An Algorithmic Study on Ridesharing Problem

by

Jiajian Leo Liang

B.Sc., Simon Fraser University, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Summer 2016

Approval

Name:	**Jiajian Leo Liang**
Degree:	**Master of Science (Computing Science)**
Title:	***An Algorithmic Study on Ridesharing Problem***
Examining Committee:	**Chair:** Dr. Binay Bhattacharya Professor

Dr. Qianping Gu
Senior Supervisor
Professor _____

Dr. Jiangchuan Liu
Supervisor
Professor _____

Dr. Andrei Bulatov
Internal Examiner
Professor _____

Date Approved: Aug 2, 2016

Abstract

The ridesharing problem is to share personal vehicles by individuals (participants) with similar itineraries. A trip in the ridesharing problem is a participant and his/her itinerary. To realize a trip is to deliver the participant to his/her destination by a vehicle satisfying the itinerary requirement. In this thesis, we focus on two optimization goals: minimize the number of vehicles and minimize the total travel distance of vehicles to realize all trips. The minimization problems are complex and NP-hard because of many parameters. We simplify the problems by considering only some of the parameters. We prove that some simplified minimization problems are NP-hard while a further simplified variant is polynomial time solvable. These suggest a boundary between the NP-hard and polynomial time solvable cases. We also propose a novel approach for finding a maximum matching in hypergraphs with special properties, extending the well-known maximum matching theorem in graphs to hypergraphs.

Keywords: Ridesharing problem; graph theory; matching; hypergraph matching; optimization algorithms

Acknowledgements

I would like to express my deepest gratitude and thanks to my senior supervisor Dr. Qianping Gu for his support and guidance. When I had questions and ideas about my research, Dr. Gu had always been there for discussions. His thoughtful ideas and encouragements were essential for the completion of my thesis. I would like to also express my appreciation to Dr. Guochuan Zhang for his input and the discussions we had about this thesis. His useful contributions were a great start for my thesis. I want to thank my other committee members, Dr. Jiangchuan Liu for being my supervisor and Dr. Andrei Bulatov for their useful comments on my thesis. In addition, a thank you to Dr. Binay Bhattacharya for serving as my examination committee chair.

Thanks to all my lab mates, colleagues and students of Dr. Andrei Bulatov for attending the weekly reading seminar to introduce and explain research results from different areas.

Table of Contents

Approval ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Figures vii

1 Introduction 1

2 Definitions and Related Work 5

2.1 Definitions and Notations 5

2.2 Problem Definition 6

2.3 Related Work 7

3 Simplified Ridesharing Problem 9

3.1 NP-hardness Result for Non-zero Detour 9

3.2 NP-hardness Result for Not Fixed Preferred Path 12

3.3 NP-hardness Result for Non-unique Destinations 13

3.4 A Greedy Algorithm 16

4 Polynomial Time Solvable Simplified Ridesharing Problems 19

4.1 All Constraints Are Satisfied - C1C2C3 19

4.1.1 Algorithm 19

4.1.2 Analysis 20

4.2 Non-unique Destinations Without Re-take 26

4.2.1 Limited Capacity 26

4.3 An Algorithm for Tree Pick-up Graph 27

4.3.1 Algorithm 28

5 Algorithm for General Pick-up Graph 31

5.1 Construction of Pick-up Hypergraph 31

5.2 Matching in Hypergraph and Solution for Ridesharing 33
5.2.1 Augmenting Graph 34
5.3 Algorithm 36
5.3.1 Augmenting path 37
5.3.2 Main search 37
5.3.3 Candidate search 39
5.4 Analysis of Optimality 40
5.4.1 Correctness of the Algorithm 48
5.5 Finding Augmenting Path in Hypergraphs 52
5.6 Algorithm for the Reachability Problem 54
5.7 Analysis 58
5.7.1 Subtle Details of the MDFS-hypergraph Algorithm 64

6 Conclusion and Future Work **66**

Bibliography **68**

Appendix A A Full Example of Algorithm 4 **70**

List of Figures

Figure 3.1 Ridesharing instance based on a given 3-partition problem instance. For each trip i, $3k+1 \leq i \leq 4k$, $n_i = 3$ and $x_i = 2M$. 10

Figure 3.2 Ridesharing instance based on a given 3-partition problem instance. 12

Figure 4.1 Different modifications of S_1^* (**move progresses**) to have a **progress**. In each figure, trips in R are represented by vertices on a path (the horizontal line). Each arc (u, v) from a vertex u to a vertex v denotes that driver u serves passenger v in some solution. The solid arcs above the path represent (S, σ), the solid arcs below the path represent (S_1^*, σ_1^*), and the dashed arcs below the path represent the modified (S_1^*, σ_1^*) after a move progress. 22

Figure 4.2 A pick-up graph G_P with four trips that have unit capacity, where $dist(p_u) = 10$, $dist(p_v) = 8$, $dist(p_x) = 3$, and $dist(p_y) = 3$. 27

Figure 4.3 G_w based on the pick-up graph G_P with four trips that have unit capacity, where $dist(p_u) = 10$, $dist(p_v) = 8$, $dist(p_x) = 3$, and $dist(p_y) = 3$. 28

Figure 5.1 An example of a pick-up graph G_P (left) and its corresponding pick-up hypergraph H (right). 32

Figure 5.2 An example of an augmenting graph of a hypergraph w.r.t a matching. The bold edges represents a matched edge, and non-bold edge represents an unmatched edge. 35

Figure 5.3 An example for forming type 2 subgraph. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge. 43

Figure 5.4 Examples for forming subgraph 2. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge. 44

Figure 5.5 Three types of subgraphs to be checked. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge. 46

Figure 5.6 Examples of non-minimal augmenting graph. The bold rounded rectangle represents a matched edge and non-bold rounded rectangle represents an unmatched edge. 49
Figure 5.7 On the left is a very simple undirected hypergraph H', where e_1 and e_3 are unmatched, and e_2 is matched. The corresponding directed hypergraph H_M is shown on the right. 53

Figure A.1 A hypergraph H with Property 5.1.2 and matching M, where the bold edges are matched edges and un-bold edges are unmatched edges. 70
Figure A.2 An augmenting graph of H w.r.t. M. 71
Figure A.3 The matching after the first augmentation. 71
Figure A.4 The matching after the second augmentation. 72
Figure A.5 Two augmenting graphs of H' w.r.t. M'. 72
Figure A.6 A hypergraph H' with maximum matching M_1. 72

Chapter 1

Introduction

Ridesharing, in its simplest form, is the shared use of personal vehicle by its driver and individual travelers who have similar itineraries and schedules. When a vehicle is selected to serve any participant, the owner of the vehicle is called a *driver* and a participant other than the driver is called a *passenger*. There are different motivations and goals for the use of ridesharing. Some ideal advantages of ridesharing include saving travel cost for both drivers and passengers (such as gas and parking costs), reducing traffic congestion, conserving fuel, and reducing air pollution [10, 22, 23]. Despite the advantages of rideshring, according to [15], the share of work trips that use ridesharing has decreased by almost 10% in the past 30 years. The average occupancy rate of personal vehicle trips is 1.6 persons per vehicle mile based on reports published in 2011 [17, 25]. Currently, ridesharing coordination is not fully regulated and organized in the transportation industry. Also, the lack of efficient methods to encourage participation in ridesharing is a major obstacle for ridesharing to become a regular transportation alternative for travelers. There are other factors that prevent ridesharing from being widely adopted, such as privacy, safety, social discomfort, and pricing. Some of these issues can be addressed by introducing reputation building system, profiling, or preferences [17]. With today's technology in GPS and smartphone, Internet-enabled mobile devices should be able to play an important role in popularizing ridesharing. There are new companies trying to reduce the gap between convenient transportation and ridesharing [2, 20], such as Uber and Lyft. Readers can refer to two relative recent surveys [2, 15] about ridesharing in general. In those two surveys, methods for general ridesharing problems are reviewed along with some approaches for encouraging participation in ridesharing.

Usually the ridesharing arrangement among the ridesharing participants is managed by a central *matching agency*. The matching agency is responsible for correctly facilitating the ridesharing services by assigning ridesharing arrangement - match between individual vehicle drivers and passengers. A *trip* in the ridesharing problem is a participant and his/her itinerary specified by many parameters such as the origin and destination in a road network, departure and arrival time, preferred path of a driver, distance/time detour

from the preferred path a driver can tolerate, vehicle capacity, and so on. To realize a trip is to deliver the participant to his/her destination by a vehicle satisfying the itinerary requirement.

In general, there are two types of ridesharing: static and dynamic. In static ridesharing, the set of participants (both drivers and passengers) is known in advance, and the ridesharing arrangement is computed in advance such that once the ridesharing arrangement is settled, no further change will be made. In dynamic ridesharing, each trip arrives online and a driver is arranged for an arrived trip without the knowledge of trips in the future. The matching agencies must be able to process real-time incoming ridesharing requests such that the ride-match should be established within a reasonable time frame, such as several minutes. The newly arranged ride-match (of a real-time incoming request) should not prevent the already established ridesharing arrangement from being carried out.

Earlier studies only focus on static ridesharing since real-time dynamic ridesharing (with reasonable response time) is only plausible recently due to GPS-enabled and Internet-enabled devices. There are slight different ridesharing systems settings that have been considered and proposed previously, both static and dynamic. We summarize the most common characteristics of the ridesharing problem that have been considered:

- **Independent** The drivers who provide the ride service are independent individuals, and the vehicles that provide the ride service belong to the drivers as well.
- **Automatic-matching** There is a central matching agency (system) for facilitating the ridesharing arrangement between the ridesharing participants, which requires a minimum communication between the agency and the participants. The automated process of ride-matching may include suggestions for routing, scheduling, and pricing.
- **Cost-sharing** The total travel cost is divided among the driver and passengers for each ride-matching such that the ridesharing participants should feel beneficial comparing to other means of transportation.
- **Carpooling** A common mode of the static ridesharing. The ridesharing participants are known in advance, and the matched commuters usually have similar schedules, start locations and destinations, or the driver who provides the ride service does not need to detour from his/her preferred route.
- **Dynamic** The ridesharing arrangement system needs to facilitate the ridesharing services in real-time based on the participants' input. The ridesharing services need to be established on short notice. Recent dynamic ridesharing is plausible due to Internet-enabled mobile devices.

The ridesharing problem is not easy to solve as a whole because many parameters and constraints are involved, especially if the central matching agency needs to balance between

different optimization goals and to provide services with high level of convenience. The aforementioned advantages and goals for the use of ridesharing can be conflicting each other. The following is a non-comprehensive list of optimization goals for ridesharing:

- Minimize the total number of drivers required.
- Minimize the total travel distance/time of drivers' trips.
- Minimize the total travel time of passengers' trips.
- Maximize the number of matched (served) requests.
- Minimize the cost for the drivers' trips.
- Minimize the cost for the passengers' trips.

For example, if the optimization goal is to minimize the cost for the ridesharing participants, it is more likely that each driver needs to serve more passengers, which in turns increases the total travel time and distance for the participants. Typically, a method can only optimize one or two goals at the same time. Reviews on recent methods for solving the ridesharing problem can be found in [2, 15].

Static and dynamic ridesharing are closely related. One can view dynamic ridesharing as a sequence of static ridesharing instances. It is important to fully understand the fundamentals of the static ridesharing problem since the solutions and methods for static ridesharing problem can be used as a basic scheduling tool for the dynamic systems. For example, a static arrangement scheme can be used as a subroutine in a dynamic ridesharing system for a batch of trips arrived in a specific time window, such as the method proposed in [19]. Thus, we look deeper into the static ridesharing problem by introducing tighter constraints and explore a boundary between the NP-hard and polynomial time solvable cases. Then, the solutions and methods of these more constrained problems possibly can be used for solving the more general static settings and dynamic systems.

In this thesis, we focus on static ridesharing. However, the ridesharing problem we study is more generic, but simplified. The difference between our generic ridesharing problem and the traditional ridesharing problem is that the role of a ridesharing participant can be assigned to be a driver or a passenger. In other words, we assume that each ridesharing participant is willing to provide ride service, whereas in traditional ridesharing problem, the sets of drivers and passengers are given as inputs. The formal definition of the generic ridesharing problem is given in Chapter 2. We make the assumption that the schedules of the ridesharing participants are always similar enough such that any ride-match arrangement can be served within the time constraints of the drivers and passengers. As a result, we only consider the ridesharing problem with the time constraint removed, and we call this problem the *simplified ridesharing problem*. This assumption may seem to have less real-life

application. However, for regularly scheduled carpooling (such as for work commute), it is likely that the commuters have similar schedules. As mentioned, we look deeper into the static ridesharing problem by introducing different constraints. Each variant of the simplified ridesharing problems is differentiated by the constraints that are applied to the simplified ridesharing problem. These constraints will be introduced in Chapter 3.

We consider two optimization goals in this thesis: (1) minimize the number of drivers and (2) minimize the total travel distance of drivers for realizing all trips. For each variant of the simplified ridesharing problem, we either show that it is NP-hard by providing a reduction or give an algorithm for solving such a simplified ridesharing problem. We extensively study one of the simplified ridesharing problems, where the algorithm and its analysis for this problem lead to a novel approach for finding a maximum matching in hypergraphs with specific properties. In particular, we extend the well-known maximum matching theorem by Berge [6] in graphs to hypergraphs and present an algorithm that finds an augmenting hyperpath in hypergraphs.

The rest of the thesis is organized as follows. In Chapter 2, we formally define the generic ridesharing problem and introduce some basic definitions and notations that are used throughout the thesis. We also review some of the related works in ridesharing. In Chapter 3, we state and prove the variants of the simplified ridesharing problem that are NP-hard (NP-complete). Chapters 4 and 5 are dedicated to polynomial time solvable simplified ridesharing problems, where Chapter 5 details the maximum matching theorem extension including a full analysis and presents an algorithm for finding an augmenting hyperpath and its analysis. Finally, we conclude this thesis and discuss some future works in Chapter 6.

Chapter 2

Definitions and Related Work

In most of the discussions, we model each variant of the simplified ridesharing problem using graph theory. Below, we introduce some graph theory basics and notations that are used throughout the thesis.

2.1 Definitions and Notations

We refer the readers to textbooks on graph theory [11, 24] for other terminologies. A graph $G = (V, E)$ consists of a finite non-empty set V of elements called vertices (or nodes) and a set E of 2-element subsets of V called edges (or arcs). The set $V(G)$ is called the vertex set of G and $E(G)$ is the edge set of G. The cardinality of V, denoted by $|V|$, is the number of vertices of G. The cardinality of E, denoted by $|E|$, is the number of edges of G. For a pair of vertices $u, v \in V$, the undirected edge e between u and v is represented as $e = \{u, v\}$. The vertices u and v are said to be *adjacent* to each other, and they are *neighbors*. The vertices u and v are said to be *incident* to e. The degree of a vertex v, denoted by $deg(v)$, is the number of edges incident to v. A vertex v is *isolated* if $deg(v) = 0$. The maximum degree of G, denoted by $\Delta(G)$, is the maximum degree of its vertices.

A directed graph (digraph) is similar to an undirected graph G defined above, except that there are directions on the edges in a digraph. Formally, a digraph $G' = (V', E')$ is a finite non-empty set V' of vertices and a set E' of ordered pairs of members of V'. E' is the set of directed edges of G'. A directed edge $e = (u, v)$ between u and v is considered to be directed from u to v. Vertices u and v are called the *tail* and *head* of e respectively. The *outdegree*, denoted by $outdeg(v)$, of a vertex v is the number of edges pointing from v (having v as a tail). The indegree, denoted by $indeg(v)$, of a vertex v is the number of edges pointing to v (having v as a head). A vertex v is called a *source* if $indeg(v) = 0$ and $outdeg(v) \geq 1$. A vertex v is called a *sink* if $indeg(v) \geq 1$ and $outdeg(v) = 0$.

A *walk* in an undirected graph G is an alternating sequence between vertices and edges:

$$\text{a walk } W : v_0, e_1, v_1, e_2, ..., e_n, v_n \qquad (\text{where } n \geq 0)$$

The *length* of the walk W is n because n edges (not necessarily distinct) are encountered. If all the edges in W are distinct, then W is a *trail*. If all the vertices in W are distinct, then W is a *path*. A walk W is *closed* if it starts and ends at the same vertex. A *cycle* (simple cycle) is a closed walk with distinct vertices and edges. A graph is *connected* if there is a path between every pair of vertices. A graph without any cycle is called *acyclic*. The most common acyclic graph is a tree. A *tree* is a connected acyclic graph $G = (V, E)$ where $|E| = |V| - 1$. An arbitrary acyclic graph is called a forest, which is a set of trees. A directed graph without any cycle is called *directed acyclic graph* (DAG).

A *matching* in a graph $G = (V, E)$ is a set $M \subseteq E$ of vertex-disjoint edges. An edge that belongs to a matching M is called a *matched edge*. Otherwise, it is called an *unmatched edge* with respect to M. A vertex v is called *matched* if v is incident with a matched edge. Otherwise, it is an *unmatched* or a *free* vertex with respect to M. A matching M in G is called *maximal* if M is not a proper subset of any other matching. A matching M in G is *maximum* if there does not exist another matching M' in G such that $|M'| > |M|$. A *perfect matching* is a matching in which all vertices of G are matched. Every perfect matching is maximum.

An *edge-weighted* graph G' is a graph where each edge of G' is associated with a real number, called a *weight*. A *weighted* graph usually refers to an edge-weighted graph, unless otherwise stated. The length of a path P in G', denoted by $dist(P)$, is the sum of the weights assigned to the edges of P. The distance between a vertex u and a path P in a graph is $dist(u, P) = \min_{v \in V(P)} dist(u, v)$. A *maximum weighted matching* M of an edge-weighted graph G' is defined as a matching where the total weights of the edges in M have a maximum value.

2.2 Problem Definition

A road network is expressed by a graph G which consists of a set $V(G)$ of vertices (locations in the network) and a set $E(G)$ of edges, each edge $\{u, v\}$ represents a road between u and v. G is weighted if each edge is assigned a weight (cost to use the road).

Given a set $R = (1, ..., l)$ of ridesharing participants' trips. We denote a trip by an integer $i \in R$ specified by $(s_i, d_i, t_{s_i}, t_{d_i}, n_i, x_i, \mathcal{P}_i)$, where

- s_i is the start location (source) of i (a vertex in G),
- d_i is the destination of i (a vertex in G),
- t_{s_i} is the earliest departure time of i,

- t_{d_i} is the latest arrival time of i,
- n_i is the number of seats (capacity) of i available for passengers,
- x_i is the detour distance i can tolerate for offering ridesharing services, and
- $\mathcal{P}_i$ is a set of preferred paths of i from s_i to d_i in G.

We say trip i can *serve* i itself and can serve trip $j \neq i$ if (1) $n_i \geq 1$, (2) i and j can arrive at their destinations by time t_{d_i} and t_{d_j} respectively using the service of i, and (3) the detour of i is at most x_i. A *detour* of a driver i is that i has to deviate from i's preferred path(s) so that i can serve other trips. A trip i can serve a set $\sigma(i)$ of trips if trip i can serve all trips of $\sigma(i)$ and the total detour of i is at most x_i. A trip i can serve at most $n_i + 1$ trips (including the driver) at any specific time point, but $|\sigma(i)|$ may be greater than $n_i + 1$ if i can serve some other passengers after the delivery of the current passenger(s), which is known as *re-take* passengers.

The *generic ridesharing problem* is that given an instance (G, R), G is the road network and $R = \{1, ..., l\}$ is the set of ridesharing trips, find a set $S \subseteq R$ of drivers and a mapping $\sigma : S \to 2^R$ such that:

- for each $i \in S$, all trips in $\sigma(i)$ can be served by i,
- for each pair $i, j \in S$ with $i \neq j$, $\sigma(i) \cap \sigma(j) = \emptyset$, and
- $\bigcup_{i \in S} \sigma(i) = R$.

We introduce some notations used throughout the thesis. We call (S, σ) a solution for the ridesharing problem. We sometimes say S is a solution leaving the mapping σ implicit, and denote $\cup_{i \in S} \sigma(i)$ by $\sigma(S)$. When a trip j is served by a driver i, we also say j is *picked-up* by i.

We study the problem of minimizing $|S|$ (the number of drivers) and the problem of minimizing the total travel distance of the drivers in S. These two optimization goals may not conflict each other. However, minimizing one does not mean the other is also minimized at the same time. We will show an example of this later (in chapter 4.2.1). As a result, each of the optimization goals is considered separately.

2.3 Related Work

The ridesharing problem is very similar to the dial-a-ride problem (DARP)[12]. In DARP, a set of designated drivers/vehicles is given who can provide shared service for ride requests from the users/passengers. The difference between the ridesharing problem and DARP is that, each vehicle in the ridesharing problem is limited to travel from its source and destination with possible detour. The DARP problem is a generalization of the vehicle routing

problem (VRP) with pick-up and delivery. Finding a feasible solution for the DARP is NP-hard because it also generalizes the traveling salesman problem with time windows (TSPTW) [26]. Because the ridesarhing problem is NP-hard in general, most previous studies focus on developing heuristics or solving some simplified variants of the ridesharing problem, such as single passenger at a time, single pick-up of a driver's trip, static ridesharing, not including pricing, or single objective function. Usually, the ridesharing problem is formulated as an IP (or MIP) problem and solved using some heuristics or meta-heuristics. In [5], Baldacci et al. propose both an exact and heuristic method to solve the car pooling problem based on two integer programming formulations. Herbawi and Weber [19] gave an IP formulation of the dynamic ridesharing problem where the objective function contains four optimization goals at the same time (multiobjective function). Each of the components in the objective function is associated with a parameter that controls which optimization goal has more emphasis. The author proposed a generational genetic algorithm to solve such an IP formulation of the ridesharing problem.

In [1], Agatz et al. developed optimization-based approaches for solving dynamics ridesharing in practical environment where new drivers and riders continuously enter and leave the system, which is called rolling horizon. They also built a simulation environment based on travel demand model data from the Atlanta Regional Commission, and use it to compare different methods. The ridesharing problem they studied is also slightly simplified - a driver can make only one pickup and one delivery. There is a similarity between the ridesharing problem studied in [1] and our ridesharing problem: each trip specifies whether the participant intends to be a driver, intends to be a passenger, or is flexible to perform either role. In a recent paper [20], Huang et al. proposed a method for large scale real-time ridesharing. They compare their method with some general approaches for the ridesharing problem, such as branch and bound algorithm and mixed integer programming approach. Their comparison is based on a large scale taxi dataset made by Shanghai taxis.

For a literature review of the ridesharing problem, the reader is referred to the survey of Furuhata et al.[15].

Chapter 3

Simplified Ridesharing Problem

Ridesharing Problems with Open Time Windows

By open time windows, we mean the departure/arrival time constraints are always satisfied in any assignment of drivers to passengers, that is, t_{s_i} and t_{d_i} are not considered. The minimization problems can be further simplified, assuming some of the parameters $s_i, d_i, n_i, x_i, \mathcal{P}_i$ satisfy certain constraints specified below:

(C1) **Single Destination**: all passengers have the same destination, that is, $d_i = D$ for every $i \in R$.

(C2) **Zero Detour**: each driver can only serve others on the way of his/her preferred path(s), that is, $x_i = 0$ for every $i \in R$.

(C3) **Fixed Path**: each trip i has a unique preferred path P_i from s_i to d_i, that is, $|\mathcal{P}_i| = 1$.

When any of the constraints is satisfied, the minimization problems become easier. However, we prove that the simplified minimization problems are still NP-hard when any two of the constraints are satisfied and the other one is not. Note that if the single destination (C1) constraint is satisfied, there is no re-take involved since every trip has same destination, driver cannot perform any re-take.

3.1 NP-hardness Result for Non-zero Detour

When constraints (C1) and (C3) are applied but the non-zero detour is allowed, we prove that the problems of minimizing the number of drivers and total travel distance in the ridesharing problem are NP-hard. The proof is a reduction from the 3-partition problem. The decision problem of 3-partition is that given a set $A = \{a_1, a_2, ..., a_{3k}\}$ of $3k$ positive integers, where $\sum_{i=1}^{3k} a_i = kM$ and $M/4 < a_i < M/2$, whether A can be partitioned into k disjoint subsets $A_1, A_2, ..., A_k$ such that each subset has three elements and the sum of integers in each subset is M. The decision problem of 3-partition is NP-complete [16].

Given an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem, we construct an instance (G, R_A) of the ridesharing problem as follows (also see Figure 3.1):

- The road network is the weighted graph G with $V(G) = \{I, D, v_1, ..., v_{4k}\}$ and $E(G)$ having edge $\{D, I\}$ of weight kM, edges $\{v_i, I\}$ of weight a_i, $1 \leq i \leq 3k$, and edges $\{v_i, I\}$ of weight kM, $3k + 1 \leq i \leq 4k$.
- $R_A = \{1, ..., 4k\}$ has $4k$ trips.
  - Each trip i, $1 \leq i \leq 4k$, has source $s_i = v_i$ and destination $d_i = D$, and each trip i has a unique preferred path between v_i and D in G.
  - Each trip i, $1 \leq i \leq 3k$, has $n_i = 0$ (i can only serve i itself) and $x_i \geq 0$.
  - Each trip i, $3k + 1 \leq i \leq 4k$, has $n_i = 3$ (i can serve up to three passengers at the same time) and $x_i = 2M$.

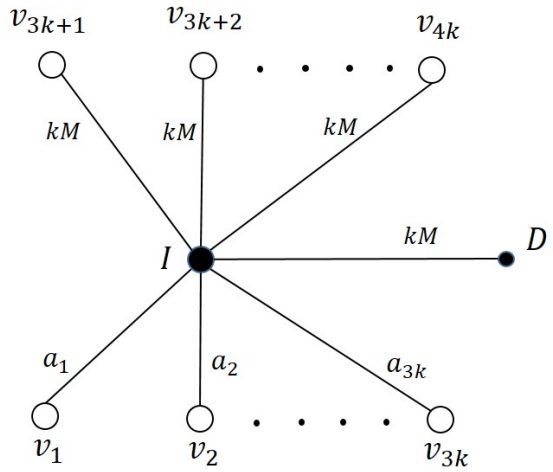


Figure 3.1: Ridesharing instance based on a given 3-partition problem instance. For each trip i, $3k + 1 \leq i \leq 4k$, $n_i = 3$ and $x_i = 2M$.

Lemma 3.1.1. *Any solution for the instance (G, R_A) has every trip i, $3k + 1 \leq i \leq 4k$, as a driver with total travel distance at least $2k(k + 1)M$.*

Proof: In the instance (G, R_A), any trip i for $3k + 1 \leq i \leq 4k$ cannot be served by any other trip $j \neq i$ because for $3k+1 \leq j \leq 4k$, the detour in such a service by j is $2kM > 2M$ for $k > 1$ and the detour limit is at most $2M$. Therefore, trip i for $3k+1 \leq i \leq 4k$ must be a driver in any solution for the instance (at least k drivers).

Let S be the set of k drivers i with $3k + 1 \leq i \leq 4k$. The total travel distance of the drivers in S is at least $2kkM$. For each trip j, $1 \leq i \leq 3k$, the total travel distance of drivers in S and trip j is at least $2kkM + 2a_j$ if j is served by a driver in S, otherwise is at least $2kkM + a_j + kM$. Since $a_j > kM$, the minimum total travel distance to realize all trips

is to have every j, $1 \leq i \leq 3k$, as a passenger and the travel distance is $2kkM + \sum_{i=1}^{3k} 2a_j = 2kkM + 2kM = 2k(k+1)M$. □

Theorem 3.1.1. *Minimizing the number of drivers in the ridesharing problem is NP-hard when constraints (C1) and (C3) are satisfied but the zero detour constraint (C2) is not.*

Proof: To get the theorem, we prove that an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem has a solution if and only if the ridesharing problem instance (G, R_A) has a solution of k drivers.

($\rightarrow$) Assume that the 3-partition instance has a solution $A_1, ..., A_k$ where the sum of elements in each A_j is M. We assign each trip i with $3k+1 \leq i \leq 4k$ to serve one set A_j for $j = i - 3k$. Since $n_i = 3$ and the sum of A_j is M, driver i can serve the three passengers corresponding to A_j with total detour distance $2M$. Hence, we have a solution of k drivers for (G, R_A).

($\leftarrow$) Assume that (G, R_A) has a solution of k drivers. By Lemma 3.1.1, every trip i with $3k+1 \leq i \leq 4k$ must be a driver in the solution. Then, each trip j for $1 \leq j \leq 3k$ cannot be a driver in the solution. By the detour limit $x_i = 2M$ and $n_i = 3$, each driver i can serve at most 3 passengers. From this and there are $3k$ passengers, each driver i must serve exactly 3 passengers in the solution. Assume, for contrary, that some driver i has a detour smaller than $2M$. Then from the fact that the sum of elements in A is kM, implying some driver i' must have a detour greater than $2M$, a contradiction. So the detour of each driver i is exactly $2M$. For each driver i with $3k+1 \leq i \leq 4k$, let A_j, $j = i - 3k$, be the subset of the three integers of A corresponding to the passengers served by i. Then $A_1, ..., A_k$ is a solution for the 3-partition problem instance.

The size of (G, R_A) is linear in k. It takes a linear time to convert a solution of (G, R_A) to a solution of the 3-partition instance and vice versa. □

Theorem 3.1.2. *Minimizing the total travel distance of drivers in the ridesharing problem is NP-hard when constraints (C1) and (C3) are satisfied but the zero detour constraint (C2) is not.*

Proof: The proof is similar to that for Theorem 3.1.1: we prove that an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem has a solution if and only if the ridesharing problem instance (G, R_A) has a solution with $2k(k+1)M$ total travel distance.

Assume that the 3-partition instance has a solution. Then there is a solution of k drivers for (G, R_A) as shown in the proof of Theorem 3.1.1. The total travel distance of this solution is $2k(k+1)M$ as shown in the proof of Lemma 3.1.1.

Assume that (G, R_A) has a solution with total travel distance $2k(k+1)M$. As shown in the proof of Lemma 3.1.1, trips i with $3k+1 \leq i \leq 4k$ are the drivers in the solution. From this, there is a solution for the 3-partition instance as shown in the proof of Theorem 3.1.1.

□

3.2 NP-hardness Result for Not Fixed Preferred Path

When constraints (C1) and (C2) are satisfied but trips may have multiple preferred paths, we prove that the problems of minimizing the number of drivers and total travel distance in the ridesharing problem are NP-hard. The proof is a reduction from the 3-partition problem (introduced in Section 3.1).

Given an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem, we construct an instance (G, R_A) of the ridesharing problem as follows (also see Figure 3.2):

- The road network is the graph G with $V(G) = \{D, I, u_1, ..., u_{3k}, v_1, ..., v_k\}$ and $E(G)$ having edges $\{u_i, I\}$ for $1 \leq i \leq 3k$, edges $\{I, v_i\}$ and $\{v_i, D\}$ for $1 \leq i \leq k$. Each edge has weight of 1.

- $R_A = \{1, ..., 3k + kM\}$ has $3k + kM$ trips.

  - Each trip i, $1 \leq i \leq 3k$, has source $s_i = u_i$, destination $d_i = D$, $n_i = a_i$ and $x_i = 0$. Each trip i has k preferred paths $\{u_i, I\}, \{I, v_j\}, \{v_j, D\}$ in G, for $1 \leq j \leq k$.
  - Each trip i, $3k + 1 \leq i \leq 3k + kM$, has source $s_i = v_j$, $j = \lceil (i - 3k)/M \rceil$, and destination $d_i = D$, $n_i = 0$, $x_i \geq 0$ and a unique preferred path $\{v_j, D\}$ in G.

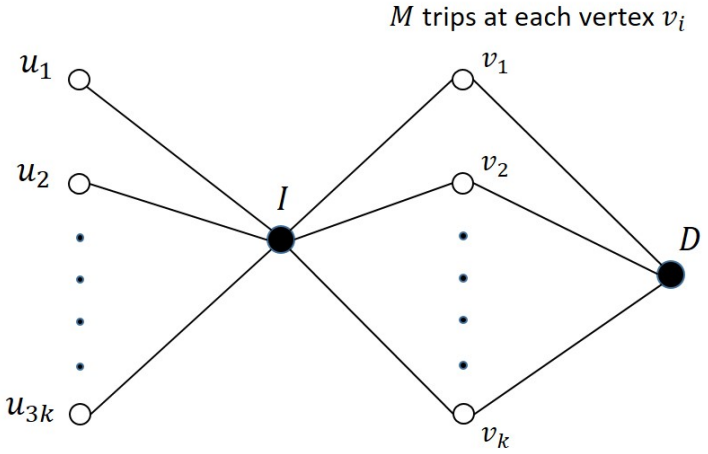


Figure 3.2: Ridesharing instance based on a given 3-partition problem instance.

Lemma 3.2.1. *Any solution for the instance (G, R_A) has every trip i, $1 \leq i \leq 3k$, as a driver and total travel distance at least $9k$.*

Proof: Since constraint (C2) is satisfied (detour is not allowed), every trip i, $1 \leq i \leq 3k$, must be a driver in any solution. A solution with exactly $3k$ drivers has total travel distance $9k$, and any solution with a trip i, $3k+1 \leq i \leq 3k+kM$, as a driver has total travel distance greater than $9k$. □

Theorem 3.2.1. *Minimizing the number of drivers in the ridesharing problem is NP-hard when constraints (C1) and (C2) are satisfied but the fixed preferred path constraint (C3) is not.*

Proof: We prove the theorem by showing that an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem has a solution if and only if the ridesharing problem instance (G, R_A) has a solution of $3k$ drivers.

($\rightarrow$) Assume that the 3-partition instance has a solution $A_1, ..., A_k$ where the sum of elements in each A_j is M. For each $A_j = \{a_{j_1}, a_{j_2}, a_{j_3}\}$, $1 \leq j \leq k$, we assign the three trips whose $n_{j_1} = a_{j_1}$, $n_{j_2} = a_{j_2}$ and $n_{j_3} = a_{j_3}$ to serve the M trips with sources at vertex v_j. Hence, we have a solution of $3k$ drivers for (G, R_A).

($\leftarrow$) Assume that (G, R_A) has a solution of $3k$ drivers. By Lemma 3.2.1, every trip i, $1 \leq i \leq 3k$, is a driver in the solution. Then, each trip j for $3k + 1 \leq j \leq 3k + 4kM$ must be a passenger in the solution, total of kM passengers. Since $\sum_{1 \leq i \leq 3k} a_i = kM$, each driver i, $1 \leq i \leq 3k$, serves exactly $n_i = a_i$ passengers. Since $a_i < M/2$ for every $a_i \in A$, at least three drivers are required to serve the M passengers with sources at each vertexv_j, $1 \leq j \leq 3k$. Therefore, the solution of $3k$ drivers has exactly three drivers j_1, j_2, j_3 to serve the M passengers with sources at the vertex v_j, implying $a_{j_1} + a_{j_2} + a_{j_3} = M$. Let $A_j = \{a_{j_1}, a_{j_2}, a_{j_3}\}$, $1 \leq j \leq k$, we get a solution for the 3-partition problem instance.

The size of (G, R_A) is polynomial in k. It takes a polynomial time to convert a solution of (G, R_A) to a solution of the 3-partition instance and vice versa. □

Theorem 3.2.2. *Minimizing the total travel distance of drivers in the ridesharing problem is NP-hard when constraints (C1) and (C2) are satisfied but the fixed preferred path constraint (C3) is not.*

Proof: Let d_{sum} be the sum of travel distances of all trips i with $1 \leq i \leq 3k$. Then the total travel distances of drivers in any solution for (G, R_A) is at least $d_{sum} = 9k$ by Lemma 3.2.1. We show that an instance $A = \{a_1, ..., a_{3k}\}$ of the 3-partition problem has a solution if and only if instance (G, R_A) has a solution with travel distance d_{sum}.

Assume that the 3-partition instance has a solution. Then there is a solution of $3k$ drivers for (G, R_A) as shown in the proof of Theorem 3.2.1. The total travel distance of this solution is d_{sum}.

Assume that (G, R_A) has a solution with total travel distance d_{sum}. By Lemma 3.2.1, trips i with $1 \leq i \leq 3k$ must be drivers. From this, there is a solution for the 3-partition instance as shown in the proof of Theorem 3.2.1. □

3.3 NP-hardness Result for Non-unique Destinations

Without the single destination constraint (C1), drivers can perform re-take, and the problem becomes harder. In fact, we show that the problems of minimizing the number of drivers

and total travel distance in the ridesharing problem are NP-hard when constraints (C2) and (C3) are satisfied but trips may have distinct destinations. The proof is a reduction from the Interval Scheduling with Machine Availabilities Problem (ISMAP) [4], which is also called the k-Track Assignment Problem [9].

The ISMAP is a machine scheduling problem: Given m machines and n jobs. Each machine $i, 1 \leq i \leq m$, has an operational interval $[a_i, b_i)$, a_i is the start time and b_i is the end time. Each job $j, 1 \leq j \leq n$, has a process interval $[p_j, q_j)$, p_j is the start time and q_j is the end time. We may simply call $[a_i, b_i)$ and $[p_j, q_j)$ an interval and call each of a_i, b_i, p_j, q_j an end point of an interval. Each job can only be processed by one machine, and each machine can process at most one job at any time point. A schedule is an assignment of jobs to machines such that every job is assigned to one machine, if job j is assigned to machine i then $[p_j, q_j) \subseteq [a_i, b_i)$, and if jobs j and j' are assigned to machine i then $[p_j, q_j) \cap [p_{j'}, q_{j'}) = \emptyset$. The decision version of ISMAP is that: Given m machines and n jobs, is there a schedule for the machines and jobs? The decision problem is NP-complete [4, 9].

Given an ISMAP instance I of m machines and n jobs, we first construct another ISMAP instance I' such that there is a schedule for I if and only if there is a schedule for I'. Then we construct an instance of the ridesharing problem such that there is an optimal solution for the ridesharing problem if and only if there is a schedule for I'.

The constructions of I' is as follows: We assume that $b_i \leq b_{i+1}$ for $1 \leq i \leq m-1$ and $a_i \geq 0$ for $1 \leq i \leq m$. For each i in I, we extend the operational interval of i to $[i-m-1, b_i+i)$ and include i as a machine in I'. Every job j in I is included in I'. For each machime $i \in I'$, we create a new job y_i with process interval $[p_{y_i}, q_{y_i}) = [i-m-1, a_i)$ and a new job z_i with process interval $[p_{z_i}, q_{z_i}) = [b_i, b_i+i)$. Then each of jobs y_i and z_i can be processed only by machine i in any schedule for I'.

Lemma 3.3.1. *There is a schedule for I if and only if there is a schedule for I'.*

Proof: Assume that there is a schedule S for I. Then in addition to S, assigning jobs y_i and z_i to machine i, $1 \leq i \leq m$, gives a schedule for I'. Assume that there is a schedule S' for I'. Then removing jobs y_i and z_i, $1 \leq i \leq m$, from S' gives a schedule for I. The size of I' is linear in $|I|$. A schedule of I can be computed from a schedule of I' in linear time and vice versa. □

Given an instance I' of m machines and $n+2m$ jobs, we construct an instance (G, R_I) for the ridesharing problem as follows:

- Graph G is the road network with $V(G) = \{u \mid u \text{ is an end point of an interval in } I'\}$ and $E(G) = \{\{u, v\} \mid u < v \text{ and no } w \in V(G) \text{ with } u < w < v\}$. Notice that G is a path.

- R_I is the set of trips defined below.
  For each machine i $(1 \leq i \leq m)$ of interval $[i-m-1, b_i+i)$, three trips are created,

(1) trip i (corresponding to machine i) with source $s_i = i - m - 1$, destination $d_i = b_i + i$, $n_i = 1, x_i = 0$, and the unique preferred path between $i - m - 1$ and $b_i + i$ in G;

(2) trip $m + i$ (corresponding to job y_i of interval $[i - m - 1, a_i)$) with source $s_{m+i} = i - m - 1$, destination $d_{m+i} = a_i$, $n_{m+i} = 0, x_{m+i} = 0$, and the unique preferred path between $i - m - 1$ and a_i in G; and

(3) trip $2m + i$ (corresponding to job z_i of interval $[b_i, b_i + i)$) with source $s_{2m+i} = b_i$, destination $d_{2m+i} = b_i + i$, $n_{2m+i} = 0, x_{2m+i} = 0$ and the unique preferred path between b_i and $b_i + i$ in G.

For each job j of interval $[p_j, q_j)$, $1 \leq j \leq n$, a trip $i = 3m + j$ is created with source $s_i = p_j$, destination $d_i = q_j, n_i = 0, x_i = 0$, and the unique preferred path between p_j and q_j in G.

Lemma 3.3.2. *Every solution for the instance (G, R_I) has every trip $i, 1 \leq i \leq m$ as a driver.*

Proof: From the zero-detour constraint (C2), a trip i can serve a trip $j \neq i$ if the interval of j is a subset of the interval of i in I' and $n_i > 0$. Since the interval of any trip $i, 1 \leq i \leq m$, is not a subset of the interval of any trip other than i, trip i cannot be served by any trip other than i. Further, any trip i, $m + 1 \leq i \leq 3m + n$, cannot serve any trip other than i because $n_i = 0$. So any solution for the instance (G, R_I) must include every trip $i, 1 \leq i \leq m$ as a driver. □

Theorem 3.3.1. *Minimizing the number of drivers in the ridesharing problem is NP-hard when constraints (C2) and (C3) are satisfied but the unique destination constraint (C1) is not.*

Proof: Given an ISMAP instance I, we construct an instance I' as shown above. By Lemma 3.3.1, I has a schedule if and only if I' has a schedule. We will prove that I' has a schedule if and only if the instance (G, R_I) has a solution of m drivers.

($\rightarrow$) Assume that I' has a schedule. Then for every job assigned to machine $i, 1 \leq i \leq m$ the interval of the job is a subset of the interval of i and thus trip i can serve the trip corresponding to the assigned job. Further, there is no overlap between the intervals of any two jobs assigned to machine i. Therefore, trip i can serve all trips corresponding to the jobs assigned to it. Thus, trips $i, 1 \leq i \leq m$, and the assignment of jobs to every trip i give a solution of m drivers for (G, R_I).

($\leftarrow$) Let (S, σ) be a solution of m drivers for (G, R_I). By Lemma 3.3.2, S has all trips$i, 1 \leq i \leq m$, as the drivers and every trip $j, m + 1 \leq j \leq 3m + n$, is served by a driver. A schedule for I' can be obtained by assigning each job corresponding to a passenger in $\sigma(i)$ to machine i.

The size of (G, R_I) is linear in $|I'|$ and $|I|$. A schedule for I' and that for I can be computed from a solution of m drivers in linear time and vice versa. □

Theorem 3.3.2. *Minimizing the total travel distance of drivers in the ridesharing problem is NP-hard when constraints (C2) and (C3) are met but the unique destination constraint (C1) is not.*

Proof: The proof is similar to that of Theorem 3.3.1. Let d_{sum} be the sum of travel distances of all trips i with $1 \leq i \leq m$ (recall that there is no detour). Then the total travel distances of drivers in any solution for (G, R_I) is at least d_{sum} because any solution must have every $i, 1 \leq i \leq m$ as a driver. We show that an instance ISMAP instance I has a schedule if and only if instance (G, R_I) has a solution with travel distance d_{sum}.

Assume that I has a schedule. Then by a similar argument as that in Theorem 3.3.1, instance (G, R_I) has a solution (S, σ) with all trips $i, 1 \leq i \leq m$, as the drivers and all trips $j, m+1 \leq j \leq 3m+n$ as the passengers. Hence, the total travel distance of all drivers in S is d_{sum}.

Assume that (G, R_I) has a solution with travel distance d_{sum}. Then the solution has all trips i with $1 \leq i \leq m$ as the drivers and all trips j with $m+1 \leq j \leq 3m+n$ as the passengers. As shown in the proof of Theorem 3.3.1, a schedule for I can be obtained. □

3.4 A Greedy Algorithm

We give a greedy algorithm for the problems of minimizing the number of drivers and minimizing the total travel distance of drivers when none of the constraints (C1), (C2) and (C3) is applied. The minimization problems are NP-hard as shown in the previous sections. An input instance for the minimization problems is (G, R), where G is an arbitrary road network and $R = \{1, 2, ..., l\}$ is a set of trips. Each trip $i \in R$ is specified by $(s_i, d_i, n_i, x_i, \mathcal{P}_i)$, where s_i and d_i are arbitrary vertices in G, $n_i \geq 0, x_i \geq 0$ and $|\mathcal{P}_i| \geq 0$. The outline of the greedy algorithm is as the follows: Given an instance (G, R), we construct a bipartite graph $H(U, V, E)$ with $U = \{u_1, ..., u_l\}, V = \{v_1, ..., v_l\}$ and $E = \{\{u_i, v_j\} \mid \text{trip } i \text{ can serve trip } j\}$. For each trip $i \in R$, compute a maximal set $\sigma(i)$ of trips that trip i can serve using the bipartite graph. Finally, we select a subset of trips which can serve all trips by a greedy approach. For each trip i, we compute a value $h(i)$ depends on the minimization problem. For the problem of minimizing the number of drivers, $h(i) = |\sigma(i)|$. For the problem of minimizing the total traveling distance, $h(i)$ is the total distance can be saved. Select the largest $h(i)$ each time until all trips are served. The details of the algorithm is given in Algorithm 1. We will only give a simple time analysis for this algorithm. In the next chapter, we will study different variants of simplified ridesharing problem that are polynomial time solvable, in particular, ridesharing problems without re-take.

Algorithm 1 A Greedy Algorithm

Input: Road network G and $R = (1, ..., l)$ of trips with each trip $i = (s_i, d_i, n_i, x_i, \mathcal{P}_i)$.
Output: a solution (S, σ) to the ridesharing instance (G, R).

for $i = 1$ to l **do**
  if $|\mathcal{P}_i| \geq 1$ **then** take one path P_i from $\mathcal{P}_i$.
  else compute a shortest path from s_i to d_i.
end for
/* Construct a bipartite graph $H(U, V, E)$ */
$U = \{u_1, ..., u_l\}, V = \{v_1, ..., v_l\}$ and $E = \emptyset$
for $u_i \in U$ and $v_j \in V$ **do**
  /* Let s_{ij} (resp. d_{ij}) be the vertex on P_i s.t.
  $dist(s_j, s_{ij}) = dist(s_j, P_i)$ (resp. $dist(d_j, d_{ij}) = dist(d_j, P_i)$) */
  if $dist(s_i, s_{ij}) \leq dist(s_i, d_{ij})$ **then** $c(u_i, v_j) = 2[dist(s_j, P_i) + dist(d_j, P_i)]$
  else $c(u_i, v_j) = 2[dist(s_j, P_i) + dist(t_j, P_i) + dist(s_{ij}, d_{ij})]$
  if $c(u_i, v_j) \leq x_i$ **then** include $\{u_i, v_j\}$ in E.
  /* Let Q_{ij} be the path with the length $dist(P_i) + c(u_i, v_j)$ for serving j by i */
end for
/* Compute a maximal set $\sigma(i)$ of trips that i can serve. Let m_i be the largest number of passengers of $\sigma(i)$ at any time point i can serve using path Q_{ij} and let c_i be the sum of detours for serving $\sigma(i)$. */
for $i = 1$ to l **do**
  $\sigma(i) := \{i\}$, $m_i = 0$, $c_i = 0$
  for every $\{u_i, v_j\} \in E$ in the increasing order of $c(u_i, v_j)$ **do**
    compute m_i for the case that v_j is included in $\sigma(i)$ and
    passengers $j' \in \sigma(i)$ are picked up in the increasing order of $dist(s_i, s_{ij'})$.
    if $c_i + c(u_i, v_j) \leq x_i$ and $m_i \leq n_i$ **then**
      Include j in $\sigma(i)$, $c_i = c_i + c(u_i, v_j)$, update m_i.
    end if
  end for
end for
/* Find a set of drivers which can serve all trips. Let $h(i) = |\sigma(i)|$ for minimizing the number of drivers and let $h(i) = \sum_{j \in \sigma(i)} dist(P_j) - c(u_i, v_j)$ for minimizing the total travel distance. */
$S = \emptyset$. Mark every trip unserved.
while $\exists$ an unserved trip **do**
  Select an unserved trip i with the largest $h(i)$.
  Include i in S, mark every trip in $\sigma(i)$ served.
  Remove every served trip from $\sigma(j)$ and update $h(j)$ for every unserved j.
end while

Time complexity Let l be the number of trips in R, $n = |V(G)|$ and $m = |E(G)|$ where G is the road network. To compute a single-source shortest path in G, it is known that it takes $O(m + n \log n)$ using Dijkstra's algorithm [14]. To find the vertex s_{ij} (resp. d_{ij}) on path P_i requires a call to Dijkstra's algorithm. Hence, it takes $O((m + n \log n)l^2)$ to construct the bipartite graph $H(U, V, E)$. For computing $\sigma(i)$, it needs to test if a trip j

(with $\{u_i, v_j\} \in E(H)$) can be included in $\sigma(i)$ due to re-take, which takes at most n_i times to see j overlaps with how many trips already in $\sigma(i)$. Since $|\sigma(i)|$ is at most l and $c(u_i, v_j)$ could have been sorted when constructing $H(U, V, E)$, it takes $O(l^2)$ to compute $\sigma(i)$ for one trip $i \in R$, total of $O(l^3)$. The while loop starts at line 28, is linear in l. Therefore, this greedy algorithm runs in $O((m + n \log n)l^2 + l^3)$.

Chapter 4

Polynomial Time Solvable Simplified Ridesharing Problems

In this chapter, we study a number of variants of the simplified ridesharing problem that can be solved in polynomial time. We will give a polynomial time algorithm or a method to solve each problem stated in this chapter. We start with the simplified ridesharing problem where all three constraints $C1, C2$, and $C3$ are satisfied. As mentioned, re-take operation is not involved when all trips in R have the same destination (C1).

4.1 All Constraints Are Satisfied - C1C2C3

We consider a special case of this problem: all the ridesharing trips lie on a single path of the road network G. In other words, for the given set $R = \{1, ..., l\}$ of trips, the graph induced by the preferred paths P_i, $1 \leq i \leq l$, is a path in G. We give a polynomial time exact algorithm for the problem of minimizing the number of drivers. For this special case, a ridesharing problem instance (G, R) can be expressed by a set $R = (1, ..., l)$ of l trips on a graph G with $V(G) = \{0, 1, ..., l\}$ and $E(G) = \{\{i, i+1\} \mid 0 \leq i \leq l-1\}$, where 0 is the common destination for all trips in R.

4.1.1 Algorithm

Given a problem instance (G, R). For every trip $i \in R$, we can assume without loss of generality that $i = (s_i, d_i, n_i, x_i, P_i)$, where $s_i = i$, $d_i = 0$, $n_i \geq 1$, $x_i = 0$ and P_i is the unique path between 0 and i in G. Given a (partial) solution (S, σ), we define the following for each driver in S:

- $\text{free}(i) = n_i - |\sigma(i)| + 1$ is the number of additional trips i can serve.
- $S_f := \{i \mid i \in S \text{ and } \text{free}(i) > 0\}$ is the set drivers in S who can serve additional trips.

The algorithm processes every trip i of R, from $i = 1$ to l. In processing i, the algorithm finds a minimum cardinality of drivers for the trips from 1 to i. We call this algorithm, Ridesharing-for-path (RFP for short), and it is given in Algorithm 2. During any execution point of Algorithm 2, whenever a trip (ridesharing participant) is assigned to be a passenger, it remains as a passenger throughout the algorithm. On the other hand, an assigned driver can be changed to a passenger when a new trip is processed.

Algorithm 2 Ridesharing-for-Path

Input: Graph G and $R = (1, ..., l)$ with $0 < s_1 < s_2 < ... < s_l$.
Output: a solution (S, σ) to the ridesharing instance (G, R) such that $|S|$ is minimized.

1: $S := \{1\}$, $\sigma(1) := \{1\}$, $S_f := \{i \mid i \in S \text{ and free}(i) > 0\}$
2: **for** $i = 2$ to l **do**
3: $S := S \cup \{i\}$, $\sigma(i) := \{i\}$, compute free(i) and S_f
4: $k :=$ Find-Target(i)
5: **while** $k \neq 0$ **do** /* serve $\sigma(k)$ by drivers in S_f and remove k from S */
6: **for** each $j \in S_f$, $k < j < i$ **do**
7: move free(j) trips from $\sigma(k)$ to $\sigma(j)$ and update free(j).
8: **end for**
9: Move the remaining trips of $\sigma(k)$ to $\sigma(i)$ and update free(i).
10: Remove k from S and update S_f.
11: **if** free$(i) \geq 1$ and $|S| \geq 2$ **then** $k :=$ Find-Target(i)
12: **end while**
13: **end for**
14: **procedure** FIND-TARGET(i)
15: **for** each $j \in S \setminus \{i\}$, gap$(i, j) := |\sigma(j)| - \sum_{a \in S_f, j < a < i}$ free(a)
16: Let $k = \arg_{j \in S \setminus \{i\}} \min\{\text{gap}(i, j)\}$;
17: **if** free$(i) \geq$ gap(i, k), **then** return k, **else** return 0.
18: **end procedure**

The procedure **Find-Target** computes a driver $k \in S$ to be removed from S and an integer *gap*. To remove a driver k from S, the trips of $\sigma(k)$ will be served by drivers j in S_f for $k < j \leq i$, first by those other than i and then by i. The value gap(i, k) is the minimum number of seats required from i to remove a driver from S and k is the driver can be removed from S by gap(i, k) seats from i.

We will discuss the more general case later in Chapter 5.

4.1.2 Analysis

In this section, we prove that Algorithm 2 Ridesharing-for-path finds a minimum solution. Let us introduce some notations. For $1 \leq j < i$, let $R(j, i) = \{k \mid j \leq k \leq i\}$. For a solution (S, σ) of R, let $S(j, i) = S \cap R(j, i)$ be the set of drivers in $R(j, i)$ and $P(j, i) = R(j, i) \setminus S$ be the set of passengers in $R(j, i)$.

Proposition 4.1.1. *Let (S, σ) be the solution found by Algorithm RFP processed $R(1, i)$ only and (S^*, σ^*) be a solution for R. If there is a driver k in S that serves passenger y, then for every driver j in $S(y+1, k-1)$, $|\sigma(j)| \geq |\sigma^*(j)|$.*

Proof: Since y is served by k in S, free$(j) = 0$ for $y < j < k$ by the algorithm when y is put into $\sigma(k)$. From this, $|\sigma(j)| = n_j + 1 \geq |\sigma^*(j)|$. □

Lemma 4.1.1. *Let (S, σ) be the solution found by Algorithm RFP processed $R(1, i)$ only. If there is an optimal solution (S_1^*, σ_1^*) for R with $S_1^*(1, i) \subseteq S$, then an optimal solution (S^*, σ^*) for R can be constructed such that $S^* = S_1^*$ and for every $j \in S^*(1, i)$, $\sigma(j) \subseteq \sigma^*(j)$ (**inclusion property**).*

Proof: Let (S, σ) be the solution found by Algorithm RFP processed $R(1, i)$ only and (S_1^*, σ_1^*) be an optimal solution for R with $S_1^*(1, i) \subseteq S$. We check the inclusion property for every element of $S_1^*(1, i)$ in the decreasing order. Let $Y_j = \sigma(j) \setminus \sigma_1^*(j)$ for $j \in S_1^*(1, i)$. Assume that $Y_k \neq \emptyset$ for some $k \in S_1^*(1, i)$ and $Y_j = \emptyset$ (inclusion property holds) for every $j \in S_1^*(k+1, i)$. Every $y_0 \in Y_k$ is served by some driver $j_0 \neq k$ in solution S_1^*. If $\sigma_1^*(k) \subset \sigma(k)$ then we move y_0 from $\sigma_1^*(j_0)$ to $\sigma_1^*(k)$. After the move, we have the following **progress:**

- The size of Y_k is reduced by one, the size of Y_j for every $j \in S_1^*(1, i)$ with $j \neq k$ is non-increasing, and the inclusion property holds for every $j \in S_1^*(k+1, i)$.

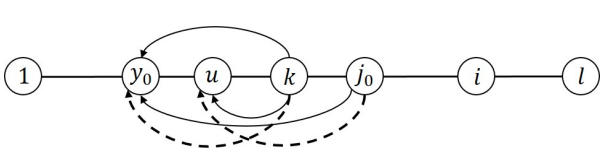
Suppose that $\sigma_1^*(k) \not\subset \sigma(k)$. Let u be a passenger in $\sigma_1^*(k) \setminus \sigma(k)$. If $u < j_0$ then we move y_0 from $\sigma_1^*(j_0)$ to $\sigma_1^*(k)$ and move u from $\sigma_1^*(k)$ to $\sigma_1^*(j_0)$ (Figure 4.1a). After this, only $\sigma_1^*(k)$ and $\sigma_1^*(j_0)$ are changed. The size of Y_k is reduced by one. Further, the size of Y_{j_0} is not changed, and even if $j_0 \in S_1^*(k+1, i)$, the inclusion property still holds for every $j \in S_1^*(k+1, i)$ because $y_0 \notin \sigma(j_0)$. Therefore, we get a progress.

Assume that $u > j_0$. Then $y_0 < j_0 < u < k$ and $y_0 \in \sigma(k)$. By Proposition 4.1.1, $|\sigma(j_0)| \geq |\sigma_1^*(j_0)|$. From this and $y_0 \in \sigma_1^*(j_0)$ but $y_0 \notin \sigma(j_0)$, there is a $y_1 \in \sigma(j_0) \setminus \sigma_1^*(j_0)$ such that $y_1 \in \sigma_1^*(j_1)$ with $j_1 \neq j_0$. We try to move y_0 to $\sigma_1^*(k)$ to have a progress by the following **move process**:

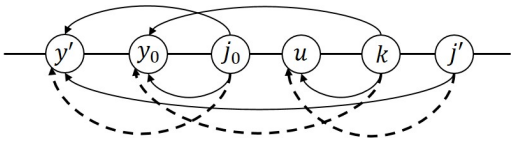
- For $y_0 \in Y_k$, if there are $u \in \sigma_1^*(k) \setminus \sigma(k)$, j_0 and j' such that $y_0 \in \sigma_1^*(j_0)$, $y' \in \sigma(j_0)$, $y' \in \sigma_1^*(j')$ and $j' > u$, then we move y_0 from $\sigma_1^*(j_0)$ to $\sigma_1^*(k)$, move y' from $\sigma_1^*(j')$ to $\sigma_1^*(j_0)$ and move u from $\sigma_1^*(k)$ to $\sigma_1^*(j')$ (Figure 4.1b).

If $j_1 > u$ then we apply the move process with $j_1 = j'$ and $y_1 = y'$. After the move process, only $\sigma_1^*(k)$, $\sigma_1^*(j_0)$ and $\sigma_1^*(j_1)$ are changed. The size of Y_k is reduced by one since $u \notin \sigma(k)$, the size of Y_{j_0} is reduced by one since $y_1 \in \sigma(j_0)$, and the size of Y_{j_1} is not increased since $y_1 \notin \sigma(j_1)$. The inclusion property still holds for every $j \in S_1^*(k+1, i)$ after the move. Therefore, we get a progress.

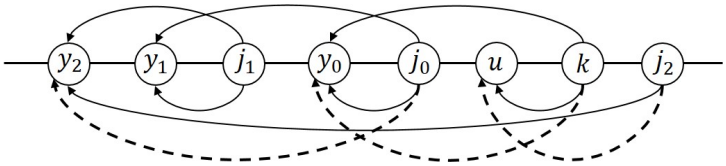
Assume that $j_1 < u$ (**extend-case**). By Proposition 4.1.1, $|\sigma(j_1)| \geq |\sigma_1^*(j_1)|$. From this and $y_1 \in \sigma_1^*(j_1)$ but $y_1 \notin \sigma(j_1)$, there is a passenger $y_2 \in \sigma(j_1) \setminus \sigma_1^*(j_1)$ such that $y_2 \in \sigma_1^*(j_2)$



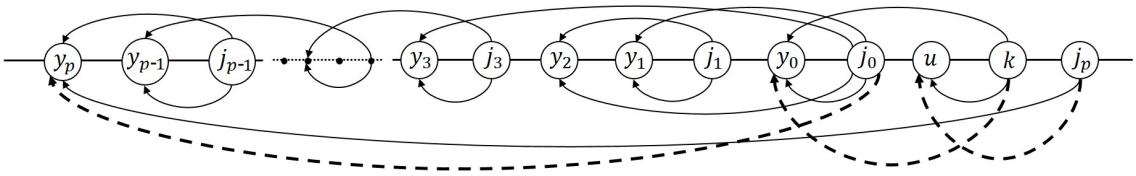
(a) A move process involving two drivers k and j_0



(b) A general move process involving three drivers k, j_0 and j'.



(c) A move process involving four drivers k, j_0, j_1 and $j_2 = j'$.



(d) A move process involving a general chain of distinct passengers $y_0, ..., y_p$. Notice that $j_2 = j_0 < j_p$.

Figure 4.1: Different modifications of S_1^* (**move progresses**) to have a **progress**. In each figure, trips in R are represented by vertices on a path (the horizontal line). Each arc (u, v) from a vertex u to a vertex v denotes that driver u serves passenger v in some solution. The solid arcs above the path represent (S, σ), the solid arcs below the path represent (S_1^*, σ_1^*), and the dashed arcs below the path represent the modified (S_1^*, σ_1^*) after a move progress.

with $j_2 \neq j_1$. Notice that $y_2 \neq y_1$ and $y_2 \neq y_0$ since $j_1 < u < k$ ($j_1 \neq j_0 \neq k$). If $j_2 > u$ then we apply the move process with $j_2 = j'$ and $y_2 = y'$ to get a progress (Figure 4.1c). Otherwise ($j_2 < u$), we have the extend-case.

Assume that the extend case continues and we have a chain of distinct passengers $y_0, y_1, .., y_{p-1}$ such that $y_c \in \sigma(j_{c-1})$, $y_c \in \sigma_1^*(j_c)$, and $j_{c-1} \neq j_c$ for $c = 1, .., p-1$, and $y_0 \in \sigma(k)$, $y_0 \in \sigma_1^*(j_0)$, and $j_0, j_1, .., j_{p-1} < u$. By Proposition 4.1.1, $|\sigma(j_{p-1})| \geq |\sigma_1^*(j_{p-1})|$. From this and $y_{p-1} \in \sigma_1^*(j_{p-1})$ but $y_{p-1} \notin \sigma(j_{p-1})$, there is a $y_p \in \sigma(j_{p-1}) \setminus \sigma_1^*(j_{p-1})$ such that $y_p \in \sigma_1^*(j_p)$ with $j_p \neq j_{p-1}$. If $j_p \neq j_c$ for every $c = 0, .., p-1$, then $y_p \neq y_c$ for every $c = 0, .., p-1$. Assume that $j_p = j_{c'}$ for some $c' \in [0, p-2]$. Since $|\sigma(j_{c'})| \geq |\sigma_1^*(j_{c'})|$ and for every $y_{c'} \in \sigma_1^*(j_{c'}) \setminus \sigma(j_{c'})$, there is a $y_p \in \sigma(j_p)$ with $y_p \neq y_c$ for every $c = 0, .., p-1$. Since the elements in the chain are distinct, the length of the chain cannot exceed $u - 1$, and there is a j_p with $j_p > u$ for the chain $y_0, .., y_p$. Then we apply the move process with $j_p = j'$ and $y_p = y'$ to get a progress (Figure 4.1d).

From the above, we can modify (S_1^*, σ_1^*) such that S_1^* does not change and the size of Y_k is reduced by one. Repeat the above, we get $Y_k = \emptyset$ and have the lemma proved. □

Recall that free(i) is the number of additional trips a driver i in a (partial) solution (S, σ) can serve. In the rest of this section, we will use $\text{free}_1(i)$, $\text{free}^*(i)$ and $\text{free}^*_1(i)$ for the numbers of additional trips a driver i in (partial) solutions (S_1, σ_1), (S^*, σ^*) and (S^*_1, σ^*_1) can serve, respectively.

Lemma 4.1.2. *Let (S, σ) be the solution found by Algorithm RFP processed $R(1, i)$ only. Then there is an optimal solution (S^*, σ^*) for R such that $S^*(1, i) \subseteq S$.*

Proof: We prove the lemma by induction on i. For $i = 1$, $S = \{1\}$ and for any optimal solution S^* for R, $S^*(1, 1) \subseteq S$. So the induction base is true. Assume that for $i - 1 \geq 1$, the lemma holds and we prove it for i. Let (S, σ) and (S_1, σ_1) be the solutions found by Algorithm RFP processed $R(1, i)$ and $R(1, i - 1)$ only, respectively. By the induction hypothesis, there is an optimal solution (S^*_1, σ^*_1) such that $S^*_1(1, i-1) \subseteq S_1$. If $S^*_1(1, i-1) \subseteq S$ then $S^*(1, i) \subseteq S$ because $i \in S$. Taking $S^* = S^*_1$, we get the lemma. So we assume that the set $W = \{w \mid w \in S^*_1(1, i - 1) \text{ and } w \notin S\}$ is not empty. We show that another optimal solution (S^*, σ^*) can be found by modifying S^*_1 with every $w \in W$ removed from S^*_1 such that $S^*(1, i) \subseteq S$.

We start the modification from a **trivial case** where $S \subseteq S^*_1(1, i) \cup \{i\}$. Let $S^* = (S^*_1 \setminus W) \cup \{i\}$. If $i \notin S^*_1$ then $|S^*(1, i)| = |S^*_1(1, i)| - |W| + 1$, and if $i \in S^*_1$ then $|S^*(1, i)| = |S^*_1(1, i)| - |W|$. Since S serves all trips in $R(1, i)$, we can move every trip in $\sigma_1(j), j \in W$ to $\sigma^*(k)$ for some $k \in S^*$. Since $|W| \geq 1$, $|S^*| \leq |S^*_1|$ and S^* is optimal. Notice that in the trivial case, $i \notin S^*_1$ and $|W| = 1$, otherwise, $|S^*| < |S^*_1|$, a contradiction since S^*_1 is optimal.

Assume that $S \not\subseteq S^*_1(1, i) \cup \{i\}$. The idea for finding (S^*, σ^*) is to modify S^*_1 by replacing a driver $w \in W$ with a driver in $(S_1 \setminus S^*_1) \cup \{i\}$. Let w be an arbitrary element of W and $Z = S_1 \setminus S^*_1$. Notice that $S \cap Z \neq \emptyset$, otherwise $S \subseteq S^*_1(1, i) \cup \{i\}$. By Lemma 4.1.1, we assume that the inclusion property holds for every $j \in S^*_1(1, i - 1)$. The modification is divided into two cases.

Case 1. There is a $z \in Z$ with $z < w$. We further assume that for some a $z \in Z$ with $z < w$, $\text{gap}(i, w) \leq \text{gap}(i, z)$, which can be written as:

$$\text{gap}(i, w) = |\sigma_1(w)| - \sum_{j \in S_1(w+1, i-1)} \text{free}_1(j) \leq \text{gap}(i, z) = |\sigma_1(z)| - \sum_{j \in S_1(z+1, i-1)} \text{free}_1(j)$$

From this, $|\sigma_1(w)| \leq |\sigma_1(z)| - \sum_{j \in S_1(z+1, w)} \text{free}_1(j)$. By the inclusion property, for every $j \in S^*_1(1, i - 1)$, $\sigma_1(j) \subseteq \sigma^*_1(j)$ and $\text{free}^*_1(j) = \text{free}_1(j) - |\sigma^*_1(j) \setminus \sigma_1(j)|$. Therefore,

$$
\begin{aligned}
|\sigma_1^*(w)| &= |\sigma_1(w)| + |\sigma_1^*(w) \setminus \sigma_1(w)| \leq [|\sigma_1(z)| - \sum_{j \in S_1(z+1,w)} \mathrm{free}_1(j)] + |\sigma_1^*(w) \setminus \sigma_1(w)| \\
&= |\sigma_1(z)| - \mathrm{free}_1(w) + |\sigma_1^*(w) \setminus \sigma_1(w)| - \sum_{j \in S_1(z+1,w-1)} \mathrm{free}_1(j) \\
&= |\sigma_1(z)| - \mathrm{free}_1^*(w) - \sum_{j \in S_1(z+1,w-1)} \mathrm{free}_1(j) \\
&\leq |\sigma_1(z)| - \mathrm{free}_1^*(w) - \sum_{j \in S_1^*(z+1,w-1)} \mathrm{free}_1^*(j) = |\sigma_1(z)| - \sum_{j \in S_1^*(z+1,w)} \mathrm{free}_1^*(j).
\end{aligned}
$$

From this, $|\sigma_1(z)| \geq |\sigma_1^*(w)| + \sum_{j \in S_1^*(z+1,w)} \mathrm{free}_1^*(j)$, implying that if we remove $|\sigma_1(z)|$ trips from $\sigma_1^*(j)$ for $j \in S_1^*$, $S_1^*(w+1,l)$ can serve at least $|\sigma_1^*(w)|$ additional trips. We apply the following **update process** to modify (S_1^*, σ_1^*):

- Add z to S_1^*, define $\sigma_1^*(z) = \sigma_1(z)$ and remove the trips of $\sigma_1^*(z)$ from $\sigma_1^*(j)$ for $j \neq z$.

After the update, $S_1^*(w+1,l)$ can serve at least $|\sigma_1^*(w)|$ additional trips. We move the trips of $\sigma_1^*(w)$ to $\sigma_1^*(j), j \in S_1^*(w+1,l)$ and remove w from S_1^*. By this, the size of S_1^* is not changed, the inclusion property still holds for every $j \in S_1^*(1, i-1)$, the size of W is reduced by one, and the size of Z is reduced by one.

Assume that for every $z \in Z$ with $z < w$, $\mathrm{gap}(i,w) > \mathrm{gap}(i,z)$. Then by Algorithm RFP, for any $z_0 \in Z$ with $z_0 < w$, z_0 is served before w ($z_0 \notin S$) and $|\sigma_1(z_0)| > \sum_{j \in S_1(z_0+1,i-1)} \mathrm{free}_1(j)$. From this, after z_0 is served, $|\sigma(j)| = n_j + 1$ for $j \in S_1(z_0+1, i)$. Furthermore, every $z \in Z$ with $\mathrm{gap}(i,w) > \mathrm{gap}(i,z)$ is served before w in Algorithm RFP ($z \notin S$). Therefore, for every $z \in S \cap Z$, $\mathrm{gap}(i,w) \leq \mathrm{gap}(i,z)$ implying $n_z \geq n_w$. We apply the update process taking some $z \in S \cap Z$. After the update, $S_1^*(1,l)$ can serve at least $n_z + 1 \geq |\sigma_1^*(w)|$ additional trips. If there is a driver $k \in S_1^*(1, w-1)$ serves at least one trip in $\sigma_1(z)$, then $|\sigma_1^*(k)| > |\sigma_1(k)|$ by inclusion property, a contradiction to Proposition 4.1.1. Thus, $S_1^*(w+1,l)$ can serve at least $n_z + 1 \geq |\sigma_1^*(w)|$ additional trips after the update. We move the trips of $\sigma_1^*(w)$ to $\sigma_1^*(j)$, $j \in S_1^*(w+1,l)$ and remove w from S_1^*. By this, the size of S_1^* is not changed, the inclusion property still holds for every $j \in S_1^*(1, i-1)$, the size of W is reduced by one, and the size of Z is reduced by one.

Case 2 For every $z \in Z$, $z > w$. Let z_0 be the minimum in Z. Then $S_1(w, z_0 - 1) = S_1^*(w, z_0 - 1)$ and $\mathrm{free}_1^*(j) = \mathrm{free}_1(j)$ for every $j \in S_1^*(w, z_0 - 1)$ by the inclusion property. Assume for contradiction, there is a $k \in S_1^*(w, z_0 - 1)$ with $\mathrm{free}_1^*(k) < \mathrm{free}_1(k)$ ($\sigma_1(k) \subset \sigma_1^*(k)$). By the inclusion property, every $y \in \sigma_1^*(k) \setminus \sigma_1(k)$ is in $\sigma_1(z)$ for some $z \in Z$. Then $|\sigma_1(k)| < |\sigma_1^*(k)|$ is a contradiction to Proposition 4.1.1 since $y < k < z$. Hence, $\mathrm{free}_1^*(j) = \mathrm{free}_1(j)$ for every $j \in S_1^*(w, z_0 - 1)$.

Assume that $\mathrm{gap}(i,w) > \mathrm{gap}(i,z_0)$. By Algorithm RFP, z_0 is served before w and $|\sigma_1(z_0)| > \sum_{j \in S_1(z_0+1,i-1)} \mathrm{free}_1(j)$. From this, after z_0 is served, $|\sigma(j)| = n_j + 1$ for $j \in S_1(z_0 + 1, i)$. Further, there is a $z \in S \cap Z$ with $w < z_0 < z$ and $\mathrm{gap}(i,w) \leq \mathrm{gap}(i,z)$,

implying $n_z \geq n_w$. We apply the update process taking this z. After the update, $S_1^*(w+1,l)$ can serve at least $n_z + 1 \geq |\sigma_1^*(w)|$ additional trips. We move the trips of $\sigma_1^*(w)$ to $\sigma_1^*(j)$, $j \in S_1^*(w+1,l)$ and remove w from S_1^*. By this, the size of S_1^* is not changed, the inclusion property still holds for every $j \in S_1^*(1,i-1)$, the size of W is reduced by one, and the size of Z is reduced by one.

Assume that $\text{gap}(i,w) \leq \text{gap}(i,z_0)$. Note that $|\sigma_1(z_0)| \geq |\sigma_1(w)| - \sum_{j \in S_1(w+1,z_0)} \text{free}_1(j)$. We apply the update process with $z_0 = z$. After the update, $S_1^*(w+1,l)$ has at least $|\sigma_1(z_0)| + \sum_{j \in S_1(w+1,z_0)} \text{free}_1(j) \geq |\sigma_1(w)|$ seats for $\sigma_1^*(w)$. Since $\text{free}_1^*(w) = \text{free}_1(w)$ we have $|\sigma_1(w)| = |\sigma_1^*(w)|$ and can move the trips of $\sigma_1^*(w)$ to $\sigma_1^*(j)$, $j \in S_1^*(w+1,l)$, and remove w from S_1^*. By this, the size of S_1^* is not changed, the inclusion property still holds for every $j \in S_1^*(1,i-1)$, the size of W is reduced by one, and the size of Z is reduced by one.

Repeat the processes in Cases 1 and 2, we get an optimal solution S_1^* with either $W = \emptyset$ or $Z = \emptyset$. For $W = \emptyset$, $S^* = S_1^*$ is the solution we want to find. For $Z = \emptyset$, $S \subseteq S_1^*(1,i) \cup \{i\}$ and by the argument in the trivial case, we can get S^*. □

Theorem 4.1.1. *Algorithm 2 Ridesharing-for-path finds a solution (S,σ) for input instance (G,R) with the minimum number of drivers.*

Proof: First, observe that (S,σ) is a solution for R since every trip in R is processed one by one and assigned to be a driver initially. By Lemma 4.1.2, there is an optimal solution (S^*,σ^*) for R such that $S^* \subseteq S$. Assume that $S^* \subset S$. By Lemma 4.1.1, we can modify σ^* such that the inclusion property holds, that is, $\sigma(j) \subseteq \sigma^*(j)$ for every $j \in S \cap S^*$. Let z be a driver in $S \setminus S_1^*$. Then all trips in $\sigma(z)$ must be served by drivers in $S^*(z+1,l)$. By the inclusion property, $\text{free}^*(j) \leq \text{free}(j)$ for every $j \in S^*(1,l)$. This implies that $|\sigma(z)| \leq \sum_{j \in S(z+1,l)} \text{free}(j)$, which is a contradiction to Algorithm RFP as $\sigma(z)$ should have been served by S. Therefore, it must be the case that $S^* = S$. □

Time complexity Let l be the number of trips in R. The road network G is expressed by the trips in R by a single path with $l+1$ vertices and l edges. Moving passengers from one set to another set (line 7 and line 9) does not require the actual general road network and can be done in $O(1)$. In each iteration, the Find-Target procedure is called multiple times, which is at most n_i times. Actually, we can have a tighter bound. Suppose that there are n_i drivers in $S(1,i-1)$ just before processing i, where $|\sigma(j)| = 1$ for every $j \in S(1,i-1)$. After i is added to S. The Find-Target procedure will be called n_i times in this iteration. As a result, $S(1,i)$ will contain only one driver, i, at the end of this iteration. When processing the next trip $i+1$, Find-Target will be called only once even if n_{i+1} is large. In fact, Find-Target is called at most $\alpha_i = \min\{n_i, |S(1,i-1)|\}$ for each iteration i. Since the size of S is increased at most one for each iteration and each time Find-Target returns $k > 0$, size of S is decreased by one, $\sum_{1 \leq i \leq l} \alpha_i$ is at most l, where $\alpha_1 = 0$ and $|S(1,1)| = 1$.

The running time for procedure Find-Target is $O(l)$ since it needs to do a scan of $S(1, i)$ to find out the driver k with minimum $gap(i, k)$. Hence, the running time for Algorithm RFP is $O(l^2)$.

4.2 Non-unique Destinations Without Re-take

In chapter 3.3, we proved that the minimization problems of minimizing the number of drivers and total travel distance in the ridesharing problem are NP-hard when constraints (C2) and (C3) are satisfied but trips may have distinct destinations. The rest of the thesis is focused on this variant of the simplified ridesharing problem where re-take is not allowed. The reduction in chapter 3.3 no longer can be applied when re-take is not allowed (each machine i can process up to n_i jobs, which is not ISMAP).

First, we formally restate this variant of the ridesharing problem. Consider the following *simplified ridesharing problem*: Given an instance (G, R), where G is the road network and R is a set of ridesharing trips. Each trip u in R is specified by a tuple (P_u, n_u), where

- P_u is a path in G with a source s_u and destination t_u, and
- $n_u \geq 0$ is the number of passengers u can serve for ridesharing,

We assume the start location of each trip u is distinct, that is, $s_u \neq s_v$ for every pair $u, v \in R$ with $u \neq v$. In the following sections, we consider three different cases for this simplified variant ridesharing problem, starting from a special case to a fully generalized case. We give a detailed analysis for the general case in Chapter 5. We will construct a pick-up relation graph, which is used in all three cases.

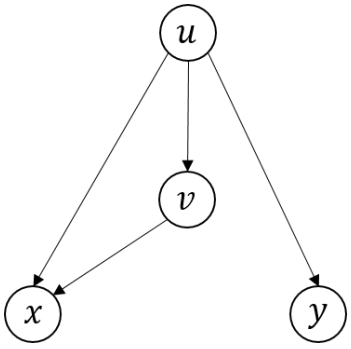
Pick-up graph. The pick-up relation graph (*pick-up graph* for short) for a set R of ridesharing trips is a digraph $G_P(V, E)$, where $V(G_P) = R$, and there is a directed edge $(u, v) \in E(G_P)$ if trip u can serve trip v.

The time it takes to construct the pick-up graph is $O(l^2 * |V(G)|)$ where $l = |R|$ and G is the road network. For each pair $u, v \in R$, u can serve v if s_v and t_v lie on the path P_u, which takes $O(|V(G)|)$ to check.

4.2.1 Limited Capacity

Consider the following special case where all trips in R can serve at most one passenger, that is, $n_u \leq 1$ for every $u \in R$. It is intuitively a matching problem between the drivers and passengers. In a maximum matching M of the underlying pick-up graph G_P (constructed based on (G, R)), each matched edge represents a single pick-up. The actual pick-up can be recovered by the edge direction in G_P. For any unmatched vertex v w.r.t. M, v drives singly (no sharing with other participants). M derives an optimal solution for minimizing

the number of drivers. As an example, consider the pick-up graph G_P of a given simplified ridesharing instance and its different optimization goals in Figure 4.2, where a maximum matching correctly finds the minimum number of drivers.



- Minimum number of drivers required is 2.

$\sigma(u) = \{u, y\}$ and $\sigma(v) = \{v, x\}$

$dist(p_u) + dist(p_v) = 10 + 8 = 18$

- Minimum total travel distance is 16 with three drivers.

$dist(p_u) + dist(p_x) + dist(p_y) = 10 + 3 + 3 = 16$

$\sigma(u) = \{u, v\}, \sigma(x) = \{x\}$, and $\sigma(y) = \{y\}$

Figure 4.2: A pick-up graph G_P with four trips that have unit capacity, where $dist(p_u) = 10$, $dist(p_v) = 8$, $dist(p_x) = 3$, and $dist(p_y) = 3$.

To minimize the total travel distance, we need to construct a different undirected weighted graph $G_w = (V_w, E_w)$ based on a given pick-up graph G_P. V_w consists of two sets V_1 and V_2, each of which is a copy of $V(G_P)$. E_w has three subsets $\{E_1, E_2, E_3\}$. $E_1 = E(G_P)$ is the set of edges between the vertices in V_1. (V_2, E_2) forms a complete graph. In addition, there are $|V(G_P)|$ edges in E_3. Each edge $e \in E_3$ is formed by each pair of the same vertices in V_1 and V_2 (each of them is a copy of $V(G_P)$). For every $e = \{u, v\} \in E_1$ (correspond to $(u, v) \in E(G_P)$), e has weight $dist(p_u)$. For every $e = \{u, u\} \in E_3$, e has weight $dist(p_u)$. The weight of every edge in E_2 is zero. We claim that a minimum weight perfect matching of G_w (which can be solved in polynomial time) provides an optimal solution for the ridesharing problem. In such a matching, a matched edge in E_1 means a pick-up, a matched edge in E_3 means a singly driving, while a matched edge in E_2 has no particular meaning, except to ensure that there is a perfect matching. The construction of G_w is shown in Figure 4.3, which is based on the pick-up graph G_P in Figure 4.2.

The minimum weight perfect matching finds a solution of the ridesharing problem with total travel distance 16 as shown in Figure 4.2. Agatz et al. [1] considered a similar matching problem, but for dynamic settings.

4.3 An Algorithm for Tree Pick-up Graph

Given an instance (G, R) with $n_u \geq 0$ for every $u \in R$. Since we assume the start location of each trip i is distinct, the pick-up graph G_P (based on (G, R)) is a directed acyclic graph (DAG). An edge (u, v) in G_P is called a short-cut if after removing (u, v), there is a path from u to v in G_P. The pick-up graph G_P is called *simplified* if every short-cut has been removed from G_P. We give an algorithm for an instance (G, R) with its simplified pick-up

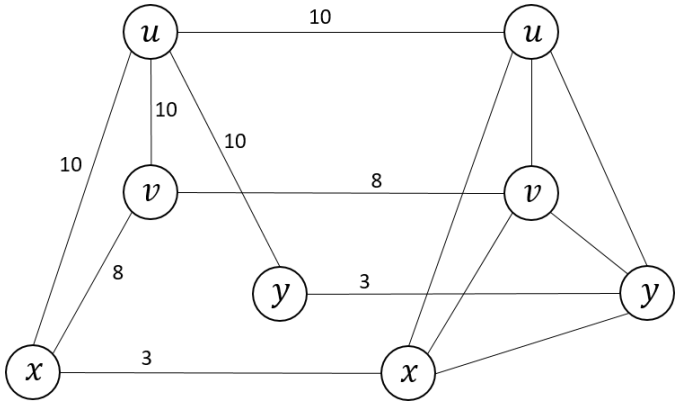


Figure 4.3: G_w based on the pick-up graph G_P with four trips that have unit capacity, where $dist(p_u) = 10$, $dist(p_v) = 8$, $dist(p_x) = 3$, and $dist(p_y) = 3$.

graph T being a forest. In what follows, we assume that the simplified pick-up graph for (G, R) is a rooted tree T.

For a vertex u in T, we define the following:

- T_u, the subtree rooted at u.
- C_u, the set of child vertices of u ($u \notin C_u$).
- D_u, the set of descendant vertices of u ($u \notin D_u$).
- For a set S of drivers, $S(T_u) = S \cap V(T_u)$ is the subset of drivers in T_u and $P(T_u) = V(T_u) \setminus S(T_u)$ is the set of passengers in T_u.
- For a mapping σ, $\text{free}(u) = \min\{n_u, |D_u|\} + 1 - |\sigma(u)|$, additional trips u can serve.

4.3.1 Algorithm

It turns out the algorithm for (G, R) with its simplified pick-up graph being a tree is almost identical to Algorithm RFP, presented in chapter 4.1. The only difference is that instead of searching a subpath, the algorithm searches a subtree. For completeness, we shall present the algorithm for tree pick-up graph as well. Given the rooted tree T (the simplified pick-up graph for (G, R)), our algorithm processes every vertex u of T in the post-order. In processing u, the algorithm finds a solution $(S(T_u), \sigma)$ for (the trips in) T_u with the minimum cardinality. For u and each descendant v of u, let $Q_{u,v}$ be the set of vertices in the path from u to v in T_u, excluding u and v. The algorithm is called Ridesharing-for-Tree (RFT for short) and is given in Algorithm 3.

Similar to Algorithm RFP, whenever a trip (a ridesharing participant) is assigned to be a passenger. It remains as a passenger throughout the algorithm. On the other hand, an assigned driver can be changed a passenger when a new ancestor of an assigned driver trip is processed.

The **Find-Target** procedure computes a vertex $w \in S(T_u)$ to be removed from $S(T_u)$ and an integer *gap*. To remove a vertex v from $S(T_u)$, the trips of $\sigma(v)$ will be served by ancestors of v, first by those other than u and then by u. *gap* is the minimum number of seats required from u to remove a vertex from $S(T_u)$ and w is the vertex which can be removed from $S(T_u)$ by *gap* seats from u.

Algorithm 3 Ridesharing-for-Tree

Input: a rooted tree T of the simplified pick-up graph for an instance (G, R).
Output: a solution (S, σ) to the ridesharing instance (G, R) such that $|S|$ is minimized.

for every vertex u in T in the post-order **do**
  /* Initialization */
  $\sigma(u) := \{u\}$, compute free(u)
  $S(T_u) := \{u\} \cup (\cup_{v \in C_u} S(T_v))$.
  /* Minimize $S(T_u)$ by removing some vertices from $S(T_u)$ */
  $w :=$ Find-Target(u)
  while $w \neq \emptyset$ **do** /* Serve $\sigma(w)$ by ancestors of w and remove w from $S(T_u)$ */
    for every $v \in Q_{u,w} \cap S(T_u)$ **do**
      move free(v) trips from $\sigma(w)$ to $\sigma(v)$ and update free(v).
    end for
    Move the remaining trips of $\sigma(w)$ to $\sigma(u)$ and update free(u).
    Remove w from $S(T_u)$.
    $w :=$ Find-Target(u)
  end while
end for

procedure FIND-TARGET(u)
for each $v \in S(T_u) \setminus \{u\}$, $\text{gap}(u, v) := |\sigma(v)| - \sum_{x \in Q_{u,v} \cap S(T_u)} \text{free}(x)$
Let $\text{gap}(u, w)$ be the minimum of $\text{gap}(u, v)$ for $v \in S(T_u) \setminus \{u\}$.
if $(S(T_u) \setminus \{u\} \neq \emptyset$ and $\text{free}(u) \geq \text{gap}(u, w))$, **then** return w, **else** return nil.
end procedure

Although Algorithm RFT and Algorithm RFP are merely the same, the analysis of Algorithm RFP cannot be directly applied to Algorithm RFT. Under inspection, Lemma 4.1.1 can be applied to Algorithm RFT. However, Lemma 4.1.2 requires more in-depth analysis. We believe that a modified version of Lemma 4.1.2 can be obtained and useful for proving the correctness of Algorithm RFT, which we stated it as the following conjecture.

Conjecture 4.3.1. *Let T_u be the subtree that has just been processed by the Algorithm 3 Ridesharing-for-Tree and $(S(T_u), \sigma)$ be the solution found by the algorithm for T_u. Then there is an optimal solution (S^*, σ^*) for R such that $S^*(T_u) \subseteq S(T_u)$.*

We leave the proof of this conjecture and correctness of Algorithm RFT as a future work. We will consider the general case in the next chapter.

Time complexity Let $n = |V(G_P)|$ and $m = |E(G_P)|$ where G_P is the pick-up graph. Then, it takes $O(mn)$ to convert a pick-up graph G_P to a simplified pick-up graph T since

each edge (u, v) needs to be checked, and it takes $O(n)$ to test reachability from u to v. To test if T is a tree (forest), it takes linear time. Assuming that the simplified pick-up graph T is a tree. Algorithm RFT does not need the actual road network any more for constructing the solution for R. Moving passengers from one set to another set (line 9 and line 11) can be done in $O(1)$. The number of nodes in T is $l = |R|$. Then, similar to Algorithm RFP, the running time for Algorithm RFT is $O(l^2)$.

Chapter 5

Algorithm for General Pick-up Graph

When the simplified pick-up graph T (of the constructed pick-up graph G_P) is no longer a forest, that is, a vertex in T can have multiple parents, the problem becomes much harder. We will not use the simplified pick-up graph T to solve this case any more, instead we construct a hypergraph (defined in Section 5.1) based on the pick-up graph G_P. We call this constructed hypergraph a *pick-up relation* hypergraph (pick-up hypergraph for short). We then find a matching in this pick-up hypergraph, which is equivalent to a solution to the simplified ridesharing problem. This is proved in Theorem 5.2.1. Let us first consider the construction of the pick-up hypergraph.

5.1 Construction of Pick-up Hypergraph

A hypergraph is a generalization of a graph. Formally, a hypergraph $H = (V, E)$ consists of a finite non-empty set V of vertices and a set E of non-empty subsets of V called hyperedges. Hyperedges in $E(H)$ can contain different number of vertices. Given a pick-up graph $G_P = (V, E)$, we construct a *pick-up* hypergraph $H = (V, E)$ based on G_P. The idea of H is inspired by the (g,f)-factor (or degree constrained subgraph) problem [7]. The construction of H works as follows:

- For each vertex $v \in V(G_P)$ where v is not a sink, create $v_1, v_2, ..., v_{n_v}$ copies of v in $V(H)$.
- For each vertex $v \in V(G_P)$ where v is a sink, create a vertex v_1 of v in $V(H)$.
- For any vertex $v \in V(G_P)$ with $n_v = 0$, create a vertex v_1 of v in $V(H)$.
- For each edge $(u, v) \in E(G_P)$, create n_u hyperedges $\{u_1, v_1, ..., v_{n_v}\}, ..., \{u_{n_u}, v_1, ..., v_{n_v}\}$ in $E(H)$.

See Figure 5.1 for an illustration of the construction of H.

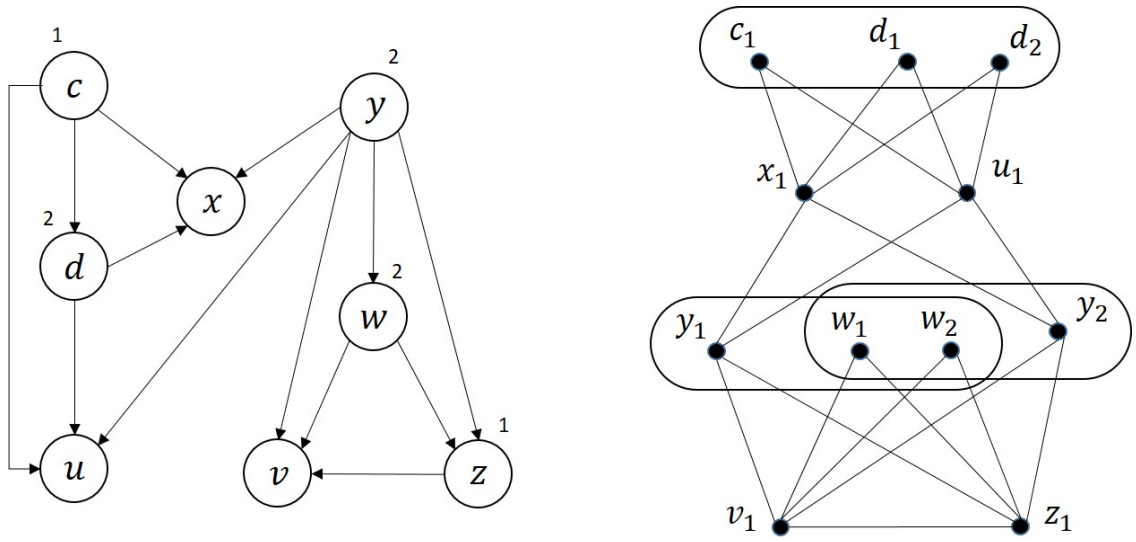


Figure 5.1: An example of a pick-up graph G_P (left) and its corresponding pick-up hypergraph H (right).

The hyperedges in H are not directed, but we will give an orientation to the edges in H based on G_P. Let (u, v) be an edge in G_P and $e = \{u_i, v_1, v_2, ..., v_{n_v}\}$ be a corresponding edge in H, for some $i \in [1, n_u]$. For e, u_i is called a *parent* of each vertex in $e \setminus \{u_i\}$, and each vertex in $e \setminus \{u_i\}$ is a *child* of u_i. Ancestor for hypergraphs is defined the same as for graphs. Vertex u is an ancestor of v if u is a parent of v or u is a parent of another ancestor of v. Each hyperedge e in H has exactly one parent and $|e| - 1$ children, and all the children correspond to the same vertex in G_P. Notice that the vertices $\{v_1, v_2, ..., v_{n_v}\}$ that correspond to the single vertex $v \in V(G_P)$ must appear as a whole in any hyperedge in H.

The pick-up hypergraph H has two properties:

Property 5.1.1. *Shortcut property: Let u, v, w be vertices in a pick-up graph G_P. If edges (u, v) and $(v, w) \in E(G_P)$, then $(u, w) \in E(G_P)$. This shortcut property also holds in H. Let u, $v_1, ..., v_{n_v}$, and $w_1, ..., w_{n_w}$ be vertices in H. If edges $\{u, v_1, ..., v_{n_v}\}$ and $\{v_1, w_1, ..., w_{n_w}\}, ..., \{v_{n_v}, w_1, ..., w_{n_w}\}$ in $E(H)$, then $\{u, w_1, ..., w_{n_w}\} \in E(H)$.*

Denoted by $S_v = \{v_1, ..., v_{n_v}\}$, the set of vertices in H that corresponds to a vertex v in G_P.

Property 5.1.2. *Each edge $e \in E(H)$ contains exactly one vertex u_i ($i \in [1, n_u]$) from the set S_u and all vertices from another set S_v, where $u_i \in S_u$ is the parent in e and S_v are children in e.*

Corollary 5.1.1. *Due to Property 5.1.2, for any two incident hyperedges e, e' in H:*

$$|e \cap e'| = \begin{cases} 1 \text{ or } |e| - 1, & \text{if } |e| = |e'| \\ 1, & \text{otherwise} \end{cases}$$

Intuitively, if $|e \cap e'| = 1$, e and e' have the same vertex $(e \cap e')$ being the parent of their respective hyperedges. It is not possible that $|e \cap e'| > 1$ and $|e| \neq |e'|$.

Let us introduce some definitions for general hypergraphs. A *path* P in a hypergraph H is a sequence of distinct vertices and hyperedges $v_1, e_1, v_2, e_2, ..., e_k, v_{k+1}$ where $v_j, v_{j+1} \in e_j$ for $j = 1, ..., k$. If $v_{k+1} = v_1$, then P is said to be a *cycle*. In fact, this is called a *Berge-cycle*. We use a more generic definition of *hypercycle*, which is given below:

Definition 5.1.1. *C is a **cycle** of length k in a hypergraph H if its vertices can be given a cyclic ordering $v_1, ..., v_j$ such that every pair of consecutive vertices v_i and v_{i+1} lie in an edge of C, and C has exactly k edges.*

Any Berge-cycle is a generic hypercycle, but a generic hypercycle may not be a Berge-cycle.

Definition 5.1.2. *An edge e in a hypergraph H is called a **portal** edge if e is incident to at least three edges e_1, e_2, e_3 such that e_1, e_2 and e_3 are pairwise disjoint.*

Notice that only hypergraphs can contain portal edges. For simplicity, a (hyper)path P of length k is denoted by its (hyper)edges only, i.e. $P = e_1, e_2, ..., e_k$.

Definition 5.1.3. *A hyperpath $P = e_1, e_2, ..., e_k$ is **simple** if P is generic hypercycle free. Equivalently, $e_i \cap e_{j+1} = \emptyset$ for $1 \leq i < j < k$ (that is, every edge $e_i \in P$ is only incident to e_{i-1} and e_{i+1} for $2 \leq i \leq k-1$), and e_1 is only incident to e_2, e_k is only incident to e_{k-1}.*

A matching M in a hypergraph H is defined the same - a set of pairwise vertex-disjoint edges. An *alternating hyperpath* in H is a *simple* hyperpath whose edges are alternately matched and unmatched w.r.t. M (an alternating hyperpath does not admit a generic hypercycle). An *augmenting hyperpath* in H is an alternating hyperpath that starts from and ends on unmatched edges, each of which is incident to at most one matched edge of H w.r.t. M.

5.2 Matching in Hypergraph and Solution for Ridesharing

Now, we are ready to show that finding a solution for this variant of simplified ridesharing problem is equivalent to finding a matching in the pick-up hypergraph.

Theorem 5.2.1. *The simplified ridesharing problem (G, R) when constraints (C2) and (C3) are satisfied but trips may have distinct destinations without re-take has a solution (S, σ) of k passengers if and only if the pick-up hypergraph H, based on the pick-up graph G_P of R, has a matching M of size k.*

Proof: $(\rightarrow)$ Given a solution (S, σ) of k passengers for (G, R). Let v be a passenger of driver u in (S, σ). Since u can serve v, there is an edge (u, v) in G_P, which corresponds

to a hyperedge $\{u_i, v_1, ..., v_{n_v}\}$ in H, for some $i \in [1, n_u]$. A driver u can serve up to n_u passengers, and there are n_u copies of u in H. Thus, the hyperedges representing the pick-ups of the same driver can be chosen in a way that they do not intersect. Hence, there are k vertex-disjoint hyperedges in H that form a matching M.
($\leftarrow$) Suppose that there is a matching M in H where $|M| = k$. Then every matched edge in H represents a valid pick-up between two trips in (G, R) as follows. Let $e = \{u_i, v_1, ..., v_{n_v}\}$ be a matched edge w.r.t. M, which corresponds to an edge (u, v) in G_P, where u is the driver and v is the passenger. Every hyperedge in H incident to e must be unmatched. This implies that no other trips will serve v, and v will not serve other trips because e contains $\{v_1, ..., v_{n_v}\}$, all copies of v. By Property 5.1.2, every hyperedge e in H that contains u_j, where u_j is a child in e and $j \neq i$, must contain all of $\{u_1, ..., u_{n_u}\}$, which means no other trip can serve u as well. For the vertices where the whole set $\{w_1, ..., w_{n_w}\}$ that are not matched, the corresponding vertices in G_P represent singly drivers, only serve themselves. Therefore, a matching M in H gives a solution (S, σ) of k passengers to instance (G, R). □

By Theorem 5.2.1, the size of M in H equals to the number of passengers in a solution (S, σ) for an instance (G, R). Hence, by maximizing the size of a matching in H, we minimize the number of drivers for the ridesharing problem. Note: neither a k-uniform hypergraph H nor a perfect matching M of H is required for solving the ridesharing problem. For simplicity, we drop the prefix "hyper" - for hyperpath, hyperdedge, and hypercycle most of the times.

5.2.1 Augmenting Graph

The key of most algorithms for finding a maximum matching in a graph is based on the maximum maching theorem by Berge [6].

Theorem 5.2.2. *Berge's maximum matching theorem: A matching M in a graph G is maximum if and only if there exists no augmenting path in G w.r.t M.*

We will extend Berge's maximum matching theorem to hypergraphs so that we know a matching in a hypergraph is maximum or not. We prove the following theorem.

Theorem 5.2.3. *A matching M in a hypergragh H is maximum if and only if there exists no augmenting graph in H w.r.t. M.*

We call this theorem, the maximum hyper-matching theorem. Augmenting graph is a generalization of augmenting path for hypergraphs. Before defining an augmenting graph, we need to define end-edges. An edge e is said to be an *end-edge* if e is incident to at most one other edge. Similarly, a vertex u is said to be an *end-vertex* if u is incident to (belongs to) only one edge. Let H be an arbitrary hypergraph and M be a matching in H. An augmenting graph T is a connected subgraph of H w.r.t M such that T satisfies the following:

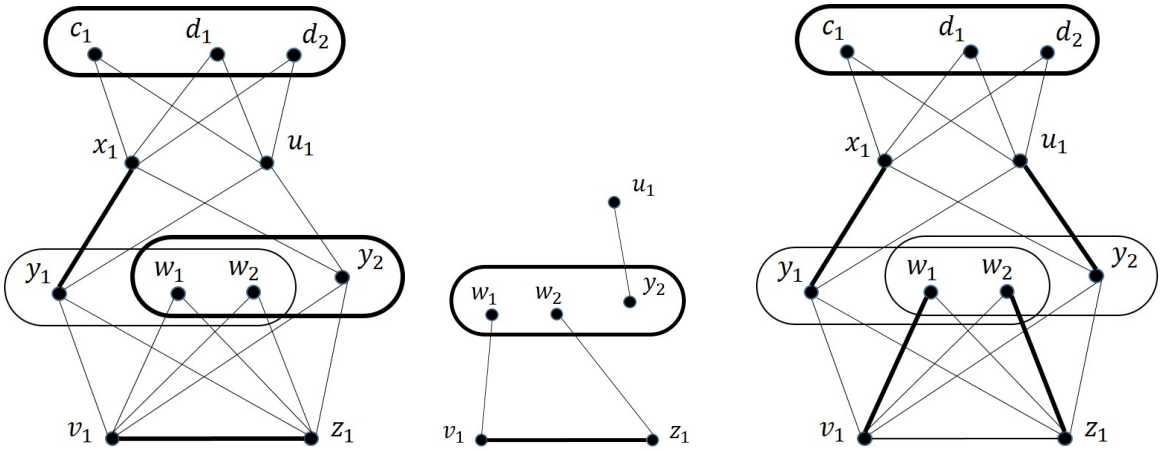
(A1) Every end-edge e in T is an unmatched edge, and e is incident to at most one matched edge in H.

(A2) For every pair of vertices u, v in T, every path in T connecting u and v is an alternating path.

(A3) All matched edges of H incident to an unmatched edge that is in T must be in T.

(A4) The number of unmatched edges in T is greater than the number of matched edges in T.

The (A2) condition is very important because it implies that unmatched edges are only incident to unmatched edges and matched edges are only incident to matched edges in T. Also, notice that Theorem 5.2.3 reduces to Berge's Theorem when H is a graph because any augmenting graph T is an augmenting path in H. Figure 5.2 shows an example of an augmenting graph of a hypergraph w.r.t. a matching.



(a) A hypergraph H with matching M

(b) An augmenting graph T of H w.r.t. M

(c) New matching after augmentation, $M = M \oplus T$

Figure 5.2: An example of an augmenting graph of a hypergraph w.r.t a matching. The bold edges represents a matched edge, and non-bold edge represents an unmatched edge.

Lemma 5.2.1. *If a hypergraph H has an augmenting graph T w.r.t. matching M, then H has a larger matching than M.*

By Lemma 5.2.1 which is proved later, if there exists an augmenting graph T in H w.r.t. a matching M, we can enlarge M by augmenting M with T, i.e. $M = M \oplus T$, see Figure 5.2. The idea for finding a maximum matching in the pick-up hypergraph is to find an augmenting graph in each iteration until none exists. We describe the algorithm for finding a maximum matching in a pick-up hypergraph in the next section.

5.3 Algorithm

Overview The proposed algorithm works as an augmenting algorithm, such as Edmonds' maximum matching algorithm [13]. There are two steps in our algorithm. First, the algorithm finds a maximal matching in H. Consider a matching M in H, which is not maximal. The algorithm finds a non-empty set of edge-disjoint alternating paths $T = \{P_1, P_2, ..., P_h\}$ in H w.r.t. M such that T has exactly one augmenting path and all other paths in T are even length alternating paths. Here, T is an augmenting graph formed by $P_1 \cup P_2 \cup ... \cup P_h$. Then by Lemma 5.2.1, augmenting M with T (i.e. $M = M \oplus T$) will increase the size of M by one. This process continues until a maximal matching M is obtained.

The algorithm enters the second step after a maximal matching M has been obtained. At each iteration of the second step, a matched portal edge e of H is picked. e is removed from M, so the vertices incident to e become free (unmatched). The algorithm will try to find a set of edge-disjoint alternating paths $T = \{P_1, P_2, ..., P_h\}$ in $E(H) \setminus \{e\}$ w.r.t $M' = M \setminus \{e\}$, where T resembles an augmenting graph. If T cannot be found, a different matched portal edge w.r.t. M is checked. If T can be found, set $M' = M' \oplus T$, which increases the size of M' by one so that $|M'| = |M|$. Then, the algorithm tries to find a larger matching w.r.t. the new matching M' by starting from the first step. Essentially, this is how the algorithm make progress. This second step stops when all matched portal edges are checked and no augmenting graph T can be found. We call this algorithm Maximum-Matching-for-Hypergraph (MMH for short), and it is shown in Algorithm 4. The details of the procedures *Search* and *CandidateSearch* are presented in Sections 5.3.2 and 5.3.3 respectively. Also, we provide a full example of Algorithm 4 in Appendix A.

Algorithm 4 Maximum-Matching-for-Hypergraph

1: Input: A pick-up hypergraph H.
   Output: A maximum matching M in H.
2: $M := \emptyset$;
3: Pick an unmatched edge $e \in H$ where e is incident to at most one matched edge
4: (Found, T) := $Search(H, M, \emptyset, e, \emptyset)$
5: **if** Found == True **then**
6: $M := M \oplus T$ /* Augment T */
7: **end if**
8: Repeat steps 3-7 until $|M|$ cannot be increased and every unmatched edge w.r.t. M is processed
9: Pick a matched portal edge $e \in E(H)$ w.r.t. M
10: $M := CandidateSearch(H, M, e)$
11: Repeat steps 8-9 until $|M|$ cannot be increased and every matched portal edge w.r.t. M is processed
12: **return** M

Note that after an augmentation, the set of matched and unmatched edges can be completely different. It needs to check every unmatched and matched edge again respectively for steps 3-8 and steps 9-11.

5.3.1 Augmenting path

The proposed algorithm (specifically in procedure *Search*) needs to find a set of alternating paths in the pick-up hypergraph w.r.t. a matching. We propose a method for finding an augmenting path in general hypergraphs by extending the algorithm in [8] for graphs to hypergraphs. It may be possible to enhance (or simplify) the current method for finding an augmenting path in hypergraphs more efficiently, such as extend from a simpler maximum matching algorithm [21], or follow a similar approach for finding a perfect matching in bipartite hypergraphs [3] (which is a recent result). However, we only focus on finding an augmenting path in polynomial time in the size of a hypergraph. The method for finding an augmenting path in general hypergraphs is detailed in Sections 5.5 to 5.7 along with its analysis. We shall note that there is a similar aspect between [3] and our approach. In [3], it relies on the hypergraph version of Hall's theorem, which is proved in [18]. Our proposed algorithm relies on the hypergraph version of Berge's theorem, that is, Theorem 5.2.3 (maximum hyper-matching theorem).

5.3.2 Main search

The core of the algorithm is to find a non-empty set of edge-disjoint alternating paths $T = \{P_1, P_2, ..., P_h\}$ in the pick-up hypergraph H w.r.t matching M such that the graph induced by $T = \{P_1, P_2, ..., P_h\}$ forms an augmenting graph. The *Search* procedure is only responsible for finding and returning an augmenting graph T, which is shown on the next page.

The first alternating path $P_1 \in T$ found by the Search procedure is an augmenting path, so P_1 has one more unmatched edge than matched edge. Each path of $\{P_2, ..., P_h\}$ is an even length alternating path. The paths in T are found by the algorithm one by one to ensure that that paths are pairwise edge-disjoint. In fact, we make sure that the paths $\{P_1, P_2, ..., P_h\}$ form an augmenting graph T of H. Once T is found, we take the symmetric difference of M and T to increase the size of the matching by one.

We now describe how the *Search* procedure finds T in detail. Let M be the current matching in the pick-up hypergraph H. The algorithm first finds an augmenting path P_1 starts from and ends with unmatched edges w.r.t. M. P_1 becomes the initial augmenting graph, $T = P_1$. If P_1 does not contain any unmatched portal edge, $M \oplus P_1$ is a valid augmentation that gives a larger matching by Lemma 5.2.1. Suppose P_1 contains some unmatched portal edges. If we perform the symmetric difference $M' = M \oplus P_1$, M' may not be a valid matching. This is because there can be matched edges in $H \setminus P_1$ w.r.t. M incident to unmatched portal edges of P_1. After augmentation $M' = M \oplus P_1$, two matched edges w.r.t. M' may be incident to each other, which is not a valid matching. To solve this problem, we try to find alternating paths that start from the matched edges incident to the unmatched portal edges of P_1.

```
1: procedure SEARCH(H, M, T, e, List)
2: Find an alternating path P_e in E(H) \ E(T) that starts from e and ends with an unmatched
   edge w.r.t. M such that P_e does not contain any edge from List, and no unmatched edge in P_e
   intersects with any unmatched edge in T.
3: if P_e cannot be found then
4:     return (False, ∅)
5: end if
6: Set T = T ∪ P_e
7: if P_e does not contain any unmatched portal edge, return (True, T)
8: List[e] := List;  /* A list of unmatched portal edges should not be included when finding a
                        different alternating path that starts from e */
9: for each unmatched portal edge e' in P_e do
10:    Let e_1, ..., e_k be the matched edges in H \ T that are incident to e'.
11:    for each e_i do /* 1 ≤ i ≤ k */
12:        (Completed, Temp) := Search(H, M, T, e_i, ∅)
13:        if (Completed == False) then
14:            Remove P_e from T; Add e' to List[e].
15:            Remove from T all the alternating paths that were found after P_e.
16:            break;
17:        end if
18:        T := Temp
19:    end for
20:    if (Completed == False) then break;
21: end for
22: if (Completed == True), return (True, T)
23: return Search(H, M, T, e, List[e])    /* find another alternating path starts from e */
24: end procedure
```

Let e be an unmatched portal edge in P_1. There can be at most $k = |e| - 2$ matched edges in $H \setminus P_1$ incident to e. For each matched edge e_i, $i = 1, ..., k$, that is incident to e, we try to find an even-length alternating path P_{e_i} starts from e_i and ends with an unmatched edge e_t w.r.t. M one by one. We allow the unmatched end-edge e_t to be incident to matched portal edge(s) of P_1 as long as it cannot be extended anymore. In other words, P_{e_i} is an alternating path ends with an unmatched edge in $H \setminus T$ w.r.t. M. In addition, we make sure that unmatched edges of P_{e_i} do not intersect with unmatched edges in T. Then, add P_{e_i} to T, and find the next alternating path starts from e_{i+1}. Suppose we can find k different alternating paths for every e_i, $i = 1, ..., k$. T will consist of alternating paths P_1, P_{e_1}, ..., P_{e_k} where P_1 has one more unmatched edge and P_{e_i} has equal number of unmatched and matched edges. $M \oplus T$ increases the size of the matching by one. If P_{e_i} also contains some unmatched portal edges, then follow the same procedure to find another set of alternating paths in $H \setminus T$ that start from the matched edges incident to unmatched portal edges.

Recursively do this until all matched edges incident to unmatched portal edges are included in T so that we are safe to perform the augmentation $M = M \oplus T$.

Suppose that there does not exist a desired alternating path in $H \setminus T$ starts from e_i that is incident to an unmatched portal edge e of the path $P_{e_y} \in T$. The algorithm will try to find another alternating path P'_{e_y} starts from e_y to replace P_{e_y}, where P'_{e_y} does not contain e. If P'_{e_y} can be found, P_{e_y} is removed from T and P'_{e_y} is added to T. In addition, T only keeps the alternating paths that were found prior to finding P_{e_y}. If P'_{e_y} does not exist, the algorithm backtracks further to the path containing the unmatched portal edge e' that is incident to e_y. This can backtrack to the initial augmenting path P_1. If a different augmenting path cannot be found, the algorithm checks the next unmatched edge that is incident to at most one matched edge.

5.3.3 Candidate search

The intuition of *CandidateSearch* can be summarized as follows. Let M be a maximal matching in the pick-up hypergraph H. We know that the current corresponding drivers in H based on M cannot serve anymore passengers. However, it is possible that a particular passenger should be a driver instead. In *CandidateSearch*, a matched portal edge $e = (u_i, v_1, ..., v_{n_v})$ is removed from M so that $v_1, ..., v_{n_v}$ become unmatched. As a result, these passengers represented by vertices in e have a chance to become drivers. The CandidateSearch procedure is shown below:

1: **procedure** CANDIDATESEARCH(H, M, e)
2: $M' := M - \{e\}$; $E(H') = E(H) \setminus \{e\}$;
3: Pick an unmatched edge $e' \in H'$ where e' is incident to at most one matched edge
4: (Found, T) := $Search(H', M', \emptyset, e', \emptyset)$
5: **if** Found == True **then**
6: $M' := M' \oplus T$ /* Augment T */
7: **end if**
8: Repeat steps 3-7 until $|M'|$ cannot be increased and every unmatched edge w.r.t. M' is processed
9: **if** $|M'| < |M|$ **then** /* no augmentation was made */
10: Set $M' = M' \cup \{e\}$;
11: **end if**
12: **return** M'
13: **end procedure**

Let $M' = M \setminus \{e\}$. Every edge incident to e is unmatched w.r.t. M' since e was a matched edge. Let H' be the hypergraph H with the edge e removed. Then procedure *Search* is called, H' is given as an input along with M'. In this way, if an augmenting graph T can be found, e is not included in T. There are four possible outcomes after one iteration in *CandidateSearch*:

(1) T is found and augmented - $M' = M' \oplus T$. As a result, $|M'|$ increased by one (so now $|M'| = |M|$). There are unmatched edges w.r.t M' have not been processed.

(2) Similar to (1), $|M'|$ increased by one, but all unmatched edges w.r.t M' have been processed.

(3) T cannot be found. $|M'|$ unchanged (so $|M'| < |M|$), but there are unmatched edges w.r.t M' have not been processed.

(4) $|M'|$ unchanged, and all unmatched edges w.r.t M' have been processed.

For both (1) and (3), call *Search* is called again to process a different unmatched edge w.r.t. M'. This is repeated until all unmatched edges w.r.t. M' are processed. For (2), no further process is required, so just proceed to check the next matched portal edge w.r.t M'. For (4), revert M' to the original matching M, and check the next matched portal edge w.r.t M.

5.4 Analysis of Optimality

As mentioned, many algorithms rely on the Berge's maximum matching theorem for finding a maximum matching in graphs. Our algorithm also relies on the hypergraph version of the theorem. Let us restate the maximum matching theorem and the hypergraph version of the theorem.

Theorem 5.4.1. *Berge's maximum matching theorem [6]: A matching M in a graph G is maximum if and only if there exists no augmenting path in G w.r.t M.*

Theorem 5.4.2. *Maximum hyper-matching theorem: A matching M in a hypergragh H is maximum if and only if there exists no augmenting graph in H w.r.t. M.*

Theorem 5.4.2 applies to arbitrary hypergraphs, not just pick-up hypergraphs. In this section, we will first prove the contrapositive of Theorem 5.4.2 in Lemma 5.4.1 and Theorem 5.4.3, for each direction.

Lemma 5.4.1. *If a hypergraph H has an augmenting graph T w.r.t. matching M, then H has a larger matching than M.*

Proof: Let M_1 be the set of edges of T belonging to M and $M_2 = E(T) - M_1$. In other words, M_1 is the set of matched edges in T and M_2 is the set of unmatched edges in T. Since T is an augmenting graph, $|M_2| \geq |M_1| + 1$. Thus, $(M - M_1) \cup M_2$ gives a larger matching than M. The new matching is valid because when an unmatched edge $e \in M_2$ becomes matched, every matched edge in H incident to e becomes unmatched at the same time, and no two unmatched edges in M_2 are incident to each other by the definition of augmenting graph. □

The following lemma is used in Berge's maximum matching theorem. Again, we will extend this to hypergraphs stated in Lemma 5.4.3.

Lemma 5.4.2. *Let M_1 and M_2 be two matchings in a graph G. Let G' be the spanning subgraph of G with edge set $E = (M_1 - M_2) \cup (M_2 - M_1)$, the symmetric difference of M_1 and M_2. Then each component of G' is one of the following types:*

1. *An isolated vertex;*
2. *An even cycle whose edges alternately in M_1 and M_2.*
3. *A simple path whose edges alternately in M_1 and M_2 such that each end-vertex of the path is unmatched with respect to exactly one of M_1 and M_2.*

Lemma 5.4.3. *Let M_1 and M_2 be two matchings in a hypergraph H. Let S be the spanning subgraph of H with edge set $E = (M_1 - M_2) \cup (M_2 - M_1)$, the symmetric difference of M_1 and M_2. Then each component of S is one of the types stated in Lemma 5.4.2 or:*

4. *A hypergraph containing an edge incident to at least three other edges and each edge of the graph is matched to exactly one of M_1 and M_2.*

Proof: Note that $\Delta(S)$ is still at most two. Hence, Lemma 5.4.2 still holds for the components in S. Let S_1 be a component of S. If S_1 is not an even cycle nor a (trivial) path, then there exists an edge in S_1 that is incident to at least three other edges. Observe that the edges of each path and even cycle in S are alternately in M_1 and M_2 since no two edges in a matching are incident to each other. It is obvious that each edge of the graph is matched to exactly one of M_1 and M_2 due to $E(S) = (M_1 - M_2) \cup (M_2 - M_1)$. □

Two other lemmas are required to prove Theorem 5.4.3. Let us introduce some notations first. In the following proofs, a (hyper)path always refer to a simple (hyper)path due to the fact that all alternating paths are simple by definition. Let M and M' be two matchings in a hypergraph H and e be a portal edge in H. Sometimes, we call a portal edge a portal only. If $e \in M$, then it is called an *M-portal*, otherwise $e \in M'$ and called an *M'-portal*. For simplicity, edges from M are sometimes called M edges. Also, when not explicitly stated, M edges can mean both M-portals or edges from M. Similarly for edges from M'.

Lemma 5.4.4. *Let M and M' be two matchings in a hypergraph H and S be the spanning subgraph of H with edge set $E = (M - M') \cup (M' - M)$. Let S_1 be a component of S and e be an M'-portal in S_1. Let e_1 be an edge in S_1 that is incident to e and T_1 be the subgraph of $S_1 \setminus \{e\}$ containing the edges that are reachable from e_1. If T_1 contains none of the two subgraphs stated below, then T_1 has more M edges than M' edges.*

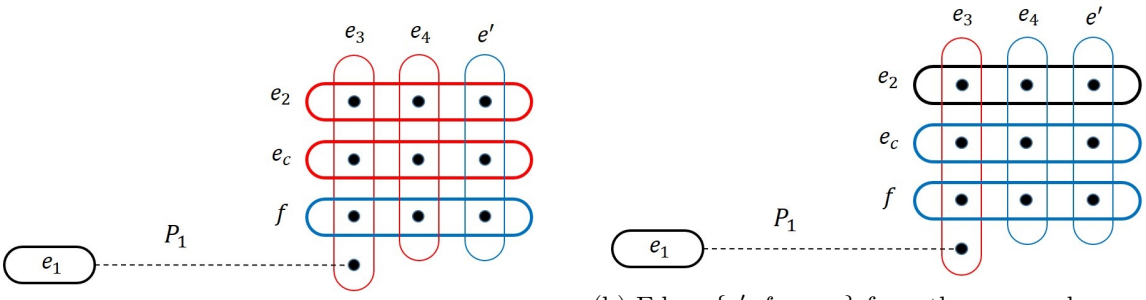
1. *A path P_1 starts from e_1 and ends at an M' end-edge e_{t_1} of T_1 (type 1).*

2. *An even length path P_1 starts from e_1 to an M' edge that is incident to an even length cycle C_1 of T_1 (type 2).*

Proof: If T_1 is a tree and every end-edge in T_1 is from M, it is obvious that T_1 has more M edges than M' edges. Suppose T_1 contains some cycles, and e_1 must go through an odd length path (starts at e_1) to reach any cycle in T_1. We will note the difference between the number of M edges and the number of M' edges in T_1 after removing a sequence of cycles from T_1.

Remove an even length cycle C from T_1, and this may divide T_1 into multiple components. Consider a component T_c of $T_1 \setminus C$ that does not contain the edge e_1. If there is an M' edge e' in T_c incident to C, then e' is the only edge incident to C. Assume, for contradictory, that there is another edge f in T_c incident to C. There exists a path $P_{e'}$ from e' to f in T_c since e' and f belong to the same component. This path $P_{e'}$ plus a sub-path of C forms another cycle C_{T_c}. The edge $e_c \in C$ incident to e' is an M-portal, and there exists a path P_1 in $T_1 \setminus C_{T_c}$ from e_1 to the edge incident to e_c. $P_1 \cup C_{T_c}$ forms the type 2 subgraph (see Figure 5.3 (a) $\rightarrow$ (b)), which leads to the contradiction that T_1 does not contain this subgraph. Thus, T_c can only have exactly one M' edge incident to C. On the other hand, if T_c does not have an M' edge incident to C, T_c can have multiple M edges incident to C. If T_c is a tree, it has at most one end-edge from M' and all other end-edges from M, so the number of M edges is at least the number of M' edges in T_c. Suppose T_c contains cycles. Remove another even length cycle C' from T_c, which may further divide T_c into more components. Let T'_c be a component in $T_c \setminus C'$. By the above analysis, T'_c can have at most one M' edge incident to C'. Some end-edges in T'_c can be incident to previously removed cycles because they may not be end-edges in T_1. However, these end-edges in T'_c must from M. Otherwise, there exists an even length path in T_1 from e_1 to one of these end-edges that connects to a cycle, which is the type 2 subgraph - a contradiction. Again, if T'_c is a tree, it has at most one end-edge from M' and all other end-edges are from M, so the number of M edges is at least the number of M' edges in T'_c. If T'_c still has cycles, then remove a cycle from T'_c one by one until there is none. Each of the resulting components is a tree that has at least as many M edges as M' edges.

Consider the component T_{c_1} of $T_1 \setminus C$ that contains the edge e_1. Every edge in T_{c_1} incident to C is an M edge. Otherwise, it will form the type 2 subgraph. If T_{c_1} is a tree, it has more M edges than M' edges since every end-edge in T_{c_1} is from M. Suppose T_{c_1} contains cycles. Similarly, remove an even length cycle C' from T_{c_1}, which may further divide T_{c_1} into more components. For the case where the components of $T_{c_1} \setminus C'$ that do not contain the edge e_1 is identical to the above. Let T'_{c_1} be the component of $T_{c_1} \setminus C'$ that contains the edge e_1. If T'_{c_1} is a tree, it has more M edges than M' edges. If T'_{c_1} has cycles, again, remove a cycle from T'_{c_1} one by one until there is none. Each of the resulting components is a tree that has at least as many M edges as M' edges, except for the tree



(a) Edges $\{e_2, e_3, e_4, e_c\}$ form cycle C and edges $\{e', f\}$ form component T_c.

(b) Edges $\{e', f, e_4, e_c\}$ form the new cycle C_{T_c}, and there is a path P_1 from e_1 to e_3, which forms type 2 subgraph.

Figure 5.3: An example for forming type 2 subgraph. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge.

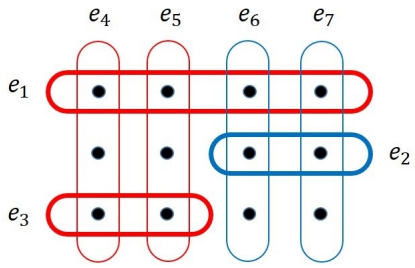
containing the edge e_1, which has more M edges than M' edges. Therefore, T_1 has more M edges than M' edges. □

Lemma 5.4.5. *Let M and M' be two matchings in a hypergraph H and S be the spanning subgraph of H with edge set $E = (M - M') \cup (M' - M)$. Let S_1 be a component of S, where S_1 contains some cycles, and every end-edge in S_1 is from M. If S_1 contains none of the two subgraphs stated below, then the number of M edges is at least the number of M' edges in S_1.*

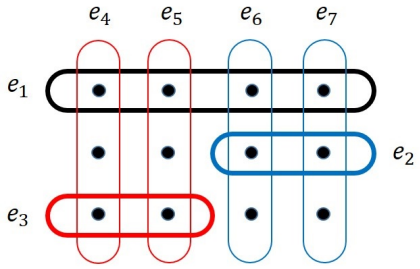
1. *Two even length cycles C_1 and C_2 in S_1 that are connected by a path P, where P is odd and has more edges from M' than M (subgraph 1).*
2. *Two even length cycles C_1 and C_2 in S_1 that share a path P, where P is odd and has more edges from M than M' (subgraph 2).*

Proof: The proof is similar to Lemma 5.4.4 - by removing a sequence of cycles from S_1. If S_1 contains only one cycle, it is trivial. Suppose S_1 contains multiple cycles. Remove an even length cycle C from S_1, which may divide S_1 into multiple components. Consider a component T_c of $S_1 \setminus C$. If there is an M' edge e' in T_c incident to C, then e' is the only M' edge incident to C since another M' edge in T_c incident to C would form subgraph 2 (see Figure 5.4 (a) → (b) and (c) → (d)).

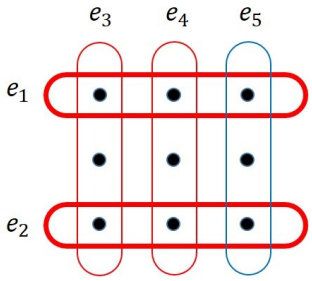
Suppose T_c is a tree. Notice that T_c cannot just be a single M' edge. Otherwise, $C \cup T_c$ forms subgraph 2 (see Figures 5.4c, 5.4d). T_c has at most one end-edge from M' and all other end-edges from M, so the number of M edges is at least the number of M' edges in T_c. Suppose T_c contains cycles. Remove an even length cycle C' from T_c, which may further divide T_c into more components. Let T'_c be a component in $T_c \setminus C'$. By the above analysis, T'_c can have at most one M' edge incident to C'. Some end-edges in T'_c can be incident to previously removed cycles because they may not be end-edges in S_1. However,



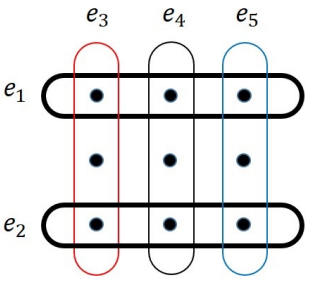
(a) Edges $\{e_1, e_3, e_4, e_5\}$ form cycle C and edges $\{e_2, e_6, e_7\}$ form component T_c.



(b) Edges $\{e_1, e_3, e_4, e_5\}$ form a cycle C_1 and edges $\{e_1, e_2, e_6, e_7\}$ form a cycle C_2. The path shared by C_1 and C_2 is e_1.



(c) Edges $\{e_1, e_2, e_3, e_4\}$ form cycle C and edge e_5 forms component T_c.



(d) Edges $\{e_1, e_2, e_3, e_4\}$ form a cycle C_1 and edges $\{e_1, e_2, e_4, e_5\}$ form a cycle C_2. The path shared by C_1 and C_2 is e_1, e_4, e_2.

Figure 5.4: Examples for forming subgraph 2. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge.

these end-edges in T'_c must from M. Otherwise, there exists an odd length path with more M' edges in S_1 connecting C and C', which is the subgraph 1 - a contradiction. Hence, T'_c has at most one end-edge from M', and all other end-edges are from M. If T'_c is a tree, the number of M edges is at least the number of M' edges in T'_c. If T'_c still has cycles, then remove a cycle from T'_c one by one until there is none. Each of the resulting components is a tree that has at least as many M edges as M' edges. □

Theorem 5.4.3. *Let M and M' be two matchings in a hypergraph H with $|M'| > |M|$, and let S be the spanning subgraph of H with edge set $E = (M - M') \cup (M' - M)$. Then there is a component S_1 of S such that there exists an augmenting graph w.r.t. M in S_1.*

Proof: Since $|M'| > |M|$, there is a component S_1 in S such that S_1 has more edges from M' than M. If every edge in S_1 is incident to at most two edges of S_1, then S_1 is an odd-length path, and thus an augmenting graph w.r.t. M (observed by Lemma 5.4.2). Otherwise, by Lemma 5.4.3, S_1 contains an edge e incident to at least three edges of S_1, which is a portal edge. Note that edges in any path in S_1 alternate between M and M'.

Case 1: Suppose S_1 is a tree - generic hypercycle free. Since S_1 has more M' edges than M edges, there is a path P in S_1 such that the two end-edges of P are M' end-edges

in S_1. We will show that an augmenting graph T w.r.t. M can be constructed from P. Initially, let $T = P$. Notice that if an M edge in $S_1 \setminus T$ is incident to an edge $e \in T$ then e is an M'-portal. If T does not contain any M'-portal of S_1, then T is an augmenting graph w.r.t. M by definition. Assume that there are M'-portals in T. Let Ω_T be the set of M'-portals of S_1 in T. We process the M'-portals in Ω_T one by one, trying to include the subtrees of S_1 incident to the M'-portals in Ω_T (trying to satisfy the A3 condition in the definition of augmenting graph). If we just include M edges incident to M'-portals in Ω_T to T, T will not have more M' edges than M edges. Thus, we try to include subtrees containing equal number of M and M' edges to T so that T has more M' edges. The way to accomplish this is quite similar to the Search procedure in Algorithm 4.

Let e be an M'-portal in Ω_T and e_1, ..., e_k be the edges in $S_1 \setminus T$ that are incident to e. Each e_i, $1 \leq i \leq k$, is a M edge. The subgraph of S_1 reachable from e_i without using any edge of T is a tree, denoted by T_i. We check every T_i. If every end-edge in T_i is an M edge, then T_i has more M edges than M' edges. We remove $e \cup T_i$ from S_1 to get a subgraph of S_1. Since S_1 has more M' edges than M edges and the number of M' edges is at most that of M edges in $e \cup T_i$, the subgraph of S_1 has more M' edges than M edges. From this, there is a component S_1' of the subgraph that is a tree and has more M' edges than M edges. Furthermore, any M' edge in S_1' is not incident to any edge in $S_1 \setminus S_1'$. Then an augmenting graph in S_1' w.r.t. M is also an augmenting graph in S_1, and we try to find such an augmenting graph anew from S_1'.

Assume that every T_i, $1 \leq i \leq k$, has an end-edge e_{t_i} which is an M' edge. Let P_i be the path from e_i to e_{t_i} and $T = T \cup P_i$. If P_i has an M'-portal, we include every M'-portal of S_1 in P_i to Ω_T. We mark edge e processed. After every M'-portal in T is processed, T is an augmenting graph w.r.t. M because every end-edge in T is an M' edge, implying T has more M' edges than M edges, and every M edge in $S_1 \setminus T$ is not incident to T.

Case 2: Suppose S_1 is not a tree, which contains cycles. In this case, a path P starts and ends with an edge from M' may not exist in S_1. In fact, there are three cases to be considered: (1) S_1 has at least two end-edges from M', (2) S_1 has exactly one end-edge from M', and (3) S_1 has no end-edge from M'. First notice that in all three cases, every cycle C in S_1 must contain at least one portal edge. Otherwise, no other edges in S_1 can connect to C, which makes C an isolated component.

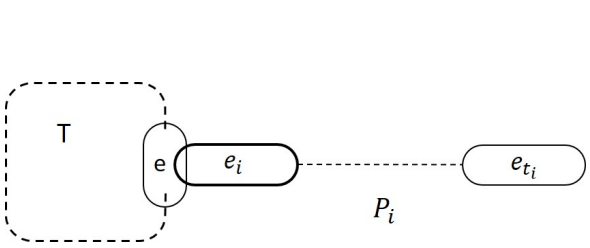
Case 2.1: Let us consider the first case: S_1 has at least two end-edges from M'. The analysis follows a similar structure as in the tree case. Let P be a path with two M' end-edges in S_1. We will try to construct an augmenting graph T w.r.t. M from P. Let $T = P$, which has exactly one more edge from M' than M. Let Ω_T be the set of M'-portals of S_1 in T. Again, if $\Omega_T = \emptyset$, T is an augmenting graph w.r.t. M by definition. Suppose $\Omega_T \neq \emptyset$. We will process the M'-portals in Ω_T one by one, and try to include subgraphs of S_1 incident to the M'-portals in Ω_T. Let e be an M'-portal in Ω_T and e_1, ..., e_k be the edges in $S_1 \setminus T$ that are incident to e. Each edge e_i, $1 \leq i \leq k$, is called a *root* edge. Let T_i be

the subgraph of S_1 containing the edges that are reachable from e_i without using any edge of T. Then, check if there exists in T_i, one of the three subgraphs stated in the following:

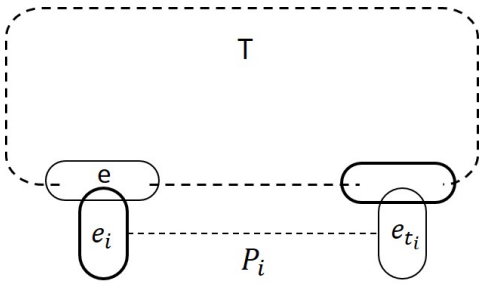
(Type 1) A path P_i starts from e_i and ends at an M' end-edge e_{t_i} of S_1.

(Type 2) An even length path P_i starts from e_i to an M' edge that is incident to an even length cycle C_i of T_i.

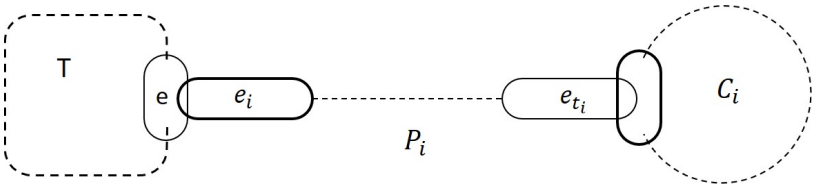
(Type 3) A path P_i starts from e_i to an edge e_{t_i} that is incident to an M edge of T.



Type 1 - e_{t_i} being an end-edge in S_1



Type 3 - e_{t_i} is incident to an M edge of T



Type 2 - e_{t_i} being an M' edge incident to an even length cycle C_i of T_i

Figure 5.5: Three types of subgraphs to be checked. The bold rounded rectangle represents an M edge, and non-bold rounded rectangle represents an M' edge.

All three types are illustrated in Figure 5.5. Type 1 and type 3 are almost the same. For type 2, the length of P_i can be zero, but C_i must include e_i then. Let G_{T_i} be one of the subgraphs above. Notice that G_{T_i} has equal number of M' edges and M edges. Add G_{T_i} to T, and T has exactly one more M' edge than M edge. Include every M'-portal of S_1 in G_{T_i} to Ω_T. Then continue to process the next root edge. Initially, T has one more M' edge than M edge, and we add to T one of the subgraphs G_{T_i} each time, where G_{T_i} has equal number of M and M' edges. Thus, T has exactly one more edge from M' than M so far. After process every M'-portal in Ω_T, if one of the subgraphs (types 1, 2, 3) can be found each time, the resulting T is an augmenting graph since T satisfies the definition of augmenting graph.

Suppose T_i does not contain the type 1, type 2, and type 3 subgraphs. Then every end-edge in T_i is an M edge. Assume that e_i is the only edge in T_i that is incident to portal edge in Ω_T. Notice that e_i must be an end-edge in T_i. By Lemma 5.4.4, T_i has more M

edges than M' edges. Remove $e \cup T_i$ from S_1 to get a subgraph of S_1. Because S_1 has more M' edges than M edges, there is a component S_1' of the resulting subgraph has more M' edges than M edges. Furthermore, any M' edge in S_1' is not incident to any edge in $S_1 \setminus S_1'$. Then an augmenting graph in S_1' w.r.t. M is also an augmenting graph in S_1, and we try to find an augmenting graph anew from S_1'.

Assume that there exists some edges in T_i, other than e_i, are incident to portal edges in Ω_T. Let Σ_T be the set of M-portals of S_1 in T, and let h be an edge in Σ_T. We will try to add to T subgraphs of $S_1 \setminus T$ incident to the M-portals in Σ_T. Let h_1, ..., h_k be edges incident to h that are not in T, and H_j be the subgraph of $S_1 \setminus T$ containing the edges that are reachable from h_j without using any edge of T, for $1 \leq j \leq k$. Check each H_j to find one of the three types of subgraphs stated below, which are very similar to types 1-3:

(Type 4) A path P_j starts from h_j and ends at an M' end-edge e_{t_j} of S_1.

(Type 5) An odd length path P_j starts from h_j to an M' edge that is incident to an even length cycle C_j of H_j.

(Type 6) A path P_j starts from e_j to an edge e_{t_j} that is incident to a M edge of T.

As a result, each type of the subgraph G_{H_j} has one more edge from M' than M. Let P_{T_i} be any path in T_i from e_i to the edge incident to an M'-portal $e' \in \Omega_T$ with $e' \neq e$. Add $G_{H_j} \cup P_{T_i}$ to T. Include every M'-portal of S_1 in $G_{H_j} \cup P_{T_i}$ to Ω_T. Since $G_{H_j} \cup P_{T_i}$ has equal number of M' edges and M edges, the new T has exactly one more M' edge than M edge. Repeat the whole process for the next M'-portal in Ω_T.

Suppose that for every $h \in \Sigma_T$, H_j does not contain the type 4, 5, and 6 subgraphs. Then every end-edge in H_j is an M edge with the possible exception of h, for $1 \leq j \leq k$. Remove T, T_i, H_j that is incident to $h \in \Sigma_T$ (for $1 \leq j \leq k$) from S_1 to get a subgraph of S_1. By Lemma 5.4.4, T_i has more M edges than M' edges, and H_j has at least as many M edges as M' edges. Since T has exactly one more M' edges than M edges, the resulting subgraph of S_1 has more M' edges than M edges. From this, there is a component S_1' of the resulting subgraph that has more M' edges than M edges. Furthermore, any M' edge in S_1' is not incident to any edge in $S_1 \setminus S_1'$. Then, we try to find an augmenting graph anew from S_1'. Notice that S_1' can have at least two, exactly one, or no end-edges from M'. For the last two cases, they are proved below, which follow the same structure.

Case 2.2: For the second case: S_1 has exactly one end-edge from M'. The proof is similar to the first cycle case (Case 2.1). Let e_s be the only end-edge in S_1 that is from M'. There must exists an odd length path P starts from e_s and connects to an even length cycle C in S_1. Otherwise, S_1 has at least as many M edges as M' edges by Lemma 5.4.4, a contradiction. Let us denote the subgraph $P \cup C$ by T, which has exactly one more edge from M' than M. Let Ω_T be the set of M'-portals in T. We then follow the construction stated above to process each M'-portal in Ω_T.

The only difference between the first case and this case is that when searching the subgraph T_i that is incident to an M'-portal edge $e \in \Omega_T$, we only look for type 2 and type 3 subgraphs since e_s is the only M' end-edge in T. Similarly when searching the subgraph H_j that is incident to an M-portal edge $h \in \Sigma_T$, we only look for type 5 and type 6 subgraphs. Then rest is identical to the construction stated in the first cycle case.

Case 2.3: Finally, S_1 has no end-edge from M'. The proof is almost identical to the second cycle case (Case 2.2). By the contrapositive of Lemma 5.4.5, there exists at least one of the two following subgraphs in S_1. (1) Two even length cycles C_1 and C_2 in S_1 that are connected by a path P, where P is odd and has more edges from M' than M. (2) Two even length cycles C_1 and C_2 in S_1 that share a path P, where P is odd and has more edges from M than M'. Either one of these subgraphs is used as the initial T, which has exactly one more edge from M than M'. Then the rest is identical to the second cycle case. □

This concludes the proof of Theorem 5.4.2. It remains to prove the MMH algorithm can find an augmenting graph if one exists. However, we will not complete the proof of the correctness of the algorithm. Instead, we describe and outline general ideas of what needs to be done to prove the correctness of the algorithm in the next section.

5.4.1 Correctness of the Algorithm

We will state two conjectures about the MMH algorithm, and in conjunction with Theorem 5.4.3 and these two conjectures, we claim that the MMH algorithm can find a maximum matching in pick-up hypergraphs. In the **Search** procedure of the MMH algorithm, it only finds a more restricted augmenting graph. The augmenting graph T found by the *Search* procedure satisfies the definition of augmenting graph (conditions A1-A4) and the following two additional conditions:

(A5) T can be decomposed into a set P of edge-disjoint alternating paths such that there is exactly one augmenting path p_0 in P, and all other paths in P are even length.

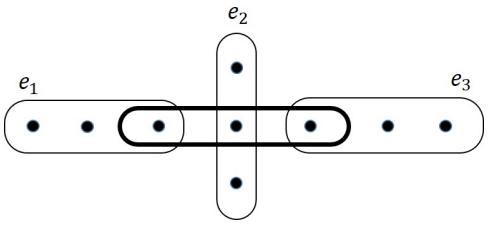
(A6) The paths in P can be given an ordering such that for each $p_i \in P$ where $i \geq 1$, the unmatched end-edge of p_i is also an end-edge in $T \setminus p_j$, $j = 0, ..., i-1$.

We call an augmenting graph satisfying conditions A1-A6 a *restricted augmenting graph.*

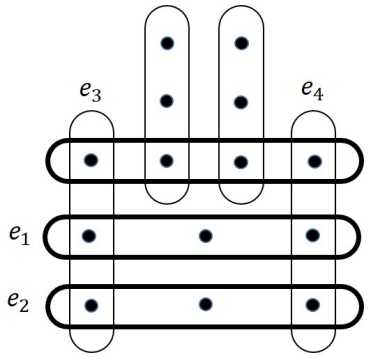
Definition 5.4.1. *An augmenting graph T of a hypergraph H is called minimal if $E(T')$ is not a proper subset of $E(T)$, where T' is any other augmenting graph of H.*

Trivial examples of non-minimal augmenting graph are shown in Figure 5.6.

The *Search* procedure precisely finds a restricted augmenting graph by finding a set P of edge-disjoint alternating paths such that P satisfies condition A6, and we state the following without a proof.



(a) Suppose the graph shown is an augmenting graph T of a hypergraph H. Then removing either e_1, e_2, or e_3 from T, it is still an augmenting graph of H.



(b) Suppose the graph shown is an augmenting graph T of a hypergraph H. Then removing $\{e_1, e_2, e_3, e_4\}$ from T, it is still an augmenting graph of H.

Figure 5.6: Examples of non-minimal augmenting graph. The bold rounded rectangle represents a matched edge and non-bold rounded rectangle represents an unmatched edge.

Conjecture 5.4.1. *Let H be a pick-up hypergraph and M be a matching in H. If there exists a restricted augmenting graph T of H, then the Search procedure in MMH algorithm finds T.*

The MMH algorithm has the following property:

- The **Search** procedure of the MMH algorithm finds only minimal restricted augmenting graph.

We will prove this property by Lemma 5.4.6.

Lemma 5.4.6. *Let H be a pick-up hypergraph and M be a matching in H. Suppose Search finds a restricted augmenting graph T of H. Then T is a minimal restricted augmenting graph.*

Proof: Recall that T found by *Search* can be decomposed into a set P of edge-disjoint alternating paths such that P contains exactly one augmenting path, and all other paths in P are even length. We will show that T is minimal by contradiction. Assume there is a restricted augmenting graph T' that is a subgraph of T such that $E_r = E(T) \setminus E(T')$ is not empty. Since T has exactly one more unmatched edge than matched edge, the number of matched edges is at least that of unmatched edges in E_r. Hence, there must exist a component T_r of $T \setminus T'$ where the number of matched edges is at least that of unmatched edges in T_r. Only unmatched edges in E_r can be incident to T'. Otherwise, T' is not an augmenting graph by definition. Any end-edge e of T_r must be unmatched for the following reasons. (1) If e is also an end-edge of T, then e must be unmatched. (2) If e is an end-edge of T_r but not an end-edge of T, then e is incident to T', so e is an unmatched edge.

Let $P' \subseteq P$ be the set of paths where every path $p_i \in P'$ has a subpath p_i^r of p_i belongs to T_r. Notice that for each $p_i \in P'$, either both end-edges of p_i^r are unmatched or the

length of p_i^r is even. Let $p_1 \in P$ be the first path containing edges of T_r added by *Search*. Consider the moment when p_1 is added to T. There are three possible situations. (1) Both the end-edges of p_1 belongs to T_r, i.e. p_1 is fully contained in T_r. Then p_1 must be the augmenting path in P. (2) Only one end-edge e_1 of p_1 belongs to T_r. Since p_1 is the first path containing edges of T_r being added, e_1 is not incident to any other path in T_r, and p_1 cannot end at a matched edge by the algorithm. Thus, e_1 is an unmatched end-edge in T_r. (3) Both end-edges of p_1 do not belong to T_r. Then a subpath p_1^r of p_1 belongs to T_r. Both end-edges of p_1^r are unmatched since they must be incident to T'. Summarizing the above, the subpath p_1^r of p_1 (p_1^r can be p_1) belongs to T_r has more unmatched edges than matched edges. All other alternating paths added to T_r after p_1 contains as many unmatched edges as matched edges. Thus, T_r has more unmatched edges than matched edges, a contradiction. Therefore, such a restricted augmenting graph T' does not exist. □

The main idea for proving the correctness of the MMH algorithm lies in the following conjecture.

Conjecture 5.4.2. *Let H be a pick-up hypergraph and M be a matching in H. Suppose there exists a minimal augmenting graph T of H. Then T can be decomposed into a set T' of edge-disjoint minimal restricted augmenting graphs w.r.t $M \setminus M'$, where $M' \subseteq M$ is a set of matched portal edges and $|T'| = |M'| + 1$.*

The proof of Conjecture 5.4.2 most likely require the pick-up hypergraph properties. In particular, Property 5.1.2 makes sure that any pick-up hypergraph is not a k-uniform k-partite hypergraph. Assuming this conjecture is true, by Lemma 5.4.1 and Lemma 5.4.6, the *Search* procedure finds all minimal restricted augmenting graphs in T' when the *ExtendSearch* procedure checks each matched edge in M. Basically, *ExtendSearch* removes $|M'|$ matched edges from M, and *Search* adds $|T'|$ unmatched edges into M one by one. Thus, $|M|$ is increased by one. From this and Theorem 5.4.3, the MMH algorithm can find a maximum matching in the pick-up. We leave the proof of Conjecture 5.4.2 as a future work. We include the following two lemmas as they might be useful for proving Conjecture 5.4.2.

Lemma 5.4.7. *Let M and M' be two matchings in a hypergraph H and S be the spanning subgraph of H with edge set $E = (M - M') \cup (M' - M)$. Let S_1 be a component of S, where S_1 contains some cycles and does not contain any M'-portal. Then the number of M' edges is at least the number of M edges in S_1.*

Proof: The proof is similar to Lemma 5.4.4. We will observe the difference between the number of M edges and the number of M' edges in S_1 after removing a sequence of cycles from S_1. Remove an even length cycle C from S_1, which may divide S_1 into multiple components. Consider a component T_c of $S_1 \setminus C$. Since there is no M'-portal in S_1, every edge in T_c incident to C is an M' edge, so there is at least one M' end-edge in T_c.

Assume T_c is a tree. If every end-edge in T_c is an M' edge, then it has more M' edges. Suppose there are some M end-edges in T_c. Let P_{c_1} be a path in T_c starts from an M' end-

edge and ends at an M end-edge, which has equal number of M' and M edges. Consider another path P_{c_2} in $T_c \setminus P_{c_1}$ where one end-edge e of P_{c_2} is incident to P_{c_1}. Since there are only M-portals in S_1, e is an M' edge. Hence, P_{c_2} has at least as many M' edges as M edges. Similarly for all other paths in T_c. Thus, T_c has at least as many M' edges as M edges.

Assume T_c contains cycles. Remove another even length cycle C' from T_c, which may further divide T_c into more components. Let T'_c be a component of $T_c \setminus C'$. Some end-edges in T'_c can be incident to previously removed cycles because they may not be end-edges in S_1. However, all these end-edges must from M' since there are only M-portals in S_1, so there can be more M' end-edges than M end-edges in T_c. Again, if T'_c is a tree, the number of M' edges is at least the number of M edges in T'_c. If T'_c still has cycles, then remove a cycle from T'_c one by one until there is none. Each of the resulting components is a tree that has at least as many M' edges as M edges. Therefore, S_1 has at least as many M' edges as M edges. □

Corollary 5.4.1. *Let M and M' be two matchings in the hypergraph H and S be the spanning subgraph of H with edge set $E = (M - M') \cup (M' - M)$. Let S_1 be a component of S, where S_1 contains some cycles and does not contain any M-portal. Then the number of M edges is at least the number of M' edges in S_1.*

Lemma 5.4.8. *Let H be an arbitrary hypergraph and M be a matching in H. Let T be a minimal augmenting graph of H. Then every cycle in T contains at least one matched portal edge of T.*

Proof: Assume for contradictory that T contains some cycles with no matched portal edge. Let C_T be the set of cycles in T without matched portal. Remove an even length cycle $C_1 \in C_T$ from T. Every edge in $T \setminus C_1$ that is incident to C_1 is a matched edge. Since C_1 has even number of matched and unmatched edges, there must exist a component T_1 of $T \setminus C_1$ has more unmatched edges than matched edges. Every unmatched edge in T_1 is not incident to edges in $T \setminus T_1$. Thus, an augmenting graph in T_1 w.r.t. M is also an augmenting graph in H. Following a similar analysis as in Theorem 5.4.3, it can be shown that T_1 contains an augmenting graph of H, which is a contradiction that T is a minimal augmenting graph of H. □

Time complexity Since we have not proved that the Search procedure can find a restricted augmenting graph if one exists. We will leave the time complexity for Algorithm MMH as a future work. The time it takes to find a restricted augmenting graph using the current algorithm may be exponential in the numbers of the edges of a pick-up hypergraph. It is possible that we could try to bound the search space by not searching certain subgraphs unless it is necessary. We leave this as an open problem:

Remark 5.4.1. *Does there exist a polynomial time algorithm in the size of a pick-up hypergraph H for finding a restricted augmenting graph in H w.r.t. a matching M in H?*

Also note that the current method relies on finding an augmenting/alternating path in the pick-up hypergraph. The algorithm introduced in the next chapter is for general hypergraphs. It is possible that it can be simplified specifically just for pick-up hypergraph.

5.5 Finding Augmenting Path in Hypergraphs

We propose an algorithm for finding an augmenting path in hypergraphs since there does not exist one. The approach from [3] is for bipartite hypergraphs, but it might be able to extend to general hypergraphs. Our approach is based on [8]. We recommend the readers to read [8] to make the proposed algorithm more easily understood.

Similar to [8], we will also construct a directed hypergraph $H_M = (V_M, E_M)$, where V_M is a set of vertices and E_M is a set of directed hyperedges, based on a given undirected hypergraph. A directed hyperedge e is a pair (T, H), where T and H are both nonempty subsets of V_M. They respectively represent the *tail* and the *head* of e. Let $H' = (V', E')$ be an undirected hypergraph without parallel hyperedges and $M \subseteq E'$ be a matching. Two hyperedges are *parallel hyperedges* if one of the edges is a subset of the other.

The construction of H_M is as follows: For each vertex $v \in V'$, we introduce two vertices v^A and v^B. $V_M := \{v^A, v^B \mid v \in V'\} \cup \{s, t\}$, where $s, t \notin V'$ and $s \neq t$.
For all pairs of edges $e_i, e_j \in E'$ such that $e_i \cap e_j \neq \emptyset$, let $e^A = \{v^A \mid v \in e\}$ and $e^B = \{v^B \mid v \in e\}$. Create four directed hyperedges $\overrightarrow{e_i e_j}^1$, $\overrightarrow{e_i e_j}^2$, and $\overrightarrow{e_j e_i}^1$, $\overrightarrow{e_j e_i}^2$ in E_M as follows:

$$\overrightarrow{e_i e_j}^1 = \begin{cases} (e_i^A \setminus e_j^A, e_i^B \cap e_j^B), \text{ if } e_i \in M. \\ (e_i^B \setminus e_j^B, e_i^A \cap e_j^A), \text{ if } e_i \in E' \setminus M. \end{cases} \quad \overrightarrow{e_i e_j}^2 = \begin{cases} (e_i^A \cap e_j^A, e_i^B \setminus e_j^B), \text{ if } e_i \in M. \\ (e_i^B \cap e_j^B, e_i^A \setminus e_j^A), \text{ if } e_i \in E' \setminus M. \end{cases}$$

$$\overrightarrow{e_j e_i}^1 = \begin{cases} (e_j^A \setminus e_i^A, e_j^B \cap e_i^B), \text{ if } e_j \in M. \\ (e_j^B \setminus e_i^B, e_j^A \cap e_i^A), \text{ if } e_j \in E' \setminus M. \end{cases} \quad \overrightarrow{e_j e_i}^2 = \begin{cases} (e_j^A \cap e_i^A, e_j^B \setminus e_i^B), \text{ if } e_j \in M. \\ (e_j^B \cap e_i^B, e_j^A \setminus e_i^A), \text{ if } e_j \in E' \setminus M. \end{cases}$$

Finally, add to E_M, a directed edge from s to vertex v^B and a directed edge from v^A to t, for each free vertex $v \in V$ such that v belongs to an unmatched edge $e \in E'$ that is incident to at most one matched edge of H'. This can create $O(|E'|^2)$ hyperedges for E_M since we check every pair (e_i, e_j) in E'. We denote the A-vertices and B-vertices belong to an edge $e \in E_M$ by $A(e)$ and $B(e)$ respectively.

A directed hyperpath from u to v in H_M is a sequence of k hyperedges $(T_1, H_1), ..., (T_k, H_k) \in E_M$ satisfying $T_i \subseteq \bigcup_{j=0}^{i-1} H_j$ for all $i = 1, ..., k+1$, where $H_0 = \{u\}$ and $T_{k+1} = \{v\}$. The vertex v is said to be *reachable* from the vertex u in H_M if there exists

a directed hyperpath from u to v. A directed simple hyperpath $P = (T_1, H_1), ..., (T_k, H_k)$ is *elementary* if $T_i \subseteq H_{i-1}$ for $i = 2, ..., k$, where T_1 and H_k are an actual tail and head respectively in E_M. Our algorithm is based on this definition of elementary hyperpath to search simple hyperpaths in H_M.

A trivial directed hypergraph H_M is shown in Figure 5.7 (on the right). The vertex t is reachable from x^A through the directed hyperpath $P = (\{x^A\}, \{v^B, w^B\}), (\{v^B\}, \{u^A\}), (\{u^A\}, \{t\})$. This hyperpath P is also simple and elementary. On the other hand, w^A does not reach t. In fact, w^A does not reach any other vertex in H_M because there does not exist a tail that consists of only $\{w^A\}$.

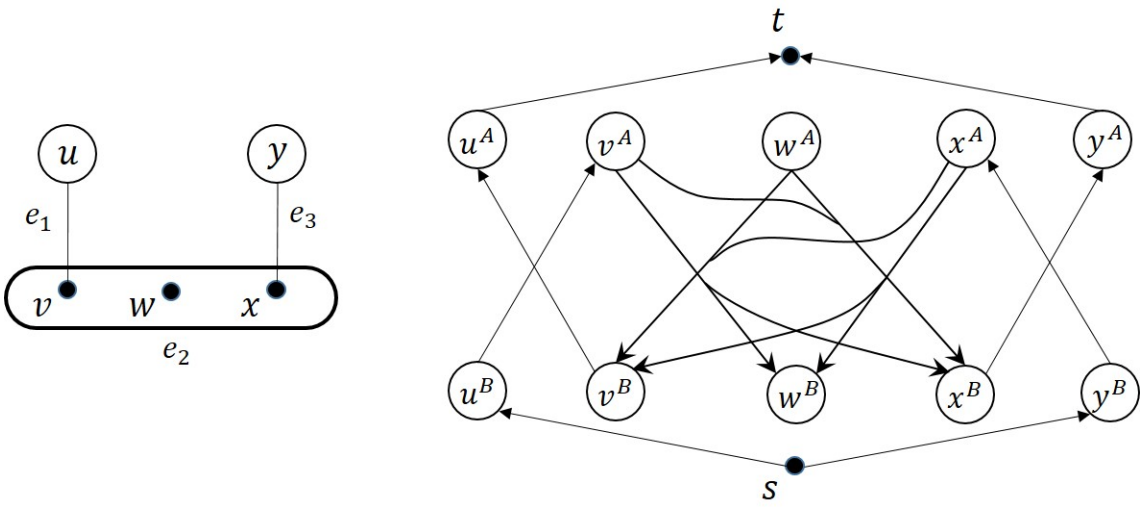


Figure 5.7: On the left is a very simple undirected hypergraph H', where e_1 and e_3 are unmatched, and e_2 is matched. The corresponding directed hypergraph H_M is shown on the right.

Remark Due to the Property 5.1.2 of a pick-up hypergraph H, if we construct H_M based on H, the hyperedges in H_M can be slightly simplified. Recall that for any two incident hyperedges e, e' in H:

$$|e \cap e'| = \begin{cases} 1 \text{ or } |e| - 1, & \text{if } |e| = |e'| \\ 1, & \text{otherwise} \end{cases}$$

As a result, for each edge $e = (T_i, H_i)$ in H_M, either $|T_i| = 1$ or $|H_i| = 1$, or both $|T_i| = |H_i| = 1$.

For simplicity, "directed" is dropped from directed hyperedge and directed hyperpath whenever there is no ambiguity. Also, we introduce two notations of a simple path $P = (T_1, H_1), ..., (T_k, H_k)$ in H_M. P can be represented with A/B vertices: $(T_1^A, H_1^B), (T_2^B, H_2^A), ..., (T_k^A, H_k^B)$, or simply $P = T_1^A, H_1^B, H_2^A, ..., H_k^B$ since $T_i \subseteq H_{i-1}$, for $i = 2, ..., k$. The path from V^A to U^B is sometimes denoted by $V^A \rightsquigarrow U^B$.

We have directed the edges in M from "A-vertices to B-vertices" and the edges in $E' \setminus M$ from "B-vertices to A-vertices". Since the distinct vertices v^A and v^B in V_M correspond to the same vertex v in V', we need to define strongly simple paths in H_M which cannot contain both vertices v^A and v^B at the same time, so that they correspond to unique paths in H'.
A (hyper)path P in H_M is *strongly simple* if

(a) P is simple, and

(b) $v^A \in P \Rightarrow v^B \notin P$, where v^A, $v^B \in V_M$.

We are now ready to extend the theorem in [8] from directed graphs to directed hypergraphs.

Theorem 5.5.1. *Let $H' = (V', E')$ be an undirected hypergraph without parallel hyperedges, M be a matching in H' and $H_M = (V_M, E_M)$ defined as above. Then there exists an augmenting path in H' w.r.t. M if and only if there exists a strongly simple path from s to t in H_M.*

Proof: ($\rightarrow$) Let $Q = (e_1, e_2, ..., e_l)$ be an augmenting path in H' w.r.t. M. Then Q is simple, i.e. $e_i \cap e_{j+1} = \emptyset$, for $1 \leq i < j < l$. e_1 and e_l are unmatched edges such that each is incident to one matched edge in H'. By the construction of H_M, there will be an edge e_s connecting s and a vertex in e_1 and an edge e_t connecting t and a vertex in e_l. Let (T_i, H_i) and (T_{i+1}, H_{i+1}) be two directed edges correspond to e_i and e_{i+1} in Q respectively. Assume without loss of generality, e_i is unmatched and e_{i+1} is matched. Then, we have (T_i^B, H_i^A) and (T_{i+1}^A, H_{i+1}^B).

By the construction of H_M, $e_i \cap e_{i+1}$ is used as the tail or head for two adjacent edges in E_M. This means that there exists two adjacent edges in E_M, say (T_i^B, H_i^A) and (T_{i+1}^A, H_{i+1}^B), such that $B(e_i) \cap H_{i+1}^B = \emptyset$ and $T_i^B \cap B(e_{i+1}) = \emptyset$. Similarly, the same property holds for A-vertices if e_i is matched and e_{i+1} is unmatched. Since Q is simple, $A(e_i) \cap A(e_{i+2}) = \emptyset$ and $B(e_i) \cap B(e_{i+2}) = \emptyset$. Hence, there exists a strongly simple path $Q' = (e_s, (T_1, H_1), ..., (T_l, H_l), e_t)$ in H_M.
($\leftarrow$) Let $P = (e_s, e_1, ..., e_k, e_t)$ be a strongly simple path in H_M. Then, $e_i \cap e_{j+1} = \emptyset$, for $1 \leq i < j < k$. e_1 and e_k are unmatched edges w.r.t. M such that each is incident to exactly one matched edge in H'. Hence, the undirected version of $P' = (e_1, e_2, ..., e_k)$ is an augmenting path in H'. □

5.6 Algorithm for the Reachability Problem

As shown by Theorem 5.5.1, finding an augmenting path in the undirected hypergraph H' is equivalent to finding a strongly simple in H_M (constructed base on H' and matching M in H'). By extending the *modified depth-first search* in [8], the proposed algorithm MDFS-hypergraph (MDFS for short) finds exactly the strongly simple paths in H_M. The MDFS

uses a stack K. TOP(K) denotes the last head (with the only exception of s) added to the MDFS-stack K. An operation POP(H^A) means the head H^A is removed from K, where H^A is the top element of K. An operation PUSH(H^A) means the head H^A is pushed onto K, which becomes the top element of K. In each step, MDFS considers an edge $e = (T_i^X, H_i^{\overline{X}})$ which has not been considered previously, where $T_i^X \subseteq TOP(K)$ and $X \in [A, B]$ (also $\overline{X}$ means *not* X). We distinguish two cases:

- Case 1: $X = A$, i.e. $e = (T_i^A, H_i^B)$ and the corresponding edge in H' is matched.

  1.1 No vertex of H_i^B belongs to a set $Z^B \in K$, but there exists at least one set $Y^A \in K$ containing vertices from H_i^A

  1.2 No vertex of H_i^B and H_i^A belongs to a set that is in K

  (i) H_i^B has not been in K, and no subset of H_i^B has been in K previously

  (ii) H_i^B has not been in K, but some subsets of H_i^B have been in K previously

  (iii) H_i^B has been in K previously

- Case 2: $X = B$, i.e. $e = (T_i^B, H_i^A)$ and the corresponding edge in H' is unmatched.

  2.1 There exists at least one set $Y^A \in K$ containing vertices from H_i^A

  2.2 No vertex of H_i^A belongs to a set $Y^A \in K$, but there exists at least one set $Z^B \in K$ containing vertices from H_i^B

  2.3 No vertex of H_i^A and H_i^B belongs to a set that is in K

  (i) H_i^A has not been in K, and no subset of H_i^A has been in K previously

  (ii) H_i^A has not been in K, but some subsets of H_i^A have been in K previously

  (iii) H_i^A has been in K previously

Note: cases 1.1 and 2.2 are symmetric; cases 1.2 and 2.3 are symmetric. There are only two sub-cases for Case 1 compare to Case 2. More precisely, there is no sub-case for some vertex of H_i^B is contained in a set $Z^B \in K$. This is due to a matched vertex can only belong to a unique matched edge. Let e be a matched edge in H', $e^B = \{v^B \mid v \in e\}$ and $e^A = \{v^A \mid v \in e\}$. Consider the following: Suppose the algorithm has visited an edge (V^A, Z^B) and pushed Z^B onto K (and still in K), where Z^B contains some vertices from e^B. The edge (V^A, Z^B) corresponds to a matched edge e' in H' because it is directed from A-vertices to B-vertices. As Z^B contains some vertices from e^B, it must be the case that $e' = e$ since a matched vertex can only belong to a unique matched edge. Hence, V^A must contain vertices from e^A such that $V^A \cup Z^B$ corresponds to all vertices in e. In order for MDFS to visit another head H_i^B containing some vertices from e^B (with $Z^B \in K$), it must first visit a tail T^A, where $(T^A, H_i^B) \in E_M$. For the same reasoning stated above, the edge (T^A, H_i^B) corresponds to the matched edge e, i.e. $T^A \cup H_i^B$ corresponds to the vertices of e. By Cases 2.1 and 2.2, T^A will never be added to K since either $V^A \cap T^A \neq \emptyset$ or

$Z^B \cap T^B \neq \emptyset$. Thus, H_i^B will never be considered while $Z^B \in K$. As a result, we will never have this sub-case in Case 1.

We say that MDFS *has found* a path $Q = (T_1^A, H_1^B), ..., (T_k^A, H_k^B)$ if and only if for $H_i^X \in Q$, $1 \leq i \leq k-1$, MDFS has performed the operation PUSH(H_i^X) and all edges on Q are considered. After performing the operation POP(H^A) from the top of stack K, MDFS always maintains a collection $L[V_i^A]$ of sets containing two types (A and B) of vertices, for each tail $V_i^A \subseteq H^A$ such that V_i^A has been used as a tail by MDFS. Each set Y^A of vertices in $L[V_i^A]$ is a head and must satisfy the requirements that:

1. MDFS has found a path Q from V_i^A to Y^A with no vertex from Y^A and Y^B belongs to a head or tail that is in Q (including V_i^A),

2. PUSH(Y^A) has never been performed, and at least one of the following must be satisfied (or both of them if can be applied)

3. POP(U^B) is performed for every head U^B containing at least one vertex from Y^B, where U^B has been pushed by MDFS. In other words, there does not exist a set $U^B \in K$ contains vertex from Y^B when Y^A is added to $L[V_i^A]$.

4. POP(U^A) is performed for every head U^A containing at least one vertex from Y^A, where U^A has been pushed by MDFS. In other words, there does not exist a set $U^A \in K$ contains vertex from Y^A when Y^A is added to $L[V_i^A]$.

Each set Z^B of vertices in $L[V_i^A]$ must satisfy the requirements that:

1. MDFS has found a path Q' from V_i^A to Z^B with no vertex from Z^A and Z^B belongs to a head or tail that is in Q' (including V_i^A),

2. PUSH(Z^B) has never been performed, and at least one of the following must be satisfied (or both of them if can be applied)

3. POP(W^A) is performed for every head W^A containing at least one vertex from Z^A, where W^A has been pushed by MDFS. In other words, there does not exist a set $W^A \in K$ contains vertex from Z^A when Z^B is added to $L[V_i^A]$.

4. POP(W^B) is performed for every head W^B containing at least one vertex from Z^B, where W^B has been pushed by MDFS. In other words, there does not exist a set $W^B \in K$ contains vertex from Z^B when Z^B is added to $L[V_i^A]$.

These requirements allow some subsets of Y^A and some subsets of Z^B to have been in K previously; this is specifically for Case 2.3.ii and Case 1.2.ii.

After performing the operation POP(H^B), MDFS also maintains a collection $L[V_i^B]$ of sets containing two types of vertices, for each tail $V_i^B \subseteq H^B$ such that V_i^B has been used as

a tail by MDFS. Each set of vertices in $L[V_i^B]$ is a head and follows the exact requirements stated above. The updating of $L[V_i^B]$ is identical as well.

Let V^A be a head and $H^X \in L[V_i^A]$, for some tail $V_i^A \subseteq V^A$ and $X \in [A, B]$. In the original MDFS algorithm [8], the path Q from V_i^A to H^X is kept implicitly after performing PUSH(H^X). For the MDFS-hypergraph algorithm, this path needs to be stored explicitly to make sure that the strongly simple property holds throughout the algorithm. We introduce a secondary Stack K_S, which keeps track of the simple path Q from V_i^A to H^X when a head $H^X \in L[V_i^A]$ is pushed onto the stack K. In Case 1.2.iii and Case 2.3.iii, the algorithm visits a head Y^A which has been in K previously. Instead of pushing Y^A onto K again, MDFS pushes a head $H^X \in L[V_i^A]$ onto K, where $V_i^A \subseteq Y^A$ is a tail. Such an operation is identical to original MDFS algorithm. Since only $H^X \in K$ and the path Q from V_i^A to H^X is not on K, we use K_S to store Q. More precisely, when the operation PUSH(H^X) is performed, Q is also pushed onto K_S. When H^X is popped from K, Q is popped from K_S as well. Again, the same process applies to $L[V_i^B]$ for some tail V_i^B.

As a result, the two cases need to be modified accordingly as the following:

- Case 1: $X = A$, i.e. $e = (T_i^A, H_i^B)$ and the corresponding edge in H' is matched.

  1.1 No vertex of H_i^B belongs to a set $Z^B \in K \cup K_S$, but there exists at least one set $Y^A \in K \cup K_S$ containing vertices from H_i^A
  - (i) $Y^A \in K$
  - (ii) $Y^A \in K_S$

  1.2 No vertex of H_i^B and H_i^A belongs to a set that is in $K \cup K_S$
  - (i) H_i^B has not been in K, and no subset of H_i^B has been in K previously
  - (ii) H_i^B has not been in K, but some subsets of H_i^B have been in K previously
  - (iii) H_i^B has been in K previously

- Case 2: $X = B$, i.e. $e = (T_i^B, H_i^A)$ and the corresponding edge in H' is unmatched.

  2.1 There exists at least one set $Y^A \in K \cup K_S$ containing vertices from H_i^A
  - (i) $Y^A \in K$
  - (ii) $Y^A \in K_S$

  2.2 No vertex of H_i^A belongs to a set $Y^A \in K \cup K_S$, but there exists at least one set $Z^B \in K \cup K_S$ containing vertices from H_i^B
  - (i) $Z^B \in K$
  - (ii) $Z^B \in K_S$

  2.3 No vertex of H_i^A and H_i^B belongs to a set that is in $K \cup K_S$
  - (i) H_i^A has not been in K, and no subset of H_i^A has been in K previously
  - (ii) H_i^A has not been in K, but some subsets of H_i^A have been in K previously

(iii) H_i^A has been in K previously

Overview of how to update $L[V_i^A]$, for any tail V_i^A, i.e. when a head Y^X , $X \in [A, B]$, should be added to and removed from $L[V_i^A]$ (similarly for B-vertices tail $L[V_i^B]$).

- Initially, $L[V_i^A] := \emptyset$.
- After an operation POP(U^B) or POP(U^A), where U^B or U^A contains vertices from Y^B or Y^A respectively, check to see if there exists another set in $K \cup K_S$ containing vertices from Y^B or Y^A. If such a set does not exist in $K \cup K_S$, add Y^X to $L[V_i^A]$ when the followings are satisfied: MDFS has found a path from V_i^A to Y^X with no vertex of Y^A and Y^B belongs to any edge on the path, and PUSH(Y^X) has never been performed.
- After PUSH(Y^A), remove the head Y^A from $L[V_i^A]$, for all $L[V_i^A]$ containing Y^A.

We need one more definition before presenting the algorithm. Let $P = (T_1^X, H_1^{\overline{X}}), (T_2^{\overline{X}}, H_2^X), ..., (T_k^X, H_k^{\overline{X}})$ be a path in H_M, where $X \in [A, B]$. We say that the path P does not *intersect* $K \cup K_S$ if the following is satisfied:

For every set V^{X_1} of $T_i^X, H_i^X, 1 \leq i \leq k$:

$\forall\ W^{X_2} \in K \cup K_S$ such that $(V^{X_1} \cup V^{\overline{X_1}}) \cap (W^{X_2} \cup W^{\overline{X_2}}) = \emptyset$, where $X_1, X_2 \in [A, B]$

The MDFS algorithm for hypergraphs is presented in Algorithm 5. We assume that $L[V_i^X]$, for every tail $V_i^X \in E_M$, $X \in [A, B]$, is *correctly computed*. The correctness proof of MDFS is also based on this assumption. The construction of $L[V_i^X]$ is described in section 5.7.1. When a head or tail is marked "pushed", it means that it has been in K. Some head and tail can contain the exact same set of vertices, so marking one of them "pushed" means both are marked "pushed".

5.7 Analysis

For the sake of completeness, we shall prove the correctness of MDFS for hypergraph H_M. The correctness proof of MDFS-Hypergraph (MDFS for short) closely follows the correctness proof in [8]. Similar to depth-first search, we also use the notion of the current search path in our correctness proof for MDFS. First note that at any execution point of MDFS, the elements in $K \cup K_S$ represent a single path from s to TOP(K). It is obvious that the elements in K represent a single path from s to TOP(K) if K_S is empty. When MDFS is in either Case 1.2.iii or Case 2.3.iii, a head H_i^X, $X \in [A, B]$, is pushed onto K, but the actual path Q_i connecting H_i^X and the last head $H_{i-1}^{\overline{X}}$ added to K is pushed onto K_S. Thus, the

Algorithm 5 MDFS-hypergraph

Input: $H_M = (V_M, E_M)$.
Output: An augmenting path P in H_M if it exists.
Let K be a stack; PUSH(s);
$SEARCH$

procedure SEARCH
$h := TOP(K)$;
if $h = t$ /* sink t */
    construct the augmenting path P which is found by the algorithm
else
    $T_K := \{T_i \mid T_i \subseteq h$ and T_i is a tail, where T_i is not marked "pushed"$\}$
    mark h "pushed";
    for every $T_i \in T_K$, mark T_i "pushed", and **do**
        for every head H_i^X such that $\{T_i, H_i^X\} \in E_M$ **do**
            Case 1: $X = B$
                Case 1.1: $\exists\ Y^A \in K \cup K_S$ such that $Y^A \cap H_i^A \neq \emptyset$
                    Case 1.1.i: $Y^A \in K$
                        no PUSH-operation is performed
                    Case 1.1.ii: $Y^A \in K_S$
                        for every path $Q \in K_S$ from tail V^{X_1} to head U^{X_2} containing every such Y^A, where $U^{X_2} \in L[V^{X_1}]$ and $X_1, X_2 \in [A, B]$, replace Q with a strongly simple path Q' not containing any Y^A if Q' exists, and **Goto** Case 1.2. Otherwise, no PUSH-operation is performed.
                end of Case 1.1
                Case 1.2:
                    Case 1.2.i: (H_i^B is not marked "pushed") and ($\nexists\ V_i^B \subseteq H_i^B$ marked "pushed")
                        PUSH(H_i^B); SEARCH;
                    Case 1.2.ii: (H_i^B is not marked "pushed") and ($\exists\ V_i^B \subseteq H_i^B$ marked "pushed")
                        PUSH(H_i^B); SEARCH; **Goto** Case 1.2.iii;
                    Case 1.2.iii: (H_i^B is marked "pushed") or ($\exists\ V_i^B \subseteq H_i^B$ marked "pushed")
                        for every pushed tail $V_i^B \subseteq H_i^B$ **do**
                            while $\exists\ H^X \in L[V_i^B]$ such that $V_i^B \rightsquigarrow H^X$ does not intersect $K \cup K_S$ **do**
                                Push the path from V_i^B to H^X onto K_S; PUSH(H^X); SEARCH;
                            end while
                        end for
                end of Case 1.2
            end of Case 1
            Case 2: $X = A$
                Case 2.1: $\exists\ Y^A \in K \cup K_S$ such that $Y^A \cap H_i^A \neq \emptyset$
                    Case 2.1.i: $Y^A \in K$
                        no PUSH-operation is performed
                    Case 2.1.ii: $Y^A \in K_S$
                        for every path $Q \in K_S$ from tail V^{X_1} to head U^{X_2} containing every such Y^A, where $U^{X_2} \in L[V^{X_1}]$ and $X_1, X_2 \in [A, B]$, replace Q with a strongly simple path Q' not containing any Y^A if Q' exists, and **Goto** Case 2.3. Otherwise, no PUSH-operation is performed.
                end of Case 2.1
                Case 2.2: $\exists\ Z^B \in K \cup K_S$ such that $Z^B \cap H_i^B \neq \emptyset$
                    Case 2.2.i: $Z^B \in K$
                        no PUSH-operation is performed
                    Case 2.2.ii: $Z^B \in K_S$

for every path $Q \in K_S$ from tail V^{X_1} to head U^{X_2} containing every such Z^B, where $U^{X_2} \in L[V^{X_1}]$ and $X_1, X_2 \in [A, B]$, replace Q with a strongly simple path Q' not containing any Z^B if Q' exists, and **Goto** Case 2.3. Otherwise, no PUSH-operation is performed.
end of Case 2.2
Case 2.3:
Case 2.3.i: (H_i^A is not marked "pushed") and ($\nexists\ V_i^A \subseteq H_i^A$ marked "pushed")
PUSH(H_i^A); SEARCH;
Case 2.3.ii: (H_i^A is not marked "pushed") and ($\exists\ V_i^A \subseteq H_i^A$ marked "pushed")
PUSH(H_i^A); SEARCH; **Goto** Case 2.3.iii;
Case 2.3.iii: (H_i^A is marked "pushed") or ($\exists\ V_i^A \subseteq H_i^A$ marked "pushed")
for every pushed tail $V_i^A \subseteq H_i^A$ **do**
while $\exists\ H^X \in L[V_i^A]$ such that $V_i^A \rightsquigarrow H^X$ does not intersect $K \cup K_S$ **do**
Push the path from V_i^A to H^X onto K_S; PUSH(H^X); SEARCH;
end while
end for
end of Case 2.3
end of Case 2
end for
POP(K_S) if h was added under Case 1.2.iii or Case 2.3.iii;
POP(K);
if $\exists$ a head H^X has not been pushed but considered s.t. H^X does not intersect $K \cup K_S$
Add H^X to $L[V_i^A]$, for all tails V_i^A such that a strongly simple path from V_i^A to H^X has been found.
end if
end for
end if
end procedure

path from s to $H_{i-1}^{\overline{X}}$, represented by $K \cup K_S$, is extended to the path $s, ..., H_{i-1}^{\overline{X}}, Q, H_i^X$. MDFS continues the search from H_i^X, which can encounter Case 1.2.iii or Case 2.3.iii again later. In either case, the path from s to H_{j-1}^X, $j > i$, is extended by another path Q_k, H_k^X. The path from s to H_k^X remains as a single path. When the top head $H_p^X \in K$ is popped, where the path Q_p connecting H_p^X and the last head $H_{p-1}^{\overline{X}}$ added to K is in K_S, the path Q_P is immediately popped from K_S as well. This results in a shorter path, i.e. the path, represented by $K \cup K_S$, is now from s to $H_{p-1}^{\overline{X}}$, which is still a single path. The search continues from $H_{p-1}^{\overline{X}}$. We call this single path represented by the elements in $K \cup K_S$ *a search path* w.r.t. that execution point. In particular, we call the single path the *current search path* w.r.t the current execution point. When we say *a previous (an earlier) search path*, we mean a search path w.r.t. any execution point prior to the current one.

For the first lemma, it shows that if there exists a strongly simple path P in H_M from U^{X_1} to W^{X_2}, $X_1, X_2 \in [A, B]$, where P has not been considered, W^{X_2} will be considered before POP(U^{X_1}) is performed.

Lemma 5.7.1. *Let $U^B \in E_M$ be a head for which MDFS performs the operation PUSH(U^B) and $U_i^B \subseteq U^B$ be a tail that is not marked "pushed". Suppose at the moment when the operation PUSH(U^B) is performed, there exists a strongly simple path P: $U_i^B = T_1^B, H_1^A, H_2^B, ...,$*

$H^B_{t-1}, H^A_t = W^A$ *such that* $\nexists\ Z^X, Z^{\overline{X}} \in K \cup K_S$ *containing vertices from* H^X_i *and* $H^{\overline{X}}_i$*, for all* $H^X_i \in P$*. Then PUSH*(W^A) *is performed before the execution of the operation POP*(U^B)*.*

Proof: Assume, to the contrary, that PUSH(W^A) is performed after POP(U^B) for the strongly simple path P. After the execution of PUSH(U^B), the edge $e = (T^B_1, H^A_1)$ must be considered before POP(U^B) by MDFS since T^B_1 is not marked pushed. Suppose PUSH(H^A_1) is performed next, then PUSH(H^A_2) would be performed after, and so on due to P is strongly simple. However, at some point, PUSH(H^X_r), $1 \leq r \leq t$ and $X \in [A, B]$, must not perform by our assumption, and MDFS starts to backtrack until POP(U^B) is performed so that it is performed before PUSH(W^A). Let H^X_r be the first head on P such that PUSH(H^X_r) has never been performed just before POP(U^B) is performed.

Suppose $X = A$. Let us consider the moment when MDFS has pushed the path $P_1 = T^B_1, H^A_1, ..., H^B_{r-1}$ and PUSH(H^A_r) is not performed next where this is the last time H^A_r is considered before POP(U^B). In other words, the edges on the subpath $T^B_1, H^A_1, ..., H^B_{r-1}$ of the strongly simple path P have been considered and pushed, and P_1 is a collection of sub-path(s) of the current search path. Do not confuse P_1 with P; P_1 may not contain any head in P. There are three cases for which PUSH(H^A_r) is not performed, namely MDFS is in Case 2.1, Case 2.2, or the tail T^B_r has already been marked "pushed" before H^B_{r-1} is pushed; in other words, PUSH(H^B_{r-1}) was performed under Case 1.2.ii.

Case 2.1 & Case 2.2 There exists sets Y^A or $Z^B \in K \cup K_S$ containing vertices from H^A_r or H^B_r respectively. If all Y^A and $Z^B \in K$, P_1 can be $T^B_1, H^A_1, ..., H^B_j, Y^A, ..., Z^B, H^A_k, ..., H^B_{r-1}$ (similarly if some set Z^B comes before Y^A). MDFS starts to backtrack. When it backtracks to H^B_j (a head on P), every set $Y^A, Z^B \in K$ has been popped. The edge (T^B_{j+1}, H^A_{j+1}) that is on the strongly simple path P would be considered before H^B_j is popped if T^B_{j+1} is not marked "pushed". Assume it is the case. Under our consideration, the head H^A_{j+1} has been pushed. In fact, the edges on the strongly simple path from H^A_{j+1} to H^A_{r-1} have been considered and pushed. Hence, $H^A_r \in L[T^A_{j+2}]$, where $T^A_{j+2} \subseteq H^A_{j+1}$ is a tail, since the path from T^A_{j+2} to H^A_r has been found by MDFS. As a result, PUSH(H^A_r) is performed next, a contradiction.

Suppose T^B_{j+1} is marked "pushed". It means that the edge (T^B_{j+1}, H^A_{j+1}) has been considered earlier by a previous search path, i.e. H^A_{j+1} is pushed and popped before the current search path, and PUSH(H^B_j) was performed under Case 1.2.ii. Because of this, the edge (T^B_{j+1}, H^A_{j+1}) is not considered during the current search. MDFS performs the operation POP(H^B_j), and moves to Case 1.2.iii. The head H^B_j is treated as it has been pushed previously. Thus, MDFS checks each tail that is a subset of H^B_j. In this case, MDFS finds T^B_{j+1} such that $H^A_r \in L[T^B_{j+1}]$ since a strongly simple path from T^B_{j+1} to H^A_r has been found under our consideration. As a result, PUSH(H^A_r) is performed next, a contradiction. Note: if some set Z^B comes before Y^A in P_1, MDFS would have been in Case 2.3.ii and Case 2.3.iii, which is symmetric to Case 1.2.ii and Case 1.2.iii.

For the case where some sets Y^A and $Z^B \in K_S$. Let $Q \in K_S$ be a path containing Y^A and Z^B. By the assumption of the lemma, $\nexists\, Y^A, Z^B \in K \cup K_S$ containing vertices from H_r^A and H_r^B at the moment PUSH(U^B) is performed. This means that the path Q from tail V^{X_1} to head U^{X_2}, where $U^{X_2} \in L[V^{X_1}]$ and $X_1, X_2 \in [A, B]$, is added to K_S after PUSH(U^B). Suppose Q is not on P_1, i.e. Q is on the current search path but comes before P_1. If P_1 is a subpath of the strongly simple path P, MDFS can replace Q with another path Q' from V^{X_1} to head U^{X_2} not containing Y^A and Z^B by the assumption that P is strongly simple path at the moment PUSH(U^B) is performed. If P_1 is not a subpath of P and MDFS cannot replace Q, then MDFS starts to backtrack. Eventually, it will backtrack to a head H_j^X, $X \in [A, B]$, such that $T_1^B, H_1^A, ..., H_j^X$ is a subpath of P. Then, by the same reason stated above, $H_r^A \in L[T_{j+1}^X]$. MDFS picks the strongly simple path P' from T_{j+1}^X to H_r^A such that Q can be replaced with another path Q', where Q' does not intersect with P, and pushes P' and H_r^A, which leads to a contradiction.

Suppose Q is on P_1 implicitly, i.e. $P_1 = T_1^B, H_1^A, ..., H_j^X, Q, H_k^X, ..., H_{r-1}^B$, where T_1^B, H_1^A, $..., H_j^X$ and $H_k^X, ..., H_{r-1}^B$ belong to K, and Q belongs to K_S. If $T_1^B, H_1^A, ..., H_j^X$ and $H_k^X, ..., H_{r-1}^B$ are two subpaths of P, then there exists a strongly simple path Q' from T_{j+1}^X to H_k^X. MDFS can replace Q with Q' due to our consideration that the edges on Q' have been considered and pushed. In fact, this replacement can be applied to all paths in K_S that are on P_1. More precisely, if all heads on P_1 that are in K also belong to P, then MDFS can apply the replacement for each path in K_S that is on P_1 so that paths in K_S will be subpaths of P. Thus, there is not any set in K_S containing H_r^A and H_r^B. If $T_1^B, H_1^A, ..., H_j^X$ or $H_k^X, ..., H_{r-1}^B$ is not a subpath of P and MDFS cannot replace Q, then MDFS starts to backtrack. Eventually, it will backtrack to a head H_p^X, $p < j$, such that $T_1^B, H_1^A, ..., H_p^X$ is a subpath of P. Then, by the same reason stated above, $H_r^A \in L[T_{p+1}^X]$. Hence, either PUSH(H_r^A) can be performed next, or Y^A and $Z^B \in K$ which is the above case, and both of them result in the contradiction that PUSH(H_r^A) is not performed.

Case 1.2.ii The tail T_r^B has already been marked "pushed" before H_{r-1}^B is pushed. This is very similar to one of the above sub-cases. It means that the edge (T_r^B, H_r^A) has been considered earlier by another search path. Then MDFS performs the operation POP(H_{r-1}^B), and moves to Case 1.2.iii. The head H_{r-1}^B is treated as it has been pushed previously. Thus, MDFS checks each tail that is a subset of H_{r-1}^B. Although $H_r^A \in L[T_r^B]$ under our consideration, where $T_r^B \subseteq H_{r-1}^B$, MDFS can only perform PUSH(H_r^A) if H_r^A does not intersect the current search path P_1, which implies that P_1 is a subpath of P. Otherwise, MDFS starts to backtrack. Eventually, it will backtrack to a head H_j^X, $X \in [A, B]$, such that $T_1^B, H_1^A, ..., H_j^X$ is a subpath of P and $H_r^A \in L[T_{j+1}^X]$. Hence, PUSH(H_r^A) can be performed next, a contradiction.

For $X = B$, MDFS is in Case 1.1 or Case 2.3.ii, which is symmetric to Case 2.2 and Case 1.2.ii respectively. □

Remark 5.7.1. *The starting vertices and ending vertices of P in Lemma 5.7.1 can be A-vertices or B-vertices, and Lemma 5.7.1 still holds.*

The following lemma is important for the MDFS algorithm. Any search path P made by MDFS continues to grow as long as P can be kept strongly simple.

Lemma 5.7.2. *MDFS constructs only strongly simple paths.*

Proof: (By induction on the number of PUSH operations). Note that only a PUSH operation can destroy the strongly simple property.

Base case: This is trivial as the first PUSH operation PUSH(s) cannot destroy the strongly simple property.

Inductive step: Suppose that all m - 1 numbers of PUSH operations do not destroy the strongly simple property. Let us consider the m^{th} PUSH operation. A PUSH operation occurs in Case 1.2 or Case 2.3. Because these two cases are completely symmetric, we only need to consider one of them. Let us consider that the m^{th} PUSH operation occurs in Case 2.3, i.e. the edge being considered is $e = (T_i^B, H_i^A)$. Recall that if MDFS is in Case 2.3, no vertex of H_i^A and H_i^B belongs to a set that is in $K \cup K_S$. (In fact, it is the same for Case 1.2)

Case 2.3.i and Case 2.3.ii For both cases, the operation PUSH(H_i^A) is performed. In order for this PUSH operation to destroy the strongly simple property, there must exist at least one set Y^A or Z^B containing vertices from H_i^B or H_i^A respectively, where Y^A or Z^B belongs to the current search path $K \cup K_S$. However, such a set Y^A or Z^B cannot exist. Otherwise, the algorithm MDFS would not have pushed H_i^A in the first place.

Case 2.3.iii Since H_i^A has been in K previously (mark pushed), the operation PUSH(H_i^A) is not performed. Instead, a head $U^X \in L[V_i^A]$ is pushed, where $X \in [A, B]$ and $V_i^A \subseteq H_i^A$ is a tail in E_M. Hence, the current search paths is extended by a path $P = V_i^A, Q, U^X$, but only U^X is pushed onto the stack K; H_i^A, Q is pushed onto K_S. Similarly, this PUSH operation destroys the strongly simple property only if there exists a set $Y^X \in P$ such that Y^X or $Y^{\overline{X}}$ intersects with $K \cup K_S$. In order for MDFS to reach Case 2.3, H_i^A (and V_i^A) do not intersect with $K \cup K_S$. Also, the algorithm only chooses a path P such that P does not intersect with $K \cup K_S$. □

The correctness of the algorithm can be derived from Lemma 5.7.1 and Lemma 5.7.2.

Theorem 5.7.1. *MDFS finds a strongly simple path from s to t if such a path exists.*

Proof: Suppose there is a strongly simple path $P : s = T_1, H_1^B, T_2^B, H_2^A, ..., T_{k-1}^B, H_{k-1}^A, T_k^A, H_k = t$ in H_M. It is clear that the edge (T_1, H_1^B) is considered by MDFS, and the operation PUSH(H_1^B) is performed. Thus, H_1^B and H_{k-1}^A fulfill the assumptions of Lemma 5.7.1 with respect to the path $H_1^B, T_2^B, H_2^A, ..., T_{k-1}^B, H_{k-1}^A$. By Lemma 5.7.1, MDFS performs PUSH(H_{k-1}^A) and hence, PUSH(t) is performed. Thus, MDFS finds a path from s to t. By Lemma 5.7.2, MDFS constructs only strongly simple paths. □

5.7.1 Subtle Details of the MDFS-hypergraph Algorithm

In this section, we will outline how to construct $L[V_i^X]$, for some tail V_i^X where $X \in [A, B]$, and how to retrieve the strongly simple path once it is discovered. In [8], a sketch of an efficient implementation of the original algorithm MDFS for graphs is provided. Such an efficient implementation requires two properties, but these two properties do not hold in MDFS-hypergraph (it may be possible to modify the MDFS-hypergraph algorithm so that a more efficient implementation is possible). Thus, we will describe a (not as efficient) method for constructing $L[V_i^X]$, including how to find the strongly simple path from V_i^X to a head $H^{[A,B]} \in L[V_i^X]$. For simplicity, X always refer to either the A-vertex or B-vertex.

The way of adding a head U^A containing A-vertices and a head W^B containing B-vertices to $L[V_i^X]$ is identical, so we only describe when to add U^A to $L[V_i^X]$ and how to find the path from V_i^X to U^A. Suppose a head U^A cannot be pushed onto K because there are some sets Y^A or Z^B in $K \cup K_S$ containing some vertices from U^A or U^B respectively (Cases 2.1 and 2.2). MDFS starts to backtrack by popping the top element, when all sets Y^A and Z^B are popped from $K \cup K_S$, we need to find all tails V_i^X where MDFS has found a simple path from V_i^X to U^A.

Finding all tails V_i^X can be accomplished by running a backward depth-first search for the simplest case. Start at U^A, and search the already considered edges (marked "pushed") backward, i.e. from head to tail instead, then consider each head marked pushed that is a super set of the just visited tail. Label each searched edge "backward pushed". When a set Y^A (Y^B) or Z^B (Z^A) that intersects with any vertex in the current search path is reached, this search path should not be extended any longer. Otherwise, it will not result in a strongly simple path. The backward search terminates when it finds all the edges of all strongly simple paths between U^A and each tail V_i^X .This may require to search every pushed edge. As a result, this backward search produces a subgraph G_{U^A} of H_M induced by all "backward pushed" edges, which consists of a sink U^A and multiple sources (tails) such that there exists a strongly simple path from each tail to U^A in G_{U^A}. Hence, we can add U^A to $L[V_i^X]$, for every tail V_i^X in G_{U^A}. We call such a graph G_{U^A} a *backward search graph*.

In the general case, we have to run a backward MDFS starting at U^A, and search the "pushed" edges backward. The purpose of the backward MDFS is to build a graph consists of all pushed tails that can reach U^A through a strongly simple path, whereas the purpose of the regular MDFS is to find a strongly simple path from the source to the destination. If we only run a regular backward depth-first search, some tails may not be strongly simple connected to U^A. Hence, we run a backward MDFS to construct a backward search graph G_{U^A} consists of the sink U^A.

Let us now explain how to find a different strongly simple path for Cases 2.1.ii and 2.2.ii. Suppose that the head H^A being considered by the algorithm intersects with at least one

set Y^A that belongs to a path $Q \in K_S$. MDFS tries to find another strongly simple path Q' not containing Y^A to replace the path Q containing Y^A (similarly for case 1.1.ii). Let $Q = V_i^X, ..., Y^A, ..., U^A$, where $U^A \in L[V_i^X]$, and G_{U^A} be the backward search graph with sink U^A. Finding a different strongly simple path Q' connecting V_i^X and U^A can be done using G_{U^A}. Remove every edge from G_{U^A} that contains vertex from H^A or H^B. If there still exists a path Q' connecting V_i^X and U^A in the resulting graph $G^R_{U^A}$, then replace Q with Q'. In other words, those removed edges do not form an edge-cut for V_i^X and U^A. The new current search path can continue from H^A, and $G^R_{U^A}$ will be used temporarily until H^A is popped. If H^A intersects with multiple paths in K_S. We can do the same for all backward search graphs one by one to find that many different strongly simple paths.

There is a more complicated case (which can be encountered by Case 2.3.iii and Case 1.2.iii as well). Consider the following: Let $Q_1, Q_2, ..., Q_k$ be paths in K_S and G_{H^X} be the backward search graph with the sink H^X. Suppose that a strongly simple path P from V_i^A to H^X cannot be found in G_{H^X} because every such path P intersects with a path Q_i in K_S, $i \in [1, ..., k]$. However, it may be possible that Q_i can be replaced with another strongly simple path Q_i' so that P can be found in G_{H^X}, but another strongly simple path Q_j in K_S needs to be replaced by another Q_j' first so that Q_i' and P can be found. This can continue, thus we need to consider all paths in K_S at once in general.

Let the current search path P, represented by elements in $K \cup K_S$, be the following: $s = T_1, H_1^B, ..., H_{c-1}^X, Q_1, H_c^A, ..., H_{d-1}^X, Q_2, H_d^A, ..., Q_k, H_l^A, ..., H_{p-1}^B$, where every Q_i, $1 \leq i \leq k$, is a path in K_S. Note that the head comes after each Q_i does not need to be A-vertices. It is made this way for simplicity. The MDFS algorithm is considering the next edge (T_p^B, H_p^A) where H_p^A (or H_p^B) intersects with paths in K_S only. We want to find a strongly simple path from s to H_p^A. Let $\mathcal{H}$ be the graph induced by the subpaths $(H_1^B, ..., H_{c-1}^A) \cup (H_c^X, ..., H_{d-1}^A) \cup ... \cup (H_l^A, ..., H_{p-1}^B, H_p^A)$ and the backward search graphs $G_{Q_1} \cup G_{Q_2} \cup ... \cup G_{Q_k}$. Then, run MDFS on $\mathcal{H}$. If it can return a strongly simple path P' from s to H_p^A, then replace the elements in $K \cup K_S$ according to P', and continue the search from H_p^A (on H_M).

To save computation, we can store every backward search graphs constructed during the call to MDFS($\mathcal{H}$) for further search.

Time complexity The time complexity of Algorithm MDFS-hypergraph depends on the implementation of backward search graphs and finding a different strongly simple path using the backward search graphs. It should be pointed out that finding a different strongly simple path using the backward search graphs may still be exponential in the numbers of the edges of the constructed directed hypergraph H_M. It may be possible to reduce the running time of Algorithm MDFS-hypergraph. We will leave this as a future work:

Remark 5.7.2. *Does there exist a polynomial time algorithm in the size of the directed hypergraph H_M for finding a strongly simple path from s to t in H_M?*

Chapter 6

Conclusion and Future Work

In this thesis, we studied the simplified ridesharing problem for the following variants:

(I) Constraints (C1) and (C3) are satisfied but (C2) detour is allowed.

(II) Constraints (C2) and (C3) are satisfied but (C1) trips can have distinct destination.

(III) Constraints (C1) and (C2) are satisfied but (C3) trips can have multiple preferred paths.

(IV) All constraints (C1), (C2) and (C3) are satisfied.

(V) Constraints (C2) and (C3) are satisfied but (C1) trips can have distinct destination and drivers are not allowed to perform re-take.

We first proved variants (I), (II) and (III) are NP-hard for minimizing the number of drivers and the total travel distance of the drivers. These results imply that if one of constraints C1, C2 and C3 is not satisfied then the minimization problems are NP-hard. Next, we provided a polynomial time exact algorithm (Algorithm RFP) for a special case of variant (IV). In this special case, we restrict the unique preferred paths of all trips lie on a same path of the road network. It is easy to see that Algorithm RFP has a polynomial time running in the size of the input. We showed that Algorithm RFP can find a solution to variant (IV) where the number of drivers is minimized.

Variant (IV) is a special case of variant (V) since satisfying C3 means re-take operations are not involved. When re-take is not allowed, the NP-hard reduction for variant (II) cannot be applied to variant (V). We then show that variant (V) can be solved in polynomial time, from a very simple case to the general case of this variant. When solving the general case, the algorithm and its analysis for this case lead to a novel approach for finding a maximum matching in hypergraphs with specific properties.

In order to prove that a matching in a hypergraph is maximum, we extend the well-known maximum matching theorem for graphs to hypergraphs (which we call the maximum

hyper-matching theorem). In particular, we define a graph structure called augmenting graph, which generalizes the idea of augmenting path for graphs to hypergraphs. The algorithm (Algorithm MMH) we proposed for finding a maximum matching in the special hypergraph (pick-up relation hypergraph) relies on the maximum hyper-matching theorem. Algorithm MMH finds a more restricted augmenting graph in each iteration. The correctness of Algorithm MMH is based on the conjecture (Conjecture 5.4.2) that an augmenting graph can be decomposed into a set of edge-disjoint restricted augmenting graphs. Therefore, it will be worthwhile to prove Conjecture 5.4.2 and complete the correctness proof for Algorithm MMH as a future work.

Although variants (I), (II) and (III) are NP-hard, it would be interesting to know whether good approximation algorithms can be made base on Algorithm RFP for these three variants. These approximation algorithms and Algorithm RFP maybe also useful for the generic ridesharing algorithm (where time constraints are considered). We only gave an algorithm to each of the variants (IV) and (V) where the number of drivers are minimized. It should be possible to have a polynomial time exact algorithm for minimizing the total travel distances of the drivers as well for each variant.

The method for solving variant (V) is a byproduct of finding a maximum matching in pick-up hypergraphs. It may be possible to solving variant (V) directly using a simpler algorithm. Another interesting idea is to come up with a better running time algorithm compared to Algorithm MMH for finding a maximum matching in pick-up hypergraphs. A more interesting but challenging direction is to find out what kinds of hypergraphs or properties a hypergraph must have such that a maximum matching in these hypergraphs can be found in polynomial time.

Bibliography

[1] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. Dynamic ride-sharing: A simulation study in metro atlanta. *Transportation Research Part B*, 45(9):1450–1464, 2011.

[2] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223:295–303, 2012.

[3] C. Annamalai. Finding perfect matchings in bipartite hypergraphs. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1823, 2016.

[4] K. Antoon, J. K. Lenstra, C. Papadimitriou, and F. Spieksma. Interval scheduling: A survey. *Naval Research Logistics*, 54(5):530–543, 2007.

[5] R. Baldacci, V. Maniezzo, and A. Mingozzi. An exact method for the car pooling problem based on lagrangean column generation. *Operations Research*, 52(3):422–439, 2004.

[6] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842–844, 1957.

[7] C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[8] N. Blum. A new approach to maximum matching in general graphs. In *Automata, Languages and Programming*, pages 586–597, 1990.

[9] P. Brucker and L. Nordmann. The k-track assignment problem. *Computing*, 54:97–122, 1994.

[10] N. D. Chan and S. A. Shaheen. Ridesharing in north america: Past, present, and future. *Transport Reviews*, 32(1):93–112, 2012.

[11] G. Chartrand and O. R. Oellermann. *Applied and Algorithmic Graph Theory*. Mcgraw-Hill, 1993.

[12] J.-F. Cordeau and G. Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007.

[13] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[14] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, 1987.

[15] M. Furuhata, M. Dessouky, F. Ordóñez, M. Brunet, X. Wang, and S. Koenig. Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological*, 57:28–46, 2013.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] K. Ghoseiri, A. Haghani, and M. Hamedi. Real-time rideshare matching problem. Final Report of UMD-2009-05, U.S. Department of Transportation, 2011.

[18] P.E. Haxell. A condition for matchability in hypergraphs. *Graphs and Combinatorics*, 11(3):245–248, 1995.

[19] W. Herbawi and M. Weber. The ridematching problem with time windows in dynamic ridesharing: A model and a genetic algorithm. *In: Proceedings of ACM Genetic and Evolutionary Computation Conference (GECCO)*, pages 1–8, 2012.

[20] Y. Huang, F. Bastani, R. Jin, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment*, 7(14):2017–2028, 2014.

[21] T. Kameda and I. Munro. A O(|V|·|E|) algorithm for maximum matching of graphs. *Computing*, 12:91–98, 1974.

[22] K. Kelley. Casual carpooling enhanced. *Journal of Public Transportation*, 10(4):119–130, 2007.

[23] C. Morency. The ambivalence of ridesharing. *Transportation*, 34(2):239–253, 2007.

[24] U. S. R. Murty and J. A. Bondy. *Graph Theory with Applications*. Elsevier Science Ltd, 1976.

[25] A. Santos, N. McGuckin, H.Y. Nakamoto, D. Gray, and S. Liss. Summary of travel trends: 2009 national household travel survey. Technical report, US Department of Transportation Federal Highway Administration, 2011.

[26] M. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4(1):285–305, 1985.

Appendix A

A Full Example of Algorithm 4

The following is a full example of the MMH algorithm for finding a maximum matching in a hypergraph H with Property 5.1.2, given in Figure A.1.

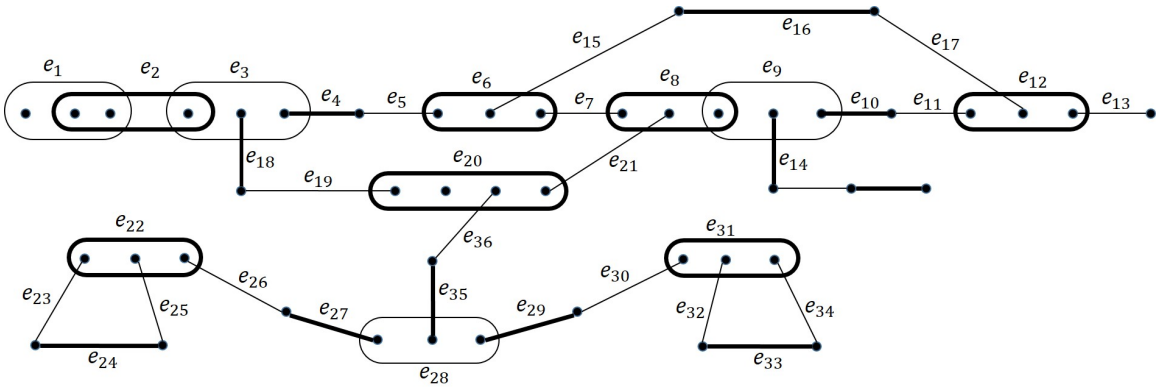


Figure A.1: A hypergraph H with Property 5.1.2 and matching M, where the bold edges are matched edges and un-bold edges are unmatched edges.

The algorithm checks every unmatched edge e where e is incident to at most one matched edge and calls the **Search** procedure. In this case, e_1 and e_13 will be checked. The algorithm checks e_1 first. Suppose that the algorithm finds the augmenting path $P = e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}$. So, initially $T = P$. Then, every unmatched portal edge in P will be processed by the algorithm. e_3 and e_9 are the only two unmatched portal edges in P.

Suppose that e_9 is checked first. The algorithm tries to find an even length alternating path starts at e_{14} in $H \backslash T$. However, such an alternating path does not exist. Thus, the algorithm backtracks and tries to find another augmenting path starts from e_1 such that it does not contain e_9. The algorithm finds another augmenting path $P = e_1, e_2, e_3, e_4, e_5, e_6, e_{15}, e_{16}, e_{17}, e_{12}, e_{13}$ and sets $T = P$ initially. Then it process the only unmatched portal edge e_3 in P. This time, it finds an even length alternating path $P_1 = e_{18}, e_{19}, e_{20}, e_{21}, e_8, e_7$. Add P_1 to T. There is no more unmatched portal edge needs to be processed. T is an augmenting graph, see Figure A.2.

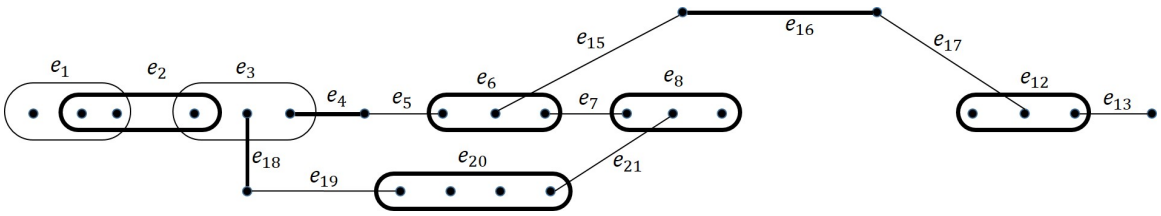


Figure A.2: An augmenting graph of H w.r.t. M.

Augment M with T, that is, $M = M \oplus T$ gives a larger matching (Figure A.3).

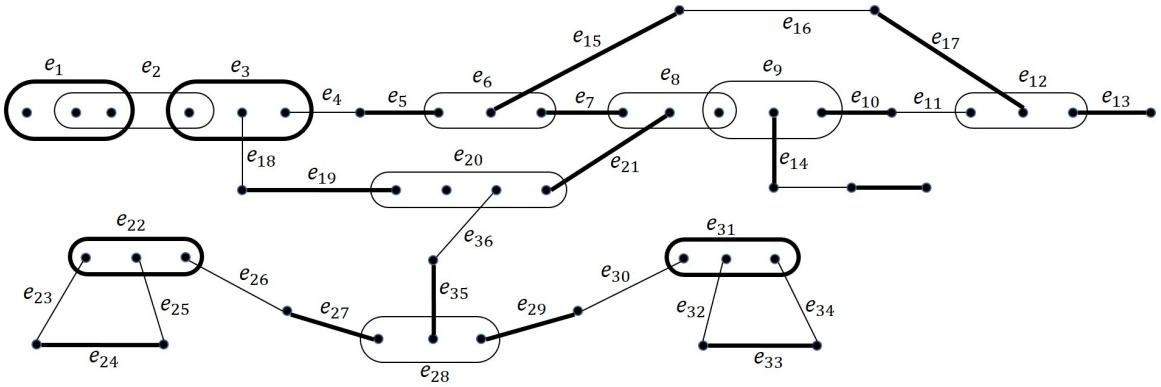


Figure A.3: The matching after the first augmentation.

After the augmentation, there is no more unmatched edge that is incident to at most one matched edge. The algorithm checks each matched portal edge and calls the **Candidate-Search** procedure. There are three matched portal edges, e_3, e_{22}, and e_{31}.

Suppose that the algorithm checks e_3 first. e_3 is removed from the current matching M temporarily. The unmatched edges that are incident to at most one matched edge is checked, i.e. a call to **Search** with $H' = H \backslash \{e_3\}$, $M' = M \backslash \{e_3\}$ and each of e_2, e_4 and e_{18} one by one. When processing e_2, no augmenting path can be found in H'. Then, it checks e_4 next and finds an augmenting path from e_4 to e_{18} in H', namely $P = e_4, e_5, e_6, e_7, e_8, e_{21}, e_{20}, e_{19}, e_{18}$. $T = P$ initially. The unmatched portal edge e_6 in P is processed. There does not exist an even length alternating path starts from e_{15} in $H' \setminus T$. The algorithm backtracks and tries to find another augmenting path not containing e_6, but there does not exist one.

Hence, it returns to CandidateSearch, and checks e_{22}. A call to Search with $H' = H \setminus \{e_{22}\}$, $M' = M \setminus \{e_3\}$ and each of e_{23}, e_{25} and e_{26}. When processing e_{23}, an augmenting path $P = e_{23}, e_{24}, e_{25}$ in H' w.r.t M' is found. There is no unmatched portal edge in P. $T = P$ is returned, and $M' = M' \oplus T$. The new matching M' has the same number of edges as M. Then, the algorithm processes e_{26} w.r.t. M'. It finds an augmenting path $P = e_{26}, e_{27}, e_{28}, e_{35}, e_{36}$ and sets $T = P$ initially. The unmatched portal edge e_{28} in P is processed. There does not exist an even length alternating path starts from e_{29} in $H' \setminus T$. The algorithm backtracks and tries to find another augmenting path not containing e_{28}, but there does not exist one. The current matching is shown in Figure A.4.

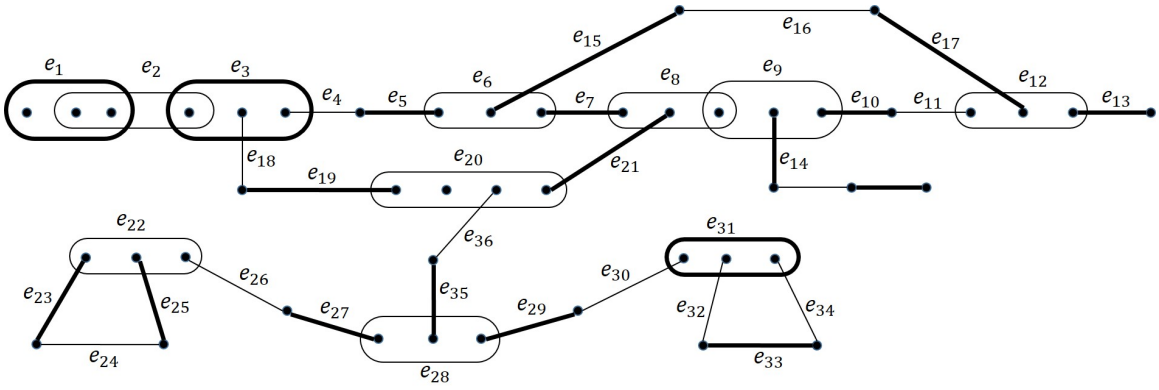
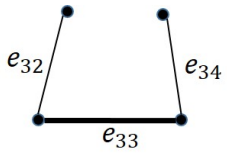
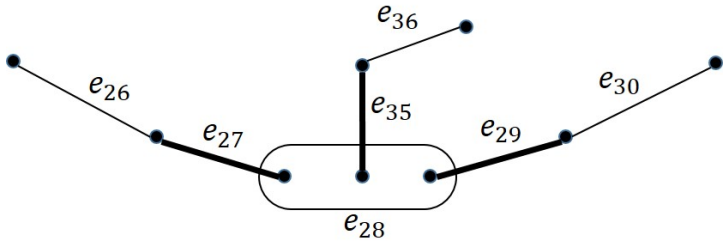


Figure A.4: The matching after the second augmentation.

The algorithm now checks the next matched portal edge e_{31} w.r.t. M'. The process is identical to checking e_{22}. Two augmenting graphs T_1 and T_2 are found w.r.t. M', shown in Figure A.5. Augmenting T_1 and T_2 gives a matching M_1 that has one more edge than M. Matching M_1 is shown in Figure A.6.



(a) An augmenting graph T_1 of H' w.r.t. M'



(b) An augmenting graph T_2 of H' w.r.t. M'

Figure A.5: Two augmenting graphs of H' w.r.t. M'.

The algorithm checks e_3 and e_{28} by calling the CandidateSearch procedure. No augmenting graph is returned. Since there does not exist an augmenting graph in H w.r.t. M_1, M_1 is maximum. In fact, this is the case by the maximum hyper-matching theorem.

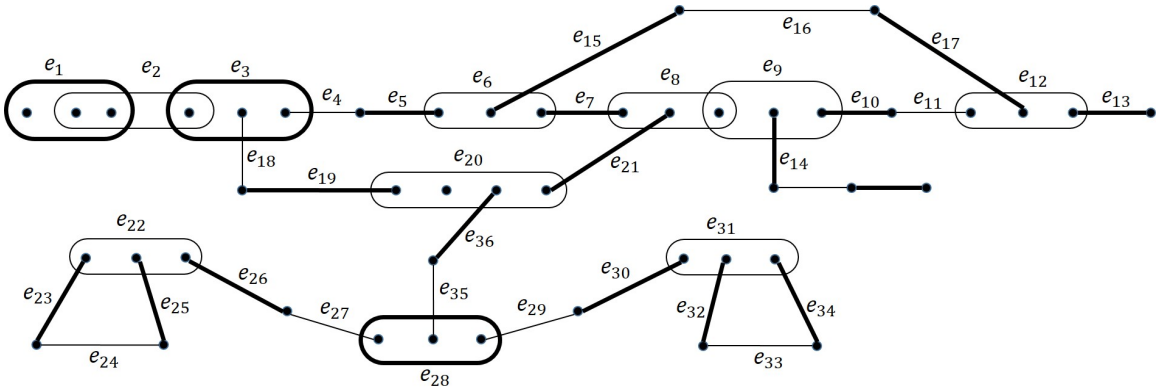


Figure A.6: A hypergraph H' with maximum matching M_1.