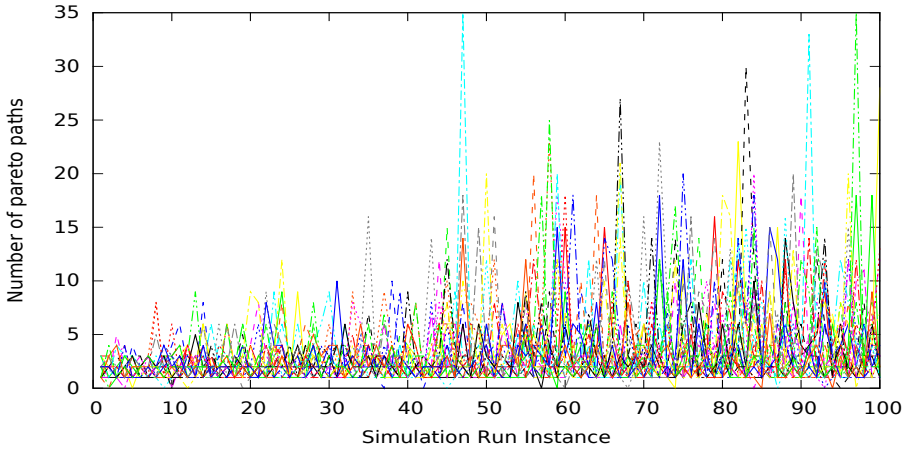


Figure 4.15: Pareto Paths Distribution for N = 500, Fixed Link Cost, Infinite Threshold Delay

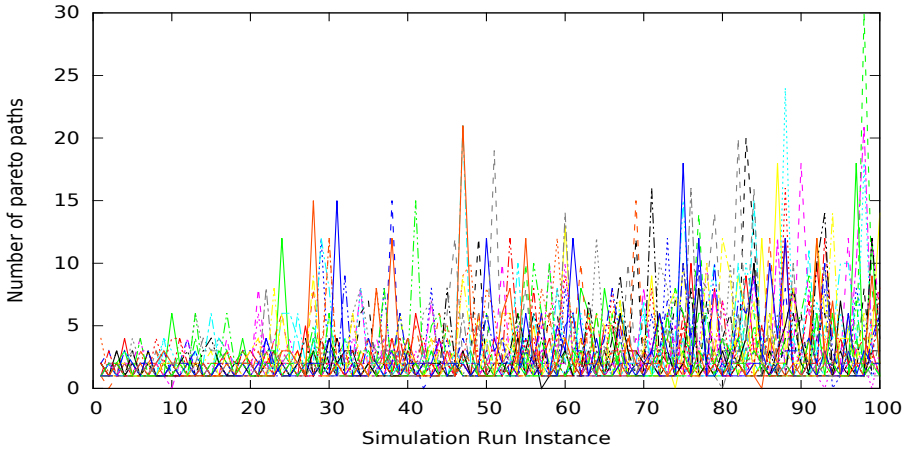


Figure 4.16: Pareto Paths Distribution for N = 600, Fixed Link Cost, Infinite Threshold Delay

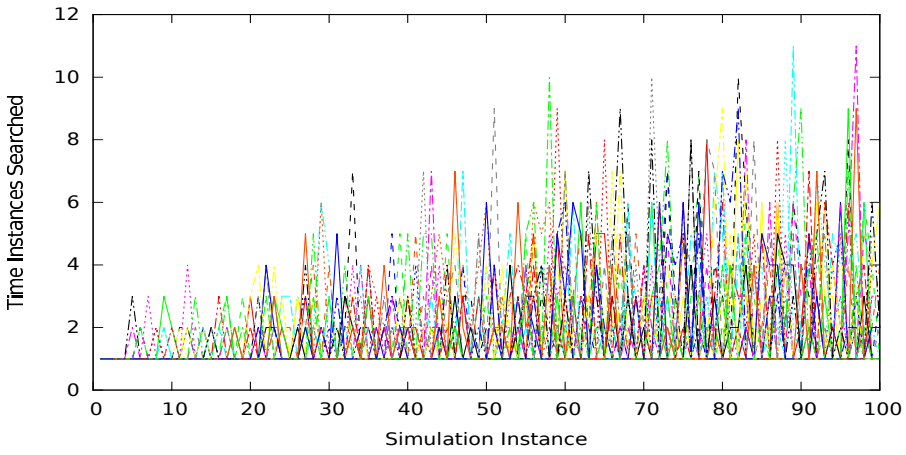


Figure 4.17: Time Instances Distribution for N = 100, Fixed Link Cost, Infinite Threshold Delay

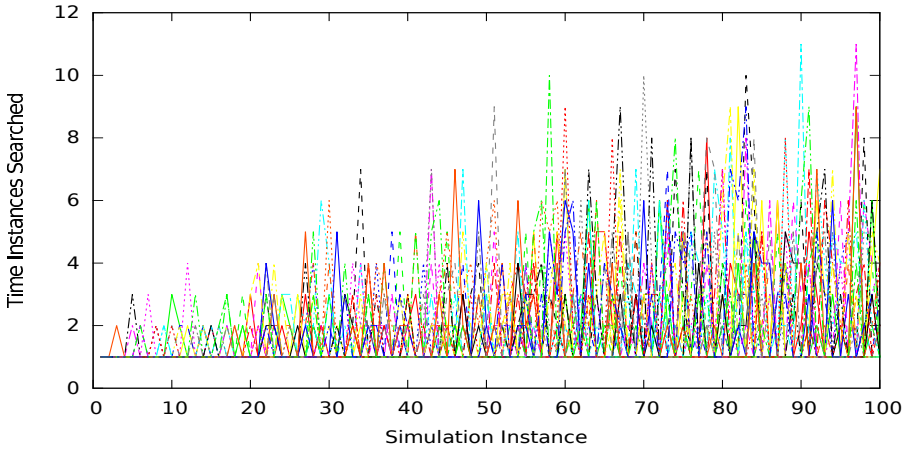


Figure 4.18: Time Instances Distribution for N = 200, Fixed Link Cost, Infinite Threshold Delay

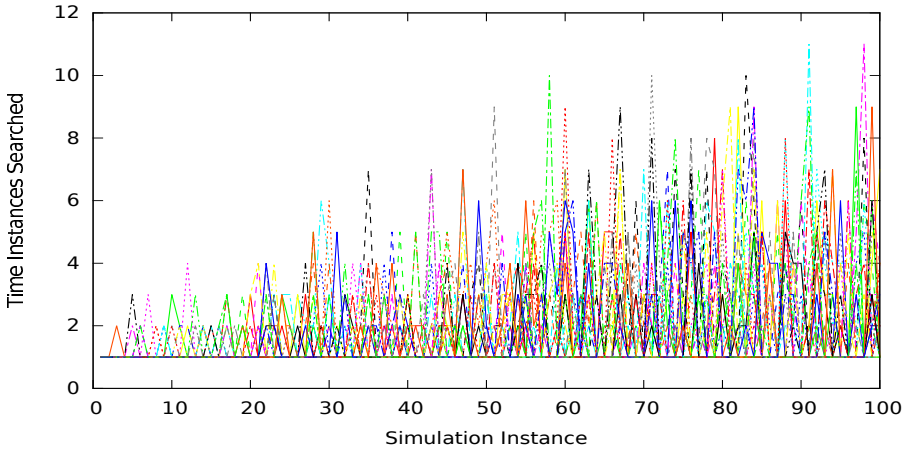


Figure 4.19: Time Instances Distribution for N = 300, Fixed Link Cost, Infinite Threshold Delay

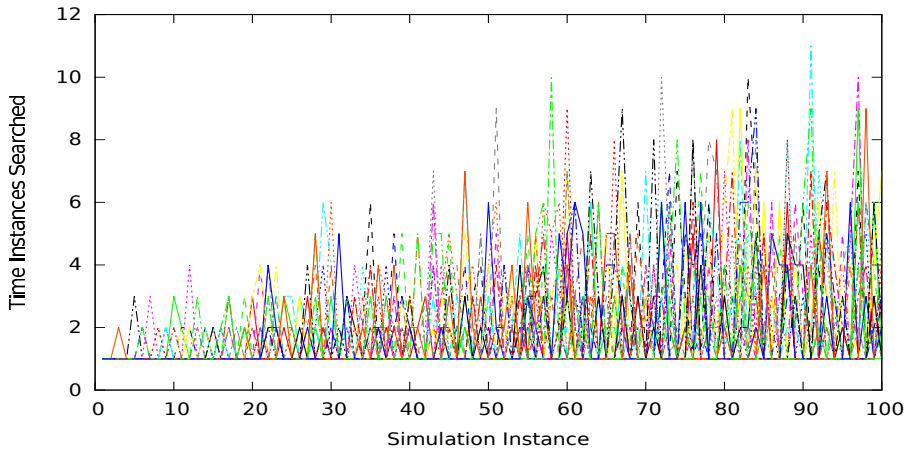


Figure 4.20: Time Instances Distribution for N = 400, Fixed Link Cost, Infinite Threshold Delay

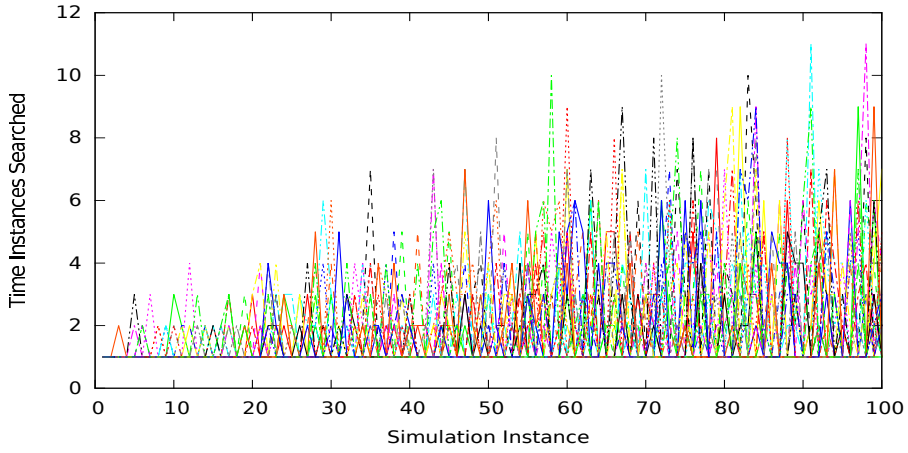


Figure 4.21: Time Instances Distribution for N = 500, Fixed Link Cost, Infinite Threshold Delay

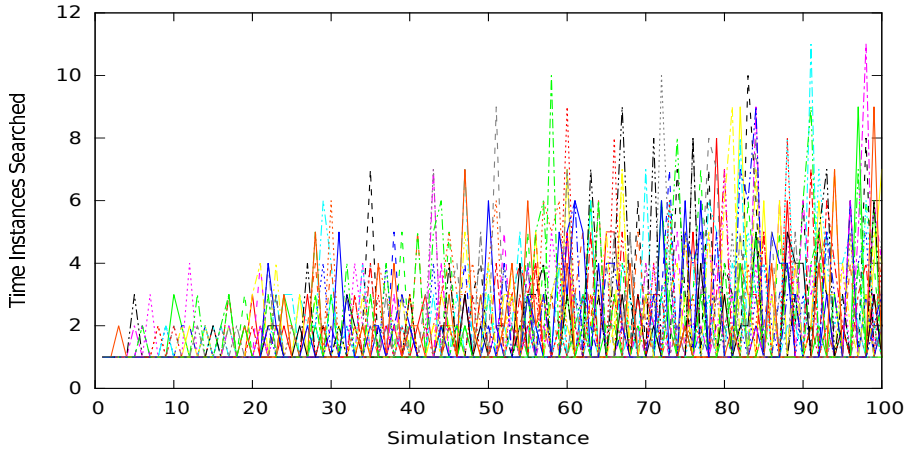


Figure 4.22: Time Instances Distribution for N = 600, Fixed Link Cost, Infinite Threshold Delay

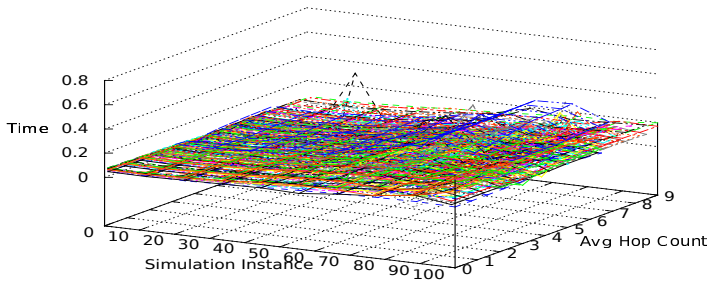


Figure 4.23: Running time vs average hop count for N = 100, Fixed Link Cost, Infinite Threshold Delay

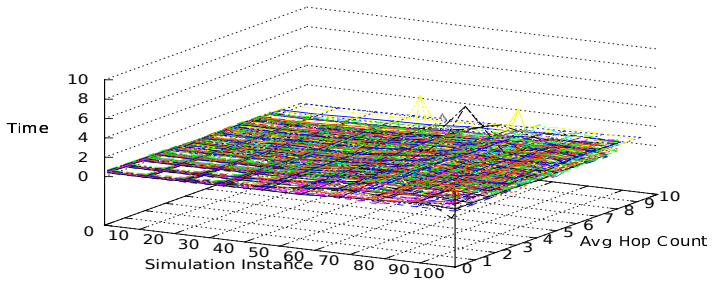


Figure 4.24: Running time vs average hop count for N = 200, Fixed Link Cost, Infinite Threshold Delay

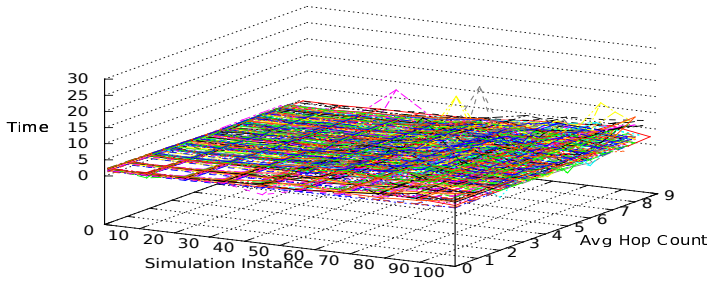


Figure 4.25: Running time vs average hop count for N = 300, Fixed Link Cost, Infinite Threshold Delay

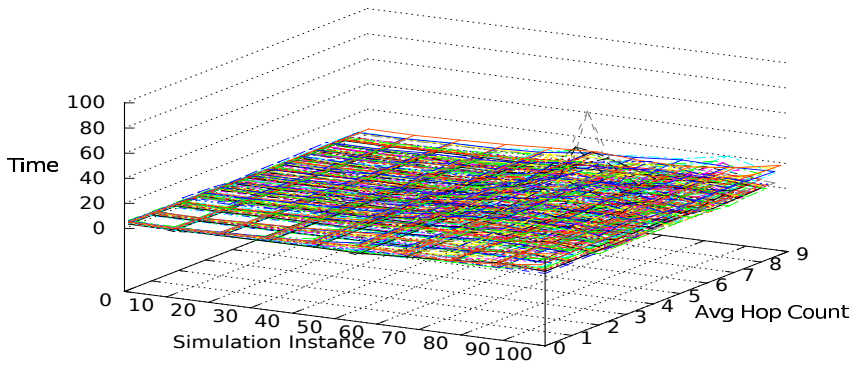


Figure 4.26: Running time vs average hop count for N = 400, Fixed Link Cost, Infinite Threshold Delay

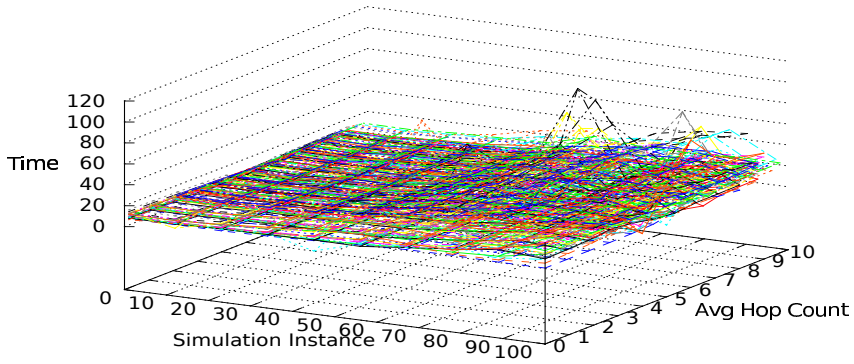


Figure 4.27: Running time vs average hop count for N = 500, Fixed Link Cost, Infinite Threshold Delay

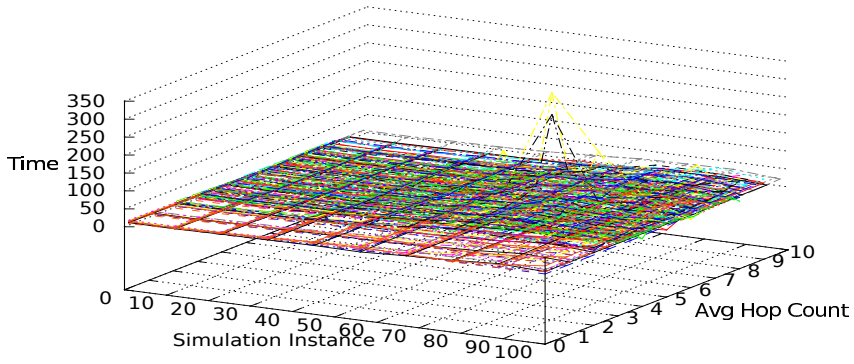


Figure 4.28: Running time vs average hop count for N = 600, Fixed Link Cost, Infinite Threshold Delay

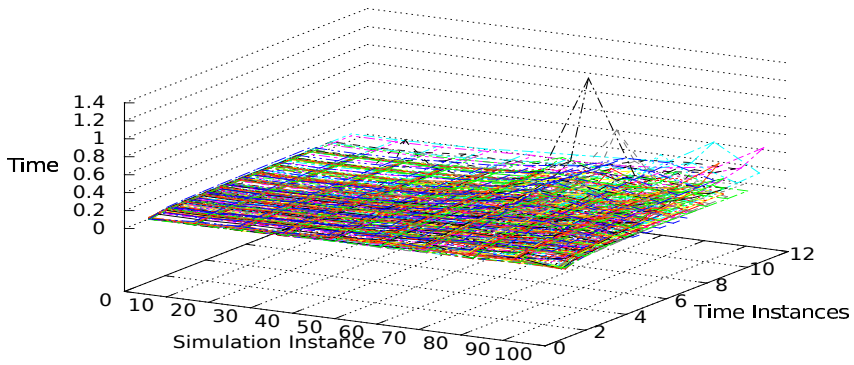


Figure 4.29: Running time vs number of time instances for N = 100, Fixed Link Cost, Infinite Threshold Delay

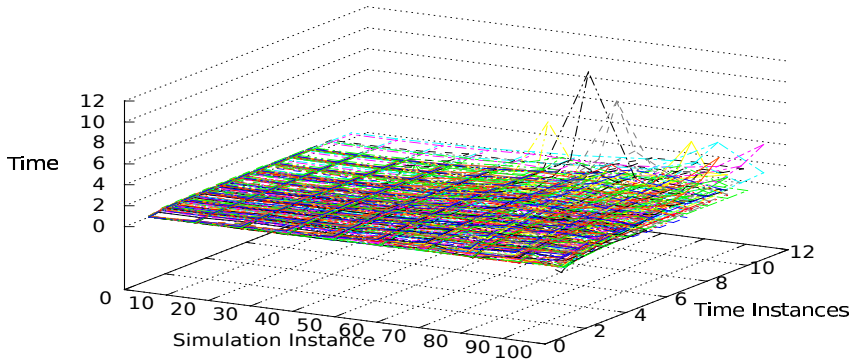


Figure 4.30: Running time vs number of time instances for N = 200, Fixed Link Cost, Infinite Threshold Delay

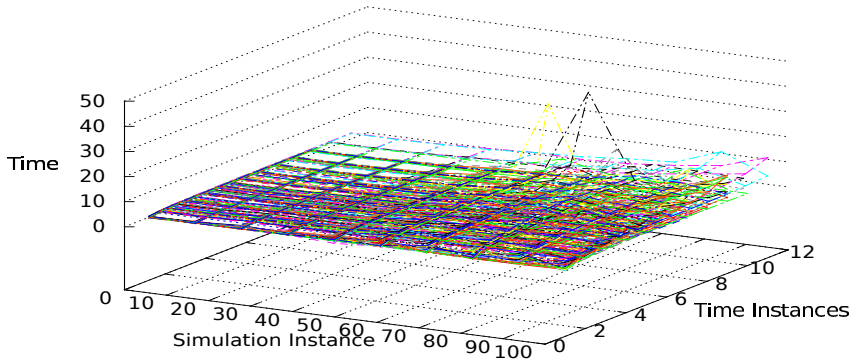


Figure 4.31: Running time vs number of time instances for N = 300, Fixed Link Cost, Infinite Threshold Delay

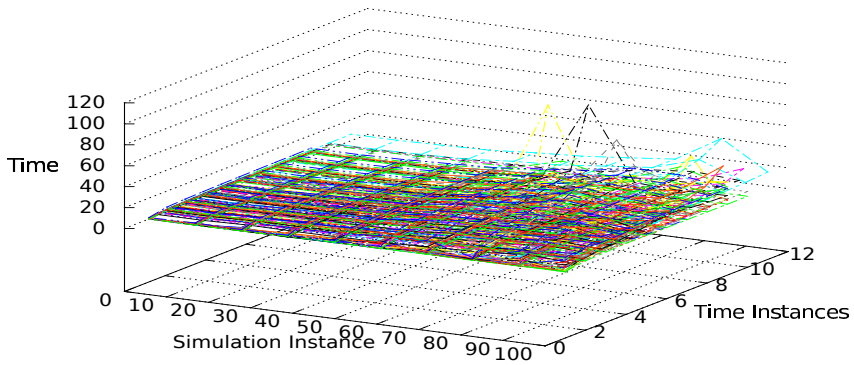


Figure 4.32: Running time vs number of time instances for N = 400, Fixed Link Cost, Infinite Threshold Delay

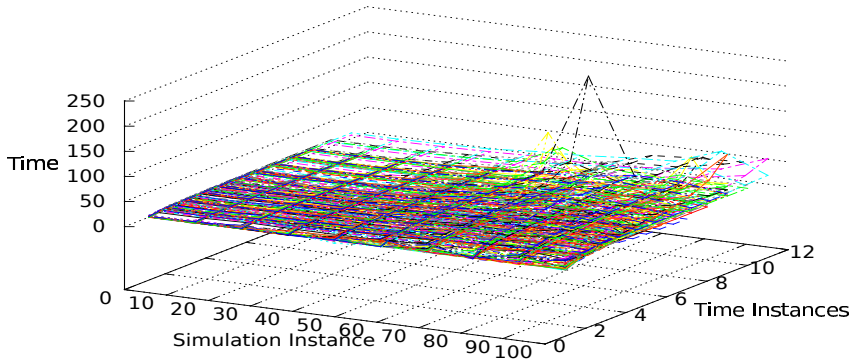


Figure 4.33: Running time vs number of time instances for N = 500, Fixed Link Cost, Infinite Threshold Delay

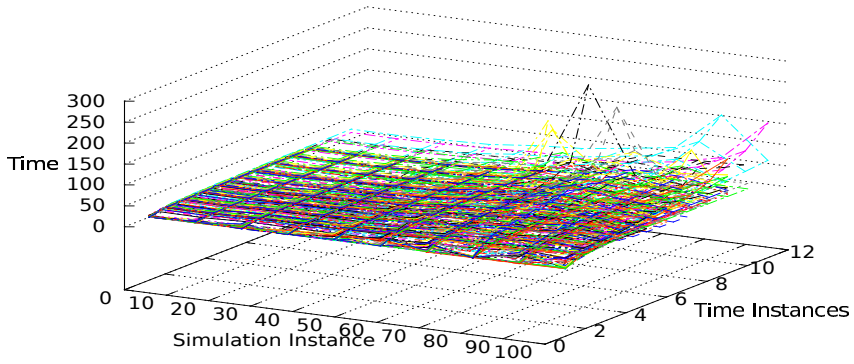


Figure 4.34: Running time vs number of time instances for N = 600, Fixed Link Cost, Infinite Threshold Delay

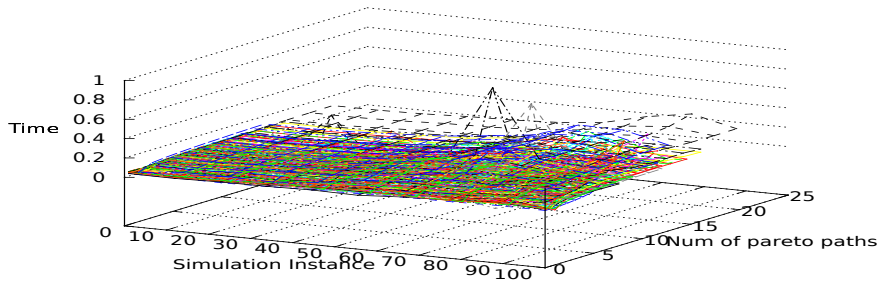


Figure 4.35: Running time vs number of pareto paths for N = 100, Fixed Link Cost, Infinite Threshold Delay

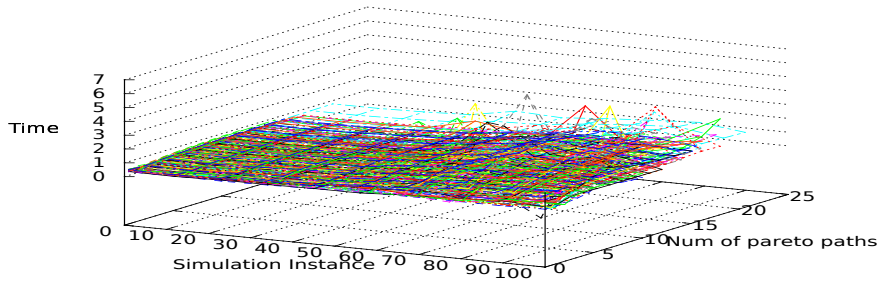


Figure 4.36: Running time vs number of pareto paths for N = 200, Fixed Link Cost, Infinite Threshold Delay

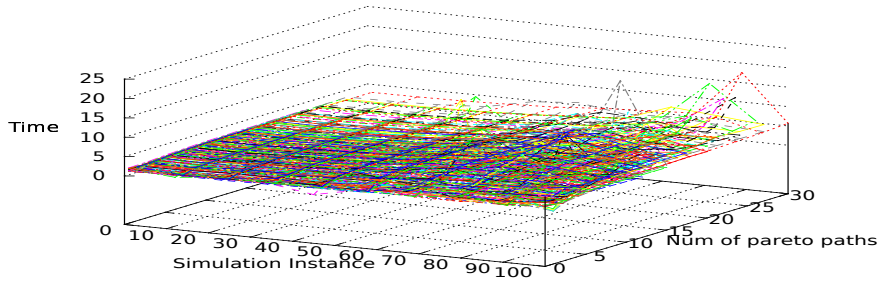


Figure 4.37: Running time vs number of pareto paths for N = 300, Fixed Link Cost, Infinite Threshold Delay

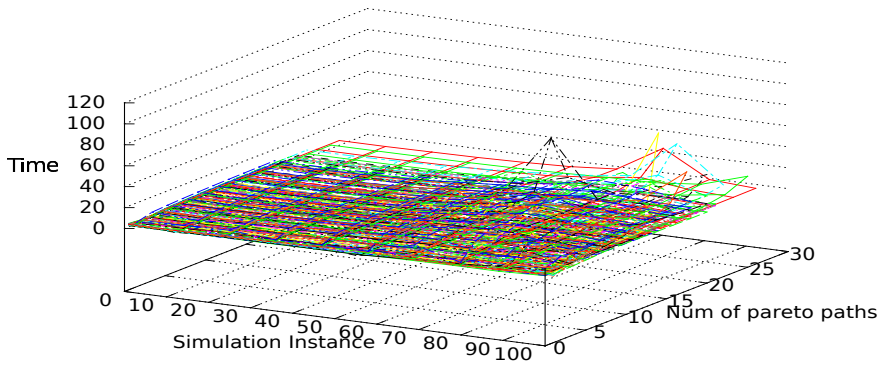


Figure 4.38: Running time vs number of pareto paths for N = 400, Fixed Link Cost, Infinite Threshold Delay

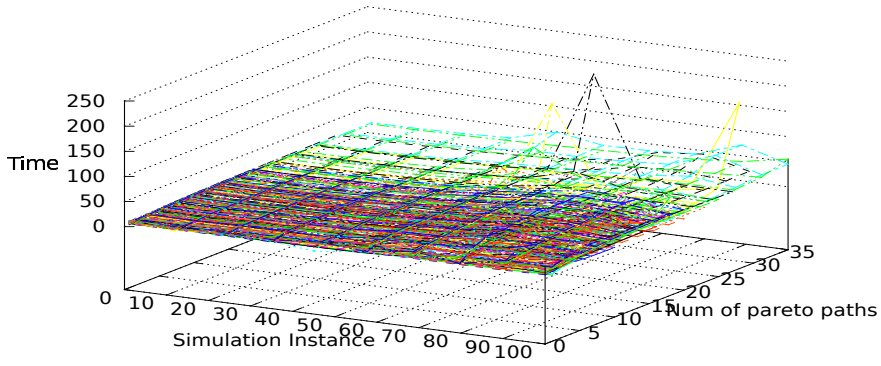


Figure 4.39: Running time vs number of pareto paths for N = 500, Fixed Link Cost, Infinite Threshold Delay

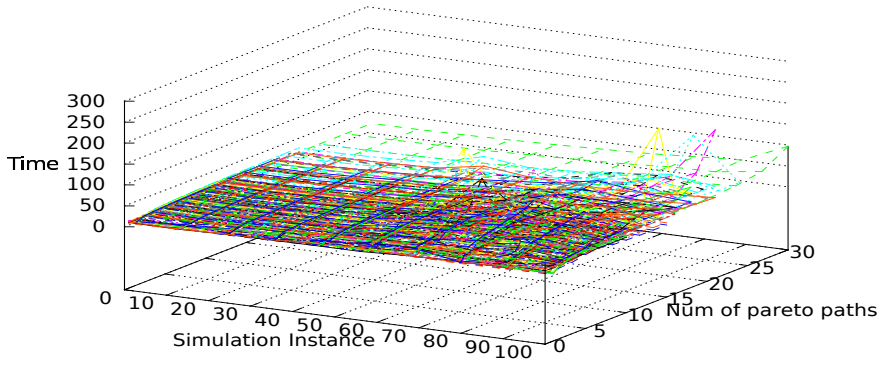


Figure 4.40: Running time vs number of pareto paths for N = 600, Fixed Link Cost, Infinite Threshold Delay

### 4.5.3 Model 3: No Threshold Delay and Cost negatively correlated to Delay

The third model is useful in two ways. From an experimental point of view, this is one of the ways of increasing the number of pareto paths using the underlying graph with the same delay and average nodal degree. From an applications point of view, taking the analogy of ISPs, a link with the maximum delay would likely represent an edge which spans across continents/countries and the low cost would be amortized by the amount of traffic going through it; taking the analogy of airlines, a link with the maximum delay would likely represent a long distance flight and the low cost would be amortized by the full utilization, while the small delays would likely represent a short connecting flight and the high cost would likely compensate for any under utilization. In the network domain, the base cost represents per Mbps and per unit time. For the airline domain, the equivalent could be per person and per unit time spent in the flight.

Figures 4.41 to 4.46 plot the distribution of the number of pareto paths over the simulation instance for N = 100 to N = 600 respectively. Figures 4.47 to 4.52 plot the distribution of the number of time instances. Figures 4.53 to 4.58 plot the distribution of average hop count of the pareto paths vs the running time (in seconds) of the algorithm. Figures 4.59 to 4.64 plot the distribution of number of time instances searched to find the pareto paths vs the running time (in seconds) of the algorithm. Figures 4.65 to 4.70 plot the distribution of number of pareto paths vs the running time (in seconds) of the algorithm. We observe that these plots show the same behavior wrt the plots for model 2, the only difference being the slightly higher running time and the higher number of pareto paths found.

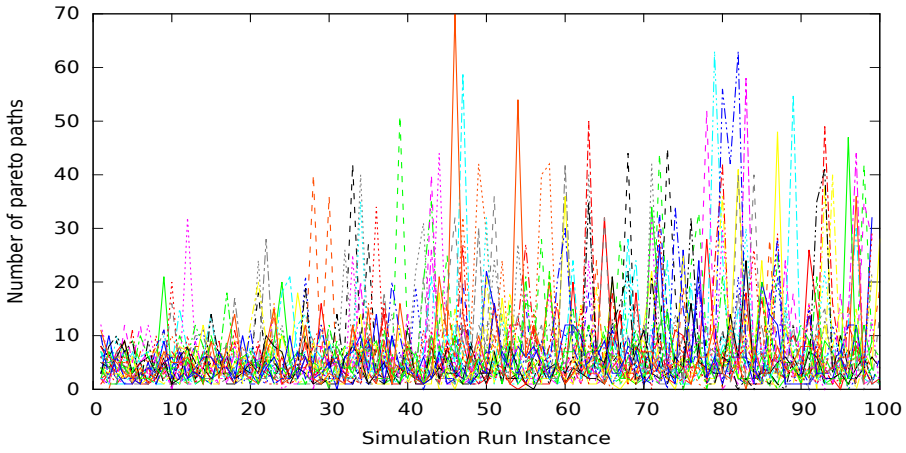


Figure 4.41: Pareto Paths Distribution for N = 100, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

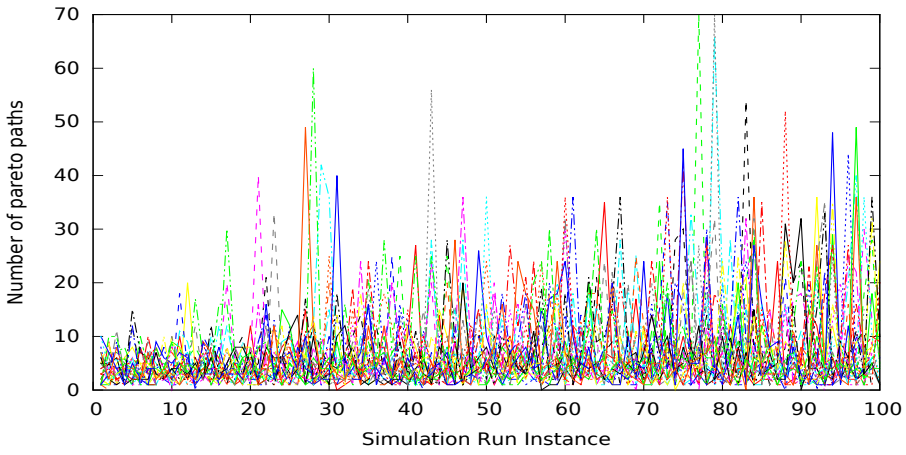


Figure 4.42: Pareto Paths Distribution for N = 200, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

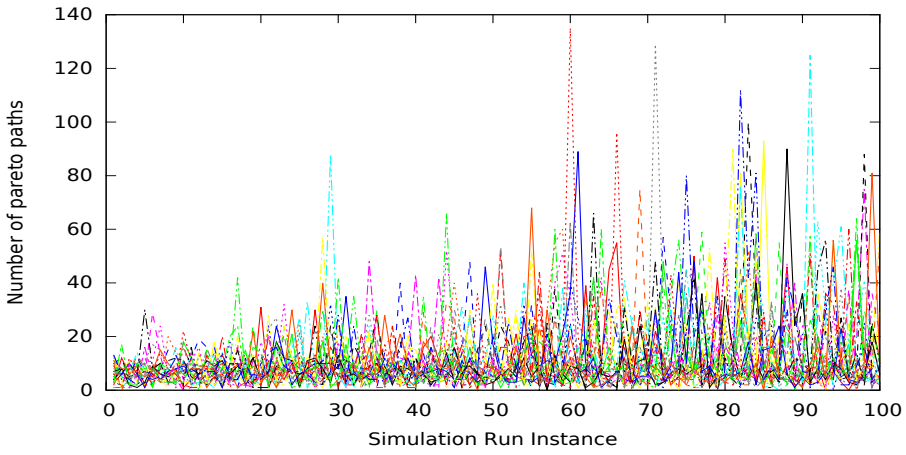


Figure 4.43: Pareto Paths Distribution for N = 300, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

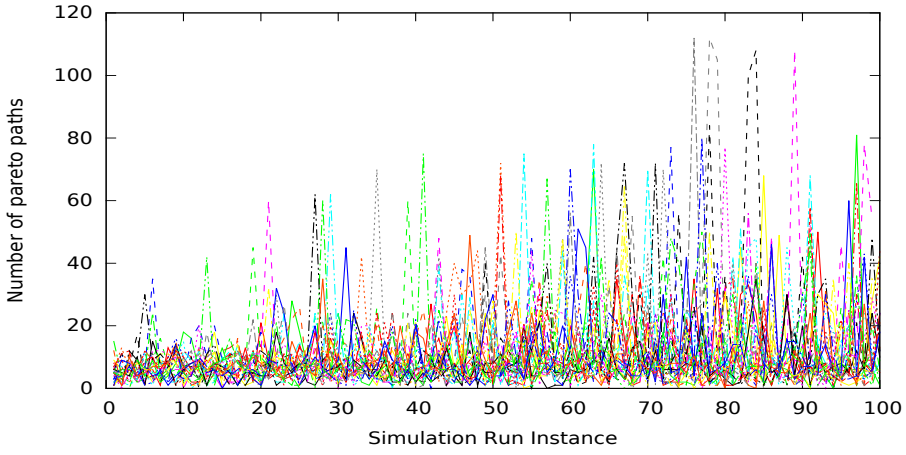


Figure 4.44: Pareto Paths Distribution for N = 400, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

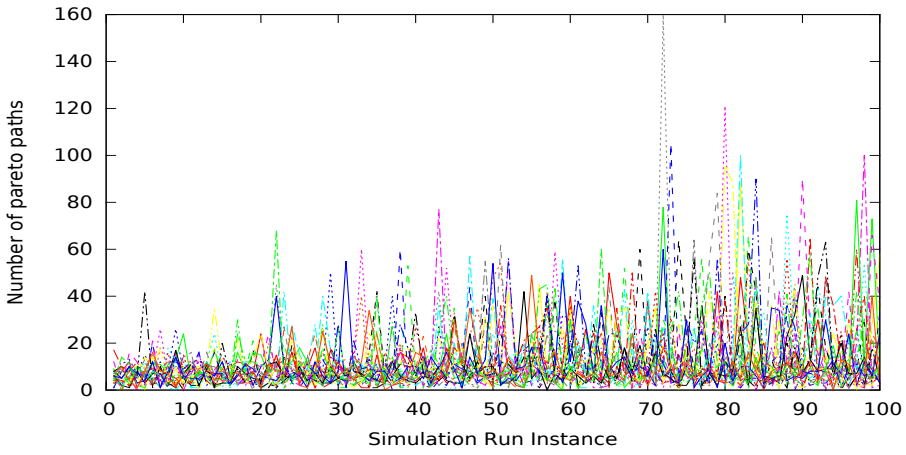


Figure 4.45: Pareto Paths Distribution for N = 500, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

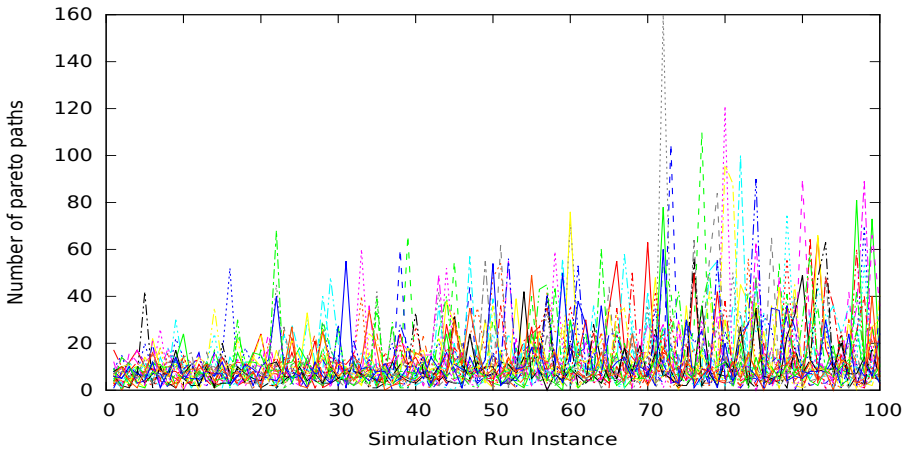


Figure 4.46: Pareto Paths Distribution for N = 600, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

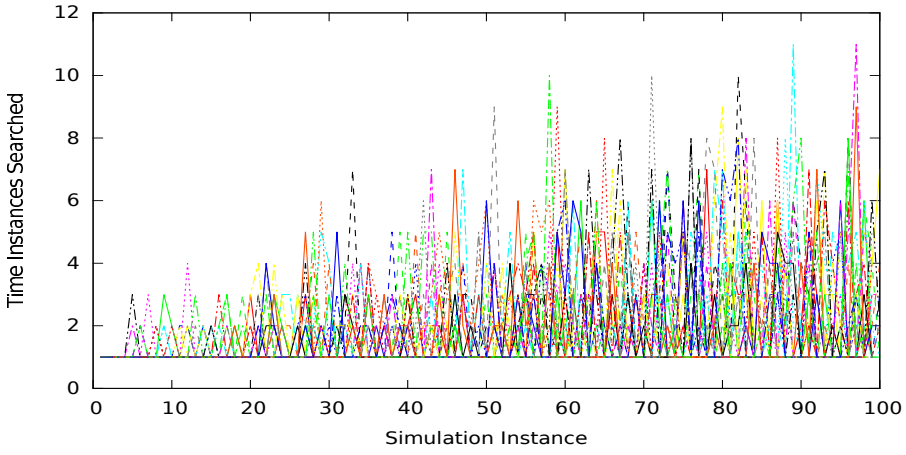


Figure 4.47: Time Instances Distribution for N = 100, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

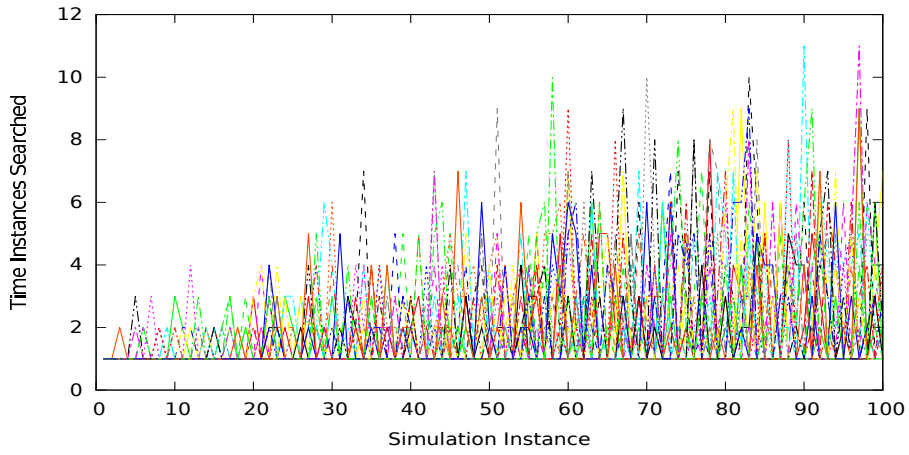


Figure 4.48: Time Instances Distribution for N = 200, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

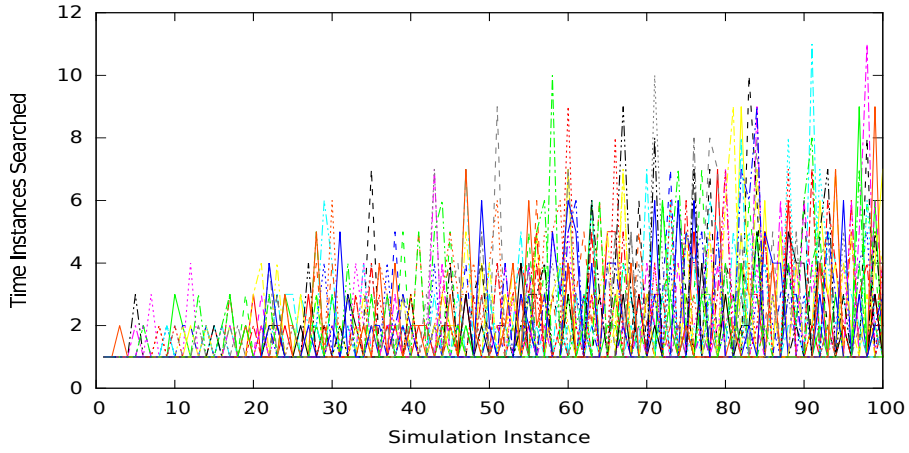


Figure 4.49: Time Instances Distribution for N = 300, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

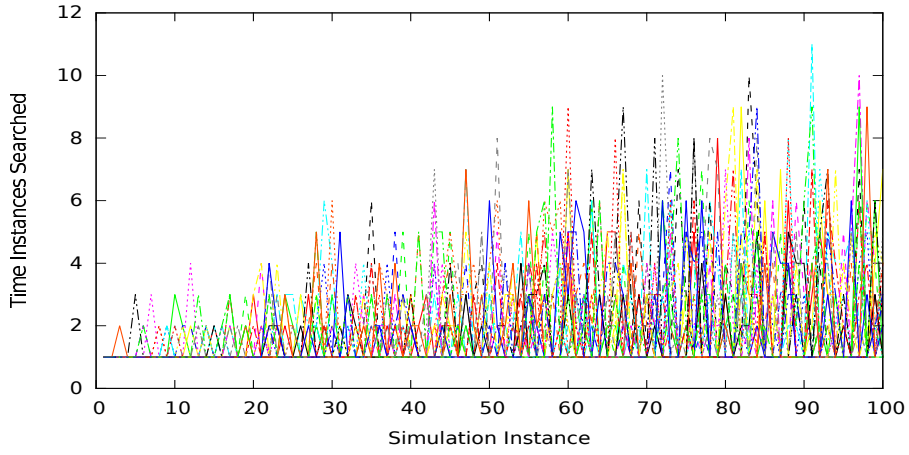


Figure 4.50: Time Instances Distribution for N = 400, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

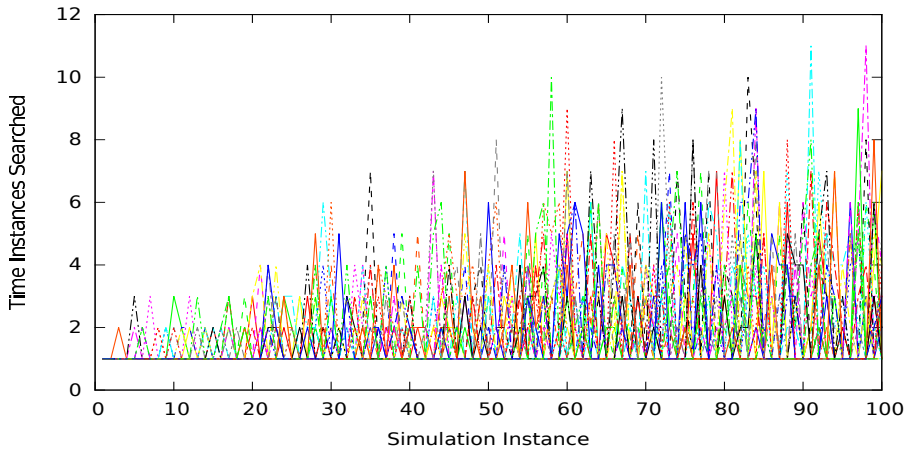


Figure 4.51: Time Instances Distribution for N = 500, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

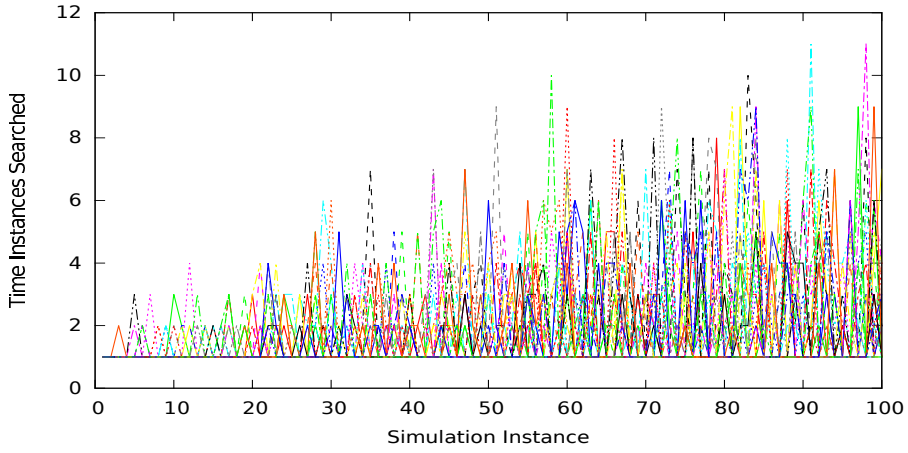


Figure 4.52: Time Instances Distribution for N = 600, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

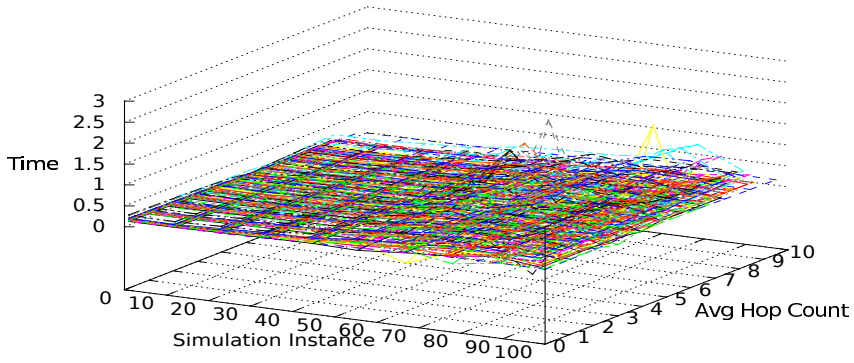


Figure 4.53: Running time vs average hop count for N = 100, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

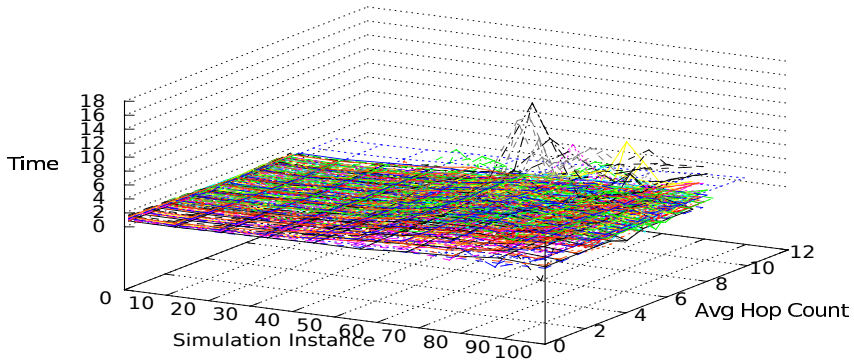


Figure 4.54: Running time vs average hop count for N = 200, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

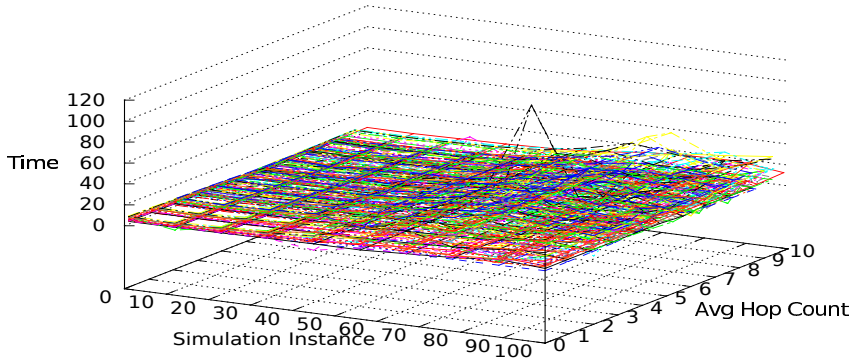


Figure 4.55: Running time vs average hop count for N = 300, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

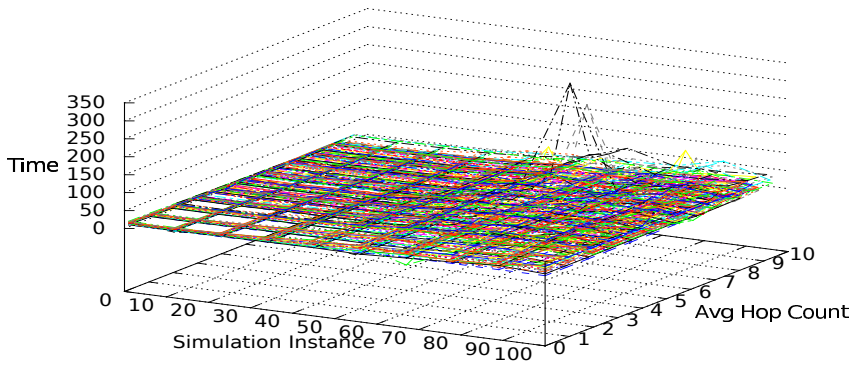


Figure 4.56: Running time vs average hop count for N = 400, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

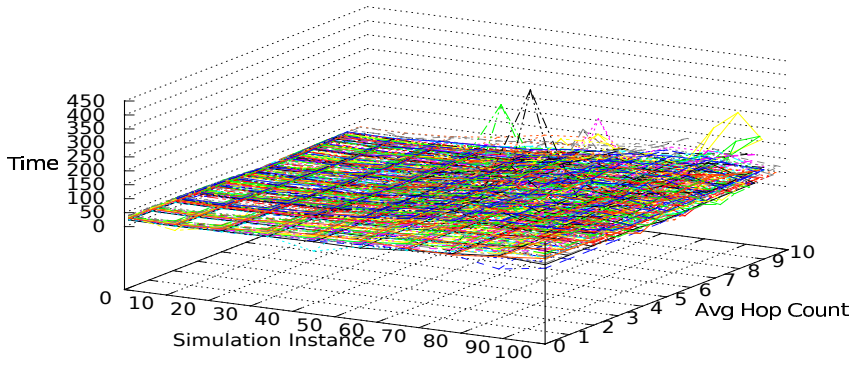


Figure 4.57: Running time vs average hop count for N = 500, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

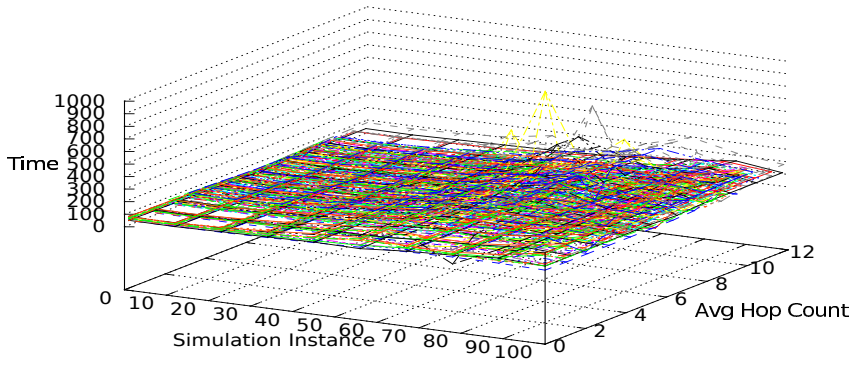


Figure 4.58: Running time vs average hop count for N = 600, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

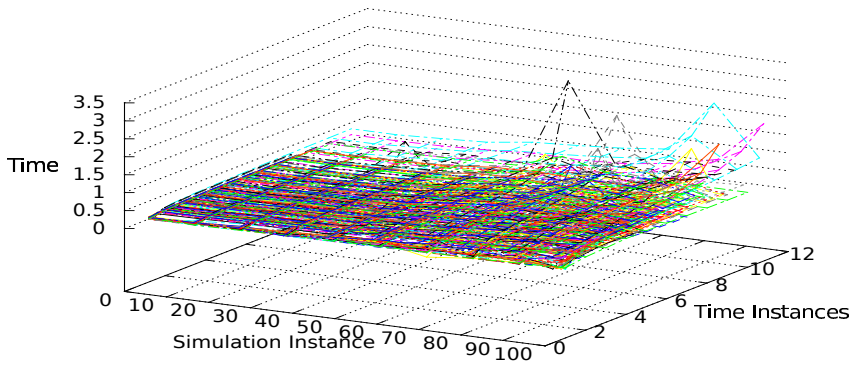


Figure 4.59: Running time vs number of time instances for N = 100, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

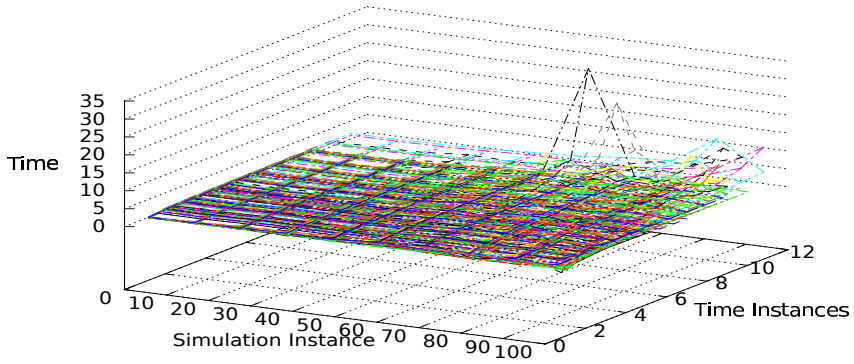


Figure 4.60: Running time vs number of time instances N = 200, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

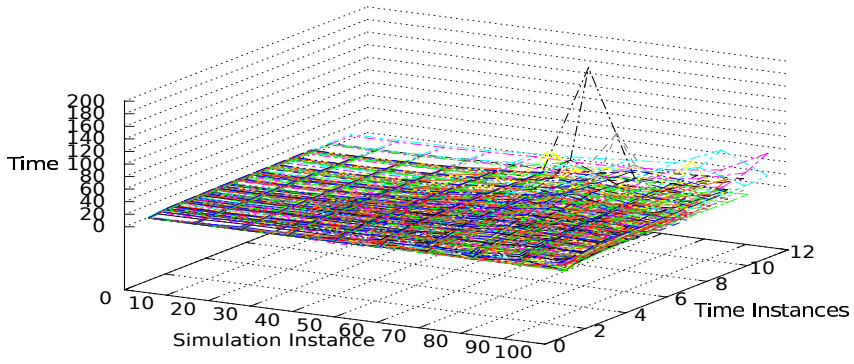


Figure 4.61: Running time vs number of time instances for N = 300, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

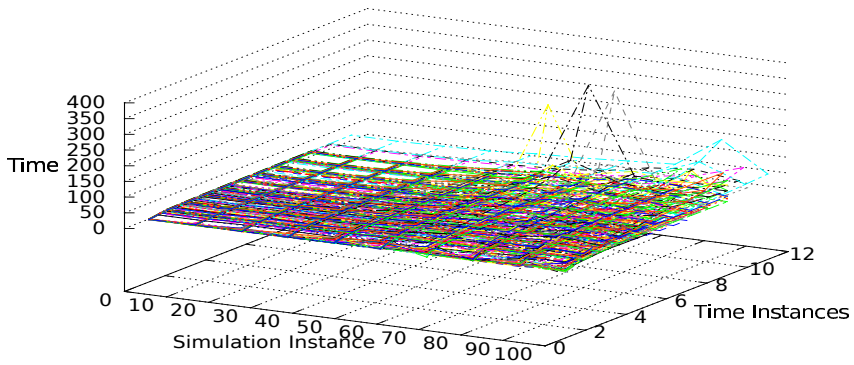


Figure 4.62: Running time vs number of time instances for N = 400, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

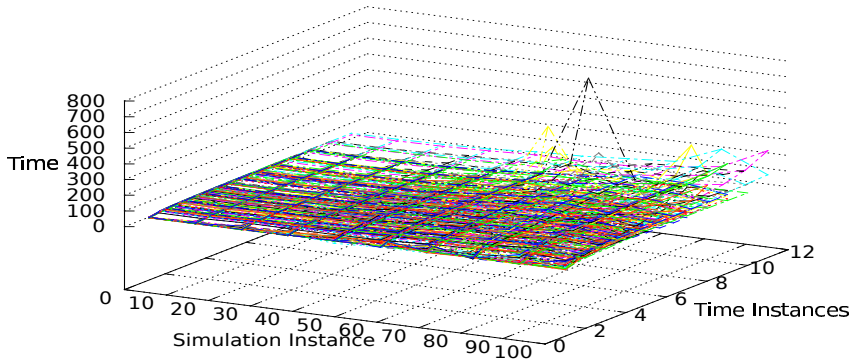


Figure 4.63: Running time vs number of time instances for N = 500, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

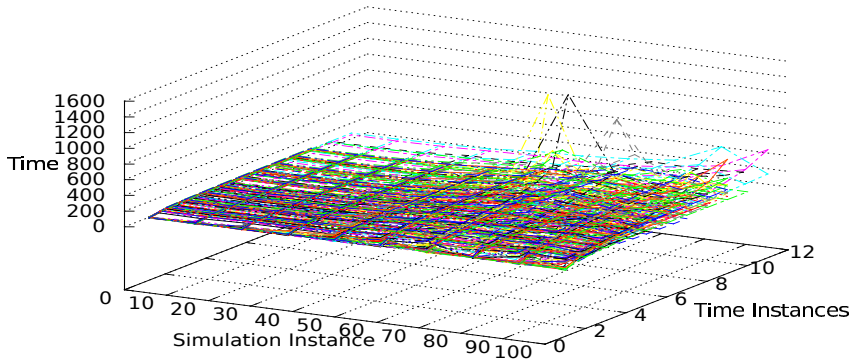


Figure 4.64: Running time vs number of time instances for N = 600, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

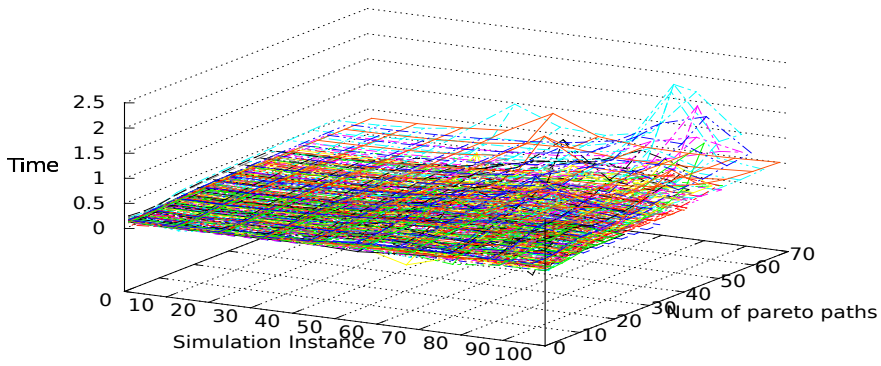


Figure 4.65: Running time vs number of pareto paths for N = 100, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

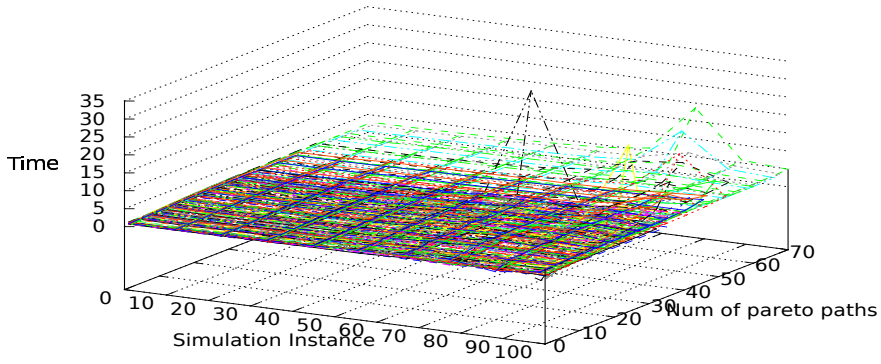


Figure 4.66: Running time vs number of pareto paths for N = 200, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

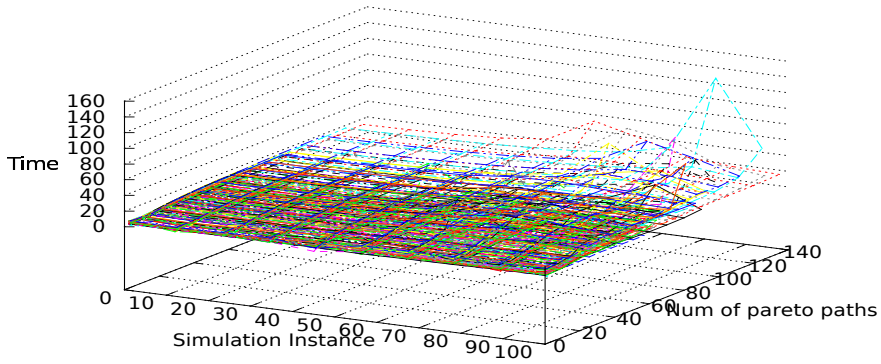


Figure 4.67: Running time vs number of pareto paths for N = 300, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

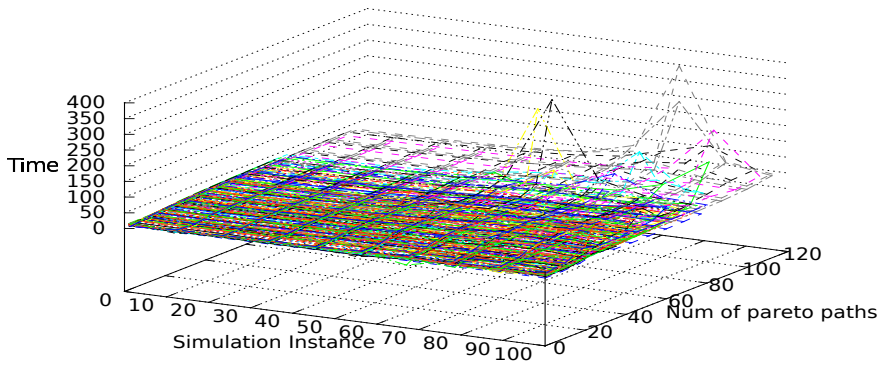


Figure 4.68: Running time vs number of pareto paths for N = 400, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

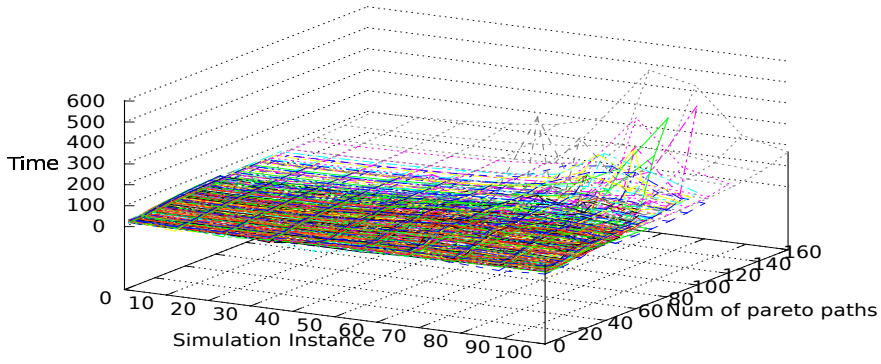


Figure 4.69: Running time vs number of pareto paths for N = 500, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

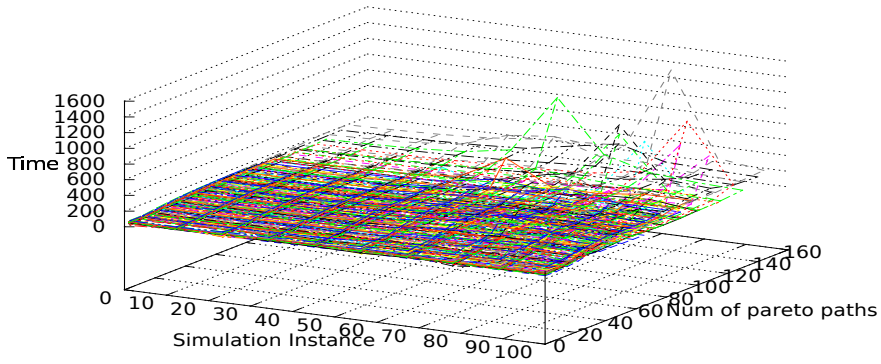


Figure 4.70: Running time vs number of pareto paths for N = 600, Cost $\propto$ 1/(Delay), Infinite Threshold Delay

### 4.5.4 Model 4: Threshold Delay and Cost negatively correlated to Delay

In this model, the link costs in the graph are negatively correlated to link delay and the user requests are modeled with finite and fixed threshold delay. We evaluate Problem 3 and present the results in Section 4.5.7.

### 4.5.5 Model 5: Three-criteria pareto paths

We extend the third model by adding another edge attribute "energy" consumption and evaluate Problem 2. In Figure 4.71 and in Figure 4.72 we plot the average number of pareto paths found and the average time it takes to find them and compare it with the two-criteria case presented in Model 3. Figures 4.73 to 4.78 plot the actual pareto solutions for N = 100 to N = 600. In Figure 4.72 we also plot the running time of a $O(N^3)$ function and a $O(N^4)$ function to compare the running time of the dynamic programming algorithm with 2-criteria and 3-criteria. We use the reference time for N = 100 to plot the extrapolated running times for N = 200 to N = 600. We expect a running time which is between $\Omega(N^3)$ and an exponential running time since we use a variation of Label Correction Algorithm and also use an adjacency matrix. Since the running time of the algorithm is slightly higher than a cubic function but well below a Quartic function we can claim that the algorithm has been efficiently implemented for both 2 and 3 criteria.

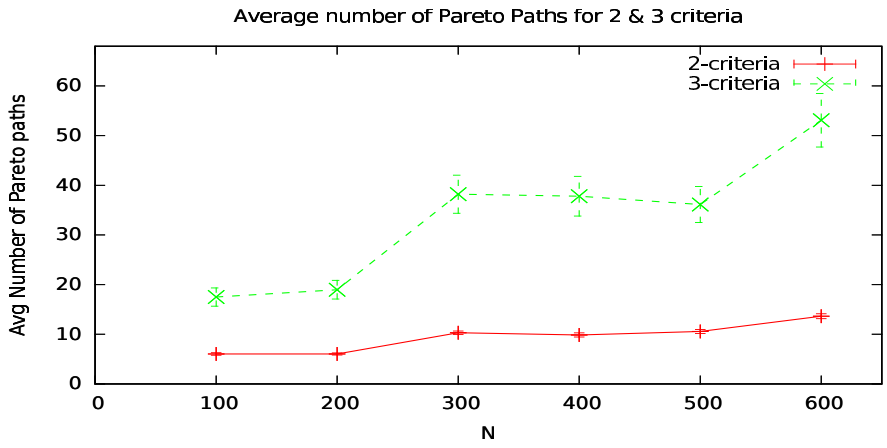


Figure 4.71: Average number of pareto paths found

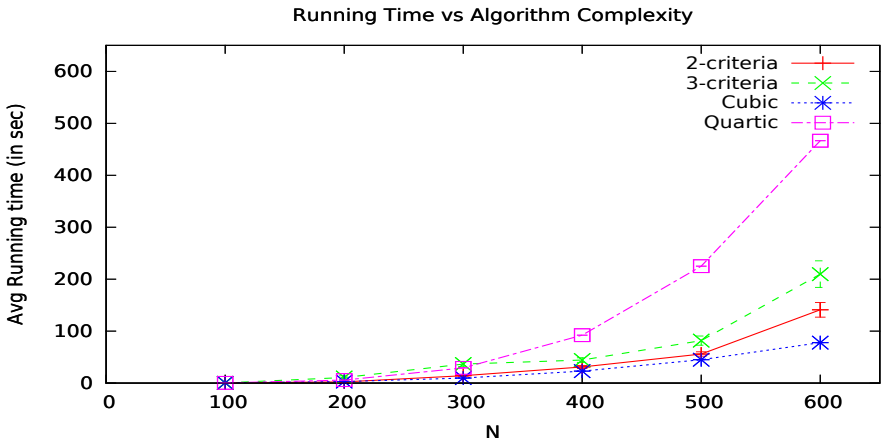


Figure 4.72: Average running time for the Dynamic Programming Algorithm

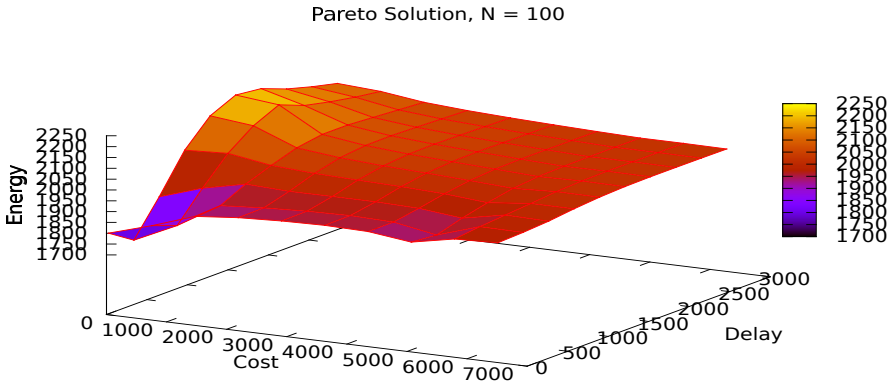


Figure 4.73: Pareto solution distribution for N = 100

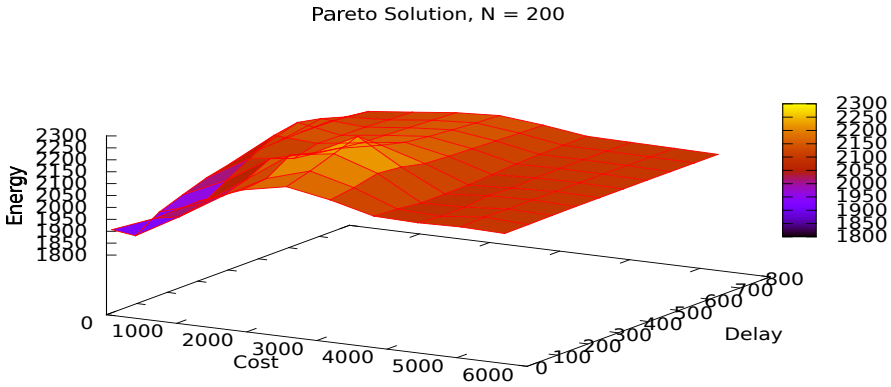


Figure 4.74: Pareto solution distribution for N = 200

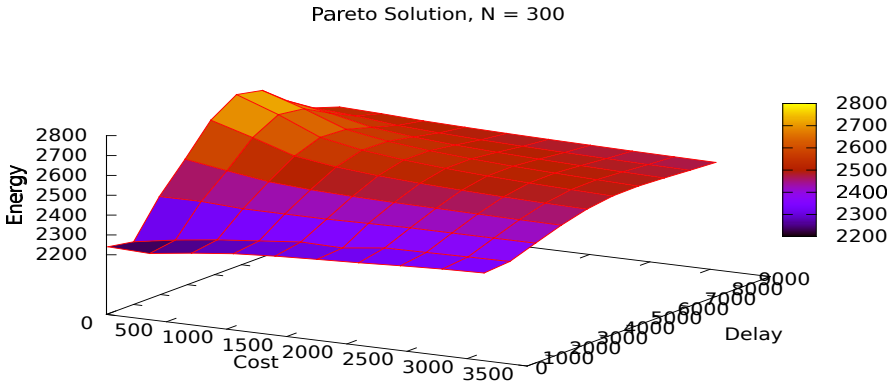


Figure 4.75: Pareto solution distribution for N = 300

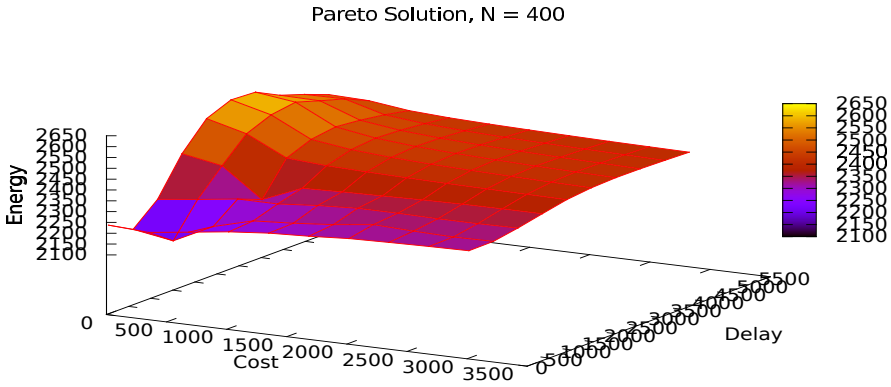


Figure 4.76: Pareto solution distribution for N = 400

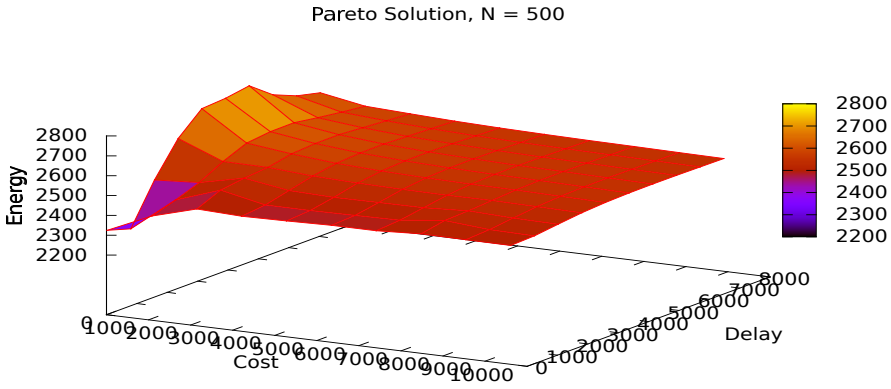


Figure 4.77: Pareto solution distribution for N = 500

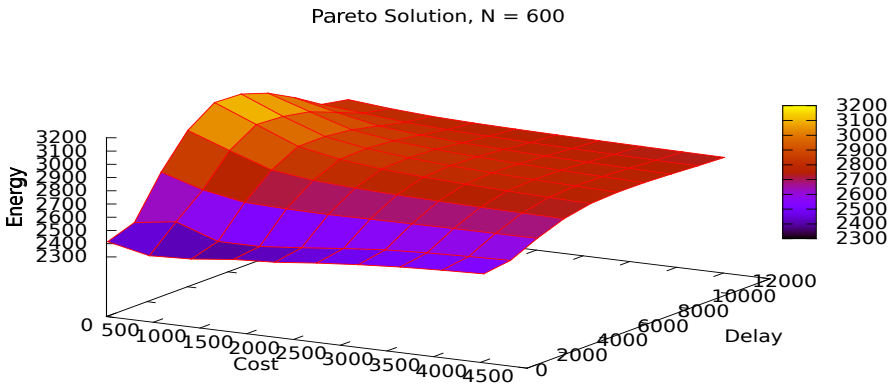


Figure 4.78: Pareto solution distribution for N = 600

### 4.5.6 Evaluation of Models 1, 2 and 3 for Problems 1 and 2

We observe that the running time of the algorithm is significantly higher for the third model compared to the first and second models. This is mainly due to more non dominant paths being stored at the intermediate nodes in the graph. We observe that the number of pareto paths has some influence on the running time of the algorithm, but the major factor influencing the running time of the dynamic programming algorithm which doesn't assume positive link attributes is N, the size of the graph.

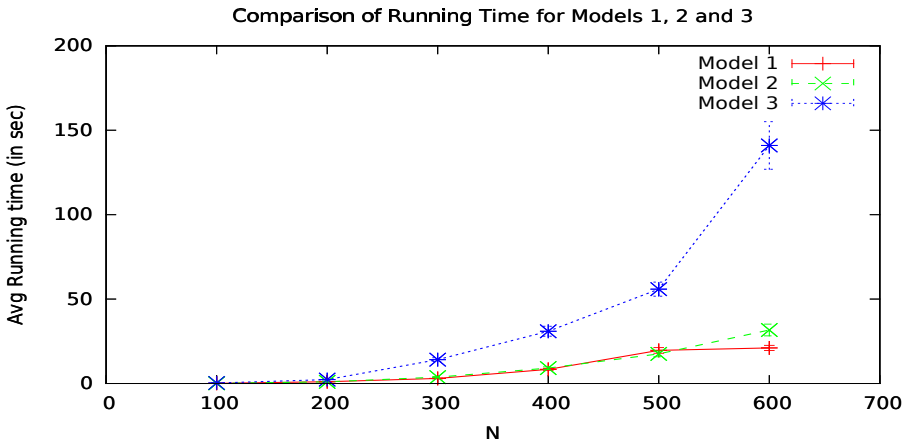


Figure 4.79: Running time as a function of N

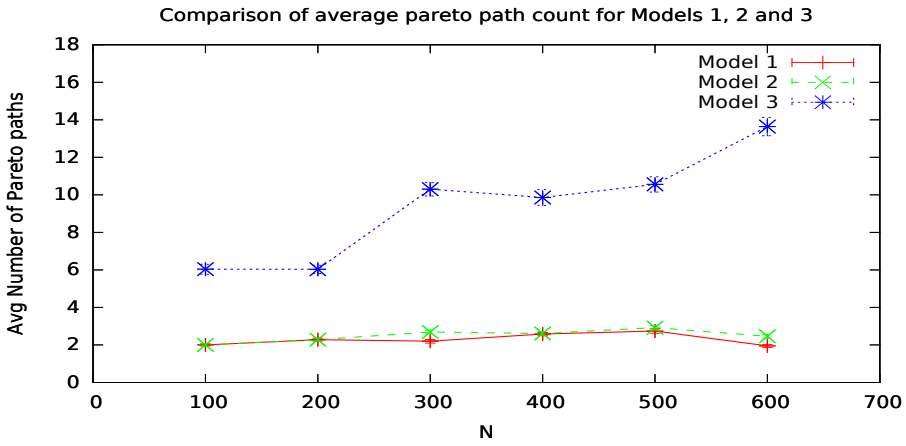


Figure 4.80: Avg number of pareto paths as a function of N

### 4.5.7 Evaluation of Models 1, 2, 3 and 4 for Problems 3 and 4

The average running time, the actual paths found when searching for $K$ paths and the pareto paths found among them, per user request with 95% confidence interval for 30 simulation runs with each simulation run consisting of 100 user requests are shown in Figure 4.81, Figure 4.82, and Figure 4.83 respectively. The figures highlight the difference between models 1 and 2.

Figure 4.84, Figure 4.85, and Figure 4.86 highlight the difference between models 3 and 4.

The average number of pareto paths among the $K$ shortest paths for $K = 5$, and $K = 10$, is small for both types of problems, but marginally higher for the problem instance without delay constraints as seen in Figure 4.81 and Figure 4.84. The difference between the two models is small to draw any definite conclusions, but we can infer that the finite threshold delay might lead to paths which are clustered around this threshold leading to paths which are very similar and also lesser in number giving a smaller number of pareto paths.

The average number of total paths is also smaller for model 1 compared to model 2 as seen in Figure 4.82 because of discarding some paths which exceed the threshold delay. This observation is consistent for model 3 and model 4 as seen in Figure 4.85.

The average running time for model 2 is higher compared to model 1 as seen in Figure 4.83, which is the result of relatively lesser paths being found which are below the threshold delay. The affect of a resource constraint i.e., threshold delay on the $K$ shortest cost path problem doesn't lead to an increase in time compared to the $K$ shortest cost problem without resource constraints for the graph instances considered in our experiments. This observation is also consistent with model 3 and model 4 as seen in Figure 4.86

The observation made while comparing the results for both the problems is influenced by the graph/user model. These observations will change for a different graph/user model.

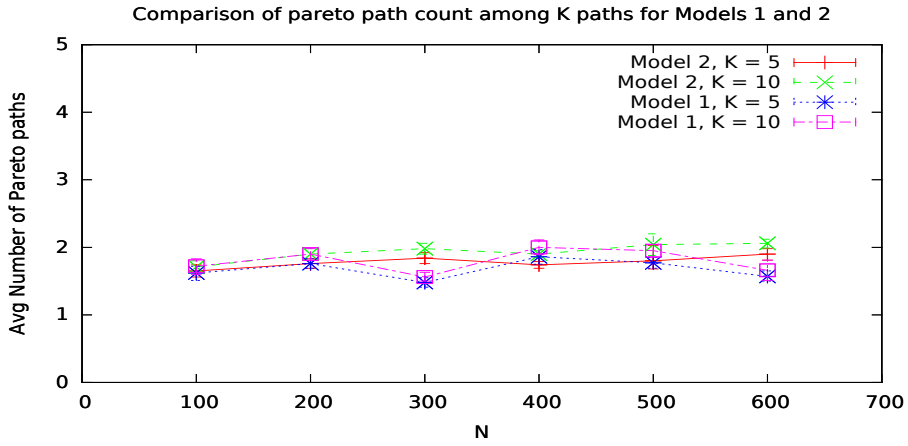


Figure 4.81: Pareto paths among K paths for Models 1 and 2

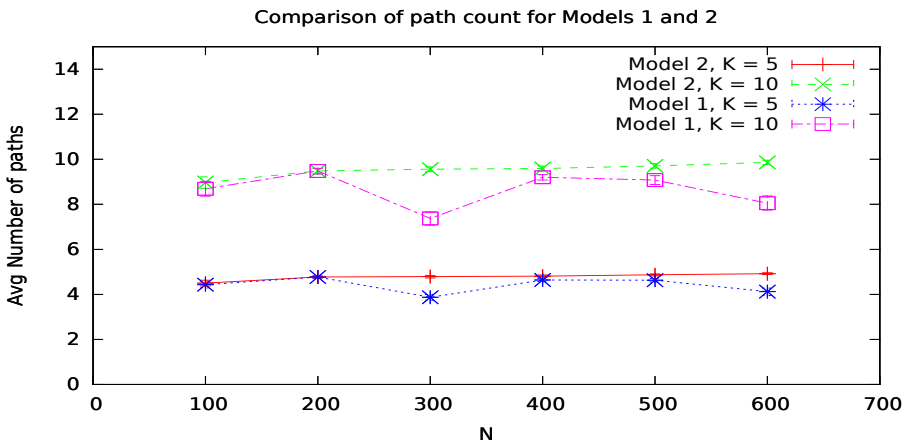


Figure 4.82: Total paths for Models 1 and 2

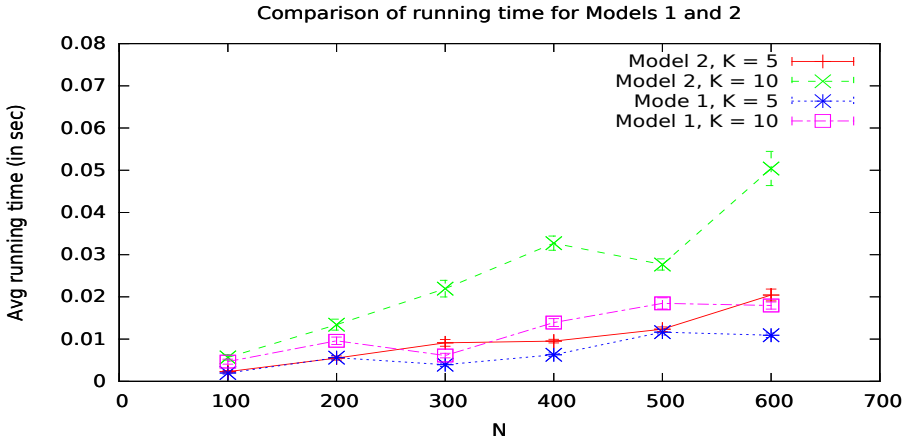


Figure 4.83: Running Time for Models 1 and 2

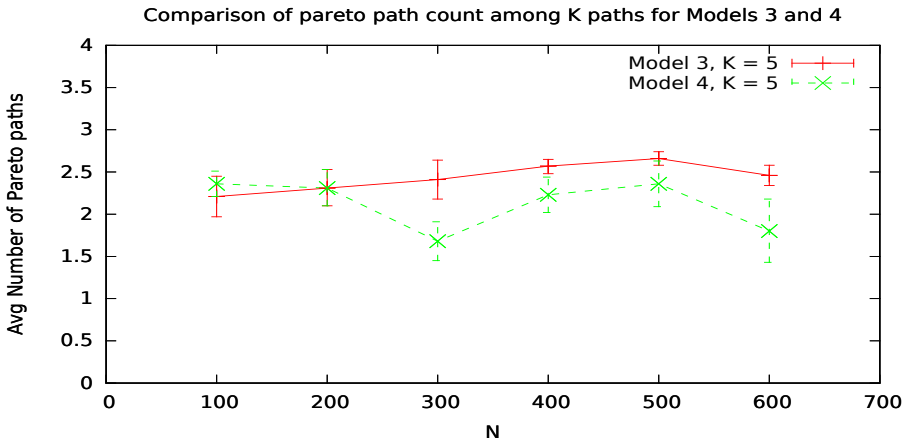


Figure 4.84: Pareto paths among K paths for Models 3 and 4

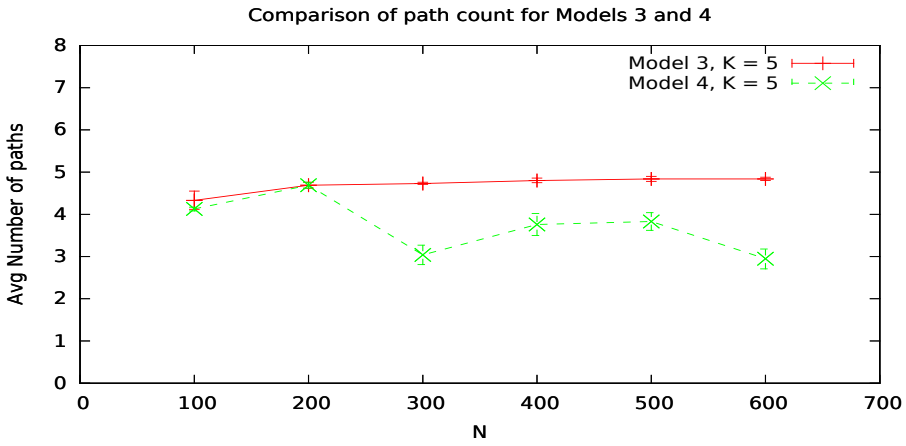


Figure 4.85: Total paths for Variation 3 and 4

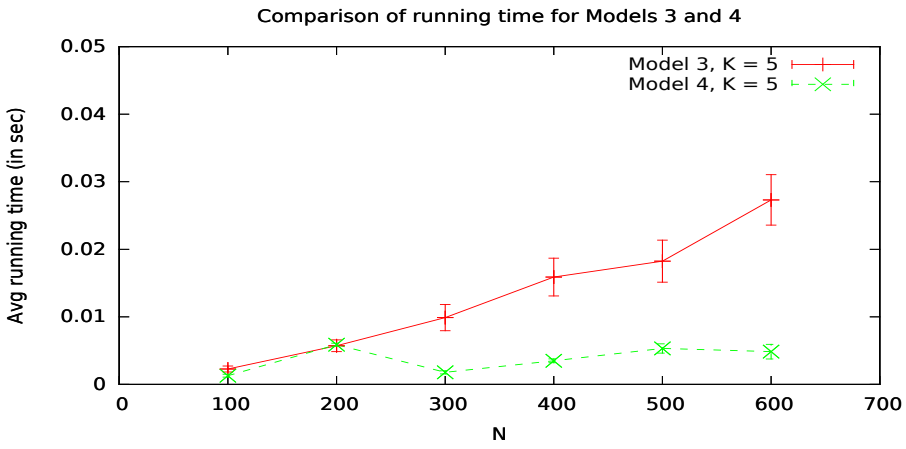


Figure 4.86: Running Time for Variation 3 and 4

# Chapter 5

# Service Routing Planner

The principles of ChoiceNet resonate in the domain of Network functions virtualization (NFV) [66]. NFV decouples the network service functionality from the underlying network, compute, and storage resources, and allows communication services to be composed by stitching together functional building blocks that may not be co-located and may be offered by different providers. Interest in NFV has grown dramatically over the past few years due to its perceived benefits to both service providers and the users of these services. One of the main challenges in realizing the potential of NFV relates to orchestration [66], i.e., the process of arrangement and coordination of multiple network services so as to deliver a desired functionality. The role of orchestration in the NFV architecture has been highlighted in previous works [67, 68] that have mostly focused on service abstraction, semantics, and the standardization of the APIs.

Whenever the NFV architecture spans networks operated by multiple distinct/competing network providers and encompasses service components that are geographically apart, orchestration requires (a) a marketplace of services, and (b) specialized routing algorithms. A marketplace [1, 69] acts as a ''service commons,'' a meeting ground where providers publish and advertise the services they offer, and users acquire services based on their requirements and have them instantiated on demand. The concept of a marketplace was also highlighted in our own ChoiceNet project [1] where we envisioned a broader set of network services but focused on introducing a network ''economy plane'' so as to boost competition among service providers. The marketplace is an essential component in opening up the network infrastructure [67] so as to develop value added services including service composition, fault-tolerance, load balancing, energy minimization, etc., by building upon more primitive virtual network function blocks.

Once the virtual services required by the user have been determined, user traffic must be steered along a path that starts at the source and visits the nodes where the virtual services are implemented, in the order in which they must be applied, before reaching the destination One variant of this routing problem, referred to as the ''node-constrained service chain routing

problem'' was studied in [70], and was solved by using a layered graph model on which conventional routing algorithms may be applied. Another variant, ''service function chaining,'' was considered in [71], and an algorithm that balances the length of the service function path and the load of service function instances was presented.

In this Chapter, we consider a general version of the service-concatenation routing problem in NFV environments where the objective is to construct a path that visits a set of nodes where virtual services are to be performed in a specific order. Specifically, our work makes the following contributions:

- We show that the service-concatenation routing problem in NFV may be modeled as a shortest path tour problem (SPTP), a problem that was first studied more than forty years ago in a different context [72, 73].
- We implement all existing algorithms for SPTP that we were able to find in the literature; some of these were originally developed for SPTP, whereas others were developed for related problems and we have modified or extended them to solve SPTP.
- We develop a new algorithm for solving SPTP that outperforms all SPTP algorithms that we are aware of.
- We carry out a comprehensive experimental study to evaluate the performance of all SPTP algorithms and identify their relative merits.

## 5.1 System Model

We consider an NFV environment spanning multiple network domains, possibly administered by distinct network providers. We assume that the NFV environment is shared by a set of service providers who compete against each other to offer network services to users. The network services may include path/routing, data storage, data modification, data analysis, and computation services. This is not an exhaustive list and may potentially be expanded to include services that do not fit into any of these categories, or services that will be developed to satisfy future demands. Service providers utilize NFV abstractions and APIs to deploy multiple instances of each service at strategic points in the network so as to better serve a geographically diverse population of users.

We further assume that the NFV architecture includes a marketplace [1, 69] as an integral component. The marketplace may be thought of as a repository of services and network functions that are available to users. The repository provides APIs for providers to publish (advertise) the services they offer, and for users (or agents acting on their behalf) to obtain lists of service offerings that are relevant to their requirements. To aid users in selecting network

services that best match their needs, the orchestration module of NFV uses a planner [74, 75]. In a travel industry analogy, service providers include the airlines, hotels, and rental car companies, whereas travel sites such as Expedia or Travelocity manage marketplaces that include planning and orchestration functions. These functions construct itineraries based on traveler (user) requirements and ensure that users may access seamlessly all the services acquired across the various flight, accommodation, and car rental providers.

An NFV marketplace planner has two main tasks [74, 75]:

- it determines the set of services/virtual functions to meet a user's requirements, and the order in which these services are to be applied to the user's traffic, and
- it constructs a path from source to destination that visits virtual nodes where instances of these services/virtual functions have been deployed.

Some of our earlier work [76, 77] demonstrates a complete lifecycle of the network services on a GENI slice [78], starting with how the network services are described using a semantics language and advertised in a marketplace, followed by how the services are purchased/acquired leading up to their instantiation, and finally how the services are used by the user. In this work, we focus on the second task above, where the objective is to direct user traffic to virtual instances of the service functions that must be applied. Note that it is the concatenation of the services in the order specified that accomplishes the functionality that meets the user's requirements. Therefore, we refer to this problem as "service-concatenation routing," and we define it as:

> *Given an ordering of a set of services, construct a path of minimum cost from source to destination, that traverses nodes where virtual instances of these services reside and may be applied in the specified order.*

The services repository of the NFV marketplace may use any convenient format or data structure to represent the services offered by the various providers. Nevertheless, we assume that the service information stored in the marketplace may be represented in a graph format that makes it possible to apply graph algorithms to solve the service-concatenation routing problem. This graph representation may be maintained internally by the marketplace itself and made available to the planner. Alternatively, the marketplace may provide appropriate APIs that allow external services to repeatedly query the repository so as to construct the graph of services, as we consider in [74]. In the latter case, planners may be offered as competing services external to the marketplace. Either way, we expect that the graph will be highly dynamic in that it will have to be updated every time users acquire new services, or release services they no longer need.

Note that the planner of a travel site takes into consideration flights from multiple airlines, many of which offer competing flights between the same pairs of cities, as well as multiple

hotels or rental car facilities within a given city. Similarly, the planner of an NFV marketplace must consider virtual services/functions from multiple providers, including virtual operators who may lease capacity from the same physical infrastructure. Consequently, the planner takes as input a topology that is a superset of the topologies representing the underlying networks. In particular, nodes and edges in the topology represent virtual entities rather than physical ones. For instance, a physical node may include multiple virtual nodes, each virtual node operated by a different service provider deploying a variety of virtual function instances. The graph may also include parallel edges between nodes that represent competing path services. Such a topology is expected to be significantly larger than the underlying physical network topology, hence path finding algorithms must scale to large graph sizes.

As a final note, we assume that the planner has knowledge of the complete topology (graph) of virtual nodes and services, and uses it to solve the service-concatenation routing problem by applying a path finding algorithm. If the NFV architecture is deployed in a software defined networking (SDN) environment, the planner may be implemented as an application of the SDN controller and use the latter's capabilities to construct and maintain this topology. However, our work does not require an SDN environment and applies to any architecture in which the planner has the means to discover and update the complete topology graph.

## 5.2 The Shortest Path Tour Problem (SPTP)

Consider the SPTP problem first studied in [72, 73]:

**Problem 5 (SPTP)** *Given*

- *a graph* $G = \{\mathcal{N}, \mathcal{E}\}$ *where* $\mathcal{N}$ *is the set of nodes and* $\mathcal{E}$ *is the set of edges,*
- *a source node* $s$ *and a destination node* $d$, $s, d \in \mathcal{N}$, *and*
- $K$ *non-empty ordered node sets* $S_1, S_2, \ldots, S_K$, *such that* $S_i \subset \mathcal{N}, i = 1, \ldots, K$,

*find the shortest path from* $s$ *to* $d$ *under the constraint that the path visit one node* $n_i \in S_i$ *of every set* $S_i, i = 1, \ldots, K$, *in the given order, i.e.,* $n_1, n_2, \ldots, n_K$.

We note that whenever each set $S_i$ is a singleton (i.e., $S_i = \{n_i\}, i = 1, \ldots, K$), SPTP reduces to loose source routing as originally specified by the IP protocol [79]. Similarly, whenever there is exactly one node set (i.e., $K = 1$), SPTP becomes similar to anycasting [80].

Recall now the service routing problem we introduced in the previous section, and let $K$ denote the number of virtual services that must be applied to the user's traffic. Without loss of generality, assume that the virtual services are labeled $1, 2, \ldots, K$, in the order in which they

must be applied. Finally, let $S_i, i = 1, \ldots, K$, denote the set of nodes where instances of virtual service $i$ reside. Since a path that solves SPTP visits a node for each virtual service, and in the order in which services must be applied, and is the minimum-cost one among all such paths, then it is also a solution to the service-concatenation routing problem defined in the previous section.

Several variants of SPTP have been studied in the literature. The constrained shortest path tour problem (CSPTP) [81] is defined as SPTP with the additional constraint that the path not include repeated edges; whereas SPTP is solvable in polynomial time, this constraints makes the problem NP-Hard. Another variant arises in travel planning applications [61], whereby there exist additional constraints related to the minimum amount of time that a traveler must stay at each node (city). The introduction of such constraints to SPTP converts the problem from polynomial time solvable to pseudo-polynomial [82]. A related problem whose objective is to find the shortest elementary path that visits all nodes in a set $S$ in an arbitrary order (i.e., the input does not include a fixed order on the nodes to be visited) is NP-Complete (NPC) [83]. Relaxing the previous problem to include paths which are not elementary still places the problem in class NPC [83]. Variants of SPTP have also been defined under the class of vertex constrained shortest path (VCSP) problems [84].

## 5.3 Algorithms for SPTP

We now consider the basic SPTP problem we defined in the previous section, and we review and classify all existing algorithms for the problem that we were able to find in the literature. We also present a new algorithm for SPTP that, as we will show later, outperforms earlier algorithms.

### 5.3.1 Path Tour Decomposition

Let us define $S_0 = \{s\}$ and $S_{K+1} = \{d\}$. It has been observed that SPTP may be decomposed into $K+1$ sub-problems, such that the $k$-th sub-problem, $k = 0, \ldots, K$, consists of constructing shortest paths from each node in $S_k$ to each node in $S_{k+1}$.

When SPTP first appeared in the literature [72, 73], it was applied to telephone and transportation networks with large, sparse topologies. Consequently, single source shortest path (SSSP) algorithms were used to solve the SPTP sub-problems. More recently, SPTP has found applications in warehouse management and control of robot motions [85, 86], where the graphs are small but dense. Therefore, researchers and developers have adopted all pair shortest path (APSP) algorithms to solve the sub-problems of SPTP, as these are more efficient for this type of graphs.

A third option for solving each subproblem of SPTP is to apply algorithms for the multiple pairs shortest path (MPSP) problem. MPSP [87--89] has a range of applications, from multi-commodity network problems to airline network problems, and is concerned with computing shortest paths for a subset of all node pairs in the network. By using algebraic shortest path algorithms [87--89], it is possible to reduce significantly unnecessary computations of either APSP algorithms (which construct paths for all node pairs) or SSSP algorithms (which must be executed multiple times, once with each node as the source node).

Therefore, we have three types of decomposition (DC) algorithms for SPTP:

- *DC-APSP:* The algorithm presented in [86] uses APSP to solve each subproblem of SPTP.
- *DC-MPSP:* Although to the best of our knowledge there has been no algorithm for SPTP that uses MPSP for the subproblems, based on our observations above, we have implemented two such algorithms:
  - *DC-MPSP-1:* This implementation uses the MPSP algorithm in [87] to compute each sub-path of the shortest tour between the source and destination nodes.
  - *DC-MPSP-2:* In this version, we apply the MPSP algorithm in [88] at each intermediate stage[1].
- *DC-SSSP:* We have implemented two algorithms that use SSSP:
  - *DC-SSSP-1:* This is a straightforward application of Dijkstra's algorithm to find shortest paths from every node of $S_k$ to every node of $S_{k+1}$. This algorithm is similar to the one employed in [71] in the context of virtual network function deployment across datacenters, and has also been discussed in [87].
  - *DC-SSSP-2:* The algorithm in [73] also uses SSSP at each stage. It differs from the straightforward algorithm DC-SSSP-1 in that it considers a virtual node $v$ that connects to each node in $S_k$ with zero-cost edges, and applies Dijkstra's algorithm to find the shortest paths from $v$ to each node in $S_{k+1}$. Hence, it is more efficient since it makes only one call to Dijkstra's algorithm in each stage.

Let $T[]$ be a $(K+2) \times N$ array such that $T[k,n]$ denotes the cost of the shortest path tour from the source node $s$ to a node $n \in S_k$; this quantity is equal to infinity if $n \notin S_k$. Also, let $D(i,j)$ denote the cost of the shortest path from node $i$ to node $j$ in the network graph. Then, the dynamic programming pseudocode of Algorithm 3 describes the operation of a generic decomposition algorithm for the SPTP problem; the only algorithm-specific operation

[1]We have also implemented the MPSP algorithm in [89] for the SPTP, but it is significantly less efficient than DC-MPSP-2 and hence we do not consider it in this study.

is the computation of the cost $D(i, j)$ of the shortest path between nodes $i$ and $j$, which may be based on the APSP, MPSP, or SSSP algorithms.

---

**Initialization:**
$T[k, n] = \infty, k = 0, \ldots, K + 1, \ \forall\ n \neq s$
$T[0, s] = 0$
**for** $k = 0, \ldots, K$ **do**
  **for** $i \in S_k$ **do**
    **for** $j \in S_{k+1}$ **do**
      $D(i, j)$ = cost of shortest path from $i$ to $j$ using APSP, MPSP, or SSSP algorithms
      $T[k + 1, j] = \min\{T[k, i] + D(i, j), T[k + 1, j]\}$
    **end**
  **end**
**end**

**Algorithm 3:** Generic decomposition algorithm for SPTP

---

### 5.3.2 Layered Graph Model

A different approach that has been used in the literature for tackling SPTP is to augment the network graph in a way that makes it possible to apply conventional shortest path algorithms to construct the path tour of minimum cost between the source and destination nodes. Specifically, the studies in [70, 85] create a layered graph of $K + 1$ layers, each layer consisting of an exact copy of the original network topology. Nodes in adjacent layers are connected with new edges such that any path from the source node (at the lowest layer) to the destination node (at the highest layer) satisfies the path tour constraints. Then, an application of Dijkstra's algorithm is sufficient to determine the minimum-cost path tour.

We have, therefore, implemented this algorithm:

- *LG:* The algorithm described in [85] to solve SPTP on a layered graph; a similar layer graph model is also discussed in [70], although an algorithmic description is not provided.

### 5.3.3 Depth First Tour Search: A New Algorithm for SPTP

We now present a new algorithm for the SPTP problem that eliminates the exploration of nodes in the graph (with respect to computing shortest path to them), whenever such exploration is determined that it will not lead to a better path tour. As a result, our algorithm is quite effi-

cient, and we will present simulation results to demonstrate that it outperforms the algorithms discussed above.

Our algorithm operates similar to Dijkstra's algorithm, but with important enhancements and modifications to make it more efficient and ensure that the SPTP constraints on the path tour are satisfied. The algorithm does not decompose SPTP in subproblems, nor does it employ a layered graph; it operates on the given network graph without modifying it. Specifically, it starts with the source node $s$ and explores nodes using the same criteria as Dijkstra's algorithm, until it reaches the destination node; at that time, the algorithm is guaranteed to have found the shortest path tour that solves the given instance of SPTP. Unlike Dijkstra's algorithm that maintains a single set of encountered nodes (i.e., nodes for which the shortest path from the source has been determined and will not change in the future), our algorithm maintains $K+1$ sets $F_i, i = 1, \ldots, K+1$, of encountered nodes: the first $K$ sets $F_i, i = 1, \ldots, K$ are associated with reaching nodes in the $K$ sets $S_i$, respectively, and the last set is for reaching the destination node $d$. Therefore, a node $x$ may be in one or more sets $F_i$ depending on which part of the tour it has been encountered; for instance, $x$ may be encountered as part of one tour from $s$ to the first set $S_1$, but it may also be encountered as part of the same or another tour from $S_1$ to $_2$.

The operation of the algorithm may be summarized as follows:

1. Initially, all the encountered sets are initialized to $\emptyset$ except $F_1$ which is initialized to contain the source node, i.e., $F_1 = \{s\}, F_i = \emptyset, i = 2, \ldots, K+1$.

2. At each iteration $l$ of the algorithm, the node $x$ with the minimum cost is selected. Unlike Dijkstra's algorithm, node $x$ is selected among all the nodes that have not been encountered as part of at least one set $F_i$. This operation is implemented efficiently by maintaining $K+1$ heaps, each associated with one of the tour stages, and then selecting the minimum cost node among all the heaps. Also note that this feature allows the algorithm to make forward progress towards the destination by continuing towards node set $S_{i+1}$ without waiting for all nodes in node set $S_i$ to be explored first.

3. Our implementation keeps track of which part of the tour node $x$ has been encountered, such that if it is part of the tour from set $S_i$ to set $S_{i+1}$, then node $x$ will now be included in set $F_{i+1}$. Also, the cost of each neighbor $y$ of $x$ is updated appropriately (i.e., as in Dijkstra's algorithm), as long as $y$ has not been encountered as part of at least one set $F_i$.

4. If node $x$ is the last node of some set $S_i$ to be explored (i.e., partial tours that reach all nodes in $S_i$ have now been constructed), then we disregard any partial tours that have only reached nodes in sets $S_{i-1}, \ldots, S_1$. Any such partial tours will have higher cost once extended to reach nodes in $S_i$, hence they cannot be part of the shortest path tour.

5. The algorithm iterates from Step 2 above, until the destination node $d$ has been reached.

Since this algorithm makes progress towards the destination beyond a set $S_i$ without waiting until all nodes of that set have been explored, it bears some similarities with depth first search; hence, we will call our algorithm depth first tour search (DFTS)[2].

Let $T[]$ be a $(K+1) \times N$ array such that $T[k,n]$ denotes the cost of the shortest path tour from the source node $s$ to a node $n \in \mathcal{N}$. Let $C_{xy}$ be the cost of the directed edge from $x$ to $y$, where $x, y \in \mathcal{N}$. Algorithm 4 provides a pseudocode description of the DFTS algorithm.

---

**Initialization:**
$F_1 = \{s\}$
$F_k = \emptyset$, $k = 2, ..., K+1$
$T[k,n] = \infty$, $k = 1, ..., K+1$
$T[1,s] = 0$
$I = 1$
**while** $d \notin F_{K+1}$ **do**
  $T[i,w] = \min\{T[i,v] + C_{vw}\}$ s.t. $v \in F_i, w \notin F_i, v \notin S_i, i = I, ..., K+1$
  **if** $w \notin S_i$ **then**
    $F_i = F_i \cup \{w\}$
  **else**
    $F_i = F_i \cup \{w\}$
    $F_{i+1} = F_{i+1} \cup \{w\}$
  **end**
  **if** $F_i \cap S_i = S_i$ **then**
    $I = i + 1$
  **end**
**end**

**Algorithm 4:** The DFTS algorithm for SPTP

---

We have the following result regarding the correctness of DFTS.

**Theorem 5.3.1** *For every connected directed graph with nonnegative edge costs, DFTS correctly constructs the shortest path tour from the source $s$ to the destination $d$.*

*Proof.* Let $L[k,n]$ be the true shortest path tour from the source node $s$ to a node $n \in \mathcal{N}$. The proof is by induction and follows the proof of correctness of Dijkstra's algorithm.

*Base Case:* $T[1,s] = L[1,s] = 0$.

*Inductive Hypothesis:* All previous found shortest path tours are correct, i.e., $\forall\ n \in \mathcal{N}r$ : $T[k,n] = L[k,n]$.

[2]Note also that the decomposition algorithms are akin to breadth first search, since they explore all nodes of a set $S_i$ before proceeding to explore nodes in set $S_{i+1}$

Table 5.1: Time Complexity

| Algorithm | Complexity |
|---|---|
| DC-APSP [86] | $O(N^3) + O(KM^2)$ |
| DC-MPSP-1 [87] | $O(N^3) + O(KM^2)$ |
| DC-MPSP-2 [88] | $O(N^3) + O(KM^2)$ |
| LG [85] | $O(KN^2) + O(KElog(KN))$ |
| DC-SSSP-1 [71] | $O(2ElogN) + O((K-1)MElogN)$ |
| DC-SSSP-2 [73] | $O((K+1)ElogN)$ |
| DFTS (this work) | $O((K+1)ElogN) + O((K+1)N)$ |

*Current Iteration:* We pick an edge $(v^*, w^*)$ which is the minimum cost edge such that $v^* \in F_i$ but $v^* \notin S_i$ and $w^* \notin F_i$, and we let:

$$T[k, w^*] = T[k, v^*] + C_{v^*w^*} = L[k, w^*] + C_{v^*w^*}$$

We distinguish two cases.

Case 1: If $w^* \in S_i$ we add $w^*$ to both $F_i$ and $F_{i+1}$. Since $w^*$ is present in $S_i$ we are now crossing the frontier of $S_i$ and we need to start exploring nodes in $S_{i+1}$, so we add $w^*$ to $F_{i+1}$.

Case 2: If $w^* \notin S_i$ we add $w^*$ to $F_i$. Since $w^*$ is not present in $S_i$ we are not yet crossing the frontier of $S_i$ corresponding to this node and we need to continue exploring nodes in $S_i$, so we add $w^*$ to $F_i$.

We now note that every path tour from $s$ to $w^*$ must have cost $\geq L[k, w^*] + C_{v^*w^*}$, therefore this is the cost of any shortest path tour. To show this, let us assume that there is a tour $P$ which has cost $< L[k, w^*] + C_{v^*w^*}$. This tour has to cross from some node explored in $F_i$ to nodes not explored in $F_i$ or nodes not explored in $F_{i+1}$. If it does, then the edge of cost $C_{v^*w^*}$ selected by the algorithm at this iteration is not the minimum-cost edge, a contradiction. ■

### 5.3.4 Algorithm Complexity

In Table 5.1, we summarize the running time complexity of the seven algorithms we described earlier in this section; we evaluate experimentally the algorithms in the following section.

The DC-APSP [86], DC-MPSP-1 [87], and DC-MPSP-2 [88] algorithms internally use three nested **for** loops to calculate the cost $D(i, j)$ of shortest paths between nodes in adjacent node sets; this computation takes time $O(N^3)$, where $N$ is the number of nodes in the graph, and is shown as the first term of the complexity expression in the top three rows of Table 5.1. The

second term in these complexity expressions corresponds to the time it takes to carry out the dynamic programming Algorithm 3. Therefore, APSP and MPSP are efficient when the graph size is not very large, the node sets $S_i$ are large such that it is necessary to compute paths for a substantial fraction of source-destination pairs, and the graph is strongly connected.

The *LG* [85] approach first constructs a modified layered graph which has $KN$ nodes and $KE$ edges; this takes time $O(KN^2)$, where $K$ is the number of layers (node sets). It then applies Dijkstra's algorithm just once on this graph, and the time for this computation is represented by the second term in the appropriate row of Table 5.1.

For the DC-SSSP-1 algorithm, the first expression in the table denotes the use of Dijkstra's algorithm once from $s$ to reach the nodes in $S_1$, and a second time from $d$ to reach the nodes in $S_K$, if we reverse the direction the edges. The second expression corresponds to the application of Dijkstra's algorithm a further $(K-1) \times M$ times to find the shortest cost distance from every node in $S_i$ to every node in $S_{i+1}, i = 1, \ldots, K-1$, where $M = \max\{|S_i|\}$. This approach works well for problem instances in which each node set $S_i$ is relatively small compared to the whole graph. DC-SSSP-2 applies Dijkstra's algorithm $(K+1)$ times for finding the shortest cost path from any node in $S_i$ to every node in $S_{i+1}, i = 0, \ldots, K+1$. DFTS applies Dijkstra's algorithm just once, but every edge may potentially be traversed $(K+1)$ times (first term in the table), and at each iteration it selects the shortest cost edge among $(K+1)$ sets (second term). To fairly compare the last four algorithms, we have implemented them using binary min-heaps, hence the logarithmic terms in the expressions shown in Table 5.1.

## 5.4 Experimental Study and Results

We now present simulation results to evaluate the seven algorithms we described in Section 5.3, namely, DC-APSP, DC-MPSP-1, DC-MPSP-2, DC-SSSP-1, DC-SSSP-2, LG, and DFTS. We evaluate the algorithms on random graphs generated using BRITE [64], a universal topology generator. We obtained undirected graphs by configuring BRITE to generate AS-Level Barabasi models; we then converted these graphs into directed ones that we used in our experiments. In generating random instances for the SPTP problem, we considered the following parameters and varied their values as described below:

- The number $N$ of nodes in the graph was varied from 1000 to 5000 in increments of 1000.
- The average nodal degree $\Delta$ of the graph was set to an integer in the range $[2, 5]$.
- The number $K$ of node sets in the tour took integer values in the interval $[1, 4]$; recall that in the service routing problem, $K$ represents the number of services to be applied to the user's traffic.

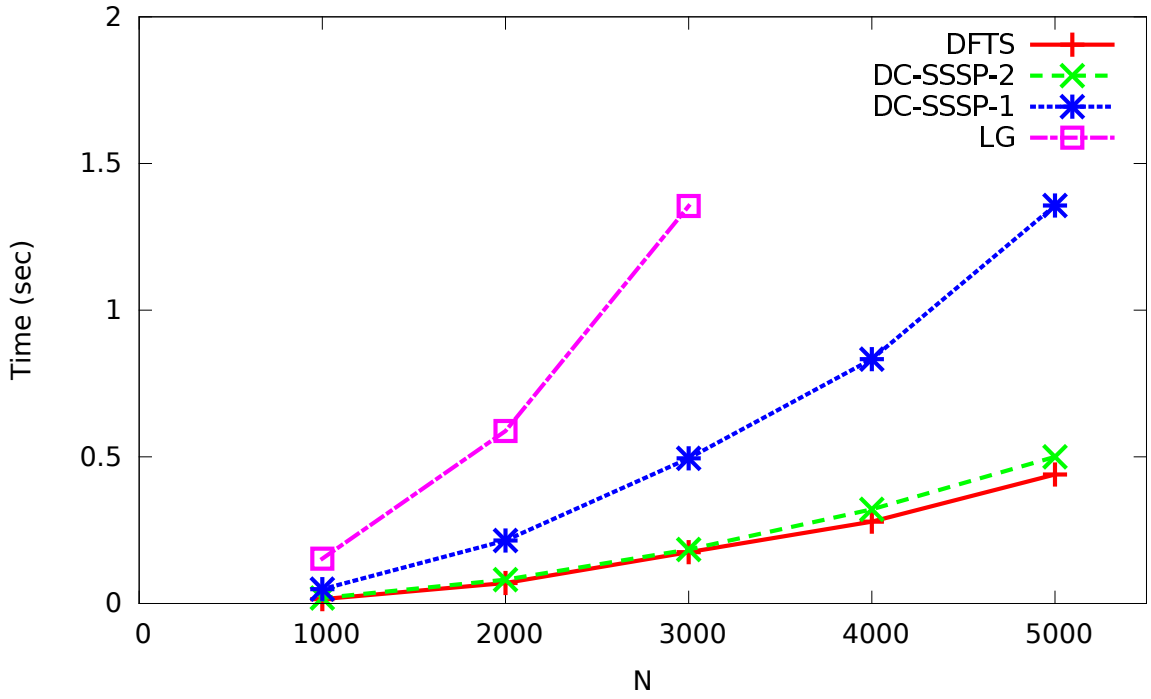


Figure 5.1: Running time comparison, most efficient algorithms, $\Delta = 3, K = 2, M = 5$

- The number $M$ of nodes in each node set was varied from 5 to 25 in increments of 5.

Since all algorithms produce the same solution to any instance of SPTP, our evaluation focuses on one metric, running time. We note that the orchestration process in an NFV environment must operate in real time and scale to large network topologies with many services and multiple virtual instances of each service. Hence, the various figures in this section explore the running time of the algorithms as a function of the various parameters listed above. With two exceptions that we discuss shortly, each data point in these figures is the average running time over 10,000 problem instances generated from the stated values of the parameters. All experiments were performed on a HPC cluster that included three processor families, Intel Xeon E5520 (2.27GHz), E5620 (2.40GHz) and E5540 (2.53GHz), all with four cores, each core having 4GB of DRAM and 8KB of cache.

### 5.4.1 Overall Comparison

Figures 5.1 and 5.2 plot the running time of the seven algorithms as a function of the number $N$ of nodes in the network. For the problem instances used in these figures, the nodal degree was set to $\Delta = 3$, the number of node sets was $K = 2$, and the number of nodes in each node set was $M = 5$. Our first observation is that the four algorithms (DFTS, LG, and the two

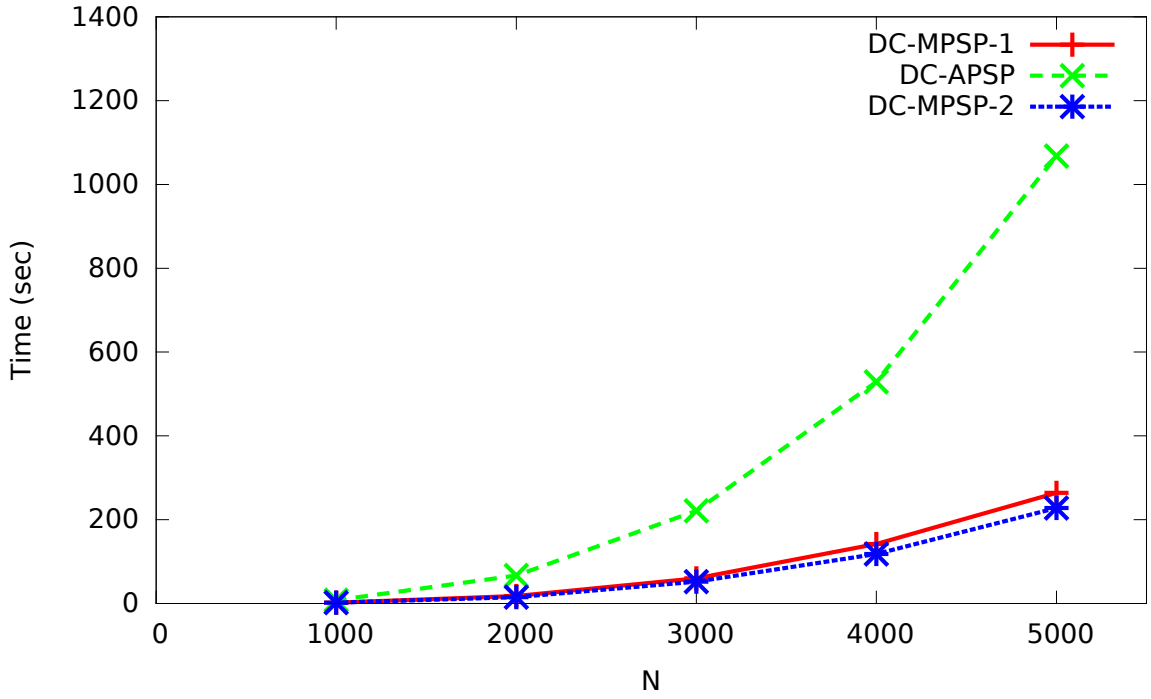


Figure 5.2: Running time comparison, least efficient algorithms, $\Delta = 3, K = 2, M = 5$

DC-SSSP algorithms) shown in Figure 5.1 take less than two seconds on average to solve these problem instances, whereas the other three (DC-APSP and the two DC-MPSP algorithms), shown in Figure 5.2 are two-to-three orders of magnitude slower - hence, it was necessary to separate them in a different figure. Furthermore, each data point in Figure 5.2, as well as in the similar Figure 5.4 discussed shortly, represents the average of only 50 problem instances, rather than the 10,000 that we used for all other figures. This value was selected as it allowed us to obtain each data point in no more than 24 hours for the largest problem instance considered in these two figures. Another interesting observation from Figure 5.1 is that no data points are shown for the LG algorithm and networks with more than $N = 3000$ nodes. Recall that the LG algorithm constructs a graph of $K$ layers of the original network topology. Consequently, as the network size grows, it is memory, not running time, that becomes the limiting factor, and we were not able to solve larger instances with the LG algorithm in the HPC cluster available to us.

Figures 5.3 and 5.4 are similar to the ones above but present results for instances generated with $\Delta = 5$, $K = 4$, and $M = 25$. Since the problem instances are larger in this case, the running times are higher than the corresponding algorithms in the previous two figures. Similarly, in both sets of figures, the running time of a particular algorithm increases with the network size

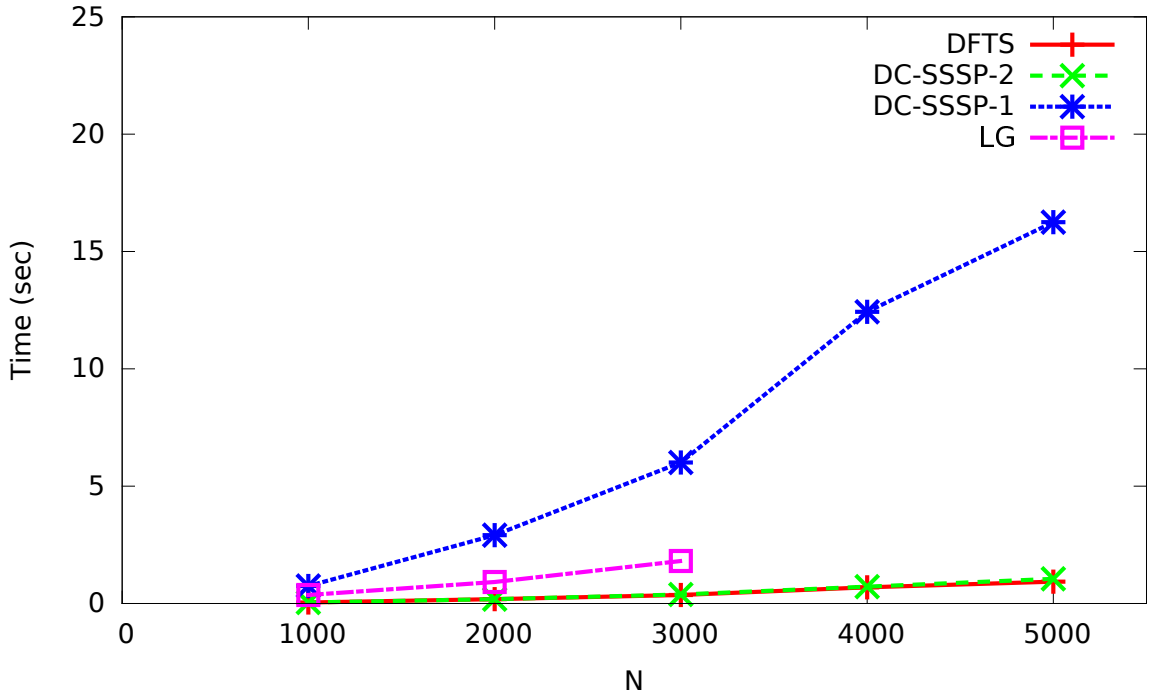


Figure 5.3: Running time comparison, most efficient algorithms, $\Delta = 5, K = 4, M = 25$

$N$.

From the four figures, it is clear that the three least efficient algorithms (DC-APSP, DC-MPSP-1, DC-MPSP-2) do not scale well and are not appropriate for real-time operation. Also, since LG replicates the network topology $K$ times, its memory requirements become a challenge for larger problem instances. Finally, as Figure 5.3 illustrates, the DC-SSSP-1 algorithm, which applies Dijkstra's algorithm multiple times at each stage, becomes one order of magnitude slower than the two best algorithms, DFTS and DC-SSSP-2, at larger problem instances considered here.

We have observed the relative behavior illustrated in the four figures above across a wide range of experiments. Therefore, in the remainder of this section we will explore further only the behavior of the two best algorithms, the DC-SSSP-2 algorithm of [73], and the new algorithm we developed, DFTS.

### 5.4.2 Comparison of DC-SSSP-2 and DFTS

Let us now investigate the performance of the two algorithms as a function of the parameters $\Delta, K$, and $M$. Figures 5.5 and 5.6 plot the running time of the DC-SSSP-2 and DFTS algorithms by varying the nodal degree $\Delta$ of the graph and keeping the values of the other parameters

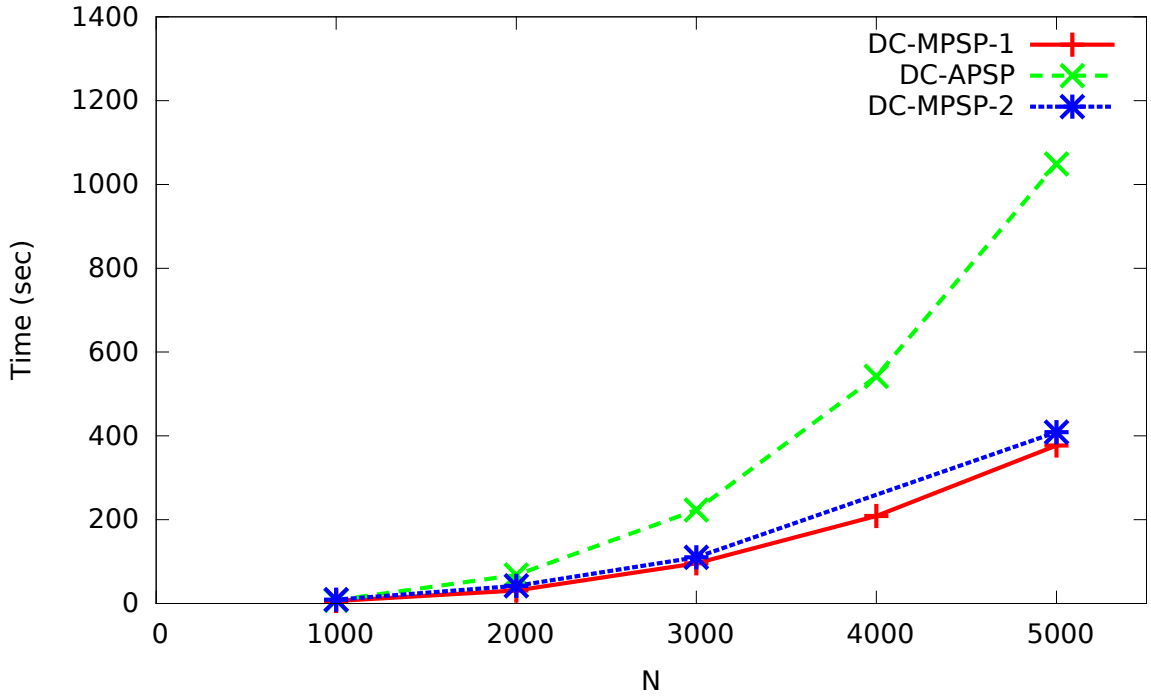


Figure 5.4: Running time comparison, least efficient algorithms, $\Delta = 5, K = 4, M = 25$

fixed. Each figure includes two sets of plots, one for networks with $N = 1000$ nodes and one with $N = 5000$. In the problem instances of Figure 5.5, the path tour must visit a node from just one set ($K = 1$) that includes five nodes ($M = 5$); whereas for Figure 5.6 the problem instances were generated with $K = 4, M = 25$.

As the average nodal degree $\Delta$ increases, the size of the network (in terms of the number of edges) grows, hence the running of the two algorithms also increases; however, as we can observe from the two figures, this increase in running time is rather moderate. Similarly, the running time curves for the larger network ($N = 5000$) sit higher than those for the smaller network ($N = 1000$) in the same figure (i.e., for the same values of the other parameters); this behavior is also expected and is due to the increase in network size and consistent with the complexity results in Table 5.1. Also, as $K$ and $M$ increase, the path tour must traverse a larger number of node sets, and there are more options (in terms of number of nodes, $M$) to be explored, hence the running time values in Figure 5.6 are higher than for the corresponding curves in Figure 5.5.

Finally, we make two important observations. First, the running time of either algorithm does not exceed one second even for the largest of the problem instances we present in the above two figures (i.e., instances with $N = 5000$ nodes, nodal degree $\Delta = 5$, $K = 5$ node sets,

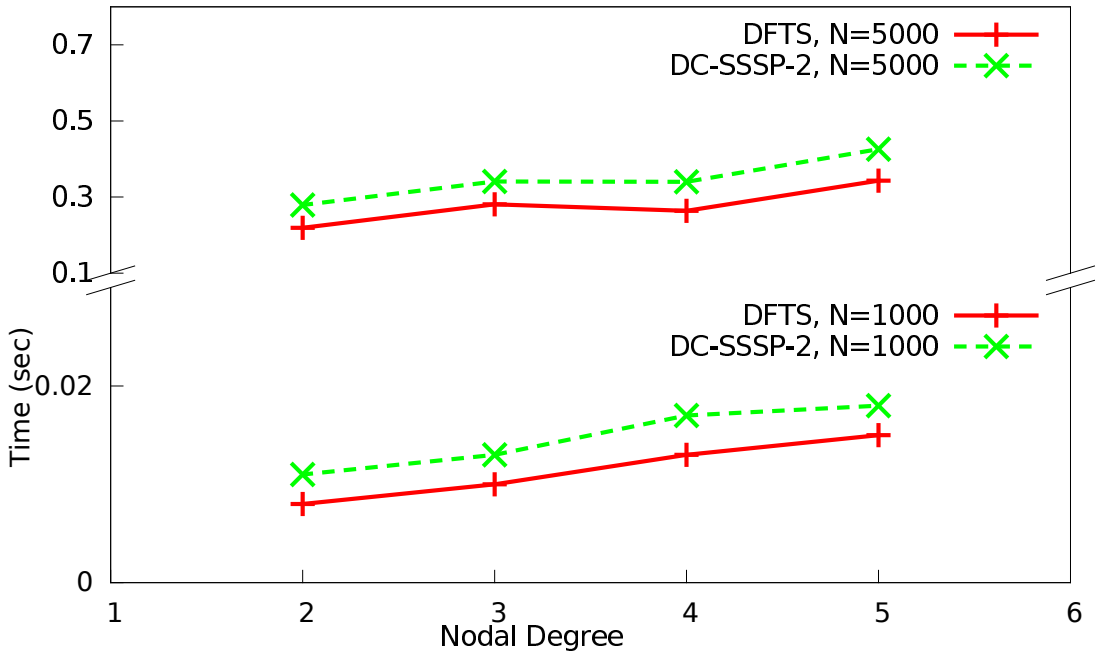


Figure 5.5: Running time vs nodal degree, $K = 1, M = 5$

and $M = 25$ nodes per set). Therefore, we conclude that these two algorithms scale well and are suitable for real-time applications. Furthermore, our new algorithm, DFTS, consistently outperforms the next best algorithm, DC-SSSP-2, across the range of parameter values that we investigated.

The next two figures, 5.7 and 5.8, are similar to the ones we just discussed but plot the running time of the DFTS and DC-SSSP-2 algorithms by varying the number $K$ of sets in a tour while keeping the other parameters fixed. With a larger number of sets, the path tour must traverse more nodes, hence it takes longer time to explore all the options to construct the tour; this intuition is confirmed by the results in the two figures. As before, we also observe that our DFTS algorithm outperforms DC-SSSP-2, and that its running time does not exceed one second, even for the largest instances.

The last pair of figures, 5.9 and 5.10, compare the running time of the two algorithms as a function of the number of set elements in a set, with all other parameters fixed. All our observations above regarding the relative and absolute performance of the algorithms are also valid for these sets of results. However, we also observe that the running time of either algorithm is largely insensitive to the size $M$ of the node sets. This is mainly due to the way the two algorithms operate. As we mentioned in Section 5.3, DC-SSSP-1 applies Dijkstra's algorithm

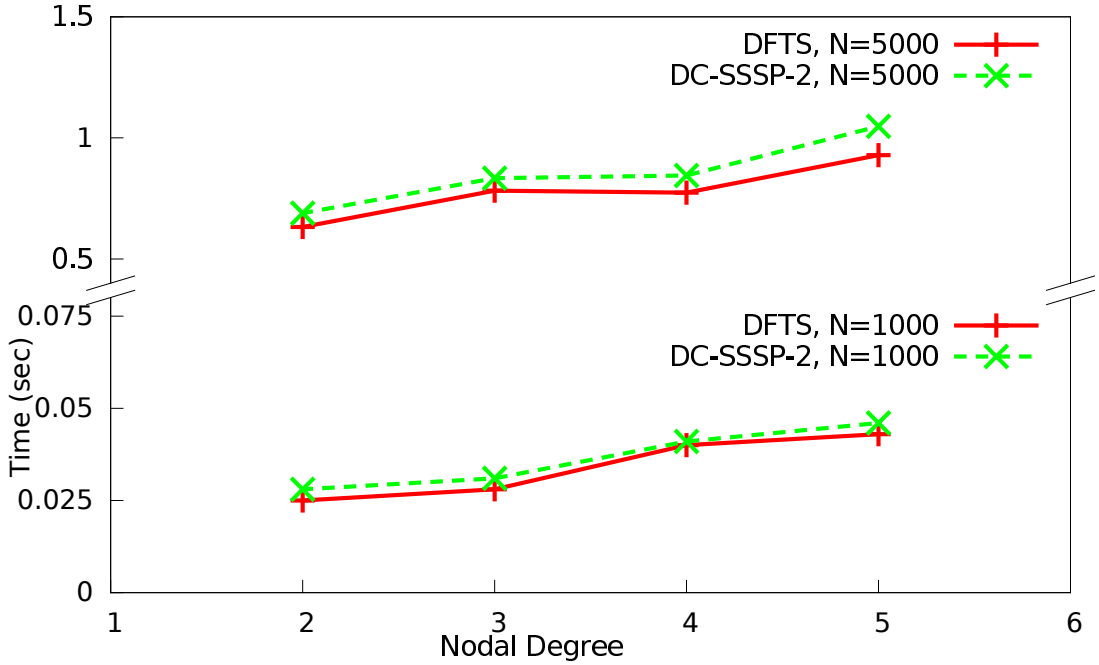


Figure 5.6: Running time vs nodal degree, $K = 4, M = 25$

once to find the shortest path from any node in set $S_i$ to any node in set $S_{i+1}$; as a result, the running time is not affected much by the size of the node sets. Similarly, our DFTS algorithm does not wait until all nodes in a set have been explored, hence, its performance is relatively independent of the set size.

There are 400 unique combinations of the values of parameters $N, \Delta, K$, and $M$ that we considered in our experiments (refer to the top of this section). In Table 5.2, we list the improvement in running time of our DFTS algorithm over the next best algorithm, DC-SSSP-2, for problem instances generated with each of these 400 parameter value combinations. As we can see, our algorithm runs faster than DC-SSSP-2 in all but 10 combinations which are highlighted in bold in the table. Across all problem instances, our algorithm achieves an improvement in running time averaging 13.62%. Overall, these results demonstrate that the new DFTS algorithm exhibits superior performance compared to existing algorithms, it scales well and is suitable for real-time applications.

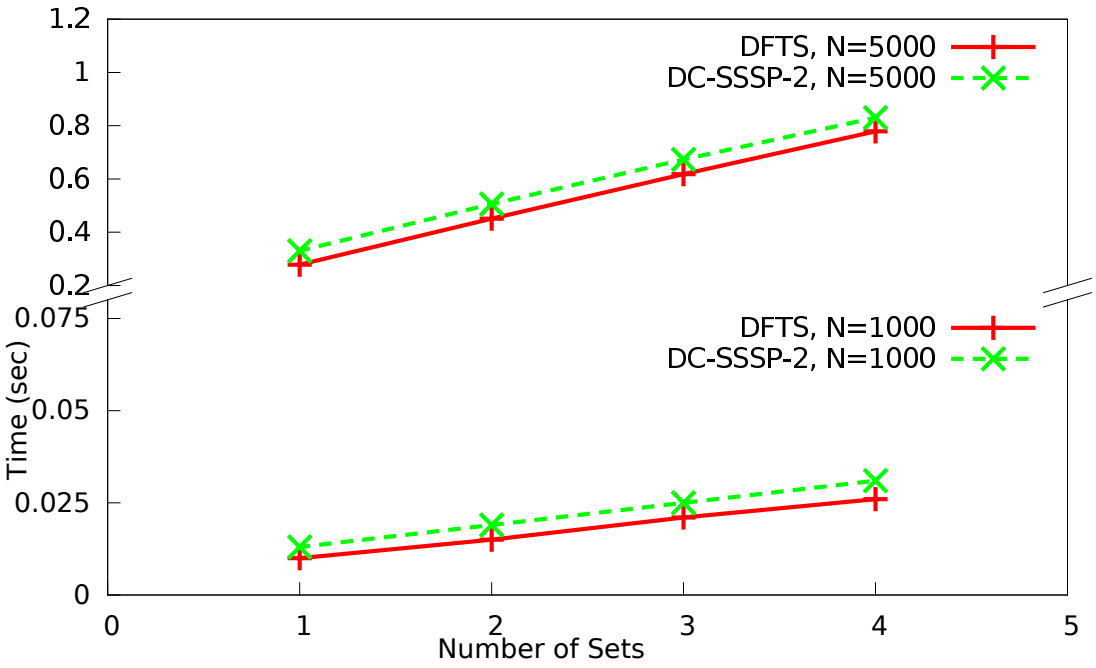


Figure 5.7: Running time vs number of sets, $\Delta = 3, M = 15$

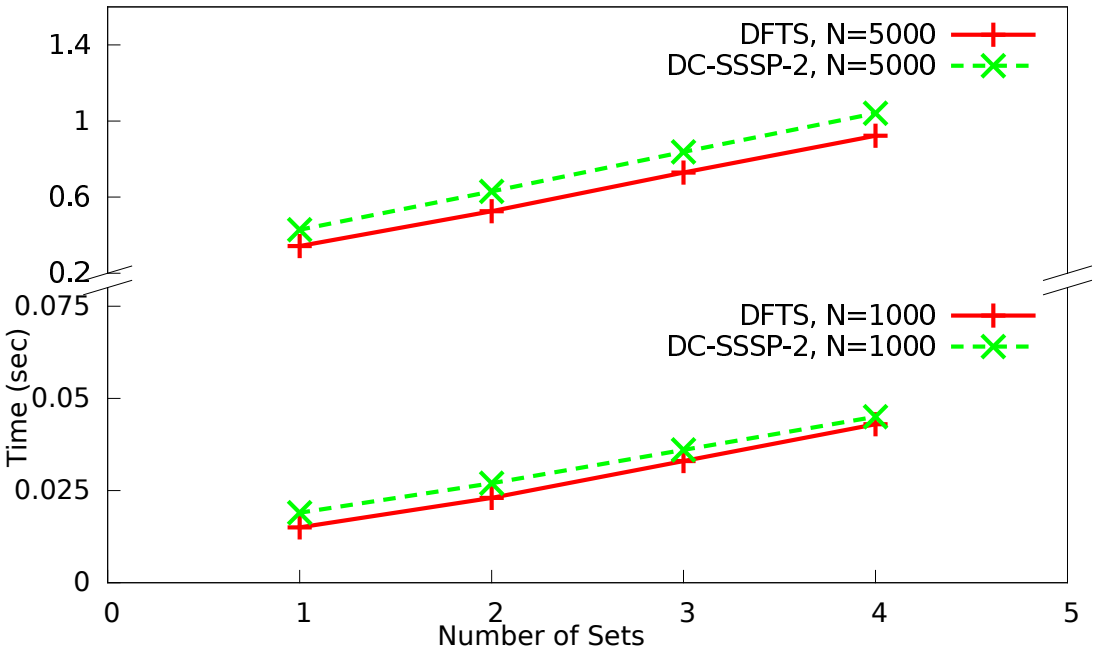


Figure 5.8: Running time vs number of sets, $\Delta = 5, M = 15$

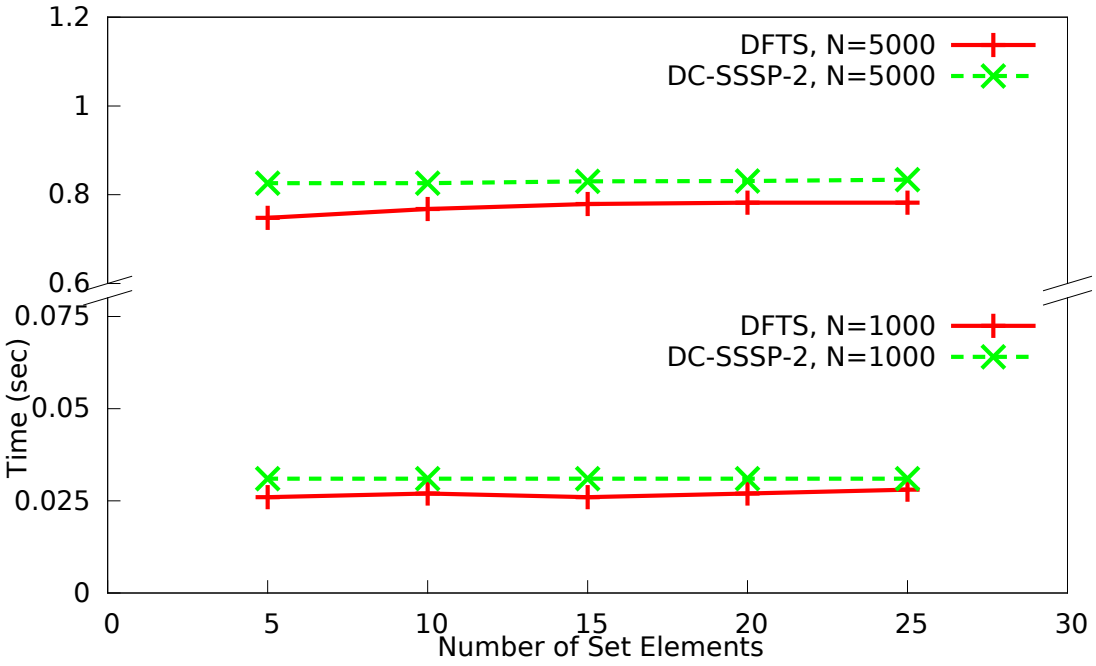


Figure 5.9: Running time vs number of set elements, $\Delta = 3, K = 4$

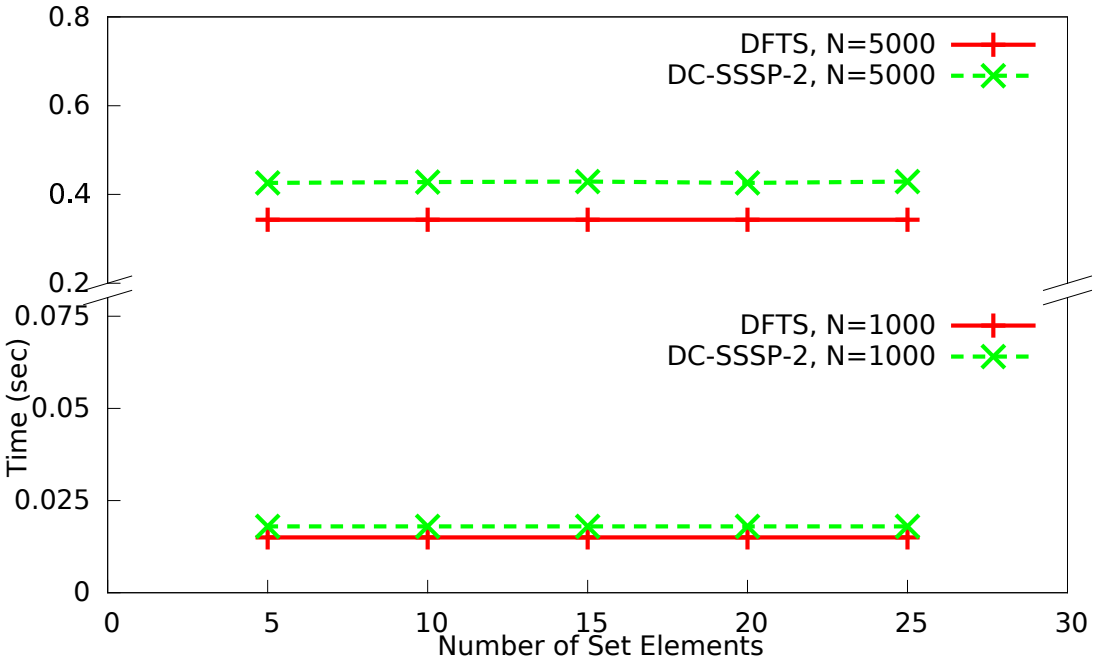


Figure 5.10: Running time vs number of set elements, $\Delta = 5, K = 1$

Table 5.2: Running Time Improvement (in %) of DFTS Relative to DC-SSSP-2

| | | $K=1$ | | | | | $K=2$ | | | | | $K=3$ | | | | | $K=4$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | $\Delta$ | $M=5$ | $M=10$ | $M=15$ | $M=20$ | $M=25$ | $M=5$ | $M=10$ | $M=15$ | $M=20$ | $M=25$ | $M=5$ | $M=10$ | $M=15$ | $M=20$ | $M=25$ | $M=5$ | $M=10$ | $M=15$ | $M=20$ | $M=25$ |
| 1000 | 2 | 26.51 | 23.83 | 28.70 | 31.73 | 30.56 | 25.37 | 19.48 | 20.74 | 24.23 | 20.91 | 23.00 | 20.14 | 18.45 | 100.00 | 16.23 | 16.33 | 14.46 | **-1.16** | 14.13 | 11.51 |
| | 3 | 21.85 | 24.77 | 24.19 | 25.16 | 25.56 | 19.77 | 13.38 | 19.89 | 8.75 | 9.38 | 17.57 | 16.71 | 15.82 | 15.94 | 19.10 | 14.70 | 6.68 | 17.01 | 14.30 | 11.20 |
| | 4 | 22.03 | 15.69 | 20.84 | 13.91 | 18.73 | 11.79 | 8.30 | 8.43 | 13.58 | 8.97 | 9.51 | **-0.13** | 9.29 | 9.90 | 9.92 | **-5.41** | 2.86 | **-10.65** | **-8.03** | 3.11 |
| | 5 | 17.51 | 18.58 | 16.87 | 19.77 | 19.38 | 13.94 | 7.42 | 15.45 | 6.03 | 10.21 | 11.72 | 7.78 | 10.62 | 7.53 | 7.06 | 7.56 | 6.50 | 6.04 | 5.90 | 5.04 |
| 2000 | 2 | 24.87 | 26.03 | 29.13 | 27.82 | 29.09 | 19.63 | 23.31 | 19.02 | 24.36 | 20.83 | 16.01 | 15.63 | 14.10 | 16.02 | 15.94 | 14.25 | 11.36 | 13.07 | 11.67 | 14.21 |
| | 3 | 18.65 | 20.32 | 9.57 | 15.60 | 18.02 | 14.10 | 11.90 | 12.01 | 12.87 | 12.50 | 12.40 | 14.22 | 10.72 | 8.98 | 9.34 | 10.22 | 7.42 | 5.41 | 6.13 | 6.27 |
| | 4 | 17.54 | 19.41 | 6.00 | 15.43 | 16.43 | 8.67 | 10.89 | 12.44 | 12.22 | 12.51 | 10.87 | 8.41 | 8.97 | 8.47 | **-0.76** | 9.95 | 6.52 | 10.39 | 5.36 | 5.91 |
| | 5 | 15.69 | 14.98 | 8.54 | 16.75 | 17.56 | 1.76 | 12.07 | 10.95 | 11.61 | 12.08 | 14.10 | 12.37 | 6.81 | 8.90 | 9.24 | 7.83 | 5.17 | 6.76 | **-5.60** | **-3.17** |
| 3000 | 2 | 18.35 | 20.83 | 19.72 | 20.10 | 20.72 | 7.49 | 14.93 | 11.72 | 14.73 | 15.55 | 14.96 | 11.06 | 10.60 | 11.72 | 11.30 | 12.98 | 9.59 | 8.94 | 8.21 | 21.75 |
| | 3 | 20.49 | 17.35 | 17.24 | 25.33 | 18.74 | 4.67 | 26.96 | 11.08 | 11.44 | 16.87 | 10.77 | 9.37 | 8.29 | 8.51 | 8.61 | 10.04 | 6.73 | 6.55 | 6.73 | 4.43 |
| | 4 | 15.48 | 15.43 | 15.42 | 14.83 | 14.98 | 3.44 | 9.69 | 10.48 | 8.61 | 9.72 | 0.37 | 7.49 | 7.81 | 7.47 | 10.65 | 8.55 | 5.92 | 10.09 | 5.30 | 5.21 |
| | 5 | 16.20 | 15.08 | 15.06 | 20.35 | 17.05 | 11.14 | 10.39 | 7.08 | 14.00 | 14.52 | 14.21 | 11.39 | 10.98 | 10.39 | 5.91 | 21.84 | **-5.53** | 3.66 | **-4.29** | 4.09 |
| 4000 | 2 | 21.25 | 20.17 | 14.16 | 19.16 | 23.08 | 21.27 | 15.02 | 14.07 | 16.62 | 16.40 | 14.15 | 11.72 | 10.44 | 11.76 | 12.63 | 13.20 | 10.40 | 7.30 | 12.23 | 8.03 |
| | 3 | 17.86 | 17.81 | 17.10 | 8.56 | 17.61 | 12.88 | 12.30 | 11.71 | 10.61 | 11.97 | 12.07 | 7.87 | 8.27 | 8.37 | 0.49 | 9.91 | 6.88 | 6.66 | 6.45 | 10.68 |
| | 4 | 23.93 | 18.24 | 21.14 | 21.59 | 21.13 | 15.20 | 15.37 | 14.10 | 15.36 | 10.70 | 9.18 | 10.67 | 11.65 | 15.33 | 12.89 | 10.63 | 10.96 | 21.37 | 9.29 | 14.22 |
| | 5 | 18.70 | 12.95 | 14.80 | 13.55 | 14.50 | 14.01 | 8.73 | 6.87 | 9.42 | 9.47 | 7.79 | 19.59 | 19.75 | 4.23 | 6.42 | 5.24 | 5.09 | 8.95 | 4.86 | 4.38 |
| 5000 | 2 | 21.44 | 21.83 | 23.67 | 24.06 | 24.58 | 18.38 | 18.02 | 18.51 | 18.95 | 18.42 | 15.14 | 13.71 | 8.31 | 15.52 | 14.16 | 14.14 | 12.19 | 12.24 | 12.15 | 8.32 |
| | 3 | 17.39 | 14.04 | 15.66 | 16.15 | 16.88 | 12.16 | 10.72 | 10.97 | 9.74 | 10.66 | 10.73 | 8.26 | 8.16 | 7.70 | 7.34 | 9.50 | 7.06 | 6.14 | 5.87 | 6.23 |
| | 4 | 22.16 | 21.45 | 22.13 | 22.43 | 22.87 | 17.62 | 13.77 | 14.53 | 16.05 | 16.01 | 14.00 | 11.13 | 11.80 | 11.27 | 11.02 | 13.54 | 9.01 | 8.93 | 8.57 | 8.45 |
| | 5 | 19.58 | 24.37 | 20.19 | 20.89 | 21.06 | 15.95 | 15.97 | 16.45 | 16.96 | 16.26 | 13.31 | 12.41 | 12.98 | 14.69 | 14.08 | 11.53 | 9.69 | 11.33 | 10.85 | 11.37 |

## 5.5 Concluding Remarks

The service-concatenation routing problem arises as an integral part of the orchestration process in NFV architectures. We have shown that service-concatenation routing is equivalent to the shortest path tour problem (SPTP). Most existing algorithmic approaches to SPTP work well only for specific classes of problem instances, and do not scale well to the large instances that arise in NFV applications. We have developed a new algorithm that applies several novel modifications to Dijkstra's algorithm to construct the shortest path tour efficiently. Our experimental study has demonstrated that our algorithm scales well to large instances and is appropriate for real-time NFV applications across a wide range of graphs.

# Chapter 6

# Summary and Future Work

The main contribution of this research has been on two fronts:

- Developing a framework where the principles of Choicenet can be realized:
  - A semantics language which provides a level playing field for services to be compared and composed
  - A economy plane protocol which defines the interactions between the various entities of ChoiceNet, essential to signing the contract for using the service
  - A mechanism for extracting services selectively from the Marketplace based on the user request
- Developing three complementary Planners which perform Network Service Orchestration
  - The first Planner presents the user with $k$ composed services using the services advertised in a Marketplace, which supports combining multiple service functionalities into one service advertisement.
  - The second Planner presents the user with multiple flavors of composed services.
    - Non dominated and resource constrained composed services
    - Non dominated composed services
    - Resource constrained $k$ composed services
    - $k$ composed services

    The composed services returned by the Planner is used to perform in-advance path reservation.
  - The third Planner presents the users with a composed service by solving the shortest path tour problem

We have showed how all the components developed in this research fit together to accomplish the goal of "Innovation" by enabling "Choice" in the Future Internet Design. We have performed extensive simulation and obtained insightful results which show the framework and the Planners work efficiently.

## 6.1 Future Work

In this work the Planners were designed to find optimal solutions for the user by performing orchestration of services, which were designed to run on a predetermined set of network nodes. This work can be extended in the following ways:

- **Service Placement:** We envision a Planner being part of the Network design. This Planner works in developing optimal Network topologies which aids the Provider in placing network services at strategic points based on user preferences and historical data. The goals of the Planner which tries to find the optimal composed service for the user may or may not be aligned with the goals of the Planner which tries to build optimal network topologies by placing services prudently.

- **Optimality Gap:** The Planners should be able to provide the "Choice" of optimal vs approximate solutions to the user. The trade-off is the running time to find the optimal solution vs the optimality gap between the approximate and the optimal solution. For the resource constrained single source shortest path problem, Lagrangian Relaxation methods and preprocessing can lead to improvements in both time and space. Similarly, for the pareto optimal paths problem the trade-off is the complete set of pareto paths vs the subset of the pareto paths, using one attribute which is a weighted average of all the attributes provides the subset of pareto paths very quickly, assuming ratio restricted lengths for the various attributes also helps in finding the pareto set quickly.

- **$k$-connectivity:** Developing a new Planner which can exploit the graph structure in particular the k-connectivity might be beneficial in further increasing the efficiency. For small values of $k$ we can divide the graph into $k$ sub-graphs and run our Planner on these sub-graphs and build the intelligence in the Planner to combine these results.

# REFERENCES


[1] Tilman Wolf, James Griffioen, Kenneth L. Calvert, Rudra Dutta, George N. Rouskas, Ilya Baldin, and Anna Nagurney. Choicenet: Toward an economy plane for the internet. *SIGCOMM Comput. Commun. Rev.*, 44(3):58--65, July 2014.

[2] A.C. Babaoglu and R. Dutta. A verification service architecture for the future internet. In *Computer Communications and Networks, 2013. ICCCN 2013. Proceedings.22nd International Conference on*, 2013.

[3] Sylvia Ratnasamy, Scott Shenker, and Steven McCanne. Towards an evolvable internet architecture. *SIGCOMM Comput. Commun. Rev.*, 35(4):313--324, August 2005.

[4] Vytautas Valancius, Nick Feamster, Ramesh Johari, and Vijay Vazirani. Mint: A market for internet transit. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 70:1--70:6, New York, NY, USA, 2008. ACM.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269--271, 1959.

[6] A. W. Brander and M.C. Sinclair. A comparative study of $k$-shortest path algorithms. In *Proceedings of the 11th UK Performance Engineering Workshop*, September 1995.

[7] S. Chen and K. Nahrstedt. An overview of quality of service routing for next-generation high speed networks: Problems and solutions. *IEEE Network*, 12(6):64--79, November/December 1998.

[8] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Transactions on Networking*, 1(3):286--292, June 1993.

[9] Mehmet Onur Ascigil and Ken Calvert. Implications of source routing. In *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, CoNEXT Student '12, pages 11--12, New York, NY, USA, 2012. ACM.

[10] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 111--122, New York, NY, USA, 2009. ACM.

[11] Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. Routing as a service. Technical Report UCB/CSD-04-1327, EECS Department, University of California, Berkeley, 2004.

[12] Landon P. Cox and Brian D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 120--132, New York, NY, USA, 2003. ACM.

[13] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp: An architecture for secure resource peering. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 133--148, New York, NY, USA, 2003. ACM.

[14] Sebastian Angel and Michael Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 195--206, New York, NY, USA, 2013. ACM.

[15] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169--182, August 2005.

[16] Rachel Singer Gordon. Understanding web services (book). *Library Journal*, 128(2):111, 2003.

[17] Liyang Yu. *A Developers Guide to the Semantic Web.* Springer Berlin Heidelberg.

[18] Michael Jeronimo and Jack Weast. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play.* Intel Press, 2003.

[19] Andre L.C. Tavares and Marco Tulio Valente. A gentle introduction to osgi. *SIGSOFT Softw. Eng. Notes*, 33(5):8:1--8:5, August 2008.

[20] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Proceedings of the First International Conference on Semantic Web Services and Web Process Composition*, SWSWPC'04, pages 43--54, Berlin, Heidelberg, 2005. Springer-Verlag.

[21] Bin Xu and Sen Luo. Efficient composition of semantic web services with end-to-end qos optimization. In Brian Blake, Liliana Cabral, Birgitta Knig-Ries, Ulrich Kster, and David Martin, editors, *Semantic Web Services*, pages 345--355. Springer Berlin Heidelberg, 2012.

[22] Freddy Lcu, Eduardo Silva, and LusFerreira Pires. A framework for dynamic web services composition. In Thomas Gschwind and Cesare Pautasso, editors, *Emerging Web Services Technology, Volume II*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 59--75. Birkhuser Basel, 2008.

[23] S. Shanbhag and T. Wolf. Automated composition of data-path functionality in the future internet. *Network, IEEE*, 25(6):8--14, Nov 2011.

[24] Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Duttai, Ilia Baldine, and Dan Stevenson. A composition algorithm for the silo cross-layer optimization service architecture. *ANTS, IEEE*, 2007.

[25] J. Y. Yen. Finding the $k$ shortest loopless paths in a network. *Management Science*, 17(11):712--716, 1971.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., New York, 1979.

[27] M. Ziegelmann. *Constrained Shortest Paths and Related Problems.* PhD thesis, Universitaet des Saarlandes, 2001.

[28] Y. Xiao, K. Thulasiraman, G. Xue, and A. Jttner. The constrained shortest path problem: algorithmic approaches and an algebraic study with generalization. *AKCE International Journal of Graphs and Combinatorics*, (2):63--86, December 2005.

[29] D. Eppstein. Finding the $k$ shortest paths. *SIAM Journal on Computing*, 28(2):652--673, 1998.

[30] N. Shi. $k$ constrained shortest path problem. *IEEE Transactions on Automation Science and Engineering*, 7(1):15--23, January 2010.

[31] M. Balman, E. Chaniotakisy, A. Shoshani, and A. Sim. A flexible reservation algorithm for advance network provisioning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1--11, Nov 2010.

[32] E.S. Jung, Y. Li, S. Ranka, and S. Sahni. An evaluation of in-advance bandwidth scheduling algorithms for connection-oriented networks. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2008)*, pages 133--138, May 2008.

[33] S. Sahni, N. Rao, S. Ranka, Y. Li, E.S. Jung, and N. Kamath. Bandwidth scheduling and path computation algorithms for connection-oriented networks. In *Proceedings of the Sixth International Conference on Networking (ICN 2007)*, pages 47--47, April 2007.

[34] Zbigniew Tarapata. Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *Int. J. Appl. Math. Comput. Sci.*, 17(2):269--287, June 2007.

[35] David J. Thuente. {TWO} {ALGORITHMS} {FOR} {SHORTEST} {PATHS} {THROUGH} {MULTIPLE} {CRITERIA} {NETWORKS}. In PETER C.C. WANG, , Arthur L. Schoenstadt, , Bert I. Russak, , and Craig Comstock, editors, *Information Linkage Between Applied Mathematics and Industry*, pages 567 -- 573. Academic Press, 1979.

[36] Pierre Hansen. *Bicriterion Path Problems*, pages 109--127. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.

[37] Ernesto Queirs Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236 -- 245, 1984.

[38] Joao Carlos Namorado Climaco and Ernesto Queiros Vieira Martins. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11(4):399--404, 1982.

[39] Joo C. N. Clmaco, Jos M. F. Craveirinha, and Marta M. B. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Networks*, 41(4):206--220, 2003.

[40] Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401--405, 1972.

[41] Ernesto de Queirs Vieira Martins, Marta Margarida Braz Pascoal, Jos Luis Esteves Dos Santos, and S. Olariu. Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science*, 10(3):247, 1999.

[42] H.C Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, 14(2):191 -- 197, 1966.

[43] Gabriel Y. Handler and Israel Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293--309, 1980.

[44] Mordechai I. Henig. The shortest path problem with two objective functions. *European Journal of Operational Research*, 25(2):281 -- 291, 1986.

[45] Arthur Warburton. Approximation of pareto optima in multiple-objective, shortest-path problems. *Operations Research*, 35(1):70--79, 1987.

[46] Refael Hassin. Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.*, 17(1):36--42, February 1992.

[47] Martin Desrochers and Francois Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191 -- 212, 1988.

[48] Horst W. Hamacher, Stefan Ruzika, and Stevanus A. Tjandra. Algorithms for time-dependent bicriteria shortest path problems. *Discret. Optim.*, 3(3):238--254, September 2006.

[49] S. Bhat and G. N. Rouskas. On routing algorithms for open marketplaces of path services. In *2016 IEEE International Conference on Communications (ICC)*, pages 1--6, May 2016.

[50] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. Shortest chain subject to side constraints. *Networks*, 13(2):295--302, 1983.

[51] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379--394, 1989.

[52] I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42(3):135--153, 2003.

[53] T.A Brown and R.E Strauch. Dynamic programming in multiplicative lattices. *Journal of Mathematical Analysis and Applications*, 12(2):364 -- 370, 1965.

[54] N. Shi. K constrained shortest path problem. *IEEE Transactions on Automation Science and Engineering*, 7(1):15--23, Jan 2010.

[55] Guy Desaulniers and Daniel Villeneuve. The shortest path problem with time windows and linear waiting costs. *Transportation Science*, 34(3):312--319, August 2000.

[56] R. A. Guerin and A. Orda. Networks with advance reservations: the routing perspective. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 1, pages 118--127 vol.1, 2000.

[57] M. Balman, E. Chaniotakisy, A. Shoshani, and A. Sim. A flexible reservation algorithm for advance network provisioning. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1--11, Nov 2010.

[58] P.C. Fishburn. *Utility Theory for Decision Making*. Wiley, New York, 1970.

[59] Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and Franois Soumis. Chapter 2 time constrained routing and scheduling. In C.L. Monma M.O. Ball, T.L. Magnanti and G.L. Nemhauser, editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 35 -- 139. Elsevier, 1995.

[60] Wei Wu and Qiuqi Ruan. A hierarchical approach for the shortest path problem with obligatory intermediate nodes. In *Signal Processing, 2006 8th International Conference on*, volume 4, pages --, Nov 2006.

[61] Jean-Franois Brub, Jean-Yves Potvin, and Jean Vaucher. Time-dependent shortest paths through a fixed sequence of nodes: application to a travel planning problem. *Computers and Operations Research*, 33(6):1838 -- 1856, 2006.

[62] Ning Shi, Shaorui Zhou, Fan Wang, Yi Tao, and Liming Liu. The multi-criteria constrained shortest path problem. *Transportation Research Part E: Logistics and Transportation Review*, 101:13 -- 29, 2017.

[63] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):pp. 712--716, 1971.

[64] A. Medina, I. Matta, and J. Byers. Brite: A flexible generator of internet topologies. Technical report, Boston, MA, USA, 2000.

[65] William B. Norton. The internet peering playbook : Connecting to the core of the internet. 2014.

[66] SDN and OpenFlow World Congress. Network Function Virtualization, updated white paper. https: //portal.etsi.org/nfv/nfv_white_paper2.pdf. October 2013.

[67] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121--136, New York, NY, USA, 2015. ACM.

[68] A. Gember, A. Krishnamurthy, S. St. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[69] G. Xilouris, E. Trouva, F. Lobillo, J. M. Soares, J. Carapinha, M. J. McGrath, G. Gardikis, P. Paglierani, E. Pallis, L. Zuccaro, Y. Rebahi, and A. Kourtis. T-nova: A marketplace for virtualized network functions. In *2014 European Conference on Networks and Communications (EuCNC)*, pages 1--5, June 2014.

[70] A. Dwaraki and T. Wolf. Adaptive service-chain routing for virtual network functions in software-defined networks. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 32--37, New York, NY, USA, 2016. ACM.

[71] A. M. Medhat, G. Carella, C. Luck, M-I. Corici, and T. Magedanz. Near optimal service function path instantiation in a multi-datacenter environment. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 336--341, Los Alamitos, CA, USA, 2015. IEEE Computer Society.

[72] C. P. Bajaj. Some constrained shortest-route problems. *Unternehmensforschung*, 15(1):287--301, 1971.

[73] A. Kershenbaum, W. Hsieh, and B. Golden. Constrained routing in large sparse networks. In *IEEE International Conference on Communications," pp. 38.14-38.18, Philadelphia, PA*, 1976.

[74] S. Bhat and G. N. Rouskas. On Routing Algorithms for Open Marketplaces of Path Services. In *Proceedings of IEEE ICC 2016*, May 2016.

[75] X. Huang, S. Shanbhag, and T. Wolf. Automated service composition and routing in networks with data-path services. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1--8, Aug 2010.

[76] S. Bhat, R. Udechukwu, R. Dutta, and G. N. Rouskas. Inception to Application: A GENI based prototype of an Open Marketplace for Network Services. In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, April 2016.

[77] R. Udechukwu, S. Bhat, R. Dutta, and G. N. Rouskas. Language of choice: On embedding choice-related semantics in a realizable protocol. In *2016 37th IEEE Sarnoff Symposium*, Sep 2016.

[78] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 -- 23, 2014. Special issue on Future Internet Testbeds Part I.

[79] Internet Engineering Task Force. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*, September 1981.

[80] Internet Engineering Task Force. *RFC 1546 -- Host Anycasting Service*, November 1993.

[81] D. Ferone, P. Festa, F. Guerriero, and D. Lagan. The constrained shortest path tour problem. *Computers & Operations Research*, 74:64 -- 77, 2016.

[82] S. Irnich and G. Desaulniers. *Shortest Path Problems with Resource Constraints*, pages 33--65. Springer US, Boston, MA, 2005.

[83] T. Ibaraki. Algorithms for obtaining shortest paths visiting specified nodes. *SIAM Review*, 15(2):309--317, 1973.

[84] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379--394, 1989.

[85] P. Festa. Complexity analysis and optimization of the shortest path tour problem. *Optimization Letters*, 6(1):163--175, 2012.

[86] P. Festa, F. Guerriero, D. Lagan, and R. Musmanno. Solving the shortest path tour problem. *European Journal of Operational Research*, 230(3):464 -- 474, 2013.

[87] I-L. Wang, E. L. Johnson, and J. S. Sokol. A multiple pairs shortest path algorithm. *Transportation Science*, 39(4):465--476, 2005.

[88] B. A. Carre. A matrix factorization method for finding optimal paths through networks. *Computer Aided Design*, 51(4):388--397, 1969.

[89] B. A. Carre. An elimination method for minimal-cost network flow problems. *J.K.Reid (Ed.), Large sparse sets of linear equations (Proc. I.M.A. Conf., Oxford, 1970). Academic Press: London*, 1971.