國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

在大型網路圖上計算完全子圖之最遠距離

Computing the Maximum Clique Distance on a Large-Scale Network

曾聖耀

Sheng-Yao Tseng

指導教授：趙坤茂 博士

Advisor: Kun-Mao Chao, Ph.D.

中華民國 100 年 6 月

June 2011

國立臺灣大學碩士學位論文
口試委員會審定書

在大型網路圖上計算完全子圖之最遠距離

Computing the Maximum Clique Distance on a Large-Scale Network

本論文係**曾聖耀**君（學號R98922095）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 100 年 6 月 20 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

[illegible]

（指導教授）

[illegible]　　[illegible]

系　主　任　呂育道

誌謝

感謝指導教授—趙坤茂教授—在我就讀碩士班期間給予莫大的協助，若無老師淳淳教誨以及諸多寶貴的指導，這本論文不可能完成。感謝教授對我的耐心與包容，以及如沐春風的鼓勵。我的性格稜稜角角，資質普普通通，老師的大力協助與不停的打氣，是成就這本論文最重要的元素。感謝老師總是適時糾正我的錯誤，學生永銘五內。

也感謝ACB實驗室，有著良好的研究討論風氣。感謝蔚茵從不間斷的支持、幫助與珍貴的討論，也祝妳往後一路順風；感謝明江學長提供我許多寶貴的意見與機會；感謝怡靜學姐、容任學長、冠宇學長、正偉學長、Roger學長、家榮學長、大餅學姐、霈君學姐、安強學長、彥緯學長，陳琨學長你（妳）們給予的經驗與建議給了我莫大的幫助，適時的打氣，是我再站起來的動力。最後要感謝同爲碩士班的成員—峻偉、秋芸、稚穎與甯之，碩士班期間經歷了種種關卡，感謝你們與我一起渡過。

最後要感謝LaTeX系統，從文書排版的過程中我得到了相當的快樂。

中文摘要

我們研究完全子圖之最遠距離（*clique diameter*）問題。給定一個無向、無權重圖，並同時給定一個集合，集合包含圖上所有給定的完全子圖，該問題問其中任兩組完全子圖間之最遠距離爲何？由於完全子圖可以代表一個理想的社群，其中每一成員均關聯於其它成員，本問題即是詢問圖上各社群之距離，以及距離最遠的一組社群。假設圖上有n個點與m個邊，同時給定r個完全子圖，我們設計一個$O(r(n+m))$時間且最多誤差爲1的近似演算法，並提供一個轉換方式，讓我們得以運用全點最短路徑演算法來解決我們的研究問題。

關鍵字：圖形演算法（graph algorithm）、完全子圖（clique）、全點最短路徑（all-pairs shortest paths）、矩陣相乘（matrix multiplication）、廣度優先搜尋（breadth-first search）。

Abstract

We study the *clique diameter* problem. Given an undirected, unweighted graph containing a fixed set of cliques, we are interested in finding the maximum distance among all pairs of cliques. In the context of social network analysis, a clique represents an ideal community, a clique distance represents the sparsity between the two communities, and a clique diameter indicates the farthest distance among the social network. Let n denote the number of nodes, m denote the number of edges and r denote the number of given cliques. We provide an $O(r(n+m))$ time algorithm to compute approximate clique distances with additive error of one. Another contribution is a reduction which transforms any instance of the clique distance problem into an instance of the all-pairs shortest paths problem.

Key words: graph algorithm, clique, all-pairs shortest paths, matrix multiplication, breadth-first search.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

We present a new computation problem, the *clique distance* problem, which is raised from the context of social network analysis. A clique, which is a complete subgraph, is an ideal community in the context of social networks analysis. Figure 1 illustrates some examples of complete graphs. In a clique-like community, each member shares a relationship between any other members in the same community. Given a set of cliques, we are interested in finding the *clique distance* between each pair of them.

Graph models or network models are powerful tools to represent relationship and are essential components in the field of computer science. Various systems can be modeled as graphs or networks. Recently the researchers have focused on the complex networks, which are the graph models of complex systems. The network models of the Metro routes map and the collaborative relationship of scientists are examples.

Traditionally the random graph models or the Erdös-Rényi model [11] is adopted

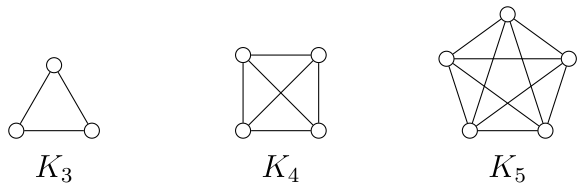


Figure 1.1: Some Examples of Complete Graphs

to represent a complex network. In the ER-model there are two parameters. The first one is the number of nodes and is denoted as n. The other parameter is the probability of an edge existing in a random graph and denoted as p. However, some graphs are different from random graph models. For example, the degree distribution of ER-model is a normal distribution, but empirical evaluation shows that the degree distribution of real world networks usually belong to Power Law distribution [26].

Community detection on graphs is an old and deeply investigated problem. The problem has numerous variants and many different objective functions [12]. However, our problem is not solving community detection problem but providing a measurement to the sparsity of communities.

Intuitively the clique distance is the shortest path joining the two cliques. The clique distance problem is motivated by measuring the distances from the two communities in a social network. The meanings of the clique distance is that the minimum number of steps needed from one community to reach another community.

We adopt a very strict definition for communities thus our measurements are worst estimates. Our contribution includes a straightforward $O(mn + n^2)$ time algorithm and a simple algorithm computing the approximate clique distance with an additive error of one. Both of them utilize the classic Breadth-First-Search algorithm.

Even we are facing a brand new problem, it is highly related to the All-Pairs Shortest Paths (APSP) problem. The classic APSP problem has a long and vastly investigated history. Moreover, it is possible to solve our problem on top of the well-developed techniques solving APSP problem with only slight rise of time complexity. However,the clique model is too strict in practice. Members in one specific community might not know all of the others. So there are several different definitions of communities, for example, *k-clique* [21] or *k-club*[2, 25]. Note that our problem adopts the clique representatives only and is purely theoretical interest.

Note that computing the clique distance while enumerating all of the cliques in the input graph at the same time are impossible because there might be exponential many of cliques in input graphs. In other words, the problem of finding all cliques is much harder than the APSP problem.

The following are brief outlines. In Chapter 2 we describe the symbols and definitions in this thesis. We also introduce the prior results of APSP problem, both the exact and approximate results are surveyed. In Chapter 3 we describe an algorithm with an additive error of one. Starting with a straightforward algorithm and some observation, we obtain the approximate algorithm. In Chapter 4 we describe a transformation from the clique distance problem to the All-Pairs Shortest Paths problem. Thus we may solve our problem by solving the APSP problem. In Chapter 5 we conclude our research results.

Chapter 2

Preliminaries

2.1 Basic Terminologies

First we define the notation $\tilde{O}(f(n))$ as the shorthand of $O(f(n)\log^c n)$ for some positive constant $c > 0$, based on the fact that $log^k(n) = o(n^\epsilon)$ for some positive constant k and ϵ [7].

By default we adopt the *Random Access Machine* (RAM) model [7].

Every graph model consists of two primary classes of elements: *vertex*, (or *node*) and *edge*, (or *link*). In this thesis we use the terms "vertex" and "node" interchangeably and do the same manner to the terms "edge" and "link". If we consider a simple graph representing the routes of air transportation, each airport is formulated into a vertex, and the route between two particular airports is formulated into an edge between the two corresponding nodes. Let $V(G)$ denote the vertex set of G and let $E(G)$ denote the edge set of G.

An edge has *weight* or *cost*. We denote $w : E(G) \to F$ as a *weight function* where F denote a *field* structure, for example, the set consists of all of real numbers. The edge weights of e is denoted as $w(e)$.

For two arbitrary nodes u, v belonging to $V(G)$, we have a *directed edge* $e =$

(u, v) representing a link from u to v. The directed edge e is an one-way link. Only u is able to reach v via e and reversed order is not allowed. We usually draw directed edges as line segments with arrows pointing to the end point v. We say an edge $f = (u, v)$ *undirected* means there is a link f between u and v and both two vertices are able to reach the others via the edge f. We draw undirected edges as plain line segments between two end points without any arrow.

The edges in one particular graph are either directed or undirected. A *directed graph* has only directed edges and an *undirected graph* has only undirected edges.

A clique is a subgraph which has its node set V' and every pair of nodes in the set V' is adjacent.

A path in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. We introduce a notation

$$\mathcal{P}(v_1, v_2, v_3 \ldots, v_{k-1}, v_k)$$

representing a path which starts by v_1, reaches v_2, v_3, $\ldots v_{k-1}$ consequently, and stops by v_k at last. The *length* or the weight of the above path is defined as:

$$l(\mathcal{P}) = \sum_{1 \le i \le k-1} w(v_i, v_{i+1})$$

The *shortest path* is the path with minimum path length among all the paths between the particular pair of nodes. Given two nodes u and v, the notation $d(u, v)$ denote the *distance*, which means the length of the shortest path(s) between u and v. Fixing two nodes, the shortest paths joining the nodes u and v may not be unique but have equal length.

The *Single Source Shortest Paths*(SSSP) problem is to find the distance from node u to every node in the same graph. If we unite all the shortest path starting from u, we obtain a *Shortest-Paths Tree* rooted at u. The *All-Pairs Shortest*

Paths(APSP) problem is to find the distance between every pair of nodes in a graph. The diameter of a graph G, denote as $\mathbf{diam}(G)$, is:

$$\mathbf{diam}(G) = \max_{u,v \in V(G)} d(u,v) \tag{2.1}$$

An *eccentricity* of a node u, denote as $\mathbf{ecc}(u)$, is the distance between u and the farthest node from u.

Given a weighted graph G with n nodes, define the *length matrix* $D = \{d_{ij}\}_{i,j=1}^{n}$ as:

$$d_{ij} = \begin{cases} w(e) & \exists e \in E(G), \quad e = (v_i, v_j), \\ \infty & \mathit{Otherwise.} \end{cases}$$

2.2 Related Works

In the clique distance problem we are interested in the inter-distance among a portion of vertices - those are covered by the given cliques. Thus our problem is highly related to the all-pairs shortest path problem. Though the two problems are not identical, some concepts could be borrowed from the APSP problem. Some related works on APSP problem are presented here. We refer Zwick's detailed survey [34] as a good reference regarding the algorithms for SSSP problem and APSP problem.

2.2.1 All-Pairs Shortest Paths Problem

A basic approach to solve APSP problem is launching Dijkstra's algorithm [9] from all of nodes. This takes $O(mn + n^2 \log n)$ time but may only adopted on the graph with nonegative edge weighted. To overcome this drawback, a *re-weighted method* is purposed [18]. The edge weight is adjusted into nonegative values and the length of all shortest paths are preserved. Combining the re-weighted methods and Dijkstra's algorithm, the APSP problem on the graph with negative weights is solvable. Nonetheless, this approach can not handle the graphs containing negative cycles.

The Flody-Warshall algorithm [7] solves APSP problem in $O(n^3)$ time. The Algorithm 2.1 describes the algorithm. Flody-Warshall can not handle the cases with negative cycles either. However, it detects the negative cycles in any graphs.

Algorithm 2.1 Flody-Warshall Algorithm

```
FLODY-WARSHALL($W$)
    ▷ Input weights matrix $W$
    ▷ Output distance matrix $D$
1   $n \leftarrow rows[W]$
2   $D^{(0)} \leftarrow W$
3   for $k \leftarrow 1$ to $n$
4         do for $i \leftarrow 1$ to $n$
5               do for $j \leftarrow 1$ to $n$
6                     do $d_{ij}^{(k)} \leftarrow \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
7   return $D$
```

Spira [29] gave an algorithm with expected running time in $O(n^2(\log n)^2)$. Karger et al. [19] presented an $O(m^*n + n^2 \log n)$-time algorithm for weighted graph – the *Hidden Paths Algorithm*, where m^* denoted the number of *optimal edges* in the input graph. Karger et al. defined any edge itself was a shortest path as an *optimal edge.* Therefore, they made no improve on the undirected graphs.

Karger et al. also proved the lower bound for any *path-comparsion-based* algorithms, which meant an algorithm revealed the weight information only by comparing the edges to see if one edge was heavier to another one. In other words, $\mathcal{A}$ only relied on order of the edge weight. McGeoch [23] independently concluded to an algorithm with identical time complexity.

2.2.2 Distance Matrix Multiplication

The APSP problem is equivalent to matrix multiplication over the closed semiring $\{\min, +\}$ [3]. Such matrix multiplication operations are defined as *distance matrix multiplication.* We demonstrate the similarity between the formulation of two operations. Given three $n \times n$ matrices A,B and C. A matrix multiplication

$C = A \times B$ is as same as the following equations:

$$c_{ij} = \sum_{1 \le k \le n} a_{ik} b_{kj} \tag{2.2}$$

where a_{ij}, b_{ij} and c_{ij} denote the element in the ith-row and jth-column of matrix A, B and C respectively. Indeed, this is a matrix multiplication over the semi-ring $\{+, \times\}$. Now we define the operation $\star$:

$$(A \star B)_{ij} = \min_k \{a_{ik} + b_{kj}\} \tag{2.3}$$

One may see the analogue between equation 2.2 and equation 2.3. Given a length matrix W, the nth distance product W^n is defined as:

$$W^n = \underbrace{W \star W \star \cdots \star W \star W}_{(n-1) \quad \text{times} \star}$$

The product W^n is a matrix with each elements w_{ij} equals to the distance between v_i and v_j. However, the distance matrix multiplication unfortunately have a $\Omega(n^3)$ lower bound, even with only minimum and sum are allowed[20].

Fredman [13] discovered that $O(n^{2.5})$ comparison and addition are sufficient to solve APSP problem, and devised a $O(n^3 (\log\log n)^{1/3} / (\log n)^{1/3})$-time algorithm by preprocessing. Fredman's algorithm was based on distance matrix and for nonegative weighted cases only. Takaoka[30] improved the time bound to $O(n^3 \sqrt{\log\log n / \log n})$. Han [16] presented an $O(n^3 (\log\log n / \log n)^{5/7})$-time algorithm for directed real weighted graph. Then Takaoka [31, 32] pushed down the time bound twice in a short period of time, to $O(n^3 (\log\log n)^2 / \log n)$ and $O(n^3 \log\log n / \log n)$. Zwick [36] gave a further improvement, a $O(n^3 \sqrt{\log\log n} / \log n)$ algorithm. Inspired by techniques used in computational geometry, Chan [4] devised an algorithm which runs in the time of $O(n^3 / \log n)$. Han [17] then devised the state-of-the-art algorithm running in $O(n^3 (\log\log n / \log n)^{5/4})$ time. Recently, Chan [5] presented a $O(n^3 \log^3 \log n / \log^2 n)$-time algorithm but for real-weighted dense graphs only

Time	References	Year
$O(mn + n^2 \log n)$	Dijkstra [9]	1959
$O(n^3)$	Flody/Warshall [7]	1962
$O(n^2(\log n)^2)$	Spira [29]	1973
$O(n^3(\log\log n)^{1/3}/(\log n)^{1/3})$	Fredman [13]	1976
$O(mn + n^2 \log n)$	Johnson [18]	1977
$O(m^*n + n^2 \log n)$	Karger [19]	1991
$O(n^3\sqrt{\log\log n/\log n})$	Takaoka[30]	1992
$O(m^*n + n^2 \log n)$	McGeoch [23]	1995
$O(n^3(\log\log n/\log n)^{5/7})$	Han[16]	2004
$O(n^3(\log\log n)^2/\log n)$	Takaoka [31]	2004
$O(n^3 \log\log n/\log n)$	Takaoka [32]	2005
$O(n^3\sqrt{\log\log n}/\log n)$	Zwick [36]	2006
$O(n^3/\log n)$	Chan [4]	2008
$O(n^3(\log\log n/\log n)^{5/4})$	Han [17]	2008
$O(n^3 \log^3\log n/\log^2 n)$	Chan [5]	2010

Table 2.1: APSP algorithms for general dense real-weighted graphs. "Year" refers to the published year of the citation.

Let $O(n^\omega)$ denote the total algebraic operations used for a multiplication between two $n \times n$ matrices. This technique is called the *fast matrix multiplication* and the best known upper bound for ω is $\omega < 2.376$ [6]. The only lower bound available on ω is the naive lower bound $\omega \geq 2$.

Inspired by Yuval [33], Alon et al. [3] devised a $\tilde{O}(n^{\frac{3+\omega}{2}})$-time algorithm utilizing the fast matrix multiplication. His algorithm was limited to directed graphs with possible edge weights in $\{-1, 0, 1\}$. Galil and Margalit [14, 15] and Seidel [27] have obtained $\tilde{O}(n^\omega)$-time algorithms for unweighted undirected graphs. Zwick gave a $O(n^{2.575})$ time algorithm.[35].

When extending to the graph with edges weighted in the range of $\{0, 1, \ldots, M\}$, the complexity of algorithms devised by Galil and Margalit were $\tilde{O}(M^{\omega+1/2}n^\omega)$. Shoshan and Zwick [28] obtained an improved time bound of $\tilde{O}(Mn^\omega)$.

Table 2.1 and Table 2.2 summarize the results we surveyed. Table 2.1 are mainly excerpted from the Table 1.1 of [5] but some additional citation are updated by us. Table 2.2 demonstrates the algorithms based on the fast matrix multiplication.

Time	References	Year	Limitation of edge weighted
$\tilde{O}(n^{\omega})$	Seidel [27]	1992	unweighted and undirected
$\tilde{O}(n^{3+\omega/2})$	Alon [3]	1997	$\{-1, 0, 1\}$
$\tilde{O}(n^{\omega})$	Galil and Margalit [14, 15]	1997	unweighted and undirected
$\tilde{O}(M^{\omega+1/2}n^{\omega})$	Galil and Margalit [14, 15]	1997	$\{0, 1, \ldots, M\}$
$\tilde{O}(Mn^{\omega})$	Shoshan and Zwick [28]	1999	$\{0, 1, \ldots, M\}$
$\tilde{O}(n^{2.575})$	Zwick [35]	2002	$\{-1, 0, 1\}$

Table 2.2: APSP algorithms based on the fast matrix multiplication.
"Year" refers to the published year of the citation.

2.2.3 Approximate Results

We summarize the approximation results of APSP problem into two types. Let the approximate distance denoted as $\widehat{d}(u, v)$. We say $\widehat{d}(u, v)$ is *stretch* t if and only if $d(u, v) \leq \widehat{d}(u, v) \leq t \cdot d(u, v)$. We say $\widehat{d}(u, v)$ *surplus* t if and only if $d(u, v) \leq \widehat{d}(u, v) \leq d(u, v) + t$.

Zwick [35] gave an stretch $1+\epsilon$ algorithm approximating all distance in directed graph in the time of $\tilde{O}((n^{\omega}/\epsilon)\log(W/\epsilon))$, where W was the largest edge weight in the graph. Aingworth et al. [1] gave a surplus 2 algorithm approximating APSP problem in the time of $O(n^{5/2}(\log n)^{1/2})$. Dor et al. [10] presented an surplus 2 algorithm in the time of $\tilde{O}(n^{3/2}m^{1/2})$ and $\tilde{O}(n^{7/3})$.

2.2.4 Diameters

Here we state some prior results regarding diameter finding techniques. Once the All-Pairs Shortest Paths problem is solved then it is possible to find diameter by searching the maximum value among the all-pairs distance. There might be more than one diameter in the same graph if all of them have a equally maximum length.

Aingworth et al. [1] gave a $2/3$-approximation algorithm finding diameter in a weighted directed graph and it runs in the time of $O(m(n \log n)^{1/2} + n^2 \log n)$.

Magnien et al. purposed [22] some lower and upper bounds of diameter, and evaluated those bounds with the real world network datasets. They considered the

Algorithm 2.2 Double-Sweep-Algorithm

```
Double-Sweep-Algorithm($G$)
    ▷ Input graph $G$
    ▷ Output the *Double-Sweep-Lower-Bound* of $G$
1   Randomly choose $r$ from $V(G)$
2   Lanuch Breadth-First-Search($r$) and let $a$ be the eccentricity of $r$
3   Lanuch Breadth-First-Search($a$) and let $b$ be the eccentricity of $a$
4   Return $d_G(a, b)$ by launching Breadth-First-Search ($a$)
```

unweighted graphs only. The following were lower bounds introduced.

Eccentricity The eccentricity gives a trivial bound $\mathbf{ecc}(v) \leq \mathbf{diam}(G) \leq 2 \cdot \mathbf{ecc}(v)$, which is computed in the time of $O(n + m)$.

Double Sweep Lower Bound See Algorithm 2.2. Since it computes a distance on the graph and diameter is the largest distance, it always serves as a lower bound.

Crescenzi et al. [8] purposed the *fringe upper bound* of diameters. The authors defined the *fringe* of some node u, denoted as $F(u)$, as the set of nodes $v \in V(G)$ such that $d(u, v) = \mathbf{ecc}(u)$. The fringe upper bound is computed in $|F(u)| + 3$ of Breadth-First-Search. Algorithm 2.3 describes how to compute it.

Algorithm 2.3 Fringe-Algorithm [8]

```
Fringe-Algorithm(G)
    ▷ Input: The input graph $G$
    ▷ Output: The fringe upper bound
 1  $\mathcal{P}(a \dots b) \leftarrow$ The path returned by Double-Sweep-Algorithm($G$)
 2  $u \leftarrow$ the halfway of $\mathcal{P}(a \dots b)$
 3  $T_u \leftarrow$ Breadth-First-Search($u$)
 4  $Max\text{-}B \leftarrow 0$
 5  if $|F(u)| > 1$
 6      then for each $z \in F(u)$
 7                  do Breadth-First-Search(z)
 8                      if $Max\text{-}B < \mathbf{ecc}(z)$
 9                         then $Max\text{-}B \leftarrow \mathbf{ecc}(z)$
10  if $Max\text{-}B = 2 \cdot \mathbf{ecc}(u) - 1$
11      then return $2 \cdot \mathbf{ecc}(u) - 1$
12  elseif $Max\text{-}B < 2 \cdot \mathbf{ecc}(u) - 1$
13      then return $2 \cdot \mathbf{ecc}(u) - 2$
14  else return $\mathbf{diam}(T_u)$
```

Though FRINGE-ALGORITHM has no theoretical guarantee to compute the exactly diameter, it reports very good estimates in practice. According to results of the experiments reported, it computes tight upper bounds which match the lower bounds in substantial cases.

Chapter 3

Clique Distance and a Straightforward Algorithm

In this chapter the precise definition is presented. Currently our problem is restricted to an undirected and unweighted graph, where any two given cliques are non-overlapped.

3.1 Problem Definition

An instance of the clique distance problem is comprised of an undirected, unweighted graph G and a set $\mathcal{C} = \{C_1, C_2, \ldots C_r\}$ collecting the given cliques. In addition, the total number of given cliques is $r = |\mathcal{C}|$. For each clique in $\mathcal{C}$, there are at least 3 nodes in it. Two degenerated cases are filtered out by the restriction. The first one is the single isolated nodes and the another is the subgraphs with only two nodes and one edge. We assume that every clique in $\mathcal{C}$ has no common node with any others, i.e. they are all non-overlapped. Without loss of generality take two cliques $C_i = \{u_1, u_2, \ldots u_s\}$ and $C_j = \{v_1, v_2, \ldots v_t\}$. Let the *clique distance* **cd** defined as:

$$\mathbf{cd}(C_i, C_j) = \min_{u_a \in C_i, v_b \in C_j} d(u_a, v_b)$$

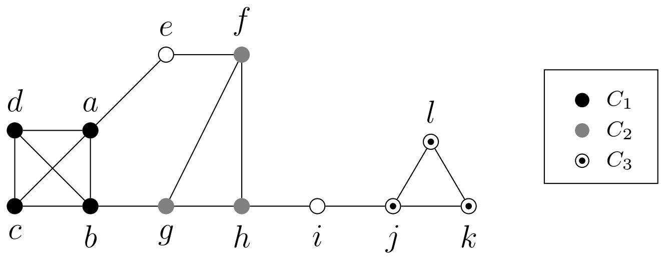


Figure 3.1: An Example of the Clique Distance Problem

and we call the shortest path between v_a and v_b as a *clique shortest path*. Consequently, the *clique diameter* of G is defined as the maximum length among all of the clique shortest paths on G.

Figure 3.1 illustrates an instance of clique distance problem. In the figure every clique is depicted in its own type of nodes. There are three cliques in the graph, $C_1 = \{a, b, c, d\}$, $C_2 = \{f, g, h\}$, and $C_3 = \{j, k, l\}$. Thus we have $\mathbf{cd}(C_1, C_2)$ equals to (b, g), and $\mathbf{cd}(C_2, C_3)$ equals to $\mathcal{P}(h, i, j)$. A clique shortest path may include any edges from the input graph except those belong to the starting or ending cliques. For example, $\mathbf{cd}(C_1, C_3) = \mathcal{P}(b, g, h, i, j)$, which includes the edge (g, h) from clique C_2.

3.2 A Straightforward Algorithm

Indeed, we intend to compute the shortest paths between some vertex u and v where the two located in two distinct, non-overlapping cliques. We may use breadth-first-search traversals to compute the shortest paths since we are restricted on unweighted graphs.

Our algorithm COMPUTE-CLIQUE-DISTANCE solves this problem by utilizing the BREADTH-FIRST-SEARCH algorithm. The details are presented in Algorithm 3.1. We are going to explain the data structures in our algorithm. The input graph is stored in the form of adjacent lists. The mapping from $V(G)$ to $\mathcal{C}$ is stored in the array *CLIQUE*. In other words, for each node v, $CLIQUE[v]$ is refereed to the clique containing v, or return NIL if v does not belong to any clique. Another variable $X = \{v | v \in C_i \quad C_i \in \mathcal{C}\}$ is a set consists of every node covered by the cliques in $\mathcal{C}$.

Algorithm 3.1 COMPUTE-CLIQUE-DISTANCE

```
COMPUTE-CLIQUE-DISTANCE($G, \mathcal{C}$)
      ▷ Input the graph $G$ and the given cliques list $\mathcal{C}$.
      ▷ Output the clique distances matrix $CD$.
 1    $r \leftarrow |\mathcal{C}|$ ,   $n \leftarrow |V(G)|$
 2    Initialize $CD$ as a $r \times r$ matrix with each element has the value $\infty$
 3    Initialize $D$ as $n \times n$ matrix with each element has the value $\infty$
 4    Initialize $\Pi$ as $n \times n$ matrix with each element has the value NIL
 5    for each $u$ in $X$
 6        do BREADTH-FIRST-SEARCH(G, $u$, $X$, $D$, $\Pi$)
 7    for each $C_i$ in $\mathcal{C}$
 8        do for each $C_j$ in $\mathcal{C}$
 9              do if $i = j$
10                    then $CD[i, j] \leftarrow 0$
11                    else  for $u \in C_i$
12                              do for $v \in C_j$
13                                    if $D[u, v] < CD[i, j]$
14                                      then $CD[i, j] \leftarrow D[u, v]$
15    return $CD$
```

Now we are going to prove the correctness of these algorithms. In the procedure COMPUTE-CLIQUE-DISTANCE, it initializes the global data structures. The initialization takes at worst $O(|X|) = O(n)$ of time. Then we perform BREADTH-FIRST-SEARCH starting from every vertex in X. After the algorithm terminated we have obtained the all-pairs shortest path for nodes in X. The algorithm finishes in the time of $O(|X|(n+m))$. However, the size of X could be n. The time complexity of Algorithm 3.1 is at worst $O(n^2+nm)$ and the space complexity is at worst $O(n^2)$.

After computing the clique distances we may find the clique distance by simply checking every element in D. This can be done in the $O(|X|^2)$ time. If the input graph is a complete graph K_n, the size of set X equals to n and we may need $O(n^2)$ time to check all of elements. So now we have an algorithm runs in $O(n^2 + nm)$ time and $O(n^2)$ space.

3.3 Some Improvements

In this section some additional lemmas are presented. We also demonstrate an approximate algorithm computing approximate clique distance in an additive error of one.

Fact 3.1 ([7]). *A tree which is the result of a breadth-first search traversal rooted at a, is a shortest path tree rooted at a.*

Lemma 3.1. *On an unweighted graph, assume two nodes u and v are located in the same clique. Take another arbitrary node x, then $|d(u,x) - d(v,x)| \leq 1$.*

Proof. Assume $d(u,x) > d(v,x)$ with no loss of generality. Assume

$$d(u,x) - d(v,x) \geq 2.$$

this implies

$$d(u,x) \geq d(v,x) + 2$$

If we take the path $\mathcal{P}(u, v, \ldots, x)$ of which its length is $1 + d(v,x)$, then we have $d(u,x) = d(v,x) + 1$, a contradiction. □

Lemma 3.2. *On an undirected, unweighted graph, every shortest path starting from a node in any clique C includes at most one edge in C.*

Proof. Assume there is a shortest path $\mathcal{P}$ contains a subpath $\mathcal{Q} = p(u_1, u_2, \ldots, u_x)$ where $u_1, u_2, \ldots, u_x$ are the vertices in clique C. We may reduce the length of $\mathcal{P}$ by replacing the subpath $\mathcal{Q}$ into one edge (u_1, u_x). Since $\mathcal{P}$ is not a shortest path, our assumption leads to a contradiction. Thus we are done. □

Lemma 3.3. *Let T be the tree returned by* Breadth-First-Search *starting by $u \in C_i$ and let w be the first node reached by the algorithm in C_j. Let $d_T(u,w)$ denote the length of the path joining u and w on T. Then we have:*

$$\mathbf{cd}(C_i, C_j) \leq d_T(u,w) \leq \mathbf{cd}(C_i, C_j) + 1$$

Proof. By definition the case such that $\mathbf{cd}(C_i, C_j) > d_T(u, w)$ is impossible. If the case such that $\mathbf{cd}(C_i, C_j) = d_T(u, w)$ or $\mathbf{cd}(C_i, C_j) = d_T(u, w) + 1$ are encountered, then we are done. This means either the node u or at least one of nodes $v \in C_i \setminus \{u\}$ is the starting points of the clique shortest path joining C_i and C_j.

For every node $x \in C_j$, by Lemma 3.1, $d(u, x) - \mathbf{cd}(C_i, C_j) \geq 2$ is true only if $x \neq w$ so $d_T(u, w) - \mathbf{cd}(C_i, C_j) > 1$ is impossible. □

Lemma 3.4. *Let T denote the breadth-first-tree returned by* Breadth-First-Search *starting by a node $u \in C_i$ and reaching w at clique C_j. If the shortest path joining u and w passes though the node $v \in C_i \setminus \{u\}$, then $d_T(v, w) = \mathbf{cd}(C_i, C_j)$.*

Proof. Clearly $d(u, w)$ is not equal to $d(v, w)$ since $u \neq v$. The length of every path joining u and w can not smaller than $1 + \mathbf{cd}(C_i, C_j)$, otherwise the Breadth-First-Search picked another routes as the shortest path. Since any subpath of a shortest path is also a shortest path, we have $d(v, w) = \mathbf{cd}(C_i, C_j)$. □

Summarize the above lemmas we devise Algorithm 3.2 and 3.3, which computes the approximate cliques distances. Algorithm 3.2, which is the procedures BFS-Revised, is slightly modified from the classic Breadth-First-Search algorithm.

We describe the variables in BFS-Revised. The variable *color*$[u]$ maintains the color of u to indicate whether the vertex u is discovered. If *color*$[u]$ = WHITE, this means none of children are reached via the operation. If *color*$[u]$ = GRAY, this means u itself is reached but some of its children may remain uncovered. If *color*$[u]$ = BLACK means the u and its children are discovered so we color u as BLACK. A distance matrix D is constructed to store the distances. In addition, the value of $D[u, v]$ is the distance from u to v. The variable $\Pi[u, v]$ contains the parent of v during the execution of the traversal starting from u. The number of cliques in the list $\mathcal{C}$ is $r = |\mathcal{C}|$. A clique distance matrix CD, which is a $r \times r$ matrix, is constructed to store the clique distance estimates during the execution of the algorithm.

Algorithm 3.2 BFS-Revised

```
BFS-Revised($G, s, X, D, \Pi, \mathcal{C}, CD$)
    ▷ $G$: The graph that operation perform on.
    ▷ $s$: The node whole operation starting with.
    ▷ $\mathcal{C}$: The given clique list.
    ▷ $D$: The distance matrix.
    ▷ $\Pi$: The predecessor matrix.
    ▷ $CD$: The clique distances matrix.
    ▷ $BRANCH$: Recording by which neighbor $s$ reached the newly discovered node.
 1  Initialize $BRANCH$ as an array with the default values NIL
 2  $C_s \leftarrow CLIQUE[s]$
 3  for each vertex $u \in V(G) - s$
 4      do $color[u] \leftarrow$ WHITE
 5         $D[s, u] \leftarrow \infty$
 6         $\Pi[s, u] \leftarrow$ NIL
 7  $color[u] \leftarrow$ GRAY
 8  $D[s, u] \leftarrow 0$
 9  $\Pi[s, u] \leftarrow$ NIL
10  $BRANCH[s] \leftarrow s$
11  $Q \leftarrow \emptyset$
12  Enqueue($Q$,$s$)
13  while $Q \neq \emptyset$
14      do $u \leftarrow$ Dequeue($Q$)
15         for each $w \in Adj[u]$
16             do if $color[w] = $ WHITE
17                   then $color[w] \leftarrow$ GRAY
18                        $D[s, w] \leftarrow D[s, u] + 1$
19                        $\Pi[s, w] \leftarrow u$
20                        if $w \in C_s$
21                           then $BRANCH[w] \leftarrow w$
22                           else $BRANCH[w] \leftarrow BRANCH[u]$
23                        if $w \in X$
24                           then $C_j \leftarrow CLIQUE[w]$
25                                $v \leftarrow BRANCH[w]$
26                                if $v \neq s$
27                                   then $CD[s, j] \leftarrow \min\{CD[s, j], D[v, w]\}$
28                                   else $CD[s, j] \leftarrow \min\{CD[s, j], D[s, w]\}$
29                        Enqueue($Q$, $w$)
30             $color[u] \leftarrow$ BLACK
```

Algorithm 3.3 CLIQUE-DISTANCE-APPROX

```
CLIQUE-DISTANCE-PRUNING($G, \mathcal{C}$)
    ▷ Input the graph $G$ and the given cliques list $\mathcal{C}$.
    ▷ Output $CD$ storing the approximate clique distances.
    $r \leftarrow |\mathcal{C}|, \quad n \leftarrow |V(G)|$
    Initialize $CD$ as a $r \times r$ matrix with each element has the value $\infty$
    Initialize $D$ as $n \times n$ matrix with each element has the value $\infty$
    Initialize $\Pi$ as $n \times n$ matrix with each element has the value NIL
    for $C_i \in \mathcal{C}$
          do Pick an arbitrary node $y$ from $V(C_i)$
              BFS-REVISED($G, y, X, D, \Pi, \mathcal{C}, CD$)
    return $CD$
```

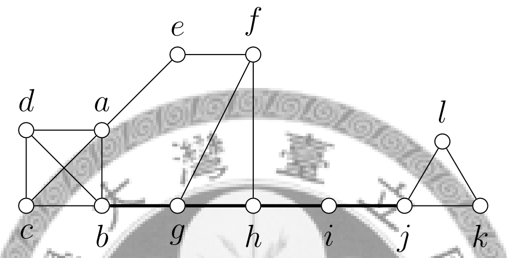


Figure 3.2: $\mathbf{cd}(C_1, C_3) = d(b, j)$ (drawing in thick lines)

The algorithm BFS-REVISED maintains a newly added variable *BRANCH*. Consider a BFS tree T computed by BFS-REVISED. Every node except s must have a parent node on T. If the operation BFS-REVISED starts from a node s in the clique C_s, then all of neighbor of s in the clique C_s are at the level one of the tree T (The node s is at level zero.) For every node w, the variable $BRANCH[w]$ refers the ancestor at the level one if s reaches w via a neighbor also in the clique C_s, or refers to s if s reaches w without passing through any neighbor belonging to C_s.

If the variable $v = BRANCH[w] \neq s$, we may reached a clique from s and the shortest path passes through one of neighbor of s in the clique C_s. By Lemma 3.4, the distance $D[v, w]$ is the clique distance. Otherwise we compare the distance $D[s, w]$ with the currently known clique distances estimation.

Note that Algorithm 3.2 computes only approximate results. To see how it failed to compute exact results, we provide the graphs showing in Figure 3.2 and

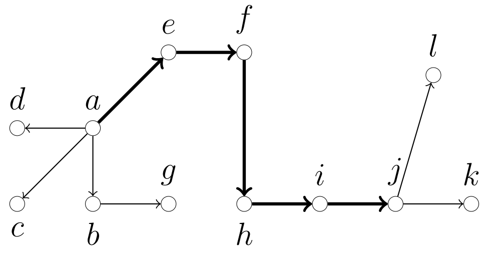


Figure 3.3: The Failed BFS-Tree Rooted at a

Figure 3.3 as examples. The correct clique distance $\mathbf{cd}(C_1, C_3)$, which is showing in Figure 3.2, is $d(b, j) = 4$. If the BFS-Revised starting from node a expands the children of e prior the expansion of b, BFS-Revised(a) reaches C_3 by node j without taking the correct clique shortest path. In this particular case Algorithm 3.2 return an approximate estimate with an additive error of one. Because Algorithm 3.2 expands nodes in arbitrary order, the failed cases are not preventable. So we conclude that Algorithm 3.2 computes the approximate clique distances with the additive error at most one.

Now we prove the correctness of our algorithms. Algorithm 3.3 is just a trivial loop. Algorithm 3.2 is a modified version of Breadth-First-Search. It traverses every node and computes the shortest path from starting node to every node in graph G. If a newly found clique is reached via a neighbor in the same clique of starting node, by Lemma 3.3 and Lemma 3.4, the correct clique distance is reported. Otherwise, we can not filter out the error cases.

The procedure BFS-Revised runs in the time of $O(m + n)$. The time complexity of Algorithm 3.3 is $O(r(m + n))$ since it perform at most r times of BFS-Revised, where r is the number of cliques in $\mathcal{C}$. Our algorithm performs better than the straightforward algorithm if the number of r is much smaller than the node number n.

Chapter 4

On Transformation to All-Pairs Shortest Paths Problem

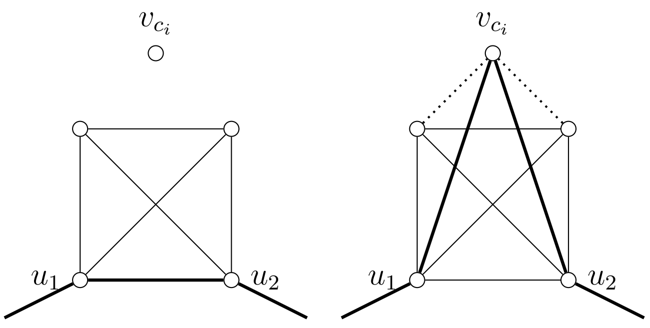
In this chapter a technique which transforms any instance of clique distances problem into an instance of APSP problem is reported. Transforming the clique distances problem into APSP problem is beneficial since we may utilize the algorithms established for APSP problem to solve the clique distance problem. However, we can not solve APSP problem by solving clique distance problem. By solving clique distance problem it is possible to obtain only a partial solution of APSP problem.

4.1 A Failed Attempt

Before introducing the final version construction, we demonstrate a failed attempt.

Definition 4.1 (Pitfall roof node)**.** For each clique $C_i = \{u_1, u_2, u_3, \ldots, u_{k-1}, u_k\}$ in $\mathcal{C}$, a newly created *roof node* v_{c_i} is inserted into $V(G)$. A roof nodes v_{c_i} are only adjacent to those nodes belong to the corresponding clique C_i and is disconnected to the rest of nodes (also disconnected from any other roof nodes). Then we put edges (v_{c_i}, u) for $u \in C_i$ where these edges are weighted in *zero*.

However, this design has a pitfall. Assume v_{c_i} is a roof node of clique C_i.



(a) Before Insertion (b) After Insertion

Figure 4.1: A Counterexample

There must be a path $\mathcal{P}(x, v_{c_i}, y)$ with its length smaller than any edges belonging to C_i. Then the shortest path found by APSP algorithms is the path including $\mathcal{P}(x, v_{c_i}, y)$. The newly found shortest path passes through the roof node v_{c_i} and is not a possible path on the originally graph. Therefore, the transformation can not fulfill our objectives.

4.2 Correction

So we must give weighed to the edges adjacent to the roof nodes.

Definition 4.2 (Roof node)**.** The construction steps remain unchanged. Instead of creating edges weighted in zero, a heavier edge weighted is given. We put edges (v_{c_i}, u_j) for $u_j \in C_i$. Those edges are weighted in $(w_{max} + 1)$ where $w_{max} = \max_{e \in E(G)} w(e)$. The set $\mathcal{R} = \{v_{c_i} | C_i \in \mathcal{C}\}$ contains all roof nodes on G.

Actually we may adopt smaller edge weights. Given a clique C_i and the corresponding roof nodes v_{c_i}. Let the maximum edge weight among the edges in C_i is w_{c_i}. Then the edges weight for those edges adjacent to the roof nodes are at least $\frac{1+w_{c_i}}{2}$ since this is enough to prevent roof nodes from being included by any shortest paths.

Lemma 4.1. *For all of pairs of clique C_i and C_j and the corresponding roof nodes v_{c_i} and v_{c_j}, we have the length $d(v_{c_i}, v_{c_j}) - 2 \times (w_{max} + 1)$ as the desired clique distance $\mathbf{cd}(C_i, C_j)$.*

Proof. By the definition 4.2 the roof nodes are constructed and inserted into G. After all the roof nodes are created, these operations result in an newly created graph G'. Because the negative edge weight is not allowed, it is possible to compute the all-pairs shortest path on G' by any algorithms solving APSP problem.

Assume the shortest path between v_{c_i} and v_{c_j} as $\mathcal{P}(v_{c_i}, s, \ldots, t, v_{c_j})$, and the node s and t belong to C_i and C_j. It must be the cases because any roof node is connected to every node in the corresponding clique only and is isolated from the rest of nodes in originally graph G. So the edge (v_{c_i}, s) must be taken. The edge (t, v_{c_j}) is also taken for the same reason.

We argue that the path $\mathcal{P}(s \ldots t)$ is the clique shortest path between C_i and C_j for the reason that the path $\mathcal{P}(v_{c_i}, s, \ldots, t, v_{c_j})$ is the shortest path between v_{c_i} and v_{c_j}. If the path between s and t are not the one with the minimal length, the shortest path algorithm must find some different node $s' \neq s$ or $t' \neq t$ where $d(s', t') < d(s, t)$. This contradicts our assumption. Since the cost of (v_{c_i}, s) and (t, v_{c_j}) are both $w_{max} + 1$ clearly the path length of $\mathbf{cd}(C_i, C_j) = d(v_{c_i}, v_{c_j}) - 2 \times (w_{max} + 1)$. □

After doing the transformation, we may compute the clique distances by computing the all-pairs shortest path among all of the roof nodes. Algorithm 4.2 illustrates the whole algorithm.

Algorithm 4.1 CLIQUE-DISTANCES-ROOF-NODES

```
CLIQUE-DISTANCES-ROOF-NODES($G, \mathcal{C}$)
     ▷ Input the graph $G$ and given cliques list $\mathcal{C}$
     ▷ Output the matrix $CD$ storing clique distances
 1   $r \leftarrow |\mathcal{C}|$
 2   Initialize $CD$ as a $r \times r$ matrix
 3   $\mathcal{R} \leftarrow \phi$
 4   for each $C_i \in \mathcal{C}$
 5        do Insert a roof node $v_{c_i}$ into $V(G)$
 6           Insert edge $(v_{c_i}, u)$ for every node $u \in C_i$
 7           $\mathcal{R} \leftarrow \mathcal{R} \cup \{v_{c_i}\}$
 8   for each $v_{c_i}$ in $\mathcal{R}$
 9        do Compute the shortest path tree rooted at $v_{c_i}$
10           $CD$[i, j] $\leftarrow d(v_{c_i}, v_{c_j})$ for any node $v_{c_j} \in \mathcal{R}$
11   return $CD$
```

The Lemma 4.1 offers the possibility to compute clique distance by any known algorithms solving APSP problem. Even though the nodes in the graph increased after we inserted those roof nodes, the insertion expands the set of nodes $V(G)$ slightly because we put a limitation on the smallest size of cliques, which is three of nodes. Let n denote the number of nodes in original graph G. We conclude that $|\mathcal{R}| = o(n)$ and the time complexity are still dominated by n. So the overall time complexity remains.

Note that the input graph is not restricted to unweighted graphs; Lemma 4.1 are safe to extend to weighted graphs. We may use the identical procedure to construct the roof nodes. The design of roof nodes also handle the cases when some of given cliques in $\mathcal{C}$ are overlapped. By definitions, the clique distances of overlapped cliques shrink down to zero immediately.

Chapter 5

Concluding Remarks

The clique diameter problem and clique distance problem provide insights about the distances of interconnection between every pair of communities. Even we solve an instance of clique distance problem, we could not solve any instance of APSP problem by the solution we obtained from clique distance problem.

Starting from a straightforward algorithm, our algorithm which runs in the time of $O(r(n+m))$ approximates clique distance in an additive error of one. If the number of cliques r is much smaller than the number of node n then our approximate algorithm runs faster than the straightforward algorithm.

We solve any instance of clique distance problem by transforming it into an instance of APSP problem. However, it is not possible to solve an APSP problem by solving a clique distance problem. After adding the roof nodes into inputed graph, we may reconstruct the clique distances from the shortest path joining the corresponding roof nodes. Any improvements on the APSP problem immediately improve on our problems.

We state the future works below. As we stated in the Chapter 1 there are different models of community structures. The first possible future work is to devise the algorithm for different model of community structures. For example, we may use the k-clique or k-club models. Since k-clique and k-club are more practical models comparing with cliques, the computation techniques based on them could capture much more precise community distances.

The second lane of future works is evaluating our algorithm on the real world datasets. Our algorithms are based on the Breadth-First-Search which has a good external-memory implementation [24] so our methods could have good performance in practice. In the future we may establish the experiments to evaluate and improve our algorithm.

Bibliography

[1] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999.

[2] Richard D. Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3:3–113, 1973.

[3] Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[4] Timothy Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50:236–243, 2008.

[5] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):2075–2089, 2010.

[6] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, Cambridge, MA, 2 edition, 2001.

[8] Pierluigi Crescenzi, Roberto Grossi, Claudio Imbrenda, Leonardo Lanzi, and Andrea Marino. Finding the diameter in real-world graphs - experimentally turning a lower bound into an upper bound. In Mark de Berg and Ulrich Meyer, editors, *ESA (1)*, volume 6346 of *Lecture Notes in Computer Science*, pages 302–313. Springer, 2010.

[9] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5):1740–1759, 2000.

[11] P. Erdős and A. Rényi. On random graphs. I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[12] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[13] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976.

[14] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Inf. Comput.*, 134(2):103–139, 1997.

[15] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *J. Comput. Syst. Sci.*, 54(2):243–254, 1997.

[16] Yijie Han. Improved algorithm for all pairs shortest paths. *Inf. Process. Lett.*, 91(5):245–250, 2004.

[17] Yijie Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest path. *Algorithmica*, 51(4):428–434, 2008.

[18] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, January 1977.

[19] D.R. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:560–568, 1991.

[20] L.R. Kerr. *The effect of algebraic structure on the computational complexity.* PhD thesis, Cornell University, Ithaca, N.Y., 1970.

[21] R. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15:169–190, 1950.

[22] Clémence Magnien, Matthieu Latapy, and Michel Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[23] Catherine C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[24] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear i/o. In Rolf H. Möhring and Rajeev Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer, 2002.

[25] Robert J. Mokken. Cliques, clubs and clans. *Quality &amp; Quantity*, 13(2):161–173, April 1979.

[26] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America*, 98(2):404–409, January 2001.

[27] Raimund Seidel. On the all-pairs-shortest-path problem. In *STOC*, pages 745–749. ACM, 1992.

[28] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *FOCS*, pages 605–615, 1999.

[29] Philip M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(\mathrm{n}^2 \log^2 \mathrm{n})$. *SIAM J. Comput.*, 2(1):28–32, 1973.

[30] Tadao Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195 – 199, 1992.

[31] Tadao Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In Kyung-Yong Chwa and J. Ian Munro, editors, *COCOON*,

volume 3106 of *Lecture Notes in Computer Science*, pages 278–289. Springer, 2004.

[32] Tadao Takaoka. An $O(n^3 \log\log n / \log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters*, 96(5):155 – 161, 2005.

[33] Gideon Yuval. An algorithm for finding all shortest paths using $n^{2.81}$ infinite-precision multiplications. *Inf. Process. Lett.*, 4(6):155–156, 1976.

[34] Uri Zwick. Exact and approximate distances in graphs - a survey. In Friedhelm Meyer auf der Heide, editor, *ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2001.

[35] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.

[36] Uri Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation*, volume 3341 of *Lecture Notes in Computer Science*, pages 841–843. Springer Berlin / Heidelberg, 2005.