Providing Location-privacy in Opportunity Mobile Social Network

by

Rui Huang

A Thesis submitted to the Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

**MASTER OF APPLIED SCIENCE**

in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

University of Ottawa

Ottawa, Canada

September 3, 2017

© Rui Huang, Ottawa, Canada, 2017

Abstract

Todo

Acknowledgements

Todo

Table of Contents

[Abstract i](#_Toc493451413)

[Acknowledgements ii](#_Toc493451414)

[Table of Contents iii](#_Toc493451415)

[List of Figures iv](#_Toc493451416)

[List of Tables v](#_Toc493451417)

[Chapter 1 Introduction 1](#_Toc493451418)

[Chapter 2 Background and Related Work 1](#_Toc493451419)

[Chapter 3 Multi-Hop Location-Privacy Protection 1](#_Toc493451420)

[Chapter 4 Appointment Card Protocol 2](#_Toc493451421)

[4.1. System Model 2](#_Toc493451422)

[4.2. Appointment Card Protocol Overview 2](#_Toc493451423)

[4.3. Appointment Card 3](#_Toc493451424)

[4.4. AC life cycle 6](#_Toc493451425)

[4.5. Exchange sending process 6](#_Toc493451426)

[4.5.1. Generate new ACs 7](#_Toc493451427)

[4.5.2. Pick dispensed ACs 8](#_Toc493451428)

[4.5.3. Pick ready ACs 8](#_Toc493451429)

[4.5.4. Send ACs 8](#_Toc493451430)

[4.5.4.1. Get the number of ACs 9](#_Toc493451431)

[4.5.4.2. Check duplicated appointment numbers 9](#_Toc493451432)

[4.6. Exchange receiving process 10](#_Toc493451433)

[4.7. Sending queries 10](#_Toc493451434)

[4.8. Transmit replies 11](#_Toc493451435)

[REFERENCES: 12](#_Toc493451436)

List of Figures

[Figure 4.1 AC exchange process (send) 9](#_Toc492582640)

[Figure 4.2 AC exchange process (receive) 10](#_Toc492582641)

[Figure 4.3 Transmit reply 11](#_Toc492582642)

List of Tables

[Table 1 Appointment Card 3](#_Toc492582645)

[Table 2 AC system parameters 4](#_Toc492582646)

[Table 3 AC states 5](#_Toc492582647)

[Table 4 friend-encounter list 5](#_Toc492582648)

[Table 5 query-send list 6](#_Toc492582649)

[Table 6 coordinate-query-send list 7](#_Toc492582650)

[Table 5 AC storages 8](#_Toc492582651)

# 

# Introduction

# Background and Related Work

# Multi-Hop Location-Privacy Protection

# Appointment Card Protocol

## System Model

The network architecture consists of two main entities: Users and Location-Based Service Providers (LBSPs). Users are mobile and communicate with others in a certain range, e.g., the communication range of their portable devices. For a user, other ones in the social network are either strangers or his friends whom he can detect when they are in his communication range. Let *RSi,j* denotes the relationship strength between user *i* and *j*, if *RSi,j* is larger than a specific friend threshold *FTmin*, user *i* and *j* (*Ui* and *Uj*) are friends. LBSPs are fixed and not connected in a social network, which provides Location-Based Services (LBSs) for users. We assume that the only information which is necessary for the LBS is the location of the original requester (*U*0), but he should still give an identity to the LBSP so that the LBSP can reply to that identity.

Attackers can access the database of LBSPs, so that they can learn everything recorded in LBSPs, including users’ identities and locations. If *U*0 sends queries with his own identity and accurate location, his will be located by attackers easily, so the key is to hide his identity and location. We propose Appointment Card Protocol (ACP) to protect the identity and location-privacy of the original requester by providing a fake identity which disenable attackers to locate the original requester with records in the LBSP database.

## Appointment Card Protocol Overview

We propose an Appointment Card Protocol (ACP) to protect original requesters when they are served by LBSPs. A user (*Agc*1) generates his own Appointment Cards (ACs) containing his identity, a unique Appointment Number (AN) and a location which are be exchanged when he encounters a friend. When *U*0 sends a query, he chooses an AC and sends the query with the identity (*Agc*1), AN and location in the AC. A LBSP replies to *Agc*0 when it receives the query. Since *Agc*1 is actually the one who generates the AC, he re-transmits it to the user (*Agc*2) whom he exchanged the AC to, and so do the following users until the reply reach *U*0.



AC

AC

AC

Reply

Query

Reply

Reply

Reply

Figure 4.1 Process example

## Appointment Card

Any user in the network can generate his own Appointment Cards (ACs) and he is called the creator of those ACs. The creator should write down his identity (*Cid*), a unique Creator Appointment Number (*Capt*) and his current location (*Clct*) on his AC. Entries of AC are shown in Table 1.

The (ready) AC is an authorization that anyone holds it can send a query to LBSPs using its creator’s identity (*Cid*) and location (*Clct*). When LBSPs receives the query, it replies to the creator (*Cid*) and attaches the *Capt* in reply, which enable the creator to determine who is the next one to transmit the reply.

Table 1 Appointment Card

|  |  |  |
| --- | --- | --- |
| ***entry*** | ***explanation*** | |
| *Cid* | The identity of the creator who generates the AC. |
| *Capt* | A unique number that distinguishes an AC from other ones generated by the same creator. |
| *Clct* | The location where the creator generates the AC. |
| *Aid* | The identity of an agency who gives the AC to the recent holder. (The previous hop of the AC) |
| *Aapt* | A unique number that distinguishes an AC from other ones transmitted by the same agency. |
| *time-out* | The time when the AC expires. |
| *EQ* | A queue (Exchange Queue) which records users who exchange the AC in order. Its length is *EQL*. |
| *dc* | A counter which is used to distribute ACs of the same creator to various users. |
| *Psd* | A pseudonym or a fake identity for the original requester receiving a reply. |

To guarantee the security, the distance between the original requester and the AC creator should be far enough, an AC must be exchanged for a specific *k* (see Table 2) times before it can be used for a requester to send a query. The value of *k* should neither be too small nor too large. If *k* is small, e.g. *k* = 1, attackers can infer the original requester from the records of his friends in the database easily. If *k* is large, replies from LBSPs can hardly reach the original requester, because replies must go through all users in their EQs.

Both *Capt* and *Aapt* are appointment numbers, which are unique random integers. When we mention “unique”, it means that there are not an identical (same value and type) existing appointment number generated by the same user in the network. Existing means that the CA which is relevant to the appointment number does not expires. For example, if a user (*Ui*) generates a *Aapt* “123456” and it does not expire, he cannot generate another *Aapt* “123456”, but a *Capt* “123456” is acceptable. He may also generate another *Aapt* “123456” after the previous one expires. It would not prevent another user (*Uj*, *j*≠*i*) generating an *Aapt* “123456” no matter whether *Ui* generates one or not.

Table 2 AC system parameters

|  |  |  |
| --- | --- | --- |
| **Parameters** | **explanation** | |
| *k* | The minimum length of the exchange record for an AC eligible to be used in a query. |
| *Seg* | dispensed ACs are exchanged when two friends encounter each other. |
| *DN* | The minimum number of ACs that a user hands out in anytime. |
| *SU* | The time unit |

The parameter *k* and the *EQL* also divides ACs into 2 states: dispensed and ready (see Table 3). Ready ACs can be used when their holder sends a query. Users still exchange ready ACs among their friends, but they do not add themselves in ACs’ *EQ* any more. Dispensed ones are not qualified to be used by current holder. Therefore, in order to make an AC useful, it should be exchanged for *k* times as fast as possible.

Table 3 AC states

|  |  |
| --- | --- |
| **State** | **Explanation** |
| *Dispensed* | *EQL* < *k* |
| *Ready* | *EQL* = *k* |

In order to manage ACs, each user maintains two lists: the *dispense-list* and the *ready-list*. It is obviously that *dispense-list* contains dispensed ACs and *ready-list* contains ready ones.

Table 4 AC storages

|  |  |
| --- | --- |
| *List Name* | *Explanation* |
| *ready-list* | It contains ready ACs. |
| *dispense-list* | It contains dispensed ACs. |

To avoid ACs generated by a creator simultaneously being dispensed to the same *k*th user, we employ the parameter *Seg* and *dc*. The *dc* is a counter which is initialized to a random integer in the interval [1, *Seg*] when a user received an AC. The AC should not be sent to the next friend until *dc* reaches 0, which enable users to split these ACs into about *Seg* batches.

## AC life cycle

The life cycle of an AC starts when it is generated by its creator. The creator initializes the *Cid*, *Capt* and *Capt* when he generates it. These three entries remain constant in the whole life cycle. In the first *k* exchanges, the user who is sending it modify its entries *Aid* to his own identity. The Aapt is also replaced by a new appointment number generated by him. he adds his identity in its *EQ* before sending it out. When the AC’s ER length reaches *k*, it is eligible to be used in a query and called a ready AC. A ready AC can be exchanged to other friends either, but senders never modify previous entries, i.e., *Aid*, *Aapt* and *EQ*. Each of ACs can be used for no more than one time. In another word, an AC expires, when reaching its time-out or it is used in a query.



Figure 4.2 AC's life cycle

As shown in the Figure 4.2, an AC starts at a generated state. When it leaves the creator, it switches to a dispensed state. If the length of its ER reaches k, it changes to the ready state. It can be time-out in either the dispensed state or the ready state. If the holder uses it in any queries, it expires, too.

## Exchange sending process

Each AC is unique and hold by only one user at one time. When a user exchanges an AC to a friend, he loses the control of that AC and cannot give it to another one. We discuss the process that users exchange ACs with their friends. The process that a user (*Ui*) exchanges ACs to his friend is shown in Figure 4.3 AC exchange process (send). When *Ui* encounters a friend (*Uj*), he would send 3 kinds of ACs to the friend, which include dispensed ACs, ready ACs and new ACs. Even though new ACs must be dispensed ACs, we discuss them separately. During the exchange process, *Ui* generates the specified number of ACs and selects a part of dispensed ACs and ready ACs, then he sends them to his friend. He also remembers whom he sends them to, which enables him to transmit replies to the friend.



Figure 4.3 AC exchange process (send)

### Generate new ACs

Each user (*Ui*) has an obligation to generate proper number of ACs for their friends, so that he guarantees that the number of ACs created by him in the network is larger than *DN* after he exchanges ACs with a friend (*Uj*). We assume that he must give *Uj* *Sg* new ACs to achieve that goal. *Ui* initializes entries for his new ACs, i.e., *Cid* (his identity), *Capt* (a new appointment number), *Clct* (his current location), *Aid* (A default value *Aiddefault*), *Aapt* (The same value as the *Cid*), *time-out*. The *EQ* in a new AC is initialized to empty, and *dc* is 0. We assume that the number of *Ui*’s ACs that are remained in the network is equal to *RNi*. The method of determining *RNi* is discussed in 4.5.4.1. Then .

### Pick dispensed ACs

*Ui* decreases *dc*s of all his dispensed ACs, so that he gets *Sd* ACs whose *dc* values are equal to zero. They should be sent to his friend. Since the parameter dc is initialed as an integer between 1 and *Seg* randomly, there are about  dispensed ACs eligible to be sent to the friend. That enables us to separate ACs which come from a common creator.

### Pick ready ACs

There are two ways for a user to provide ready ACs: 1. give his own ready ACs to friends directly; 2. dispensed ACs whose *EQL* values are equal to *k*-1. Since ready ACs are precious for every user, the strategy that a user shares ready ACs with his friends should save his own ready ACs. the number of ready ACs which would be sent to the friend is



, where *Sd*,*k*-1 denotes the number of dispensed ACs whose *EQL* values are equal to *k*-1. We assume that *Uj* has no ACs at all when they encounter. After their exchange, if *Uj* has more ready ACs than *Ui*, *Ui* must have all his old ready ACs with him; if *Ui* loses some of his ready ACs, *Uj* does not have more ready ACs than *Ui*. In another word, *Ui* should do his best to help his friend *Uj*, but *Ui* is not in a dilemma worse than his friends. That strategy makes users who sends more queries than friends supplement their AC consumption quickly, while protecting their friends’ profits. Then *Ui* picks *Sr* ready ACs randomly and sets their *dc*s to 0.

### Send ACs

An AC should be sent to *Uj*, if and only if its *dc* is equal to 0. Before it is sent to *Uj*, *Ui* adds its identity to the its *EQ*. Besides, *Ui* replaces its *Aid* and *Aapt* with his own identity and a new appointment number. The old *Aid* is recorded in his *agency-list*, so do both old and new *Aapt*s. The structure of the *agency-list* is shown in Table 5. When the AC comes from his *dispense-list*, he checks its *EQL*. If it is equal to k and the last entry of *EQ* is himself, he generates a unique pseudonym for the *AC* (*Psd*) and records the pseudonym in the *agency-list* (*Nxt*). Otherwise, he records the identity of his friend in *Nxt*. Since items which is relevant to an expired ACs would be deleted, each item in the *agency-list* is related to a distinct existing AC. Information in it can be used for statistics showing in the following sections.

Table 5 Agency\_list entries

|  |  |
| --- | --- |
| *List Name* | *Explanation* |
| *Aidold* | The previous agency identity. |
| *Aaptold* | The previous agency appointment number |
| *Aaptnew* | The current agency appointment number |
| *Nxt* | The identity of the next friend. |
| *time-out* | The time when the AC expires |

#### Get the number of ACs

*Ui* can get the number of existing ACs which is generated by him based on the information in his *agency-list*. He counts the items in the list whose *Aidold* value is equal to *Aiddefault*, that is the *RNi*.

#### Check duplicated appointment numbers

All existing appointment numbers generated by a user are recorded in his *agency-list*. If *Aidold* value is equal to *Aiddefault*, the *Aaptold* is a *Capt*; Otherwise, the *Aaptnew* is a *Aapt*. When the user generates a new appointment number, he must check the *agency-list* to guarantee it is unique.

## Exchange receiving process

The process that *Ui* receives ACs from his friend is shown in Figure 4.4 AC exchange process (receive). ACs are divided into two groups based on their *EQL* values, which are dispensed ACs and ready ACs. The *dc* value of a dispensed AC is assigned to a random integer in the interval [1, *Seg*], then the AC is moved into his *dispense-list*. Ready ACs are moved into his *ready-list*, and their dc values are set to -1.



Figure 4.4 AC exchange process (receive)

## Sending queries

When a user wants to send a query, he picks an AC from his *ready-list*. He gives preference to the AC that has the earliest time-out. As a result, he sorts ACs in the *ready-list* in ascending order based on their remaining time and selects the first AC (e.g. *ac*) to send his query. That means the query is sent under the name of *ac*’s creator (i.e., *Cid*), and the location of the query is also *ac*’ location (i.e., *Clct*). Besides, *ac*’ *Capt* is also included in the query. The query can be transmitted to LBSPs by an arbitrary DTN routing protocol (e.g., Spray and Wait). When a LBSP receives the query, it provides services based on ac’s location and reply to ac’s creator (ac’s *Cid*). Users can also use a general DTN routing protocol to deliver the reply.

## Transmit replies

When a user receives a reply, he checks his *agency\_list*. If the sender identity is the LBSP, he searches for a record in his agency\_list, whose *Aidold* is *Aiddefault* and *Aaptold* is equal to the *Aapt* in the reply. If the sender identity is a user, he looks up for an item whose *Aidold* is equal to the sender identity and *Aaptold* is equal to the *Aapt* in the reply.

Based on the record he finds, he modifies information in the reply. The sender identity is set to his own identity; the destination is set to the user whose identity is *Nxt*; *Aapt* is assigned to *Aaptnew*. The item related to that AC can be deleted after the replacement, then he replies to the next friend (*Aidnew*).

After the original requester sends the query out, he claims himself as the one whose identity is *Psd*. That identity is equal to the *Nxt* in the *k*th user’s *agency\_list*. As a result, nobody knows the identity of the original requester. The process is shown in Figure 4.5.



Figure 4.5 Transmit reply

## Example

We assume there are four users in our social network: Alice, Bob, Charlie and David and there are 3 pairs of friends (i.e. Alice and Bob, Bob and Charlie, Charlie and David) in our example. We assign the system parameter *k* to 3, so that each Appointment Card must be exchanged for three times before it can be used. The system parameter *DN* is equal to 100, so that each user guarantees there are more than 100 ACs generated by him in the network.

When Alice encounters Bob, both of them generate 100 new ACs and give all the new ACs to the other one. Table 6 shows the number of ACs hold by Alice and Bob after the exchange where ACAlice,1 denotes the ACs generated by Alice and with *EQL* equal to 1.

Table 6 Alice and Bob exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Alice | Bob |
| Dispensed ACs | 100 ACBob, 1 | 100 ACAlice, 1 |

Then Bob encounters a friend Charlie who is a new user. Since Bob has already generated 100 ACs and none of them are used, he generates no AC for Charlie. But, Bob should give half of his dispensed ACs to Charlie while Charlie generates 100 new ACs and give them all to Bob.

Table 7 Bob and Charlie exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Bob | Charlie |
| Dispensed ACs | 100 ACCharlie, 1  50 ACAlice, 1 | 50 ACAlice, 2 |

When Charlie encounters David, Charlie gets 100 new ACs from David. David get 25 ACs which are generated by Alice from Charlie. The 25 ready-ACs hold by David can also be exchanged to other users, as shown in Table 8.

Table 8 Charlie and David exchange ACs

|  |  |  |
| --- | --- | --- |
| Users | Charlie | David |
| Dispensed ACs | 100 ACDavid, 1  25 ACAlice, 2 |  |
| Ready ACs |  | 25 ACAlice, 3 |

The route of one of the ready ACs (for example, the ACAlice, 3) after it is received by David is shown as follow. For convenience, we donate it by AC*x*. David gives a pseudonym (Say *p*) to AC*x* before he exchanges it to another user. We assume that David gives it to Elizabeth and Elizabeth gives it to Franklin, and Franklin would use the AC*x* to send the query. Franklin sends the query pretending he is Alice and receives the reply pretending he is *p.* More specifically, in the replying phase, LBSP replies to Alice and then the reply would be transmitted by Bob, Charlie and received by David. After that, David broadcast the reply to *p*, other users would ignore that reply while Franklin realized it is the reply of his query since he uses the AC with pseudonym *p.*

Table 9 The exchange history of ACx

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| User name | *Cid* | *Capt* | *Aid* | *Aapt* | *Pseudonym* |
| Alice | Alice | 233 | Alice | 557 | -- |
| Bob | Alice | 233 | Bob | 743 | -- |
| Charlie | Alice | 233 | Charlie | 691 | -- |
| David | Alice | 233 | David | 283 | *p* |
| Elizabeth | Alice | 233 | David | 283 | *p* |
| Franklin | Alice | 233 | -- | -- | -- |

When Franklin wants to send a query to a LBSP, he uses Alice’s identity and attaches the *Capt* (i.e. 233) in the query. Then he uses the pseudonym *p* as his identity when he receives a reply.

The LBSP replies to Alice, and the *Capt* (i.e. 233) is included in the message. When Alice receives the message, she checks her *agency\_list* and finds the entry whose *Aidold* is equal to *Aiddefault* and *Aaptold* is equal to 233. Based on the entry, she attaches the *Aaptnew* (i.e. 557) in the reply and sends it to *Nxt* (i.e. Bob). We collect entries which are related to the AC*x* in Table 10. Then Alice remove the item from her *agency\_list*. Also, the number of ACs generated by her decreases to 99 (=100-1), so that she should generate a new AC when she encounters a friend.

When Bob receives the reply from Alice, he also checks his *agency\_list* and learns that the reply is related to the AC*x* based on the sender identity (i.e. Alice) and the *Aapt* (i.e. 557). He modifies the *Aapt* to 743 in the reply message and relays it to Charlie. Charlie also follows the same process. Different from the previous 3 friends, David relays the message to a pseudonym *p*, because the *Nxt* value is equal to *p* in his *agency\_list*. Since Franklin is using the pseudonym *p* as his identity, he can get the reply message at last.

Table 10 Users' agency\_list

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| User Identity | *Aidold* | *Aaptold* | *Aaptnew* | *Nxt* |
| Alice | *Aiddefault* (LBSP) | 233 | 557 | Bob |
| Bob | Alice | 557 | 743 | Charlie |
| Charlie | Bob | 743 | 691 | David |
| David | Charlie | 691 | 283 | *p* |

Table 11 Important information in the reply messages

|  |  |  |  |
| --- | --- | --- | --- |
| Transmit | Sender | Receiver | *Aapt* |
| LBSP → Alice | LBSP | Alice | 233 |
| Alice → Bob | Alice | Bob | 557 |
| Bob → Charlie | Bob | Charlie | 743 |
| Charlie → David | Charlie | David | 691 |
| David → *p* | David | *p* | 283 |

In a nutshell, the *Aapt* in the reply is the key for each user (friend) to relay the reply message to the next one. After a user relays the reply message to the next one, he removes the entry from his *agency\_list*.

# Vertex reduce algorithm

Movement of network nodes is a significant factor for the performance of DTNs. To evaluate the performance of different DTN protocols, researchers presented different kinds of movement models, like in [2]. Since shortest path algorithm is an essential and time-cost part of creating many movement models, it is an important topic to optimize the time-cost for simulators calculating the all-pairs shortest path problem in the entire map.

The best known non-negative edge weight undirected map all-pairs shortest path algorithm [3] has a complexity , which is still expensive when *n* is huge. For most real-world city map, there are thousands of points in a single square kilometer. Since a tens square kilometers map is reasonable for a DTN protocol evaluation, a simulator must cut down the time-cost of the calculation of the all-pairs shortest path.

Real-world map developers often use short straight lines to present curves, like in [4], so that a several-meters curves may contain tens of points. It is obviously that we do not need to calculate shortest path for every one of these points. In this paper, we present an Vertex Reduce Algorithm (VRA) to reduce the number of points before the process of the shortest path algorithm. The basic idea is that VRA removes all points whose degrees are less than 3 from the map, while keeps the result of all-pairs shortest path algorithm correct.

The rest of this paper is organized as follows: 5.1 presents some basic lemmas and definitions. The process of VRA is described in 5.2 and 5.3.

## Ignorable vertex and reserved vertex

Vertices in the graph are considered as ignorable and reserved ones. In each iteration of VRA, we remove the ignorable vertex from the graph, while the reserved vertices compose a new graph.

Definition: the vertex whose degree is larger than 2 is the reserved vertex (R).

Definition: the vertex whose degree is smaller than or equal to 2 is called the ignorable vertex (G).

Definition: if two reserved vertices (e.g. ) are connected by a sequence of ignorable vertices (e.g. ), then the sequence from to (i.e., ) is a line-segment (LS).

A reserved vertex can belong to different line-segments, while an ignorable vertex belongs to a unique line-segment. Both the reserved vertex and the ignorable vertex are called vertices (V)

Definition: the shortest path between two vertices inside a line segment is called the inner shortest path (SPI), Let denote the inner shortest path between two vertices (i.e. ) inside a line segment.

Lemma1: If two vertices (i.e. , ; ) are in the same line-segment, the shortest path between them is

Since an all-pairs SPI has a complexity equal to or smaller than and the *n* here is much fewer than the number of points on the entire map, the time-cost for SPI is ignorable comparing to the time-cost of the entire map all-pairs shortest path calculation.

Lemma2: We assume that there are two different line-segments (i.e., and ). We pick a vertex from and a vertex from , then the shortest path between and is

where . Here and are in the same line-segment, so we can use lemma1 to calculate it, so do the and . The is the only part we need to calculate using all-pair shortest path algorithms. We should notice that and could be the same vertex, but it does not make any difference to the lemma.

Therefore, the key is getting an all-pair shortest path of all reserved vertices.

## Vertex reducing

VRA iteratively removes ignorable vertices from the graph until only reserved vertices remained. In each iteration, we remove all ignorable vertices from the graph and remain their lines to connect reserved vertices. As a result, each pair of reserved vertices might have various connection lines. we keep the shortest one and remove others.

### Remove ignorable vertices

As degrees of ignorable vertices are no more than 2, an ignorable vertex has only 0, 1 or 2 neighbours. In the case of 0 or 1 neighbour, we just delete the ignorable vertex. If it has two neighbours, we connect its two neighbours before it is removed, as shown in Figure 5.1. When we remove an ignorable vertex, the weight of the line which connects its two neighbours is equal to the sum of its recent two lines’ weights (i.e., ). As soon as we remove all ignorable vertices in a line-segment, the two reserved vertices at the ends of the line-segment are connected with a line directly, whose weight is the sum of all the intermediate ones’ (i.e., ).



Figure 5.1 Remove ignorable vertices

### Tidy reserved vertices connections

After we remove all ignorable vertices, all the reserved vertices are connected directly. However, it is possible that there are several connections between a pair of reserved vertices. The shortest connection is the short-cut between that two reserved vertices obviously, so that we remove all other connections except the shortest one, as shown in Figure 5.2. We assume that is the shortest line in their three connections, then and are removed.



Figure 5.2 Tidy reserved connections

### Iterations

The whole algorithm is shown in Algorithm 1. The input of the algorithm is the original entire map, which is called . In the *i*th iteration, VRA makes a copy of as . If , the is the output of the previous (i.e., ) iteration. VRA removes all ignorable vertices in the as described in 5.2.1 and remove all unnecessary connections between reserved vertices as described in 5.2.2. In other words, is smaller and smaller as *i* increases, because we always remove ignorable vertices from them. If is equal to , which means that the *i*th iteration makes no modification on the graph, the algorithm ends.

Algorithm 1 vertex reducing

**function** reduce(*Graphi*)

Copy *Graphi* to *Graphi*+1

**for** every ignorable vertex *Gu* in *Graphi*+1 **do** remove(*Gu*) (see 5.2.1)

**for** every pair of reserved vertices **do** remove redundant connections (see 5.2.2)

**end function**

**function** VRA(*Graph*0)

i=0

**while** i=0 or there are any differences between *Graph*i-1 and *Graph*i

**do**

**call** reduce(*Graph*i)

i=i+1

**end**

**end function**

## Vertex assembling

We assume that the reducing process stop at the *i*th iteration. As a result, the is the input of an all-pairs shortest path algorithm. Let denote the shortest path from to in . We can infer the all-pairs shortest path of based on with lamme1 and lamme2. The algorithm ends when we get the result of . The algorithm is shown in Algorithm 52.

Algorithm 52

function

for all ignorable vertices in as

for all other vertices in as

if and are in the same line-segment then

Use lemma 1 to calculate their shortest path.

else

Use lemma 2 to calculate their shortest path.

endif

end

end

end function

function

while i > 0 do

call

end

end function

We start from the and assemble all intermediate graphs, until we get the result of . For any intermediate , we have already get the all-pairs shortest path result of its reserved vertices in the previous iteration (i.e. ). Then we calculate the shortest path from any ignorable vertices to all others. When the two vertices are in the same line-segment, we use lemma1 to calculate their shortest path. The lemma 1 includes 4 parts: , , and . Since we already get the result of in the previous iteration, we just need to calculate 3 parts, which is easy and has a small scale. If the two vertices are in different line-segments, we use lemma2 to calculate the shortest path. We also already get all the parts from the previous iteration so that we just need to deal with their parts.

**REFERENCES**:

1. A. Lastname1, B. Lastname2, C. Lastname3 and D. Lastname4, “Article name,” in Publication name, vol. 13, issue 2, pp. 223–244, 2017.
2. Working Day Movement Model
3. all pairs shortest path algorithm with expected running time O(n2logn)
4. https://www.openstreetmap.org/