Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Road Graph Simplification for Minimum Cost Flow Problem

Daria Lebedeva

Supervisor: Ing. Martin Schaefer
May 2019

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Lebedeva Daria**	Personal ID number:	**465869**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Road Graph Simplification for Minimum Cost Flow Problem

Bachelor’s thesis title in Czech:

Zjednodušení silničního grafu pro výpočet nejlevnějších toků

Guidelines:

Consider the minimum cost multicommodity flow problem on a road network. The computation of the problem is hard due to the complexity of the road graph. A standard approach is to consider only the main roads of the analysed area and to assume that the simplified graph of the main roads represents well the original one. The goal of the thesis is to formalize the process by explicitly considering the road network parameters and the given combinatorial problem in the simplification process.
Research the literature for approaches to graph simplification and the literature for the related flow problem.
1. Design a simplification algorithm and the mapping of the solution to the original road network.
2. Implement your solution.
3. Evaluate the performance of the solution regarding the simplification level and the problem solution quality.

Bibliography / sources:

[1] Toivonen, Hannu, Sébastien Mahler, and Fang Zhou. "A framework for path-oriented network simplification." International Symposium on Intelligent Data Analysis. Springer Berlin Heidelberg, 2010.
[2] Chekuri, Chandra, Thapanapong Rukkanchanunt, and Chao Xu. "On element-connectivity preserving graph simplification." Algorithms-ESA 2015. Springer Berlin Heidelberg, 2015. 313-324.
[3] Ruan, Ning, Ruoming Jin, and Yan Huang. "Distance preserving graph simplification." 2011 IEEE 11th International Conference on Data Mining. IEEE, 2011.

Name and workplace of bachelor’s thesis supervisor:

Ing. Martin Schaefer, Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.01.2019** Deadline for bachelor thesis submission: **24.05.2019**

Assignment valid until: **30.09.2020**

Ing. Martin Schaefer	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to express my deepest gratitude to my supervisor Martin Schaefer for guiding my work in the right direction, and for persistent readiness to answer any questions. I am also grateful to my boyfriend and parents for their endless moral support.

Declaration

Abstract

In this work we consider the Minimum Cost Multicommodity Network Flow (MCMNF) problem as a key problem for traffic routing. The routing problem is recurring, it should be solved many times a day on a daily basis. So we present a solution that may be successfully used in the long term. We make use of a periodic demand pattern, i.e. vehicles' directions are in general recurring daily. Our improvement is based on column generation method, that allows us to reuse vehicles paths from previous days in the solution process. We achieved a 40% reduction of computational time, while the optimal solution is preserved.

Keywords: Network flows, MCMNF, column generation, traffic routing

Supervisor: Ing. Martin Schaefer

Abstrakt

V této práci se zaměřujeme na problém výpočtu nejlevnějších toků jako na klíčový problém pro řízení dopravního provozu. Tento problém se řeší pravidelně během dne, tj. nejde o nalezení řešení jednou, ale o dlouhodobý proces, ve kterém se pořád hledá řešení toho samého problémů s různými vstupy. Proto představujeme řešení, které může být úspěšně použito v dlouhodobém horizontu. Předpokládáme, že v poptávce existuje periodický vzor, tj. směr vozidel se obecně opakuje denně. Naše zlepšení je založeno na metodě generování sloupců, která umožňuje opětovné použití cest vozidel z předchozích dnů při vyhledávání řešení. Dosáhli jsme snížení výpočetního času o 40% při zachování optimality řešení.

Klíčová slova: Toky v sítích, nejlevnější toky, metoda generování sloupců, řízení provozu

Překlad názvu: Zjednodušení silničního grafu pro výpočet nejlevnějších toků

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays, traffic congestion became a real scourge of humanity. In 2018 every American spent an additional 97 hours travelling, and $87 billion were lost in total due to traffic jams [9]. There are some general approaches to fight this problem: road network may be extended, the public transport system should cover as much area as possible and should be suitable and comfortable for everyone. Alternative modes of transport, such as bicycles or electric scooters, should also be promoted - for this purpose rental systems could be set up. However, these are all expensive long-term approaches. Assuming the road network and the demand are fixed, we can try to avoid congestion by routing the traffic.

Figure 1.1: Times Square traffic jam in New York City.
Photo credits: joiseyshowaa from Freehold, NJ, USA [CC BY-SA 2.0]

It is desirable for every driver to avoid congestion, so there are congestion aware navigation systems that use real-time traffic data to suggest routes that are possibly not the shortest but the fastest for each driver. So if we assume that everyone is following the navigation device, leading navigation providers have the power to influence so much traffic that they can cause congestion on the other roads. Ideally, we would like to get to the situation when we have perfect information where all the traffic is going; then we could plan a route that would be the best response to the current traffic situation. We are approaching so-called Nash equilibrium, when everyone's route is the best with respect to routes of others.

If our goal is to minimise total travel time for all drivers, we may achieve better result only if we coordinate the drivers, i.e. we are moving from user equilibrium to system optimum. However, it means that not everybody would use the fastest route, and some drivers are required to sacrifice and choose slower routes for the good of the system [14]. In fact, it is unlikely with the human driver but is feasible in the possible future of transportation, where mobility becomes a service, and all the demand is served by a fleet of autonomous vehicles. In this setting fleet could be controlled centrally, and the routes could be coordinated.

Obviously, congestion is caused by too many vehicles on the same road at the same time. The relation of density, volume and speed of traffic is extensively studied research field of congestion modelling. We consider the simplest threshold congestion model, where number of cars travelling through a road segment is constrained by a maximum number of cars per time unit. We refer to this number as a capacity of the road segment. We also assume that if the number of vehicles on the segment is smaller than its capacity, the speed on this segment is static (so-called free flow speed). To minimise total travelling time, we need to know the amount of time required to drive through each road segment. We may assign the length of the segment divided by maximum allowed speed as travel time for this segment (assuming that capacity is not exceeded there). If we are talking about a city, number of simultaneous requests is vast, and, as we mentioned before, we want to consider all of them at once. So to say, we combine individual requests with the same origin and destination points and obtain a demand, which consists of a large number of requests for movement of a certain number of vehicles from one point (origin) to another (destination). If we represent the road network as a graph, where nodes are crossroads and edges are road segments between them, and assign capacity and travel time parameters to each edge, the goal will be to find a path for each request (origin-destination pair) such that total travelling time for all paths[1] will be minimized and capacity constraint will be satisfied for each edge[2]. This formulation naturally leads to network flows

[1]Travelling time for the path is a sum of travelling times for all edges which it contains.

[2]Capacity constraint is satisfied if number of paths containing the edge is less than or equal to its determined capacity value.

problem, where we need to find a flow with minimum cost (cost is travel time in our case). More precisely, we may formulate this task as a Minimum Cost Multicommodity Network Flow (MCMNF) problem [18].

Not surprisingly, the size of this problem is huge, especially for large cities such as New York City (NYC). To be more specific, the NYC road network converted to a graph consists of 126 987 nodes and 335 407 edges, that means more than 10^{10} of potential origin-destination pairs. If we examine the yellow taxis' data [5], which make up only a part of the traffic, there are more than 10^7 requests in August 2015, i.e. more than 220 requests per minute. There are several solutions for the MCMNF problem, but due to the size of the problem, all of them require much memory and computational time.

We consider the MCMNF problem as a key problem for routing approaches that would make congestion-free routing possible for large cities. Now its solution requires significant computational costs, first of all, because of the size of the problem. So, the purpose of this thesis is to find a way to simplify the problem so that it is scalable with the size of the city and practically applicable for the traffic routing. Of course, to defeat traffic jams, it is not enough to solve this problem only once. As we need to be able to solve it at every moment for many days, we consider the routing problem to be repeated. So to say, our goal is not to route vehicles at this particular moment as fast as possible, but to invent a simplification that helps to solve the repeated routing problem in the long term. It may require some computations once, but as a result we obtain a faster way to solve the problem on a daily basis.

The remainder of this work is organised as follows. In Chapter 2 we briefly review existing problem approaches and techniques, that may be useful. Then we formulate the necessary theoretical preliminaries and formalise the MCMNF problem in Chapter 3. We describe the solution of MCMNF problem, that we use as a base, and introduce our improvements to the solution algorithm in Chapter 4. Different ways to use our approach, results of experiments and comparison with a basic solution are presented in Chapter 5. Chapter 6 contains the conclusion.

Chapter 2
Related work

In this chapter, we discuss the ways to solve MCMNF problem and existing simplification techniques. MCMNF could be understood as a set of single-commodity network flow subproblems, which may be solved independently. However, there are general constraints, that force subproblems to interact with each other. Each subproblem requires sending a certain number of vehicles from one point to another, and this is a part of the problem, where subproblems are completely independent of each other. But there is also a capacity constraint on each edge, which should be taken into account for all subproblems together, so that total flow (number of vehicles) on each edge should be less or equal to its capacity. This explains why MCMNF problem is much more complex than a set of single-commodity network flow subproblems. MCMNF is naturally defined as a linear program, where total flow cost is minimised with respect to three types of constraints: capacity constraint for each edge, *mass balance* constraint, which is responsible for difference between input and output flow in each node and non-negativity constraint for each variable, because flow on edge could not be negative. We show the exact definition of this linear program in Chapter 3.

2.1 Solution techniques

In general MCMNF problem could be solved by means of a linear program. The problem is that this program is too large to be solved by common methods. There are several techniques that allow us to solve large-scale linear programs, we describe two of them below.

The first technique for solving MCMNF problem is **Relaxation algorithm**, which is based on Lagrangian relaxation. For each node a weight (so-called node potential) is determined. On each iteration of the algorithm Lagrangian relaxation is done by relaxing mass balance constraints, i.e. we subtract those constraints multiplied by determined weights from the objective function. Therefore, we do not have to satisfy those constraints, but we are penalised for violating and rewarded for satisfying them. Penalty or reward is proportional to the weight of the node and the size of imbalance in this node. Relaxation algorithm works iteratively and on each step performs one of two following operations: 1) modifies node potentials if for those potentials exists a solution with higher objective function value, i.e. moving to solution with smaller penalty, or 2) replaces already found solution with another, that is also optimal, but excess in at least one node is decreased, or, in other words, new solution is less constraints violating in terms of original problem. On each operation, objective value is increased or remains unchanged with decreasing of solution infeasibility [1].

The second solution technique is a **column generation** method, based on the idea that in the optimal solution only some variables are used, e.g. in our problem flow on most edges/paths is zero. Problem is then divided into two parts: *restricted master problem (RMP)*, which is the original problem but considering only subset of variables (columns in a coefficient matrix) and *the pricing problem*, used to determine new variables (columns) which could potentially improve the solution. The solution algorithm then consists of two repeated steps: on the first step RMP is solved on the subset of variables, and its solution is used on the second step to determine new variable(s) to be added by solving the pricing problem. The steps are repeated until the optimal solution is found [11].

Some more algorithms for solving this problem are described in [1]. In [19] is shown, that the column generation method is generally faster than the relaxation algorithm for the MCMNF problem. There are also other features of this algorithm, which make it the most suitable for our purposes, we discuss them in Chapter 4.

2.2 Simplification techniques

In this section, we review existing simplification techniques, which are based on changing the graph as an input of the algorithm. The rationale behind that approach is that if we reduce the graph, the solver will need less time to find a solution. All simplifications may be generally divided into two categories: one preserves the original solution, the other only resembles it. If edges or nodes, which are excluded, do not appear in any solution, this solution belongs to the first category. Such solution is, of course, more reliable, but it can be costly to calculate such a simplification. The general rule is that it should be possible to reconstruct the solution in the original graph, and any solution may be reconstructed this way. We refer to such simplifications as *solution preserving*. In the second category structure of the graph may be changed, some elements may be ignored, so it may not be possible to construct the same solution as in the original graph. However, the solution in a simplified graph may be fairly similar to the original one, meaning "relevant" edges are preserved. It is important not to exclude paths, that may be relatively good in terms of the problem, i.e. may appear in some solution. We call those simplifications *solution reducing*.

One of the examples of the solution preserving simplifications is removing "useless" edges with respect to each commodity. Algorithm for single-commodity network flow is introduced in [2] and improved in [12]. The key idea is to remove edges which do not belong to any simple path from source to sink node, hence will not be used in the solution. Such edges are named "useless". The task of finding all useless edges in a directed graph is NP-complete, so the algorithm removes only part of them. In case of a planar graph, there is an algorithm which removes all useless edges (introduced in [2]). However, it imposes restrictions on the graph, that can be bypassed in case of maximum flow problem by transforming the graph, but this is not our case. We can naturally expand this algorithm (the first one, without the requirement of planarity) for multicommodity network flow and remove edges, that are useless for all source-sink pairs. But due to a large number of demand pairs and fairly good connectivity of the road network, it is not rational in our case.

One of the common techniques belonging to the solution reducing category is to consider only main roads everywhere or on certain segments of the path, i.e., near the origin and destination node all edges could be considered, but in the middle of the path only main roads are used. Edges are filtered out based on the class of the road (highways, freeways etc.) or capacity with respect to some limiting lower value [8],[10]. This leads to absolute ignorance of paths containing small streets in the middle of the way. Although this speeds up the calculations, the lack of some paths makes this method not very reliable.

Next example of a solution reducing algorithm is described in [17]. Its key idea is to define a measure function for the paths and preserve the best paths between all pairs of nodes. Common examples are preserving the maximum flow (i.e., maximum path capacity between each two nodes, where capacity of the path is a minimum of its edges' capacities, remains the same) or the cheapest path (i.e. minimum cost of the path between any two nodes remains the same, where the cost of the path is the sum of its edges' costs). However, none of these approaches fully describe our problem, as we need to preserve both costs and capacities. If we restrict to the shortest[1] paths, the capacity of remaining edges may not be sufficient to solve the problem so the original solution may not fit into a pruned graph. Similarly, if we use a maximum flow parameter to prune the graph, we may remove the cheapest edges with small capacities, belonging to the original solution. Furthermore, this approach does not guarantee any better solution, as we are focused on preserving maximum flow, which is not necessarily in the solution, because edges with bigger capacities may have higher costs.

There are also some simplification techniques that can be categorised as solution reducing and can be used, for example, in network design, but are not applicable to our problem. We give two examples of such simplifications

[1]From now on, we use the term "shortest" for the cheapest path in terms of edges' costs.

based on the idea of identifying a set of important nodes and somehow preserving connectivity between them. This approach is similar to clustering, described in Chapter 4. But in case we want to preserve information about edges' costs and capacities, such complete restriction to a subset of nodes is undesirable. First algorithm based on element-connectivity is described in [4]. A subset of graph's nodes is called "terminals", and element-connectivity between two nodes is measured as a number of paths which do not have joint edges or non-terminal nodes. Then a reduced graph is computed by removing or contracting edges between non-terminals, while element-connectivity is preserved for each pair of terminals. As a result, non-terminals make up an independent set, i.e., there are no edges between them. This approach is perfect for analysing the structure of a graph between some important nodes when other nodes do not matter, but is not applicable in our case, because with contracting and removing edges all edges between non-terminals we lose the structure of the graph, which is necessary to solve our problem. Another example of such simplification is described in [15]. This simplification tries to preserve shortest paths' distances with the graph restricted to a smaller set of nodes. Its key idea is based on determining *gate nodes*, which are used to reconstruct the shortest path between each pair of nodes. With some threshold $\epsilon > 0$ each pair of nodes is called *local* if the distance between them is lower than a threshold and *non-local* otherwise (if a path between those nodes exists). Gate nodes are chosen so that for every non-local pair of nodes there is a path that consists only of gate nodes (except start and end nodes), every two consecutive nodes in a path are local and path length (i.e. the sum of distances between consecutive nodes) equals original distance between start and end nodes. We can say that gate nodes are showing the topology of the graph. Then some edges which do not contribute to any shortest path are removed. The major drawback of this approach is that it is defined on an unweighted and undirected graph, which makes it impossible to use in our case. Also, the shortest paths are not enough to solve the MCMNF problem, as they do not necessarily hold all the vehicles that need to be sent.

Chapter 3

Preliminaries

In this chapter, we give two definitions of network flows: in addition to the standard definition, we introduce a definition using paths, which is more useful for the purposes of this thesis. We also formulate the Minimum Cost Multicommodity Network Flow (MCMNF) problem as a linear program in both cases.

3.1 MCMNF (basic formulation)

First, let us define a MCMNF problem. Given a graph G (road network in our case) and a demand M (set of requests giving number of vehicles going from some origin to destination per time unit), we want to find a flow f with a minimum cost which describes movements on graph's edges for each request. Here we are dealing with so-called steady-state flow, i.e., given input is not changing over time, demand is static. So, given requests may be interpreted as a constant load between origins and destinations. Our goal is to compute so-called output flows - routing policy for all vehicles. As the demand is static, we do not consider vehicles' movements in time, i.e. capacity of each edge gives the number of vehicles that could go through it per time unit.

3.1.1 Network flow definition

$G = (V, E, c, \sigma)$ is a directed graph with capacitated edges, where V is the node set and $E \subseteq V \times V$ is the edge set. Nodes correspond to road junctions and edges to road links. The capacity of road links is denoted by $c : E \to \mathbb{R}_{>0}$ where $c(e)$ corresponds to the capacity of the road link e, measured in the number of vehicles per time unit. Each road link, in addition, has a cost per unit, which is defined by $\sigma : E \to \mathbb{R}_{\geqslant 0}$ and may be understood for example as fuel consumption or travel time on this link.

Demand $M = \{(s_i, g_i, d_i),\ i = 1, ..., |M|\}$ is a set of tuples (s_i, g_i, d_i) where s_i and g_i is an origin-destination pair, $s_i, g_i \in V$ and d_i gives number of vehicles to be routed per time unit.

For each component of the demand we are looking for a flow $f_i : E \to \mathbb{Z}_{\geq 0}, i = 1, ..., |M|$ which satisfies the laws of network flows and fulfils the demand (inspired by [6]):

$$\sum_{e \in E^-(v)} f_i(e) = \sum_{e \in E^+(v)} f_i(e), \qquad \text{for all } v \in V \backslash \{s_i, g_i\},\ i = 1, ..., |M| \tag{3.1}$$

$$\sum_{e \in E^-(s_i)} f_i(e) + d_i = \sum_{e \in E^+(s_i)} f_i(e), \qquad \text{for all } i = 1, ..., |M| \tag{3.2}$$

$$\sum_{e \in E^-(g_i)} f_i(e) = \sum_{e \in E^+(g_i)} f_i(e) + d_i, \qquad \text{for all } i = 1, ..., |M| \tag{3.3}$$

$$\sum_{i=1}^{|M|} f_i(e) \leq c(e), \qquad \text{for all } e \in E \tag{3.4}$$

Here and beyond we assume that sum over an empty set equals zero. Meaning of laws is worded as follows: (3.1) stands for equality of input and output flow in each node except for origin and destination. (3.2) and (3.3) give that required amount of flow coming out from origin node and coming into the destination. (3.4) shows that capacity constraints should be respected. $E^-(v)$ and $E^+(v)$ stands for incoming and outgoing edges for the node $v \in V$ respectively.

3.1.2 Linear program

Most of existing solutions deals with MCMNF problem as a linear program, so it is a good moment to formulate it. In fact, it is not necessary to consider the flow for each demand component separately. It is sufficient to find a flow on each edge, while three types of constraints should be satisfied: capacity constraint for each edge, mass balance constraint for each node and non-negativity constraint for flow. Mass balance constraint stands for the difference between input and output flow in a node. Their difference should correspond to the difference between sum of demands for which the node is a destination and sum of demands for which the node is an origin. If a node is not an origin or destination for any demand, both differences should be zero. The problem could be stated as follows (based on [1]):

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} \sigma(e) x_e \\
\text{subject to} \quad & \\
& x_e \leq c(e), && \text{for all } e \in E \\
& \sum_{e \in E^-(v)} x_e - \sum_{e \in E^+(v)} x_e = b(v), && \text{for all } v \in V \\
& x_e \geq 0, && \text{for all } e \in E
\end{aligned}$$

where $b(v)$ stands for the mass balance in the node v and can be computed as follows:

$$b(v) = \sum_{(s,v,d)\in M} d - \sum_{(v,g,d)\in M} d \tag{3.5}$$

Variable x_e stands for the flow on edge e.

3.2 Formulation using paths

In this section, we introduce another definition of flows on the graph, which is fairly natural while we are looking for a path for each request. This definition is based on the idea that every vehicle going from origin to destination node goes by some simple path[1], and routes of all vehicles could be described as a set of paths between origin and destination nodes. In this case, flow f_i is not a mapping on edges, but on paths. That is to say, for every simple path $f_i : P_i \to \mathbb{Z}_{\geq 0},\ i = 1, ..., |M|$ gives amount of flow on this path, where P_i stands for the set of all possible simple paths between origin s_i and destination g_i. Amount of flow, given by f_i, is the same for all edges along the path. We define the set of paths which contain an edge e as $N(e)$, the set of paths coming into the node v (going through or ending in it) as $E_{in}(v)$ and the set of paths coming out of the node v (going through or starting in it) as $E_{out}(v)$.

$$N(e) = \{p \mid e \in p \wedge \exists\ i : p \in P_i\}$$

$$E_{in}(v) = \bigcup_{e\in E^-(v)} N(e),\ E_{out}(v) = \bigcup_{e\in E^+(v)} N(e)$$

Laws of the network flows must be similarly satisfied for this formulation:

$$\sum_{i=1}^{|M|} \sum_{p\,\in\, E_{in}(v)\cap P_i} f_i(p) - \sum_{i=1}^{|M|} \sum_{p\,\in\, E_{out}(v)\cap P_i} f_i(p) \ = \ b(v),\ \forall v \in V \tag{3.6}$$

$$\sum_{i=1}^{|M|} \sum_{p\,\in\, N(e)\cap P_i} f_i(p) \ \leq \ c(e),\ \forall e \in E \tag{3.7}$$

where $b(v)$ is node's mass balance defined in (3.5). Meaning of laws remains the same: (3.6) shows that required amount of flow should come out from each node, and come into each node, i.e. difference of input and output flow in each node should be equal to the difference of demands, for which this node is destination and origin node respectively. (3.7) corresponds to capacity constraints, i.e. the sum of flows going through each edge should be less or equal to its capacity.

[1]Simple path is a path in which all nodes are distinct.

Linear program using path model does not require any major changes, except that variables stand for flow amount on paths, not edges:

$$\text{minimize} \quad \sum_{p \in P} \sum_{e \in p} \sigma(e) x_p \tag{3.8}$$

subject to

$$\sum_{p \in N(e)} x_p \le c(e), \qquad \text{for all } e \in E \tag{3.9}$$

$$\sum_{p \in E_{in}(v)} x_p - \sum_{p \in E_{out}(v)} x_p = b(v), \qquad \text{for all } v \in V \tag{3.10}$$

$$x_p \ge 0, \qquad \text{for all } p \in P \tag{3.11}$$

where $P = \bigcup_{i=1}^{|M|} P_i$ is a set of all simple paths between some origin-destination pair from the demand.

This formulation of the linear program has an advantage that the solution on the graph is easy to reconstruct: as output we get the number of vehicles that should be sent by each path. In contrast, in the edge model, as a result, we receive only amount of flow on each edge, and are additionally required to reconstruct the flow for each demand element. However, in edge model $|E|$ variables and $|V| + 2|E|$ constraints are presented, while in path model there are $|P|$ variables and $|E| + |V| + |P|$ constraints. Number of simple paths in the graph is huge, even if we consider only a subset of them (P consists of paths between origin-destination pairs). It is nearly impossible to solve the problem in the path model until we are using column generation method, which allows to consider only a subset of paths and iteratively generate new paths when needed.

Chapter 4

Solution

In this chapter, we explain the evolution of our approach. We also describe the column generation technique, which is used by MCMNF solver that we have been given, in greater detail. As we mentioned before, we need to solve the problem repeatedly on a daily basis. Thus, we call the solver for each time slot to find a solution. Our purpose is to improve this solver by re-using paths which were computed for previous time slots. Our simplification may be classified as *solution preserving*, as we do not remove any elements of the graph completely. Its key idea is that solver receives a set of paths that are likely to be used in the solution, so it is not needed to recompute them in each solver run.

4.1 Naive approach

Now we describe the evolution of our approach: where it comes from, and why a path model described in Section 3.2 is useful to MCMNF problem. First, we describe a naive problem-solving approach, on which we are based, and which is improved in Section 4.3.

In the absence of capacity constraints, the shortest paths from origin to destination node for each demand component are sufficient to solve the problem. If those paths are not sufficient in reality (i.e., some capacity constraints are violated), we keep as many vehicles as possible while not exceeding any capacity value and look for the second shortest path, trying to route remaining vehicles by it and so on. However, this might not help at all, if those two shortest paths overlap in the bottleneck (i.e., they have a joint edge with exceeded capacity), or help only partially if some constraints are still violated, and we have to find the next shortest path. By continuing such approach, we get an iteratively increasing set of paths on which we are trying to route the vehicles.

As we said before, we consider the routing problem to be repeated. Given that, repeated computation of the shortest paths between nodes is not desirable.

We may want to store all possible simple paths ordered by their costs, but it is impossible in practice due to graph size. If we knew in advance, which paths we would need, we could consider only those, but, of course, it is also impossible. We may compute some small number (e.g., 10) of shortest paths for every pair of nodes and hope that this number of paths would always be enough to solve the problem. However, if it would not, we are dealing with congestion and we have no ability to construct an optimal solution. In this situation, we may want to add some paths to our set, and there is nothing stopping us from saving them for further use in case the similar situation appears. Our solution is based on the idea of saving used paths and reusing them in further computations. This approach suggests using column generation technique, which solves the optimisation problem using some subset of variables and iteratively increases this set until the optimal solution is found. Variables denote paths in our case. Using this technique, we may pass the set of stored paths to the solver at the start of computation, so there is no need to recompute them.

4.2 Column generation technique

Key idea of column generation technique [11] applied to MCMNF problem is based on iterative addition of paths between each origin-destination pair in demand. First, we take the shortest paths for every pair and use only them to fulfill the assignment (i.e. send a needed amount of flow through each path). Most likely there will be overloaded edges (i.e. total flow on the edge violates its capacity). We adjust the cost of each edge so that overloaded edges have higher costs and will less likely be chosen in the shortest path. Then we recompute the shortest paths for each origin-destination pair and add them to paths we have found before. Then MCMNF problem is solved using only this set, and overloaded edges are found. We repeat those steps until the optimal solution is found, i.e. there are no overloaded edges or possibility to improve the solution.

We will now formulate this technique more formally. As we mentioned in Section 2.1, this is an improvement of linear programming solution, so we should start with the definition of the linear program (LP). However, our formulation, given in Section 3.2, is not very convenient for this method: (3.10) constraint, which stands for the difference of input and output flow in each node, should be recomputed whenever we are adding a new path for each node in this path. This constraint can be reformulated so that for each demand element there is a required amount of flow, which should be equal to the sum of flows over all paths considered for this demand:

$$\sum_{p \in P_i} f_i(p) = d_i, \text{ for all } i = 1, \ldots, |M|$$

Also, we could not solve given program straightforward using column generation method, because restricted master problem (RMP), where we use only subset of paths, is infeasible most of the time (we have strict capacity constraints, and it is quite possible, that given set of paths is not sufficient with respect to edges' capacities). Therefore, we should add an overflowing variable for each edge, i.e. capacity on the edge may be exceeded, but it is penalised proportionally to the exceeding flow on this edge and penalty value is much bigger than any possible total flow cost. With this formulation, RMP is feasible all the time, and its optimal solution refers to the least capacity violating flow (or minimum cost flow, if the solution with no overflowed edges exists).

So, LP could be reformulated as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{p\in P}\sum_{e\in p}\sigma(e)x_p + N_{penalty}\sum_{e\in E} f_e && && (4.1)\\
\text{subject to} \quad & \sum_{p\in N(e)} x_p - f_e \le c(e), && \text{for all } e\in E && (4.2)\\
& \sum_{p\in P_i} x_p = d_i, && \text{for all } i=1,\dots,|M| && (4.3)\\
& x_p \ge 0, && \text{for all } p\in P && (4.4)\\
& f_e \ge 0, && \text{for all } e\in E && (4.5)
\end{aligned}$$

Here x_p stands for the amount of flow on path p and f_e is overflow on the edge e, i.e. allowed number of vehicles going through this edge in addition to its capacity, and N_{number} is a penalty for each vehicle on overflowed edge, which does not fit in its capacity. Dual LP is constructed as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_{e\in E} c(e)y_e + \sum_{i=1}^{|M|} d_i z_i \\
\text{subject to} \quad & y_e \le 0, && \text{for all } e\in E\\
& z_i \in \mathbb{R}, && \text{for all } i=1,\dots,|M|\\
& \sum_{e\in p} y_e + z_i \le \sum_{e\in p}\sigma(e), && \text{for all } p\in P_i,\ i=1,\dots,|M|\\
& y_e \ge -N_{penalty}, && \text{for all } e\in E
\end{aligned}$$

In the beginning subset of paths consists of the shortest paths for each origin-destination pair. All overflowing variables are presented, i.e. we are restricting only set of paths' variables x_p. On each iteration of the algorithm, RMP is solved with a subset of variables (paths), then a solution of pricing problem for each origin-destination pair (PP_i) helps us to determine variables that should be added, based on dual values of RMP solution. In our case it means that costs of edges are recomputed based on values of dual variables y_e, which correspond to capacity constraint (4.2): they indicate which constraints are active (for inactive constraints corresponding dual values are zero). Pricing problem for each demand element is defined as follows:

$$PP_i(\{y_e\}_{e\in E}) := \operatorname*{argmin}_{p\in P_i} \sum_{e\in p} \sigma(e) - y_e$$

In fact, as y_e is non-positive, we are increasing the cost of edges where capacity constraints are active (i.e. those edges are overflowed or on the verge of its capacity), and then looking for the shortest paths contained in P_i, i.e. going between origin and destination for i-th demand element, with respect to new costs. Found paths for each origin-destination pair are then added to the set of variables P.

Algorithm 1: Column generation

```
1  $P \leftarrow \bigcup_{i=1}^{|M|} \mathrm{PP_i}(\mathbf{0})\}$;
2  do
3      $(\{x_i^*\}, \{y_e\}) = \mathrm{RMP}(P)$;
4      newcols $\leftarrow$ False;
5      for $i \leftarrow 1 \ldots |M|$ do
6          $p \leftarrow \mathrm{PP}_i(\{y_e\})$ ;
7          if $p \notin P$ then
8              $P \leftarrow P \cup \{p\}$;
9              newcols $\leftarrow True$;
10         end
11     end
12 while newcols;
13 return $\{x_p^*\}$ ;
```

4.3 Improved approach

Back to naive approach described in Section 4.1, we now see that it is very simplified. There we do not consider the interaction between demand elements. For example, we may not want to consider the second shortest path between some origin and destination, if it overlaps with the first shortest path for other origin-destination pair in some bottleneck edge, or we want to find a trade-off in using these paths. Column generation method deals with this problem by recomputing the costs: bottleneck edge in the example will have higher cost, and the path containing it probably will not be chosen.

This interaction between demand components may vary in time. Thus, final set of used paths depends on the given demand. Therefore, we come up with the idea of making use of periodic demand patterns. For example, on weekday mornings majority of people go to work from residential districts to office areas, and backwards in the evenings. For each time slot information about the same time slot from the previous days is used for presumptive calculations of paths that may be engaged in movements, i.e. may appear in the solution. We pass set of those paths to the solver, which uses them for routing. Solver may in addition generate new paths if needed. Set of used paths is then expanded with new generated ones and may be used in the next day.

The set on which we are trying to solve MCMNF problem consists of paths between origins and destinations. Solver iteratively expands this set until an optimal solution is found. As a result, we receive a set of origin-destination paths on which optimal solution exists, i.e. fleet can be routed with minimal cost. We save this set and provide the next solver instance with it. While only paths from origin to destination may be used to solve the problem, the instance chooses only needed paths from a given set (i.e. going from some origin to destination in its demand). Thus the first solver iteration is computed on a bigger set of paths (basic solver uses only set of the shortest paths instead), and an optimal solution may be found on this iteration. If this did not happen, solver iteratively expands its set of paths until a solution is found. After the solution was found, we expand the set of paths with the new generated paths and pass it on to the next solver instance.

Chapter 5

Implementation and evaluation

In this chapter, we describe the obtained results and compare a basic solver, which uses the column generation technique, with our improved one, which in addition uses results from previous runs to speed up the calculations. We also present the different ways in which improved solver can be used.

First, we describe implementation details and used data in Section 5.1. Then, in Section 5.2 we check the periodicity of the demand. That is to say, demand has a periodic pattern, e.g. directions of vehicles at 8 am on Monday, August 3 in general correspond to those at 8 am on Monday, August 10. Finally, we use this knowledge to present different approaches of using our solver for repeated routing in Section 5.3.

All code is written in Python 2.7, using OSMnx [3] and NetworkX [7] libraries.

5.1 Data

In this section, we describe the data we have used. Records from NYC Yellow taxis are used as a demand, considering tiny part of NYC (1.6 km $\times$ 1.6 km square area). Graph, downloaded using OSMnx, is presented as *MultiDiGraph* from NetworkX library. We use clustering to decrease the number of different origin-destination pairs in the demand. As we solve a steady-state flow problem, we collect all requests from some time interval and use them as a demand for the solver.

5.1.1 Trips data

As a demand to test solver on, we use data from NYC Yellow taxi trip records freely accessible on the official web of NYC Taxi & Limousine Commission [5]. We store it in a PostgreSQL database [16] with the PostGIS extension [13] to allow spatial operations on it (e.g. looking for the nearest node in the graph for a given location). It is worth mentioning that nowadays exact pickup and dropoff locations are not published, and there is access only to taxi zone number from 1 to 263, which are not accurate enough for our purposes. Because of it, we use data from 2015, where information is complete (i.e. pickup and dropoff coordinates are present). CSV file with trips for a given month may be downloaded from [5], the structure of the file is also published there. In this work data from August, September and October 2015 is used. First, we import CSV file as a table to our database, then we remove incomplete rows, where either pickup or dropoff longitude or latitude equals 0. To allow further interaction with data we create columns *pickup_geo* and *dropoff_geo* with corresponding Geography objects [13].

5.1.2 Road network

The task of converting city map to the road network graph is quite complicated. Fortunately, there is a python library OSMnx, which allows converting OpenStreetMap street network into MultiDiGraph from NetworkX. Furthermore, it does correction and simplification of the topology, which helps in the further graph's analyse [3].

We operate with a tiny part of the NYC streets for test purposes. It is an area around the Empire State Building, 800 meters in each of cardinal directions, which we downloaded as a graph using OSMnx and then restricted to the biggest strongly connected component to avoid the absence of paths between nodes, 15 of 191 nodes (7.9%) were deleted. The examined area is shown on Figure 5.1, nodes that were deleted due to strong connectivity are marked red.

Coordinates of the nodes are stored in the database to allow further operations with trips data. We filter out requests from trips data where pickup or dropoff point is not lying in the examined area.

5.1.3 Clustering

Key idea of our algorithm is a repetition of the approximate drive directions. We also mentioned that we may reuse only paths for the same origin-destination pair. For better performance of our algorithm, we introduce clustering of the graph nodes, i.e. we choose a number of clusters and divide nodes into groups using K-means algorithm. The central node is then found in each cluster as the nearest node to the centroid. Then for each of pickup and dropoff locations we find the nearest node from chosen central nodes. All nodes are still present in the graph and used for routing, but requests are made only between cluster centres, e.g. two neighbouring houses are considered as a single point of departure. In our case, we divide nodes into 30 clusters using *ST_ClusterKMeans* function from PostGIS [13] (an average of 5.9 nodes in a cluster). Chosen central nodes are shown on Figure 5.2.

As a result, we are dealing with the graph with 176 nodes and 330 edges, demand consists of 932 995 requests distributed over three months (initially there was 34 140 319 valid requests for the whole NYC).

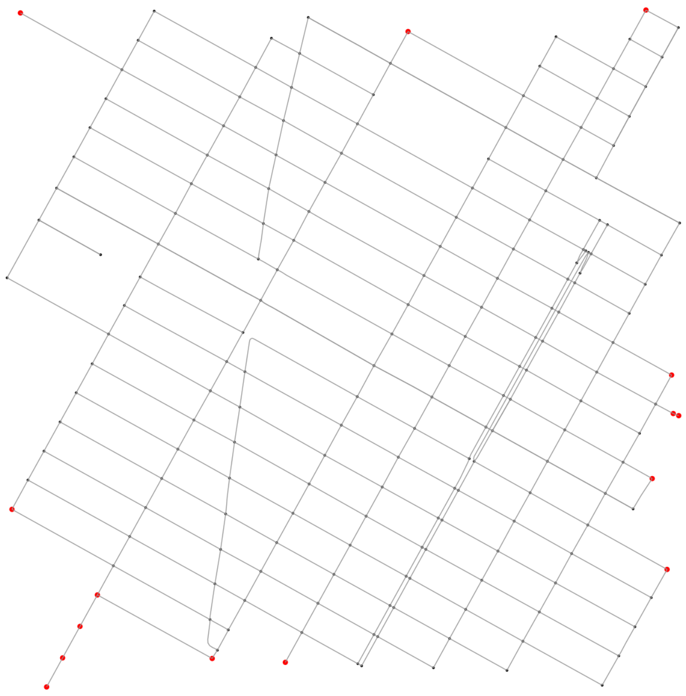


Figure 5.1: Examined area: deleted nodes (red) due to strong connectivity

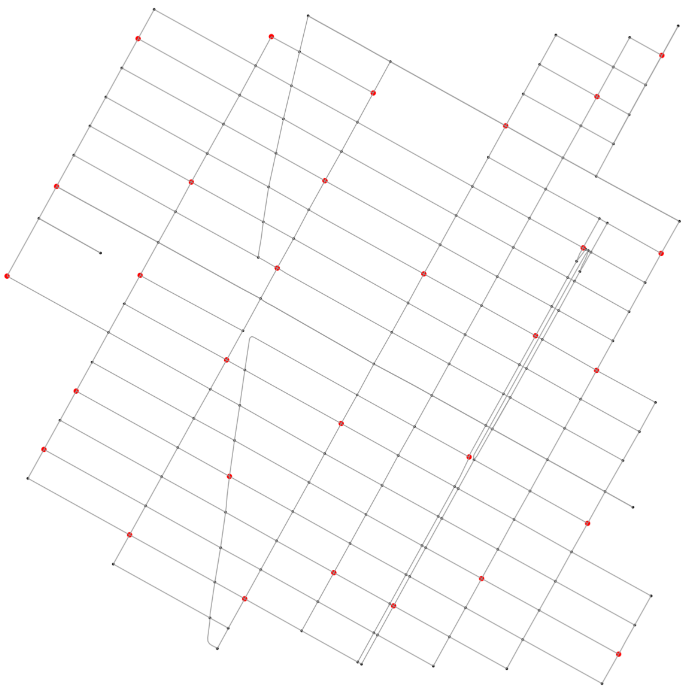


Figure 5.2: Clustering: chosen central nodes (red)

5.2 Demand distribution

First, we examine demand distribution over days and hours to test our theory of periodic demand patterns. As can be seen in Figure 5.4, the number of requests is weekly recurring. On Figure 5.5 we sum up the number of requests in each two-hour interval over three months for each day of the week. It can be seen that distributions for business days are fairly similar, while demand values are varying depending on the time interval. It indicates that the same periodic demand pattern may exist on different business days. So we came up with the idea to reuse paths from the same time intervals on all business days, e.g. paths used at 8-10 a.m. on Monday will be used for the same time interval on Tuesday etc. From now on, we consider only weekdays in our experiments. Weekends may be explored separately, and it is required to test if movements on Saturdays are similar to movements on Sundays. Of course, demand's periodicity is just a hypothesis. If it turns out to be true, our algorithm will work well.

As we have 30 cluster nodes, maximum number of demand elements is $30 * 29 = 870$. In fact, an average number of demand elements in the two-hour interval is 358, i.e. about 40% of possible origin-destination pairs occurs on average in the examined interval. More detailed distribution could be seen in Figure 5.3, where we compute an average number of origin-destination pairs in each two-hour interval, considering only weekdays.

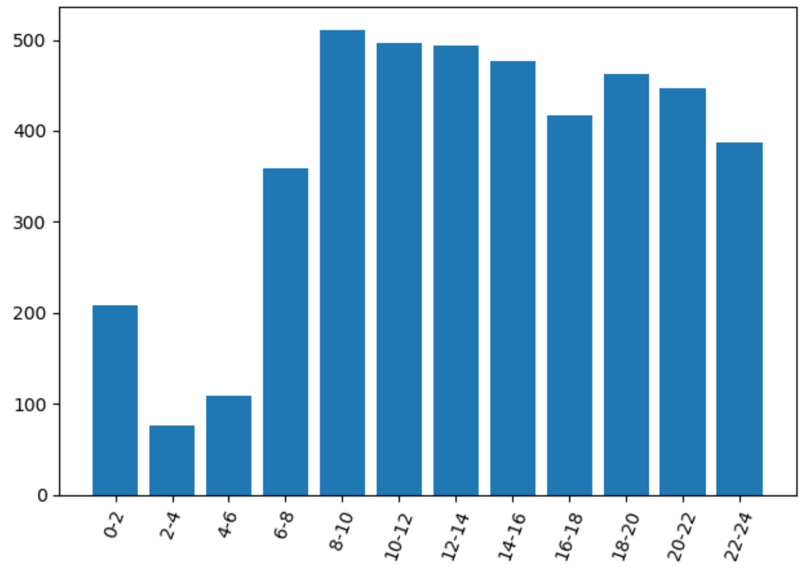


Figure 5.3: Average number of origin-destination pairs (weekdays)

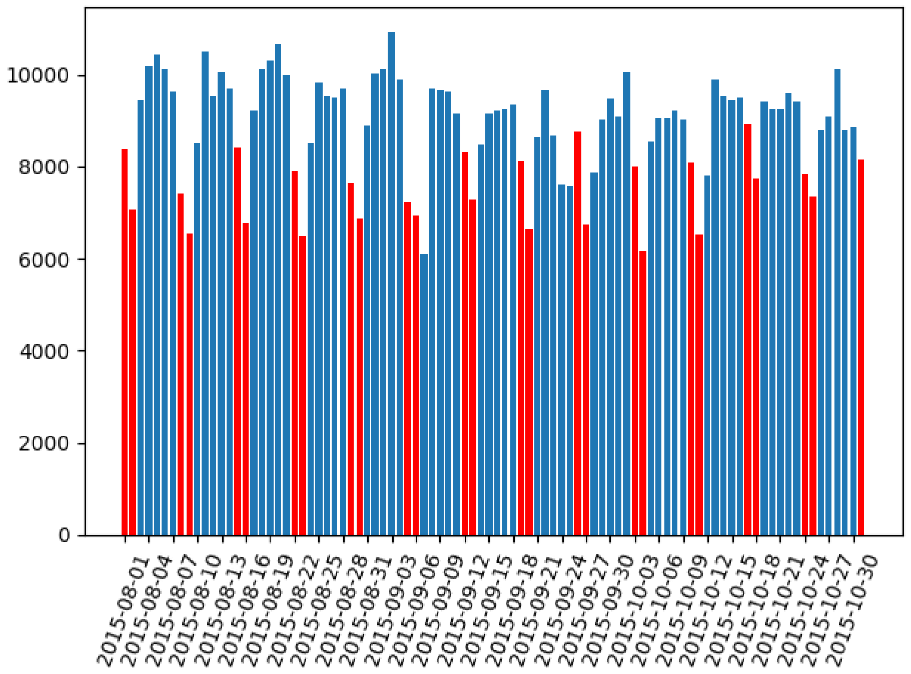


Figure 5.4: Number of requests on each day (red bars show weekends)

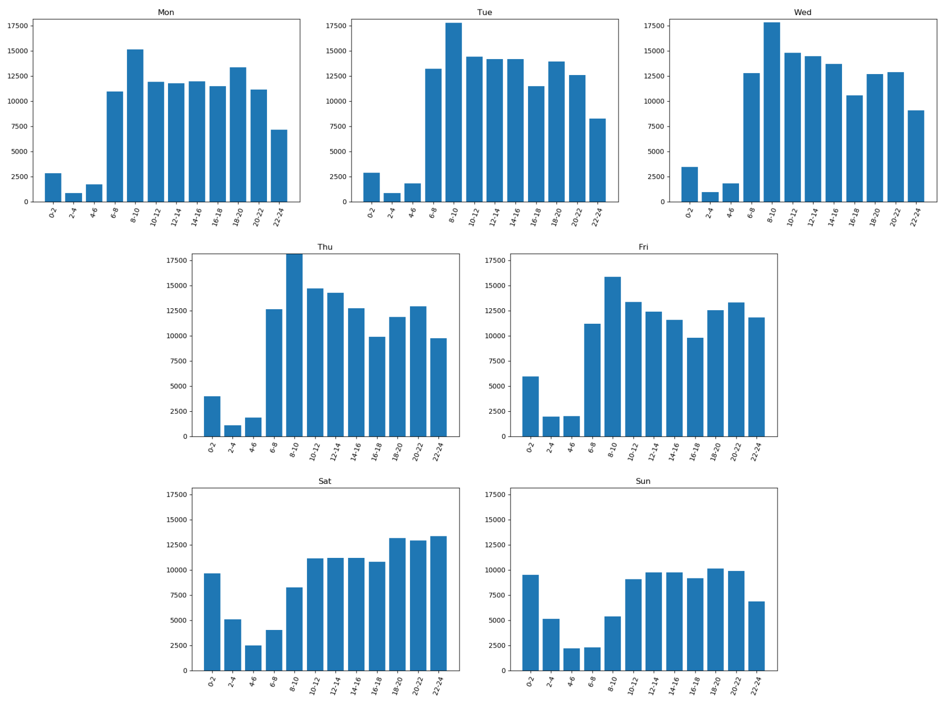


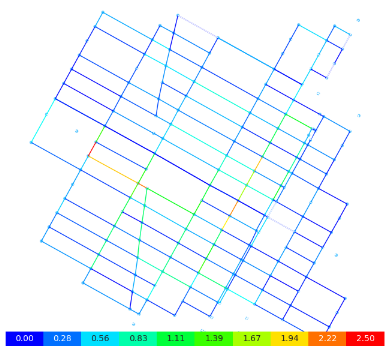
Figure 5.5: Total number of requests for each day of the week in given time interval over three months

5.3 Evaluation

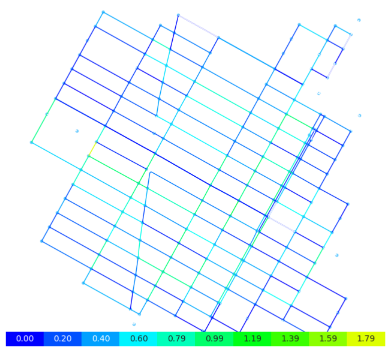
We should determine a capacity value for each edge to get relevant results. We do not have any data on road occupancy, and taxis represent only a part of the traffic, so we decided to determine it experimentally. It is obvious that this value should be proportional to the number of lanes on a given road segment. We determine the capacity scale so that the problem is solvable most of the times. Obviously, it is not possible to determine the value which meets two conditions at once: feasible solution for the problem should exist, but the shortest paths should not be sufficient to solve the problem (in this case basic solver needs only one iteration to find the solution, we can not improve it). It can be seen for example from demand distribution on Figure 5.5, that there are much less requests at night than in the morning rush hours. Thus, we have chosen the value so that for most time intervals (excluding rush hours and night calm) the conditions are met. In case of infeasibility solver will find the solution which is the least capacity violating due to overflow variables described in Section 4.2. Capacity value for each edge is a number of lanes there multiplied by capacity scale. We use capacity scale 100 in following experiments.

As we said before, we work with existing solver that solves the MCMNF problem using the column generation technique described in Section 4.2, as the implementation of such a solver is not a purpose of this work. Our goal is to improve it so that MCMNF problem is solved faster in the long term. So to say, we want to speed up calculations in total, while we are looking for the solution in each time interval on each day. We are doing so by reusing previous results.

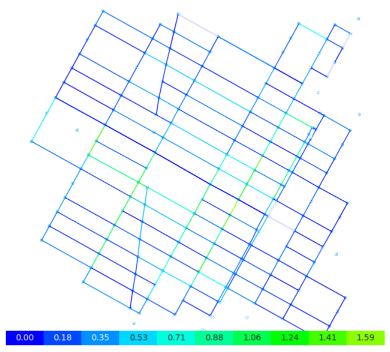
To introduce the solver, we compare its result with a shortest paths approach: for each weekday we computed the shortest path between origin and destination for each request and used it for routing. We chose 8 am - 10 am interval as morning rush hours, where it is not possible to avoid congestion, but it is possible to reduce it, and 12 pm - 2 pm interval, where the solver is able to avoid congestion at all, while the shortest paths approach is not sufficient. On the Figure 5.6 we show average roads load for each of approaches in each interval. Color of the edge corresponds to the average number of cars on this edge divided by its capacity. It can be seen that the shortest paths approach could not completely get rid of traffic jams in both cases (traffic jams occur when the value on the edge exceeds 1), while the solver was able to reduce the congestion in the first case and completely get rid of it in the second.



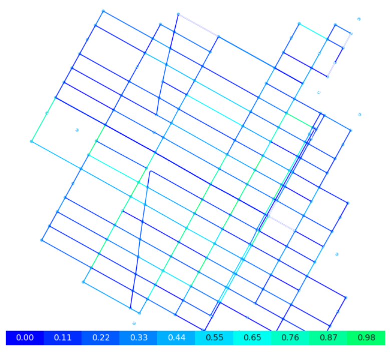
(a) : The shortest paths, 8am - 10am



(b) : Solver, 8am - 10am

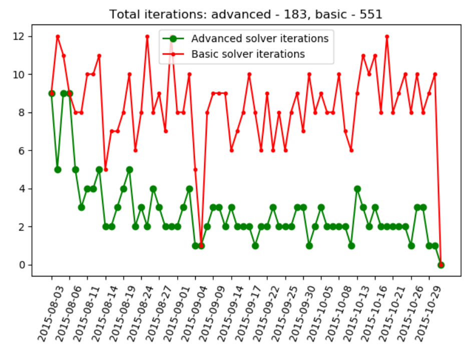


(c) : The shortest paths, 12pm - 2pm

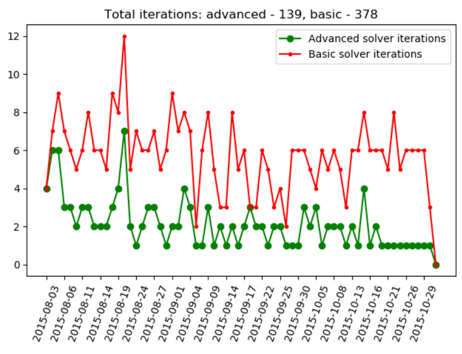


(d) : Solver, 12pm - 2pm

Figure 5.6: Average roads load



(a) : 8am - 10am



(b) : 12pm - 2pm

Figure 5.7: Number of iterations basic vs. advanced solver

Now we compare our solver, improved by reusage of previously computed paths, with the original one. We measure computational time and number of solver iterations (i.e. how many times new shortest paths were computed for all demand pairs). This will give us better idea of the quality of our solution, as the finding the RMP solution on each iteration is usually a bottleneck for a large problem, however, in our testing scenario size of the problem is quite small, so the auxiliary computations (e.g. initializing the solver) takes considerable time compared to the calculation of the shortest paths. Also number of iterations could be used to determine a *simplification level*. We define it as a percentage "improvement", i.e. ratio of difference between number of iterations of basic and advanced solver and number of iterations of basic solver in percents.

$$\text{simplification level} = \frac{\text{basic iterations} - \text{advanced iterations}}{\text{basic iterations}} * 100\%$$

First, we measure the number of iterations in each weekday for the same time intervals: 8 am - 10 am and 12 pm - 2 pm. Result can be seen on the Figure 5.7. We refer to our improved solver as "advanced", and "basic" is the solver without any changes. It can be seen, that number of iterations has dropped significantly, and improved solver mostly needs only one or two iterations to solve the problem, i.e. known paths are (almost) sufficient to route vehicles. Simplification level is 67% in the first and 63% in the second interval.

Now we examine different approaches of re-using data. It is possible, that our hypothesis is wrong, and it makes no sense to reuse only data only from the same time intervals - any reusage will be successful. To test that, we compared three possibilities of re-using data: *daily reusage* (i.e. use data from the same hours from the previous days, paths from 8am-10am in Monday are used for 8am-10am interval in Tuesday), *hourly reusage* (i.e. use data from the same data from previous time intervals, paths from 8 am - 10 am on Monday are used for 10 am - 12 pm on Monday) and *overall reusage* (use all paths computed before, iterate over all time intervals).

	Iterations	Time [min]	Paths	Simplification level [%]
Basic solver	3669	75.4	0	0
Daily reusage	1449	**44.1**	25012	61
Hourly reusage	2524	71.2	**2438.4**	31
Overall reusage	**1055**	56.0	4319	**71**

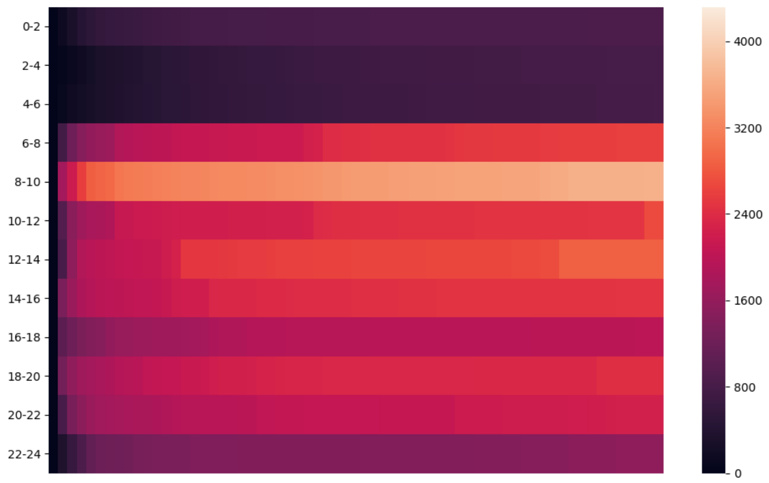
Table 5.1: Comparison of different approaches

We summarise the result in Table 5.1, where we show a number of iterations, total solver time and a number of stored paths for different approaches. Values are summarised over all time intervals in three months (August -

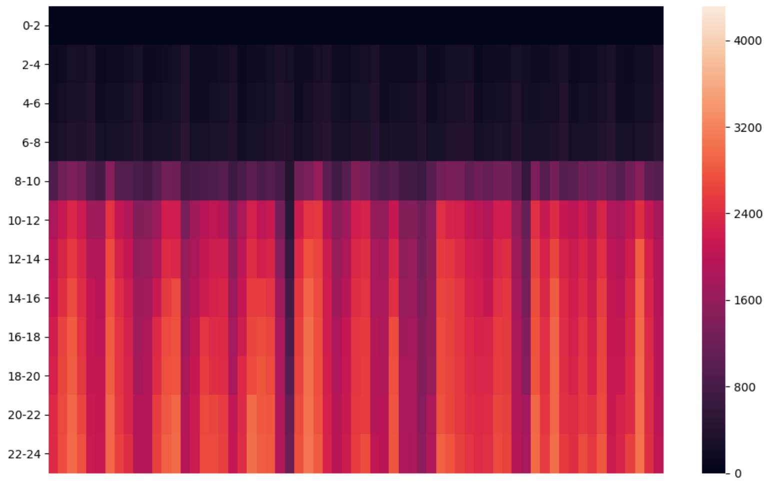
October), only weekdays are considered. In *daily reusage* paths are stored for each two-hour interval, and a total number of stored paths grows with the number of days. For *hourly reusage* number of stored paths increases throughout the day, but all of them may be deleted at the end of the day, as we do not use them in the next days. So, in the table we show an average number of stored paths at the end of the day. In *overall reusage* number of paths increases after each solver run, however, quite quickly the number of paths reaches the level, after which new paths are rarely added. Evolution of the number of paths could be seen in Figure 5.8.

Detailed heatmaps with a number of iterations and ratio between basic and advanced solver in each time interval could be found in Appendix B.

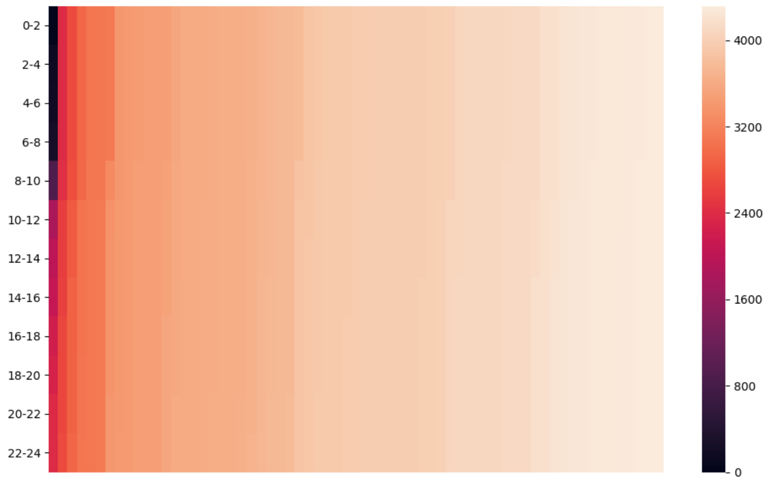
It is evident that *daily reusage* approach works much better than *hourly reusage*. The reason is that solver does not always compute ordinary shortest paths, because costs are changing during iterations. So chosen paths depend on the interaction between demand components at this moment, which vary at different times of the day. Thanks to the obtained results, we can say that our theory is correct, and there are actually periodic patterns in the demand. *Overall reusage* also works, but spent time is higher, as we use all stored paths (over 4000) even for "small" intervals, where shortest paths (i.e. one path per demand element) are sufficient for the solution.



(a) : Daily reusage



(b) : Hourly reusage



(c) : Overall reusage

Figure 5.8: Number of stored paths

Chapter 6
Conclusion

In this thesis, we are supposed to present a simplification for a Minimum Cost Multicommodity Network Flow problem. We decide to move away from the general graph simplification techniques and focus on a specific problem. That is, we do not try to come up with a technique that would be useful for all kinds of graph algorithms. Since the task of traffic routing is currently quite acute, we focus on applying MCMNF problem to it and assume that the demand is in general predictable based on data from previous days. We choose the column generation method as the most appropriate one for our idea and presented an improvement for the solver based on this method. We simplify the graph as an input of the solver passing only subset of graphs' paths, leaving the solver to generate additional paths if needed. Thus, we also make the problem easier for the solver, as it does not need to generate paths it has been given.

Our improvement helps to solve everyday routing problems effectively, taking only 60% of time used by the existing column generation solver to find the solution and reducing number of solver iterations by 60%. There are at least two possibilities to use it depending on what is more important: runtime or required memory. If we use *daily reusage* of paths, we will achieve a shorter runtime, but are required to store used paths for each time slot. Conversely, if we use *overall reusage*, it will take more time to achieve the solution (75% of basic solver time), but the number of stored paths will be almost 6 times less for the two-hour intervals. Moreover, optimal solution will be always found with our improvement, i.e. we are not sacrificing optimality for computational time.

6.1 Future work

There are a few things that may be addressed in future work:

- Some days may be out of the picture. For example, it may be a public holiday or a huge social event. In this case, we store paths that are unlikely to be used in the future. To avoid it, usage of stored paths may be examined, and paths that are not used for several consecutive days should be removed.
- Evolution of the number of stored paths should be examined on a larger dataset. Will the number of paths in *overall reusage* always stop growing at relatively small values? Testing on a larger dataset may also show some new interesting dependencies and repeatabilities, as more scenarios of taxi usages will be covered (we considered an area smaller than $3km^2$ in the centre of the city, which definitely does not include for example shopping trips).
- Taxi rides data are quite one-sided. If a dataset contained data on all trips, including personal and company vehicles, rented cars, were available, the results would be noteworthy. We are guessing that periodic demand pattern will be found there, too.

Appendix A

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: theory, algorithms, and applications*, Prentice-Hall, Inc., 1993.

[2] T. C. Biedl, B. Brejová, and T. Vinař, *Simplifying flow networks*, in International Symposium on Mathematical Foundations of Computer Science, Springer, 2000, pp. 192–201.

[3] G. Boeing, *Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks*, Computers, Environment and Urban Systems, 65 (2017), pp. 126–139.

[4] C. Chekuri, T. Rukkanchanunt, and C. Xu, *On element-connectivity preserving graph simplification*, in Algorithms-ESA 2015, Springer, 2015, pp. 313–324.

[5] *Tlc trip record data*. Retrieved from `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`. [Online; accessed 2019-04-21].

[6] L. R. Ford Jr and D. R. Fulkerson, *Flows in networks*, Princeton university press, 2015.

[7] A. Hagberg, P. Swart, and D. S Chult, *Exploring network structure, dynamics, and function using networkx*, tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[8] Y.-W. Huang, N. Jing, and E. A. Rundensteiner, *Hierarchical path views: A model based on fragmentation and transportation road types.*, in ACM-GIS, Citeseer, 1995, p. 93.

[9] *Inrix: Congestion costs each american 97 hours, $1,348 a year*. Retrieved from `http://inrix.com/press-releases/scorecard-2018-us/`. [Online; accessed 2019-05-02].

[10] K. Ishikawa, M. Ogawa, S. Azuma, and T. Ito, *Map navigation software of the electro-multivision of the'91 toyoto soarer*, in Vehicle Navigation and Information Systems Conference, 1991, vol. 2, IEEE, 1991, pp. 463–473.

[11] M. E. Lübbecke and J. Desrosiers, *Selected topics in column generation*, Operations research, 53 (2005), pp. 1007–1023.

[12] E. Misiołek and D. Z. Chen, *Two flow network simplification algorithms*, Information Processing Letters, 97 (2006), pp. 197–202.

[13] *Postgis 2.5.3dev manual.* Retrieved from `http://postgis.net/docs/`. [Online; accessed 2019-05-19].

[14] T. Roughgarden and É. Tardos, *How bad is selfish routing?*, Journal of the ACM (JACM), 49 (2002), pp. 236–259.

[15] N. Ruan, R. Jin, and Y. Huang, *Distance preserving graph simplification*, in 2011 IEEE 11th International Conference on Data Mining, IEEE, 2011, pp. 1200–1205.

[16] *Postgresql 11.3 documentation.* Retrieved from `https://www.postgresql.org/docs/11/index.html`. [Online; accessed 2019-05-19].

[17] H. Toivonen, S. Mahler, and F. Zhou, *A framework for path-oriented network simplification*, in International Symposium on Intelligent Data Analysis, Springer, 2010, pp. 220–231.

[18] J. Tomlin, *Minimum-cost multicommodity network flows*, Operations Research, 14 (1966), pp. 45–51.

[19] D. Weibin, J. Zhang, and S. Xiaoqian, *On solving multi-commodity flow problems: An experimental evaluation*, Chinese Journal of Aeronautics, 30 (2017), pp. 1481–1492.

Appendix B
Figures

In this appendix we present detailed results in comparison of three different approaches: *daily reusage*, *hourly reusage* and *overall reusage*. We used three months' data for testing, however, for better legibility, we show the results for each month separately.

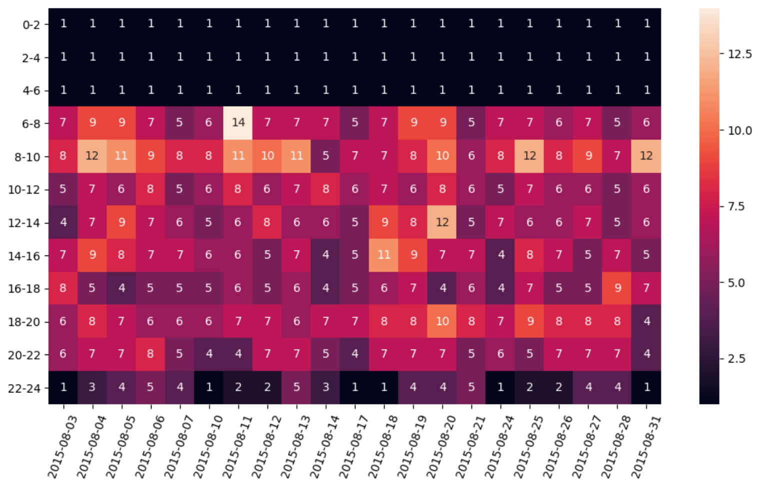
Let us consider the first solvers' comparison figure as an example. In Figure B.1 we see a comparison of basic and improved solver on all two-hour intervals on weekdays in August.

First heatmap shows the number of iterations required for basic solver to find a solution in each interval. Maximum it needed 14 iterations in 6 am - 8 am interval on August 11th. It could also be seen, that in the night (0 am - 6 am) one iteration is sufficient to solve the problem. This is because the demand is little at this time.

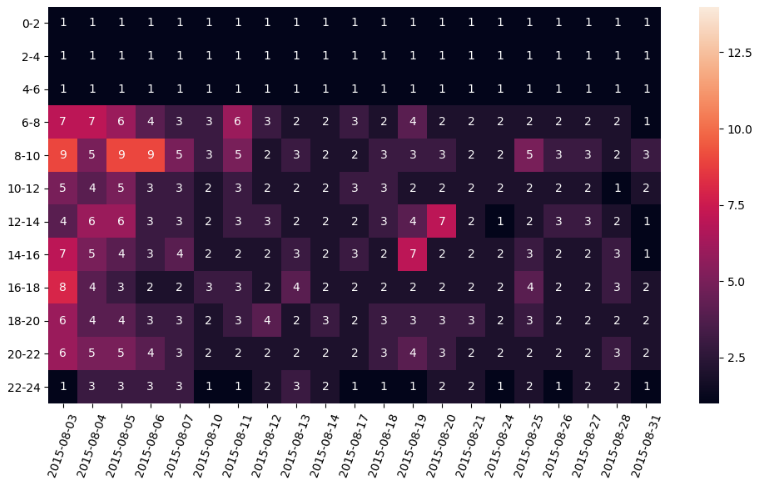
Next heatmap shows the number of iterations required for improves solver to find the same solution. Maximum it needed 9 iterations, always in the morning rush hours (8 am - 10 am). In 6 am - 8 am interval on August 11th it needed only 5 iterations in contrast to basic solver, which needed 14.

The last heatmap shows ratio of two previous ones, i.e. ratio of the number of iterations of the improved solver to the number of iterations of basic one. It could be seen, that "1" (which means that the improved solver could not reduce the number of iterations) appears mostly on intervals, where basic solver has already needed only one iteration, i.e. there is nothing to improve. Those situations appear mostly at night or in the first day, where is no stored paths to use for improvement.

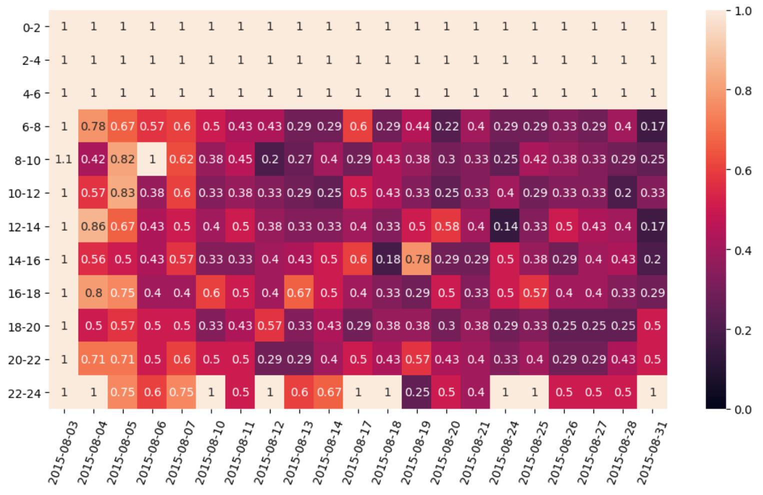
One should not be surprised by the values greater than 1 in ratios (e.g. in Figure B.4c): the number of iterations depends on chosen paths, and this process is not fully deterministic. That is why the number of iterations may vary between two runs of the solver. However, this difference (in our experience) does not exceed 1.



(a) : Basic solver iterations

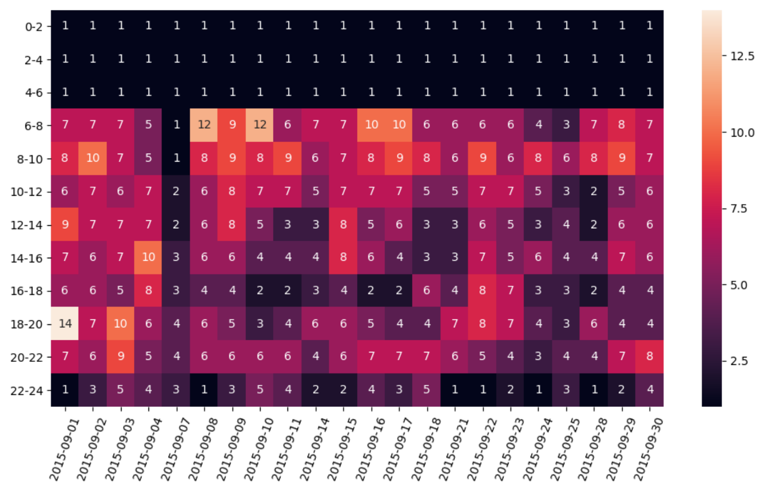


(b) : Advanced solver iterations

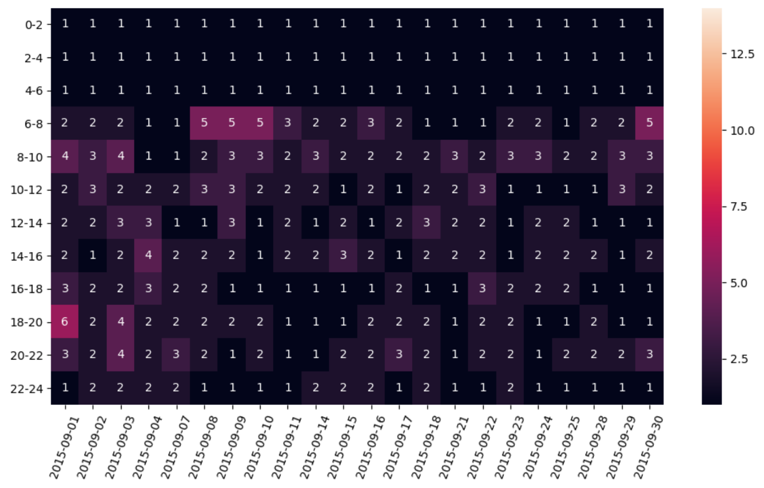


(c) : Ratio advanced/basic

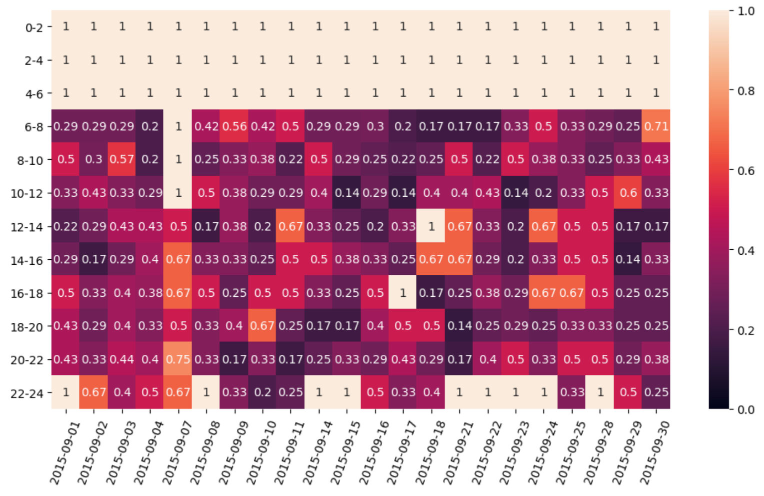
Figure B.1: Daily reusage (August)



(a) : Basic solver iterations

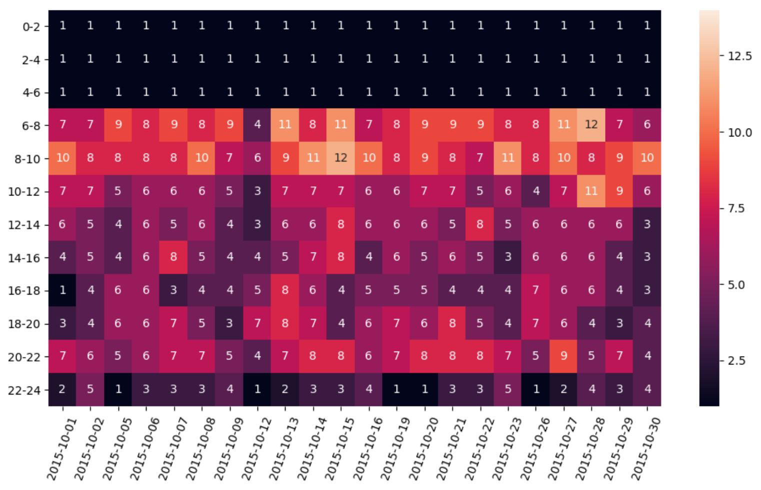


(b) : Advanced solver iterations

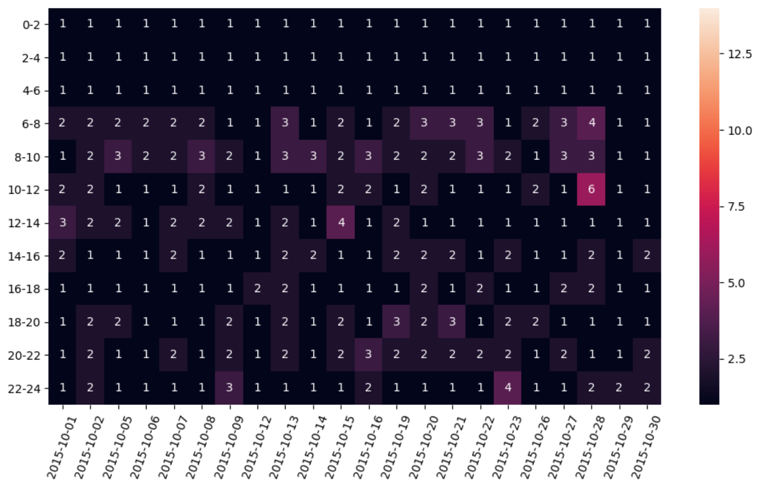


(c) : Ratio advanced/basic

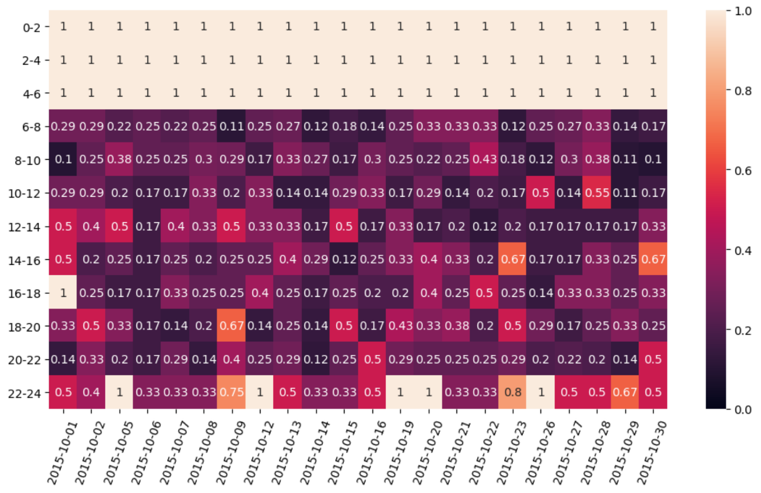
Figure B.2: Daily reusage (September)



(a) : Basic solver iterations

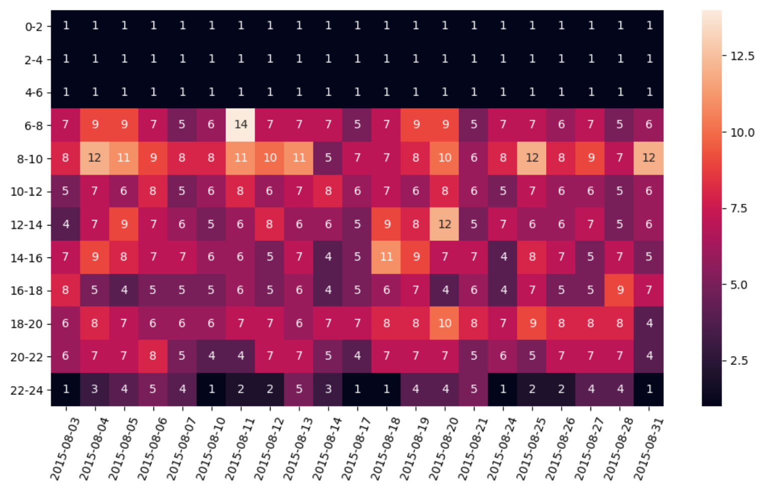


(b) : Advanced solver iterations

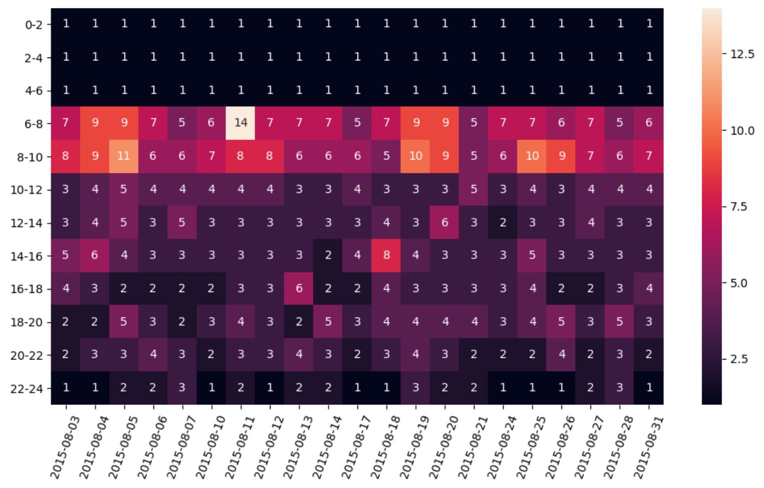


(c) : Ratio advanced/basic

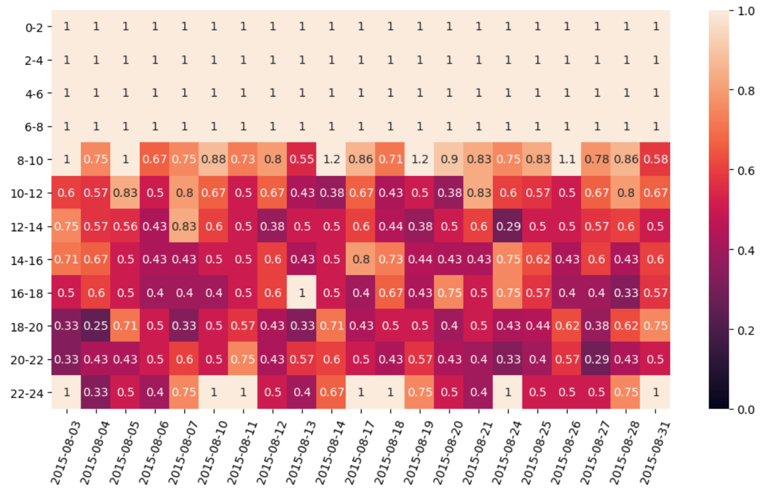
Figure B.3: Daily reusage (October)



(a) : Basic solver iterations

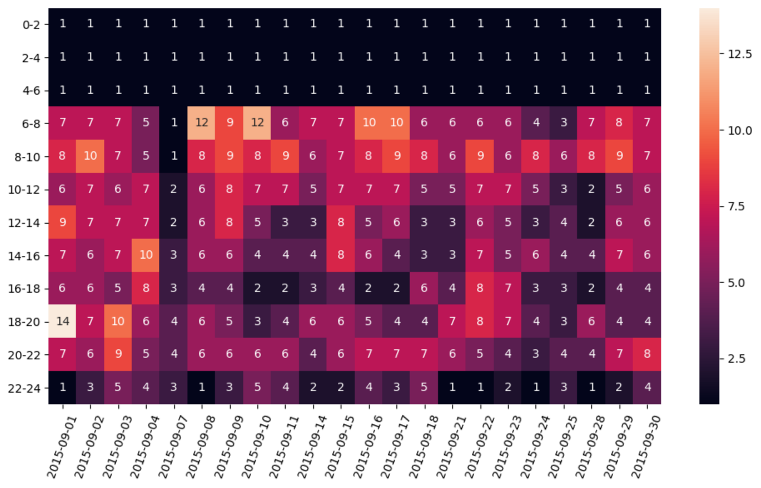


(b) : Advanced solver iterations

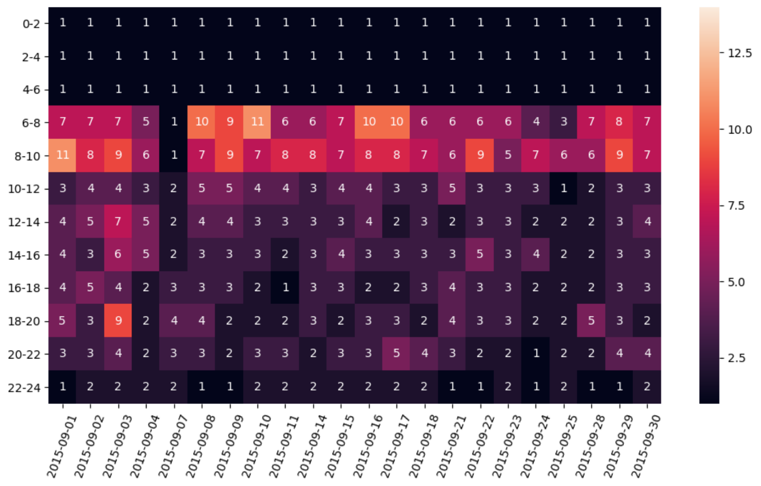


(c) : Ratio advanced/basic

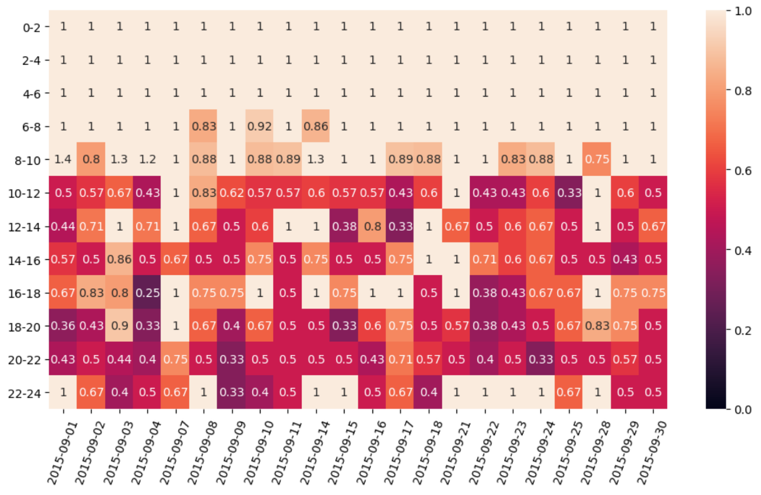
Figure B.4: Hourly reusage (August)



(a) : Basic solver iterations

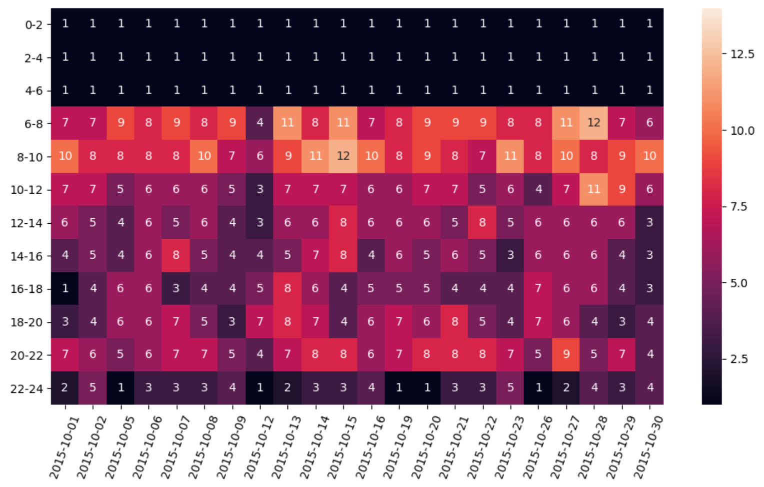


(b) : Advanced solver iterations

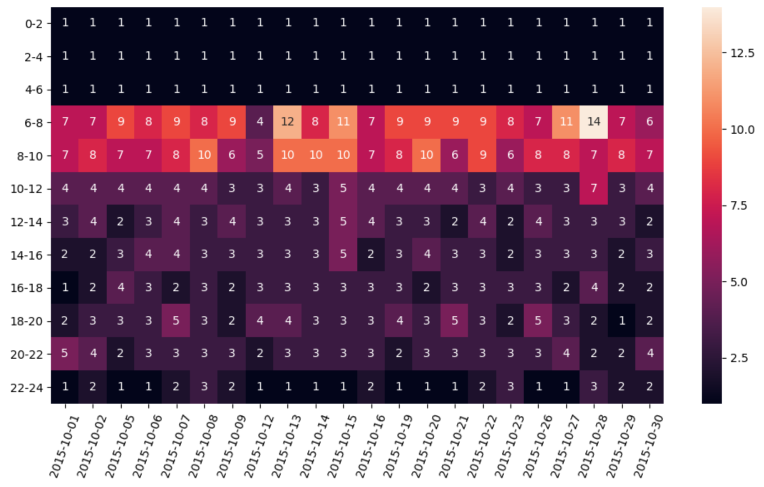


(c) : Ratio advanced/basic

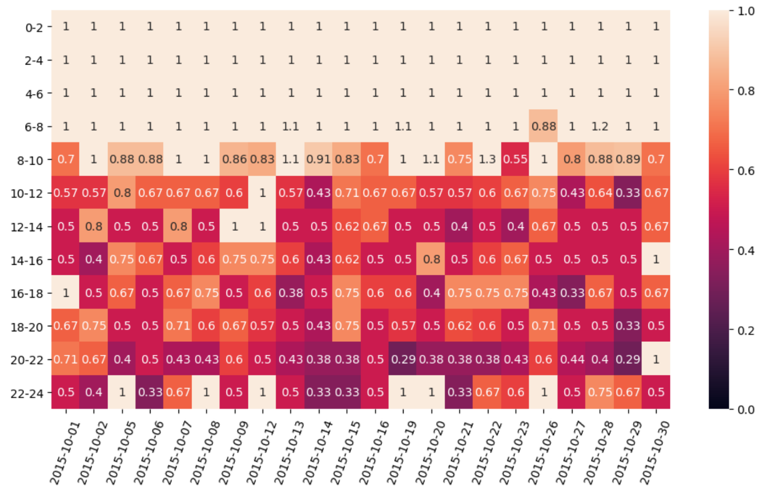
Figure B.5: Hourly reusage (September)



(a) : Basic solver iterations

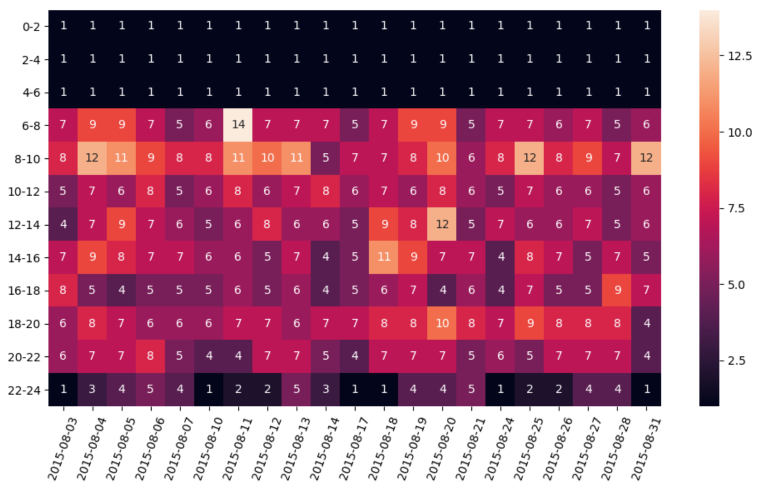


(b) : Advanced solver iterations

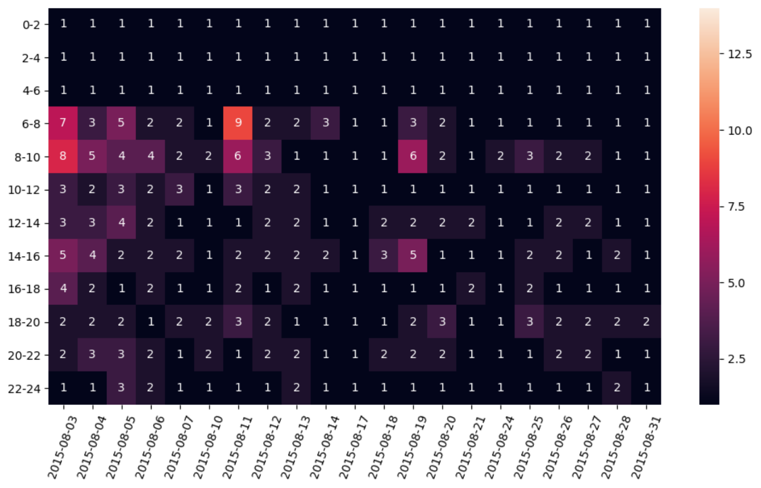


(c) : Ratio advanced/basic

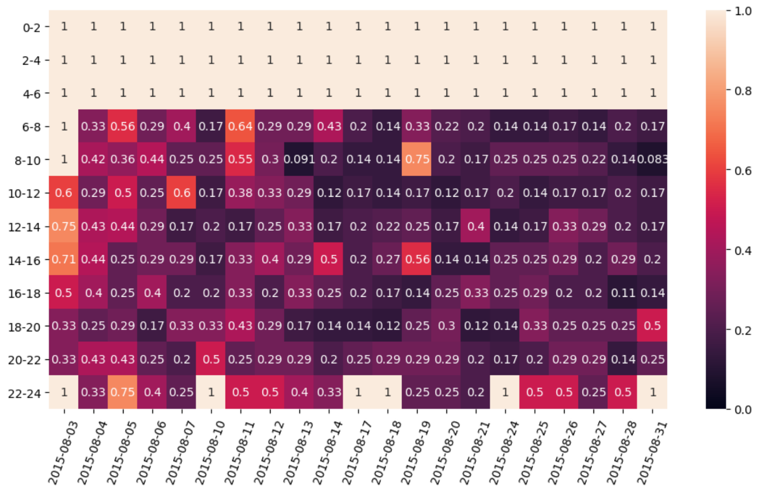
Figure B.6: Hourly reusage (October)



(a) : Basic solver iterations

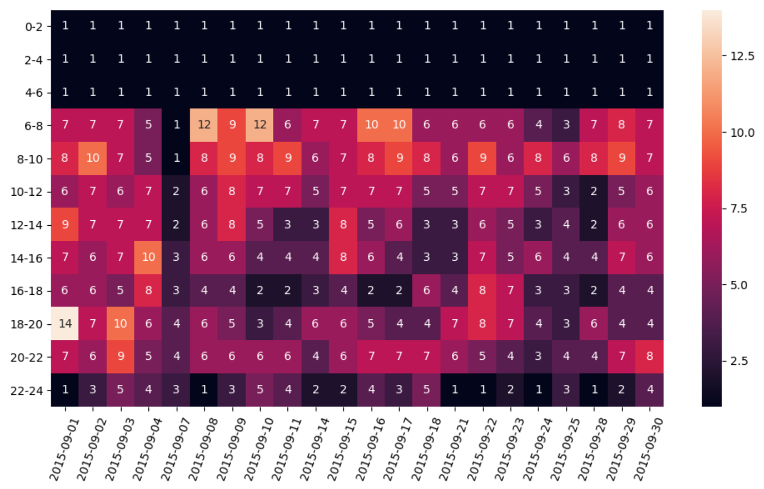


(b) : Advanced solver iterations

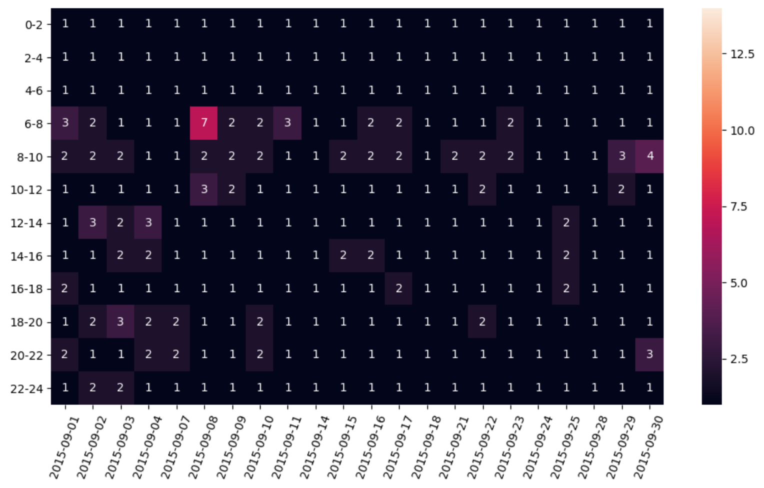


(c) : Ratio advanced/basic

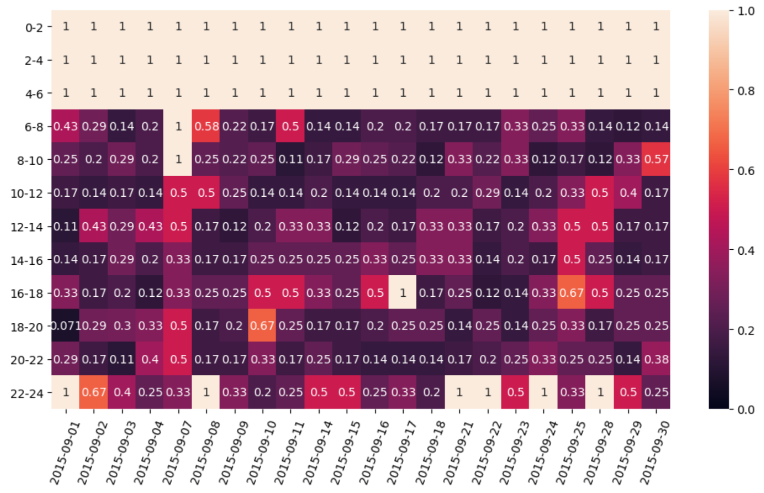
Figure B.7: Overall reusage (August)



(a) : Basic solver iterations

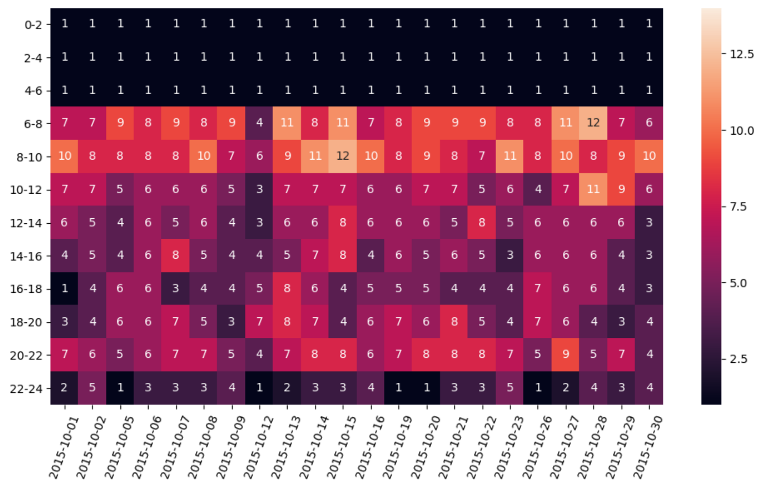


(b) : Advanced solver iterations

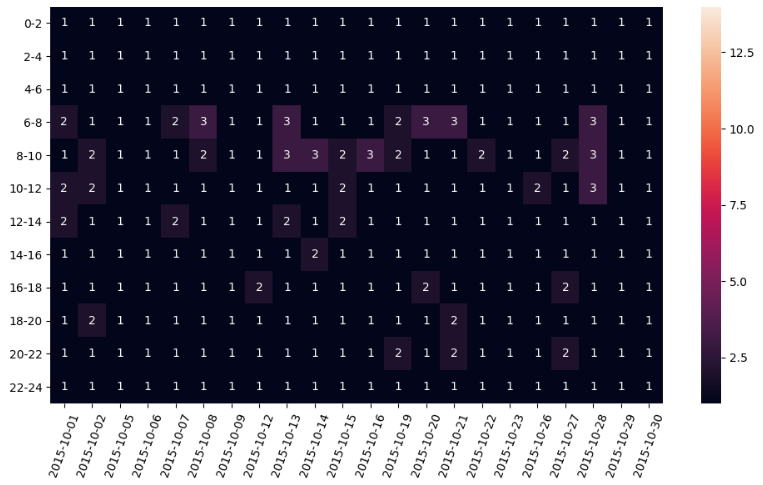


(c) : Ratio advanced/basic

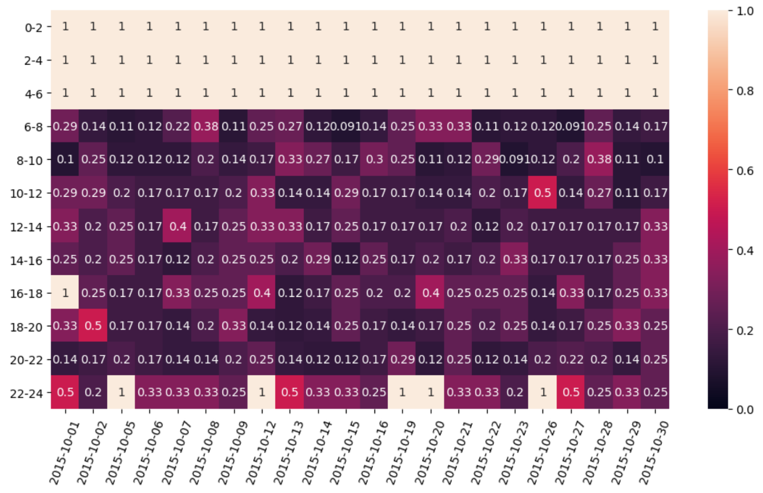
Figure B.8: Overall reusage (September)



(a) : Basic solver iterations



(b) : Advanced solver iterations



(c) : Ratio advanced/basic

Figure B.9: Overall reusage (October)

Appendix C

CD content

```
/
├─ bachelor_thesis.pdf:  text of this work
└─ src:  directory containing all source codes used in this work
   ├─ data:  directory with used graphs
   │  ├─ part.gpickle:  a graph on which all experiments were
   │  │                 conducted
   │  └─ small_example.gpickle:  a smaller graph for introduction
   │                             purposes
   ├─ readme.pdf:  a quick guide to the source code
   ├─ query.sql:  necessary commands to fill in the trips
   │              database
   ├─ download_graph.py:  a script for downloading the graph on
   │                      which all experiments were conducted,
   │                      storing it into the database and
   │                      filtering the trips data
   ├─ shortest_paths.py:  a script applying a shortest paths
   │                      approach to the graph
   ├─ run_solver.py:  a solver we have been given with
   │                  improvements we made (described in the file)
   ├─ small_example.py:  a script which shows possibilities of
   │                     our code base without having to create a
   │                     database of trips
   ├─ analyze_demand.py:  a script for examining demand
   │                      distribution
   ├─ analyze_multidays_interval.py:  comparison of the shortest
   │                                  paths approach, basic and
   │                                  advanced solver
   ├─ analyze_multidays.py:  comparison of the basic solver and
   │                         three different approaches:  daily,
   │                         hourly and overall reusage
   ├─ load_save_utils.py:  auxiliary functions for retrieving the
   │                       demand, storing and loading graphs
   └─ plot_utils.py:  auxiliary functions for plotting
```