# Computing Exact Bottleneck Distance on Random Point Sets

Jiacheng Ye

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Sharath Raghvendra, Chair
Lenwood S. Heath
Edward A. Fox

May 8, 2020
Blacksburg, Virginia

Keywords: bipartite graph, bottleneck matching, random points

# Computing Exact Bottleneck Distance on Random Point Sets

Jiacheng Ye

(ABSTRACT)

Given a complete bipartite graph on two sets of points $A, B$ where $|A| = |B| = n$, in a bottleneck matching problem, we want to find an one-to-one correspondence, also called a matching, that minimizes the length of its largest edge; the length of an edge is simply the Euclidean distance between its end-points. As an application, consider matching taxis to requests while minimizing the largest distance between any request to its matched taxi. The length of the largest edge (also called the *bottleneck distance*) has numerous applications in machine learning as well as topological data analysis. For any parameter $\delta > 0$, a $\delta$-*disc graph* will have $A$ and $B$ as the vertex set and will include an edge between any two points $a \in A$ and $b \in B$ provided the Euclidean distance between them is $\leq \delta$. One simple way to find the bottleneck distance is to determine the smallest value of $\delta$ for which the corresponding $\delta$-disc graph has a maximum matching of size $n$. Such a $\delta$ can be identified in a straight-forward way by a simple binary search in $O(\log n)$ time. Within each guess of the binary search, one can use the classical Hopcroft-Karp (HK-) algorithm to find a maximum cardinality matching in $O(m\sqrt{n})$ time where $m$ is the number of edges in the $\delta$-disc graph. In this thesis, we consider the case where $A$ and $B$ are points that are generated uniformly at random from a unit square. Instead of the HK-Algorithm, we implement a new algorithm by Lahn and Raghvendra (Symposium on Computational Geometry, 2019), and we call it the FAST-MATCH algorithm. Furthermore, we test the FAST-MATCH algorithm against a standard implementation of the HK-Algorithm and find the following.

- For a $\delta$-disc graph where $\delta$ is close to the bottleneck distance, the HK-algorithm seems to exhibit its worst case behaviour of executing $\approx \sqrt{n}$ iterations. This suggests that the problem that we consider may indeed be a hard instance for the HK-Algorithm.
- Similar to the HK-Algorithm, when $\delta$ is the bottleneck distance, we also see that the algorithm of Lahn and Raghvendra seems to exhibit its worst case behavior and executes approximately $n^{1/3}$ iterations indicating that this setting may also be a hard instance for the algorithm of Lahn and Raghvendra.
- We compare the two algorithms along various parameters and find substantial improvements ($\geq 1.5$ times faster) in actual execution times for $n \geq 10^6$.

# Computing Exact Bottleneck Distance on Random Point Sets

Jiacheng Ye

(GENERAL AUDIENCE ABSTRACT)

Consider the problem of matching taxis to an equal number of requests. While matching them, one objective is to minimize the largest distance between a request and its match. Finding such a matching is called the bottleneck matching problem. In addition, this optimization problem arises in topological data analysis as well as machine learning. In this thesis, I conduct an empirical analysis of a new algorithm, which is called the FAST-MATCH algorithm, to find the bottleneck matching. I find that, when a large input data is randomly generated from a unit square, the FAST-MATCH algorithm performs substantially faster than the classical methods.

# Dedication

*To my family...*

# Acknowledgments

First, I would like to express my sincere gratitude to my thesis advisor, Dr. Sharath Raghvendra. His guidance helped me in my research work and writing. This thesis would not be possible without his constant motivation and patience.

Second, I would like to thank the rest of my committee members: Dr. Edward A. Fox and Dr. Lenwood S. Heath. I appreciate their insightful comments about my thesis and hard questions in my defense.

My sincere thanks also goes to my friends: Nathaniel Lahn, Rachita Sowle and Kaiyi Zhang. They supported and inspired me in the past semesters.

Last but not the least, I would like to thank my parents Yongbing Ye and Xiaohua Shan, for supporting me throughout my life.

# Contents

# Chapter 1

# Introduction

Consider two sets $A$ and $B$ of $n$ points each in a 2-dimensional plane. Suppose we have a graph $G(V, E)$, $V$ is the set of vertices, and $V = A \cup B$ , and each vertex represents a point; $E$ is the set of edges in $G$, and $E \subseteq A \times B$. Let $a \in A$ and $b \in B$ be two points from different groups. Let the cost $c(a, b)$ of an edge $(a, b)$ be the Euclidean distance between the points $a$ and $b$, i.e., $c(a, b) = \|a-b\|$. A *matching* of size $k$ is a set of $k$ vertex-disjoint edges. A perfect matching is a matching of size $n$. Let the *bottleneck weight* or the *bottleneck distance* of any matching $M$ be simply the weight of its largest edge. The *bottleneck matching problem* is the problem of finding a perfect matching with the smallest bottleneck cost.

**$p$-Wasserstein distance:** For any $p \geq 1$, the $p$-Wasserstein distance is simply the cost of the matching $M$ that minimizes the following objective function: $(\sum_{(a,b)\in M} \|a - b\|^p)^{1/p}$. Wasserstein distance is often used as a measure of similarity between discrete distributions and there are also some applications about Wasserstein distance in machine learning, including Generative Adversarial Network [2], which is a neural network architecture that generate an new, synthetic instances of data that simulate the real data that we feed to the neural network. Note that as $p$ tends to $\infty$, the $p$-Wasserstein distance approaches the bottleneck distance, so the bottleneck distance is also equivalent to $\infty$-Wasserstein distance.

We highlight two applications of bottleneck distance.

- Consider a ride-sharing application like Uber that receives the longitude and latitude of all the requests that arrived in the past few minutes. It also has the locations of all the taxis that are available and would like to match each request to one available taxi. One objective for them would be to come up with an assignment that minimizes its largest cost edge; here cost can be measured as either the distance traveled or the response time.

- Topological Data Analysis (TDA) is an emerging field in data analysis where one comprehends data by identifying and interpreting the features of the shape it forms. The shape of a point set at a scale $\delta > 0$ is defined by taking the union of discs of radius $\delta$ around these points. We obtain a sequence of shapes as we increase $\delta$ from 0 to $\infty$. Any topological feature (for instance, a hole in the shape) is formed at some value of $\delta$, say $x$ and gets filled at some value of $\delta$, say $y$. One can create the so-called persistence diagram of the point set by mapping each of the features to the point (x,y) denoting the time of birth and death of the feature [12]. To compare the similarity of two point clouds, one can use the bottleneck distances to match their features.

There are several other applications of bottleneck matching; see for instance applications in Robotics [4], Geographic Information Science [5], Geoinformatics [7], and Bioinformatics [6].

## 1.1 Related Work

### 1.1.1 Arbitrary Graphs

For arbitrary unweighted bipartite graph $G(V, E)$, where $|V| = n$, $|E| = m$ and $V = A \cup B$, Ford and Fulkerson's algorithm can compute its maximum matching by iteratively

searching for an augmenting path in each iteration [9]. The time complexity of Ford and Fulkerson's algorithm is $O(mn)$. Hopcroft-Karp's (HK-) algorithm [11] improves upon Ford and Fulkerson's algorithm by searching a maximal set of vertex-disjoint shortest augmenting paths in each iteration, which reduces the total number of iterations from $n$ to $\sqrt{n}$. As a result, the HK-Algorithm runs in $O(m\sqrt{n})$ time. For a parameter $0 < \lambda \leq 1$ and a graph that has an easily computable balanced vertex separator of size $n^{\lambda}$, Lahn and Raghvendra [16] presented a new approach to find a maximum cardinality matching in $O(mn^{\frac{\lambda}{1+\lambda}})$, where $\lambda$ is a parameter of the graph. Their approach improves upon the HK-Algorithm [11] when $\lambda \in [1/2, 1]$.

A graph $G(V, E)$ is a weighted graph if every edge $e$ in the graph also has a weight, or cost, $w(e)$. In a weighted matching problem, we wish to compute a matching while minimizing or maximizing an objective function that depends on the costs. The *assignment problem* also called the minimum-cost matching problem, for instance, aims to compute the matching that minimizes the sum of the cost of its edges. One of the earliest works about the assignment problem is the Hungarian algorithm [14], which has the time complexity of $O(|V|^3)$. For a general bipartite graph with integral edge costs, the Gabow-Tarjan algorithm [10] computes the minimum-cost matching with the time complexity of $O(\sqrt{n}m \log(nC))$, where $C$ denotes the largest magnitude of cost. For a planar bipartite graph where each edge has an integer edge cost, Asathulla et al. [3] presented an $\tilde{O}(n^{4/3} \log(nC))$ algorithm that computes minimum-cost perfect matching. For $K_h$ minor-free graphs with $h = O(1)$ and integer edge costs, Lahn and Raghvendra [15] presented an $\tilde{O}(n^{7/5} \log(nC))$ time algorithm to compute a minimum-cost maximum cardinality matching.

### 1.1.2 Geometric Graphs

Given a graph in geometric settings, vertices are points in a fixed $d$-dimensional space. The weight, or cost, of an edge between $a \in A$ and $b \in B$ is $\|a-b\|^p$, where $p \geq 1$ is a fixed integer, and $\|a-b\|$ is the Euclidean distance between $a$ and $b$. The weight of a matching $M$ is defined by $\left(\sum_{(a,b)\in M} \|a-b\|^p\right)^{1/p}$, which is the $p$-Wasserstein distance. We are interested in computing a perfect matching with the minimum weight for any fixed $p \geq 1$. When $p = 1$, the problem is called the Euclidean bipartite matching problem. The Euclidean bipartite matching problem can be computed in $\tilde{O}(n^{3/2+\delta})$ time for an arbitrary small $\delta > 0$ [19]. When $p = \infty$, we try to find a perfect matching that minimizes its largest weight edge, and the problem is called the bottleneck matching problem. In the work of Efrat, Itai and Katz [1], they show the following. Suppose there exists a dynamic nearest neighbor search data structure $D_r(S)$ that stores the point set $S$ and the radius $r$, and if given a point $q \in \mathbb{R}^2$, the cost of returning a point $s \in S$ s.t. $d(q,s) < r$ is bounded by $T(|S|)$. Hopcraft-Karp Algorithm with such a data structure can be implemented in $O(n^{3/2}T(n))$ on any $\delta$-disc graph containing $2n$ points. In the planar case, Efrat *et al.* show we can construct such a data structure in $O(n\log(n))$ time, and $T(n) = O(\log(n))$. Therefore, the time complexity of the Hopcroft-Karp's algorithm is improved to $O(n^{3/2}\log(n))$, and the bottleneck matching problem in a plane can be solved in $\tilde{O}(n^{3/2})$ time.

### 1.1.3 Approximation Algorithms in Geometric Settings

There are several matching algorithms that compute a matching with a cost that is approximately optimal. For instance, for the Euclidean bipartite matching problem in $d$-dimensional space, Sharathkumar and Agarwal [20] present an $\epsilon$-approximation algorithm, for $0 \leq \epsilon \leq 1$, which means that the matching returned by their algorithm has a cost of $\beta < (1+\epsilon)\beta^*$,

where $\beta^*$ is the cost of the optimal matching. Their algorithm executes in $\tilde{O}(n/\epsilon^d)$, where $0 < \epsilon < 1$. For the bottleneck matching problem, Lahn and Raghvendra [16] present an algorithm that can compute a $\epsilon$-approximate bottleneck matching in $\tilde{O}(n^{4/3}/\epsilon^4)$ time.

Another important problem in geometric settings is the geometric transportation problem. In geometric transportation problem, we are given a set of points $P$ in a $d$-dimensional Euclidean space. and each point $p$ carries $\mu(p)$ units of supply, where $\mu(p)$ is a positive or negative integer, and $\sum_{p \in P} \mu(p) = 0$. The goal is to find a flow that transport all positive supplies from points to points carrying negative supplies, and also minimize the total distance traveled by transportation. The geometric transportation problem is a generalization of matching problem. Khesin et al. [13] presented a $(1 + \epsilon)$-approximation algorithm for geometric transportation problem in time near-linear in $n$, and polynomial in $\epsilon^{-1}$ and in the logarithm of the total supply.

## 1.2 Bottleneck Distance via $\delta$-disc graph

For a parameter $\delta > 0$, a $\delta$-disc graph is a bipartite graph that connects $a \in A$ to $b \in B$, where $A$ and $B$ are two groups of vertices in the graph, with an edge if and only if $\|a - b\|$ is at most $\delta$. The $\delta$-disc graph is useful to model the topology of wireless communication network, where each device has the same and fixed transmission radius, and two devices can only communicate if the distance between them is with the transmission radius.

We can solve the bottleneck matching problem by simply finding the smallest $\delta$ such that the corresponding $\delta$-disc graph has a perfect matching. The Bottleneck matching problem, therefore, can be solved using algorithms that compute maximum cardinality matching. Since there are at most $n^2$ possible bottleneck distances, we need compute maximum cardinality matching of at most $n^2$ graphs to know the bottleneck distance. Efrat, Itai and Katz's

work [1] shows a better approach: they use the binary search to find the bottleneck distance, and we only need to compute maximum cardinality matching for $O(\log(n))$ graphs. Therefore, the efficiency of solving a bottleneck matching problem is proportional to efficiency of the algorithm that we choose to compute the maximum cardinality matching.

In this thesis, we consider the problem of computing the exact bottleneck matching when $A$ and $B$ are points are chosen uniformly at randomly from a unit square. I implement a new exact algorithm for the bottleneck matching problem. This algorithm can be seen as an adaptation of the $\epsilon$-approximation algorithm for the bottleneck matching by Lahn and Raghvendra [16] which runs in $O(n^{4/3} \log n)$ time. In contrast, the classical HK-Algorithm based approach takes $O(n^{3/2} \log n)$ time. We refer to this new algorithm as the FAST-MATCH algorithm.

## 1.3 Results

There are three objectives in our research. The first one is to implement and test the FAST-MATCH algorithm that computes the maximum cardinality matching and use it to find the bottleneck distance. Our objective it to analyze the experimental behavior of the FAST-MATCH algorithm when the input points are chosen uniformly at random from a unit square. The second objective is to compare the FAST-MATCH algorithm with a classical HK-Algorithm based approach and compare its performance with the performance of FAST-MATCH algorithm. Our final objective is to understand the empirical performance of Hopcroft Karp's algorithm for the case where points are chosen uniformly at random from a unit square.

Through our experiments on points chosen uniformly at random from a unit square, we find that the FAST-MATCH algorithm based approach outperforms the approach based

on Hopcroft-Karp's algorithm for the bottleneck matching problem: for datasets with more than $10^6$ points, the FAST-MATCH algorithm based approach runs 1.5 times faster than the approach based on the Hopcroft-Karp's algorithm. We also find that the approach based on Hopcroft-Karp's algorithm seems to exhibit its worst-case behavior when points are chosen uniformly at random from a unit-square. More precisely, when $\delta$ is the bottleneck distance, an execution of the HK-Algorithm on $\delta$-disc graph seems to take the worst case $\Omega(\sqrt{n})$ iterations.

## 1.4 Thesis Outline

In Chapter 2, we present how to efficiently construct and store a $\delta$-disc graph. In Chapter 3, we present details of the Hopcroft-Karp algorithm. In Chapter 4, we present the FAST-MATCH algorithm along with its efficient implementation details. In Chapter 5, we describe the experimental setup, and analyze the results of our experiments and conclude.

# Chapter 2

# Generating the $\delta$-disc Graph Efficiently

Before we describe the FAST-MATCH algorithm for computing the maximum cardinality matching problem, we need to discuss how the $\delta$-disc graph is generated and stored. There are two different ways to do this and we discuss this next.

1) *Brute Force Approach:* One obvious way to generate the graph is to store only those edges of the complete graph explicitly to see if they are within a distance of at most $\delta$. This process has an execution time of $\Theta(n^2)$ which will be too slow. In this chapter, we describe a rather simple procedure that allows the computation of the edges in the graph in time proportional to the number of edges in the graph.

2) *Dynamic Data Structure Approach:* We note that the best theoretical algorithm for bottleneck matching does not store the $\delta$-disc graph. Instead, it implicitly computes the edges of the graph as and when needed. We begin by noting that the classical HK-Algorithm relies heavily on conducting BFS and DFS on the $\delta$-disc graph. An atomic operation in both BFS and DFS is to identify an unexplored neighbor in the $\delta$-disc graph of the 'current' vertex. One can do this by simply querying a dynamic nearest neighbor data structure that stores all the unexplored nodes and checking if this neighbor is within a distance $\delta$. Dynamic nearest neighbor data structures are

quite challenging to implement and have very large constants making this technique impractical. Practitioners have used KD-Trees to build practical dynamic nearest neighbor structures that gives good practical results [12].

Given that $A$ and $B$ are chosen uniformly at random, KD-Tree will start to resemble a quad-tree in its geometry. Motivated by this observation, we present a rather simple approach to compute and store the $\delta$-disc graph next.

## 2.1 Compute the $\delta$-disc graph

Datasets in our experiments contain points that are randomly generated in a unit square. The intuitive idea of our approach is to use a grid to partition the bounding square into smaller squares with side-length of $\delta$. To compute the edges of the $\delta$-disc graph that are incident on an arbitrary point $p$, we only need to compute distance between $p$ and points that are in the same square as $p$, or the squares that share their boundary with $p$'s square. This is because the shortest distance between two points in non-adjacent squares is greater than $\delta$ and so, we can ignore squares that are not adjacent to $p$'s square. We do not need to generate all the squares explicitly. Instead, we can assign points to the correct squares by just sorting.

Figure 2.1 shows an example: a bounding square is partitioned into 25 $\delta \times \delta$ smaller squares. Suppose we want to compute edges for points in the pink square. In the brute-force approach, we need to compute distances between points in the pink square and all points in the entire bounding square. However, for our approach, we only need to consider points in the pink and yellow squares. The shortest distance between the pink square and an arbitrary green square is greater than $\delta$, so we can just ignore them.

In my implementation, I assign each point to the corresponding square implicitly. Suppose we

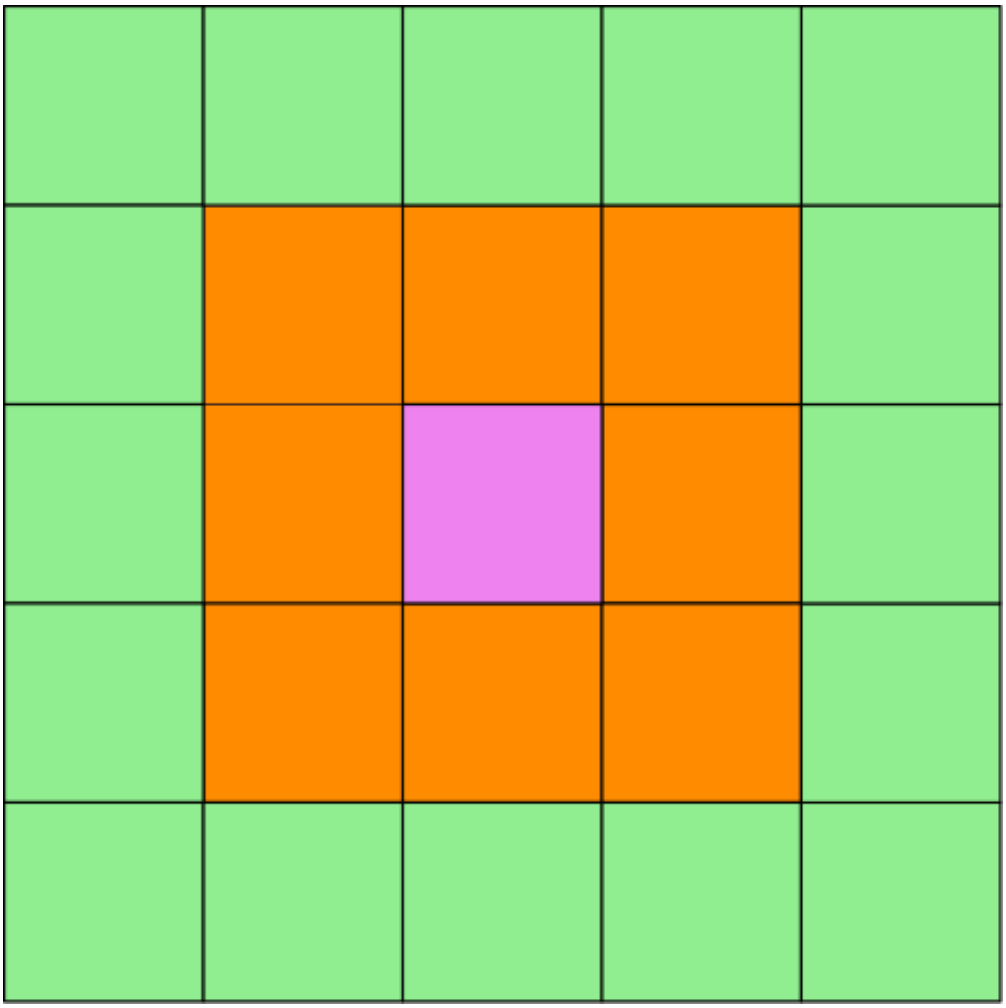

Figure 2.1: An Example

have points in a 2-dimensional space, and all points are stored in a list. We first sort all points based on their $x$ coordinates, and get a sorted list of points. Then we split the bounding square to $\frac{1}{\delta}$ columns with the width of $\delta$ (1 is the side length of the bounding square), and assign each point to the corresponding column. This can be done implicitly by partitioning the sorted list using indexes. For each column, which actually is just a continuous interval in the sorted list, we sort points in this column based on their $y$ coordinates. Then we do the same thing: we split this column to $\frac{1}{\delta}$ rows, and each row has the height of $\delta$. Each point in this column is assigned to the corresponding row, and both width and height of such

"row" are $\delta$, so each point is actually assign to the square we want. Because what we do is just sorting lists where total number of vertices is $2n$ (each vertex is sorted twice), the total time complexity of assigning vertices to the corresponding squares is $O(n\log(n))$. On the other hand, columns and rows are constructed implicitly by splitting the list using indexes, therefore space complexity is just $O(n)$. This approach can be easily extended to a higher dimension.

**Algorithm 1** Partition

Input: $P = \{p_1, p_2, ...\}$ is the list that stores points
Input: $ax$ is an indicator for the axis, or dimension
Sort $P$ by ax-coordinate of points, and we call the new list $P_{ax}$
Suppose $P_{ax} = \{pax_1, pax_2, ...\}$
Let $currLevel = \left\lfloor \frac{pax_1}{\delta} \right\rfloor$
Let *boundary* be an new empty list
Add *currLevel* to *boundary*
**for** $i \in \{2, 3, 4...\}$ **do**
    $level = \left\lfloor \frac{pax_i}{\delta} \right\rfloor$
    **if** $level > currLevel$ **then**
        Add *level* to *boundary*
        $currLevel \leftarrow level$
    **end if**
**end for**
return *boundary*

Given a list $P$, we define $P[i : j]$ to be the sub-list of $P$ that starts with the index $i$, and ends with the index $j$ (not include $P[j]$).

Suppose when we search for a square of a point, we have already updated the list $P$ where we store points, and got the correct list of *columns* and *rows*, which are returned by the Algorithm 2.

**Algorithm 2** Generate $\delta$-disc graph in 2-dimensional case

Input: $P = \{p_1, p_2, ...\}$ is the list that stores points
Let $columns = partition(P, axis_x)$
Let $head = c_1$
Let $rows$ be an empty list
**for** $i \in \{2, 3, 4...length(columns)\}$ **do**
    $tail = c_i$
    Let $P_i = P[head : tail]$
    $row_i = partition(P_i, axis_y)$
    Add $row_i$ to $rows$
**end for**
return $columns$, $rows$

**Algorithm 3** Search for a square

Input: A point $p = (p.x, p.y)$
Input: $P = \{p_1, p_2, ...\}$ is the list that stores points
Input: $columns$ and $rows$
Use binary search in $columns$, we can find two indexes $c_i, c_{i+1}$ s.t. $p \in P[c_i : c_{i+1}]$
Use binary search in $rows[i+1]$, we can find two indexes $r_j, r_{j+1}$ s.t. $p \in P[c_i : c_{i+1}][r_j : r_{j+1}]$
$P[c_i : c_{i+1}][r_j : r_{j+1}]$ is the square the $p$ belongs to (It is a set of points).
return $P[c_i : c_{i+1}][r_j : r_{j+1}]$

## 2.2 Improvement in practice

In $d$-dimensional case, the new approach has an expected time complexity of $O\big(\frac{\delta^d \times n^2}{S^d}\big)$, where $S$ is the side-length of the bounding square. We run experiments to compare our approach to the brute-force approach when $\delta$ is the bottleneck distance. In experiments, for each number of vertices in $\{100, 1000, 5000, 10000\}$, we generate 10 datasets where points are randomly generated in a bounding square, and for each dataset we generate a $\delta$-disc graph with $\delta =$ the bottleneck distance, and record the running time of generating the graph. Figure 2.2 compares the running times of the two approaches, and the $x$-axis represents the number of vertices in the graph, and the $y$-axis represents the average running time to construct the $\delta$-disc graph. Our approach is substantially faster in generating a $\delta$-disc graph when $\delta$ is the bottleneck distance.

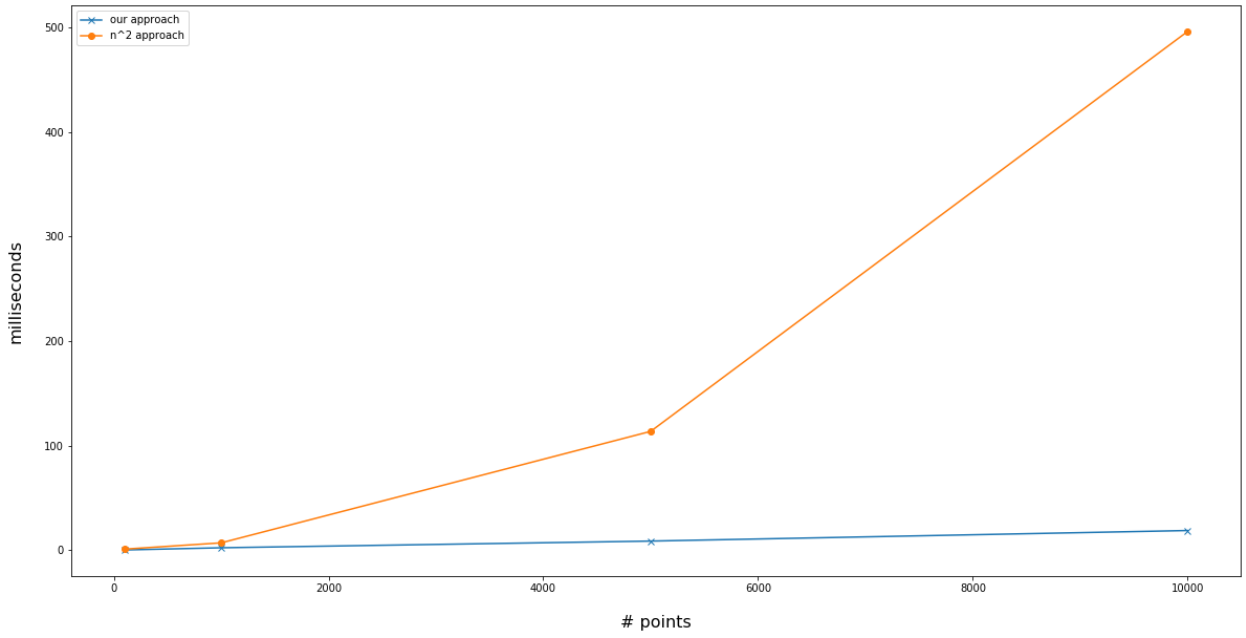


Figure 2.2: The new approach v.s. the $O(n^2)$ approach

# Chapter 3

# Hopcroft-Karp Algorithm

The Hoprcraft-Karp(HK-) algorithm is a classical algorithm that takes a bipartite graph $G(V, E)$ as the input and outputs a maximum cardinality matching $M$. Its time complexity is $O(m\sqrt{n})$, where $m$ is the number of edges and $n$ is the number of vertices in the graph. In this chapter, I will first give some background information needed to understand the matching algorithms presented in this thesis. Then, I will describe the Hopcroft-Karp's algorithm, and give a brief analysis of its complexity. At the end, I will discuss its empirical performance of the HK-Algorithm based approach for computing the bottleneck matching.

## 3.1 Background Information

We begin by introducing basic concepts of matching. Using these concepts, we describe the HK-Algorithm. Consider an arbitrary bipartite graph $G(V, E)$, $V = A \cup B$, $E \subseteq A \times B$, $|A| = |B|$, $|V| = n$, $|E| = m$. We define the edge between arbitrary $v_1, v_2 \in V$ as $e(v_1, v_2)$. If $G(V, E)$ is a weighted graph, $\exists w : E \to \mathbb{R}$, and the weight of the edge $e \in E$ is $w(e)$.

For a directed graph, we use $e(a, b)$ to denote an edge directed from $a$ to $b$. Note that, for a directed graph $e(a, b)$ and $e(b, a)$ denote two different edges. On the other hand, for an undirected graph, $e(a, b) = e(b, a)$.

### 3.1.1 Matching

For an arbitrary unweighted bipartite graph $G$, let $M \subseteq E$, $M$ is a matching if and only if the edges in $M$ are vertex-disjoint, which means $\forall e_1, e_2 \in M$ and $e_1$ does not share any common vertex with $e_2$. The *cardinality*, or *size* of $M$ is the number of edges in $M$, i.e., $|M|$. $M$ is perfect matching if $|M| = |A| = |B|$. $M$ is a maximum cardinality matching if $|M| = \max(|M'|)$, $\forall$ matching $M'$ in $G$. If the graph is weighted, the cost of a matching $M$ is $\sum_{e \in M} w(e)$.

### 3.1.2 Augmenting Path

Given any matching $M$, let $A_F$ and $B_F$ denote the vertices of $A$ and $B$ that are not matched in $M$. We refer to these vertices as *free* vertices. An alternating path $P$ is simply a path that alternates between edges that are in the matching and those that are not in the matching. An *augmenting path* is an alternating path that starts and ends at a free vertex. We define the length of $P$ as the number of edges in $P$.

We can augment a matching $M$ along an augmenting path by updating the matching as follows: $M \leftarrow M \oplus P$, here $\oplus$ denotes the symmetric difference. It is easy to see that augmenting a matching along $P$ increases the size of the matching $M$ by 1. Many matching algorithms, including those studied in this thesis, iteratively find and augment the matching along the augmenting path until we arrive at a maximum matching.

### 3.1.3 Residual graph

Given a bipartite graph $G(V, E)$, and $V = A \cup B$, and a matching $M$, we can create a residual graph $G'$ w.r.t $G$ in the following way: $G'$ is a directed graph, and contains the same set of

vertices $V$ as $G$; $\forall e(a,b) \in E$ where $a \in A$ and $b \in B$, if $e \in M$, there is an edge $e'$ that is directed from $b$ to $a$ in $G'$, and if $e \notin M$, $e'$ is directed from $a$ to $b$ in $G'$. Residual graph makes it easy to search for augmenting paths, because in a residual graph, any directed path from a free vertex $a \in A$ to a free vertex $b \in B$ is an augmenting path.

### 3.1.4 Ford-Fulkerson Algorithm

Before we describe the Hopcroft-Karp's algorithm, we describe the simpler Ford-Fulkerson's algorithm. The Ford-Fulkerson algorithm is an algorithm that solves the max-flow min-cut problem, and the maximum cardinality matching problem in bipartite graph can be transformed to a max-flow min-cut problem. In the Ford-Fulkerson's algorithm, we first create a residual graph $G'(V', E')$ w.r.t the input graph $G$ and the current matching $M$. In $G'$ we create two auxiliary nodes, $s$ and $t$, and they don't participate in matching. We generate directed edges in $G'$ from $s$ to all free vertices in $A$, and also generate directed edges in $G'$ from all free vertices in $B$ to $t$. Therefore, $V' = V \cup \{s, t\}$, $E'$ contains all directed edges we described above. After we finish constructing $G'$, we start to searching for an augmenting path from $s$ to $t$ using BFS or DFS. Once we find an augmenting path, we augment it and update $M$, and also update $G'$ according to the new $M$. Such phase is repeated, and we will find an augmenting path in each phase. Because each augmenting path guarantee $|M|$ will be increased by 1, there are at most $n$ phases, and each phase has a time complexity of $O(m)$, so the total time complexity of the Ford-Fulkerson algorithm is $O(mn)$.

## 3.2 Algorithm Description & Complexity

In this section, we consider an arbitrary unweighted bipartite graph $G(V, E)$, where $V = A \cup B$, and $E \subseteq A \times B$. We also use $M$ to denote the current matching maintained by the algorithm. Let $A_F$ and $B_F$ represents the set of free vertices in $A$ and $B$ correspondingly. Initially, $M = \emptyset$, $A_F = A$, and $B_F = B$. The Hopcroft-Karp's algorithm is executed in phases. In each phase, the algorithm first constructs the residual graph $G'(V', E')$ with two auxiliary vertices $s$ and $t$. Then, the algorithm execute BFS with $s$ as the source. Let $l_v$ be the shortest augmenting path length from $s$ to any vertex $v \in A \cup B$ of $G'$.

$$l = \min_{v \in B_F} l_v.$$

After that, the algorithm constructs a new graph $G_M(V_M, E_M)$ based on $G'$ s.t. $e(v, u) \in E_M$ only if $e(v, u) \in E'$ and $l_u = l_v + 1$. We call $G_M$ the *admissible graph*. Then, we execute DFS starting from $s$, and search for vertex-disjoint shortest augmenting paths (they are vertex-disjoint except they share the two auxiliary vertices $s$ and $t$) in $G_M$. There is at least one vertex-disjoint shortest augmenting path in $G_M$, and the algorithm tries to find a maximal set of shortest vertex-disjoint augmenting paths in each phase, and augment the matching $M$. The algorithm keeps running until BFS can't find an augmenting path, which means $M$ is already the maximum cardinality matching.

In their paper, Hopcroft and Karp [11] show that the length of the shortest augmenting path increases by at least 1 after each phase. After $\sqrt{n}$ phases, assume that there are $t$ vertex-disjoint augmenting paths left to find the maximum cardinality matching, each augmenting path has a length of at least $\sqrt{n}$. Note that the the symmetric difference of the current matching $M$ and the maximum cardinality matching $M^*$ contains $t$ vertex disjoint augmenting paths. Each of the vertex-disjoint augmenting path is of length at least $\sqrt{n}$.

**Algorithm 4** Hopcroft-Karp's algorithm

```
1: M = ∅
2: while True, ... do
3:     Construct the residual graph G' according to the current matching M
4:     Run BFS in G' starting with s, and get l
5:     if l doesn't exist then
6:         Finish the algorithm
7:     else
8:         Let G_M be the admissible graph
9:         Run DFS in G_M starting with s
10:        Find the maximum set of shortest augmenting paths {P_1, P_2, ...}
11:        Augment {P_1, P_2, ...}, and update M
12:    end if
13: end while
```

Furthermore, the total length of these augmenting paths is $|M| + |M^*| \leq 2n$. Therefore, there are at most $\frac{2n}{\sqrt{n}} = 2\sqrt{n}$ augmenting paths, i.e., $t \leq 2\sqrt{n}$. It takes at most another $2\sqrt{n}$ phases to get the maximum cardinality matching since each phase can find at least one augmenting path. Therefore, the total number of phases is at most $2\sqrt{n} = O(\sqrt{n})$. For each phase, the algorithm explores each edge at most once, so the total time complexity of the algorithm is $O(\sqrt{n}) \times O(m) = O(m\sqrt{n})$. When the graph is dense, i.e., $|E| = \Omega(n^2)$, HK-algorithm's time complexity is $O(n^{2.5})$.

## 3.3 Hopcroft-Karp for bottleneck matching

Although the time complexity of Hopcroft-Karp's algorithm is $O(m\sqrt{n})$, which is the worst case, its time complexity is far way from the worst case in many cases. One example is Erdo-Renyi Graph. For a parameter $0 < p \leq 1$, the Erdos-Renyi Graph [8] is a random graph $G(n, p)$ with a vertex set $V$ that contains $n$ vertices. In Erdos-Renyi Graph, for any pair $v_1, v_2 \in V$, the probability that the $e(v_1, v_2)$ exists in the graph is $p$. Rajeev Motwani [17] showed that the Hopcroft-Karp algorithm only takes almost linear time on

Erdos-Renyi Graph. In fact, in his paper, Motwani [17] says that the HK-Algorithm rarely exhibits its worst-case running time. For many graphs, he shows that the number of phases is significantly smaller than $O(\sqrt{n})$; for instance, for the Erdos-Renyi graph, the number of phases was shown to be $O(\log n)$. In this section, we show a natural input instance where the HK-Algorithm seems to exhibit its worst-case behavior of $O(\sqrt{n})$ phases.

The Hopcroft-Karp's algorithm can be used to solve bottleneck matching problem. The most straightforward way is to compute all possible distances between vertices. There are at most $n^2$ such distances. For each distance $d$, we generate the $\delta$-disc graph with the unit distance as $d$, and run Hopcroft-Karp's algorithm on it. We can try all possible distances, until we find the smallest distance that gives us a perfect matching. If we select, or guess, the distances in a smart sequence, like executing a binary search on the sorted distances, the total time complexity of finding the exact bottleneck can be improved to $O(log(n)n\sqrt{m})$; here $m$ is the number of edges in a $\delta$-disc graph for the largest guess of $\delta$. By the use of dynamic nearest-neighbor structure, Efrat *et al.* [1] improve the time taken to execute each phase of HK-Algorithm to $\tilde{O}(n)$ leading to an overall execution time of $\tilde{O}(n^{3/2})$ which is independent of the number of edges in the $\delta$-disc graph.

We conduct experiments and analyze the number of phases taken by HK-Algorithm when $A$ and $B$ are chosen uniformly at random from a unit square. Our experiments seem to suggest that, when $\delta$ is set to the bottleneck distance, the number of phases executed by HK-Algorithm may be $\Omega(\sqrt{n})$. If this is true, there are two interesting consequences. First, we have a natural and simple instance where HK-Algorithm exhibits its worst case behavior of $\Omega(\sqrt{n})$ phases. Second, this would also imply that the analysis of the algorithm of Efrat *et al.* [1] is almost tight. We provide the details of our experiments next.

We run experiments to test the performance of Hopcroft-Karp's algorithm at various values of $\delta$. We test the datasets containing 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000,

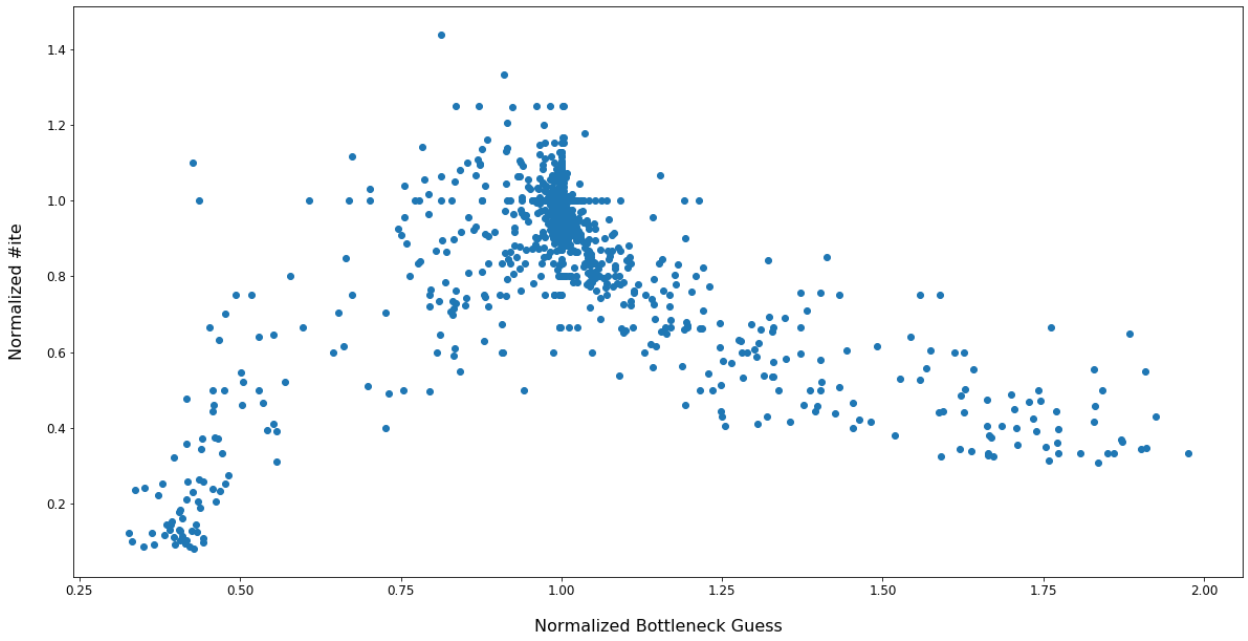


Figure 3.1: Normalized Bottleneck Guess v.s. Normalized number of ite

1500000 points. For each number of points, we generate 10 datasets where points are uniformly chosen in a $128 \times 128$ bounding square. For each dataset, we run the HK-algorithm to find the exact bottleneck distance. In each experiment, the algorithm tries many bottleneck guesses ($\delta$) before getting the exact bottleneck distance. For each guess, we plot the point $\left(\frac{BottleneckGuess}{ExactBottleneckDistance}, \frac{\#ite(BottleneckGuess)}{\#ite(ExactBottleneckDistance)}\right)$ so different experiments can be compared. In the Figure 3.1, we plot all such points. There are two observations we can get from the Figure 3.1. The first one is the the points are clustered very close to $(1, 1)$, which means most of guesses are distances near the exact bottleneck distance. Another observation is the normalized number of iterations seems to increase as the bottleneck distance guess comes close to the exact bottleneck distance.

Figure 3.2 plots the increase in the number of iterations as $|V| = n$ increases. We restrict our analysis to the case where $\delta$ is the bottleneck distance. Theoretically, Hopcroft-Karp's algorithm requires $O(\sqrt{n})$ iterations. From the experiments, we see the number of iterations

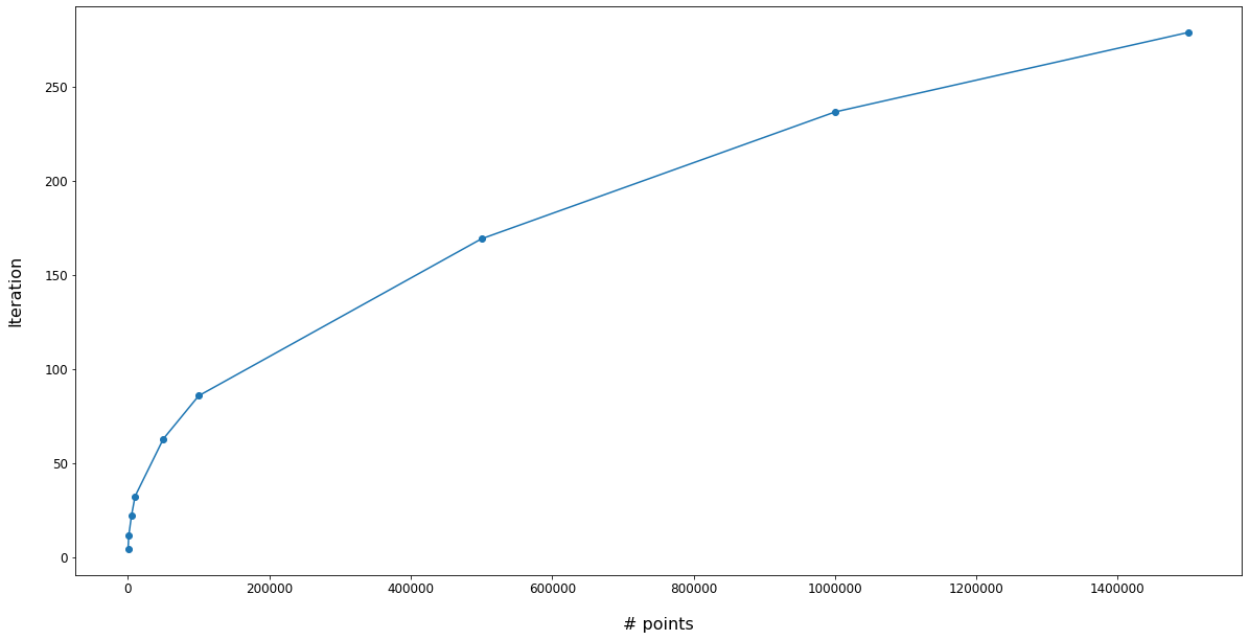


Figure 3.2: Number of Hopcroft-Karp's iterations

of Hopcroft-Karp increases in a pattern that is close to $\sqrt{n}$ when $\delta$ is the bottleneck distance. Figure 3.3 plots $n$ v.s. $\frac{\sqrt{n}}{iteration(n)}$. It seems $\frac{\sqrt{n}}{iteration(n)}$ is either a constant or a slowly increasing function of $n$.

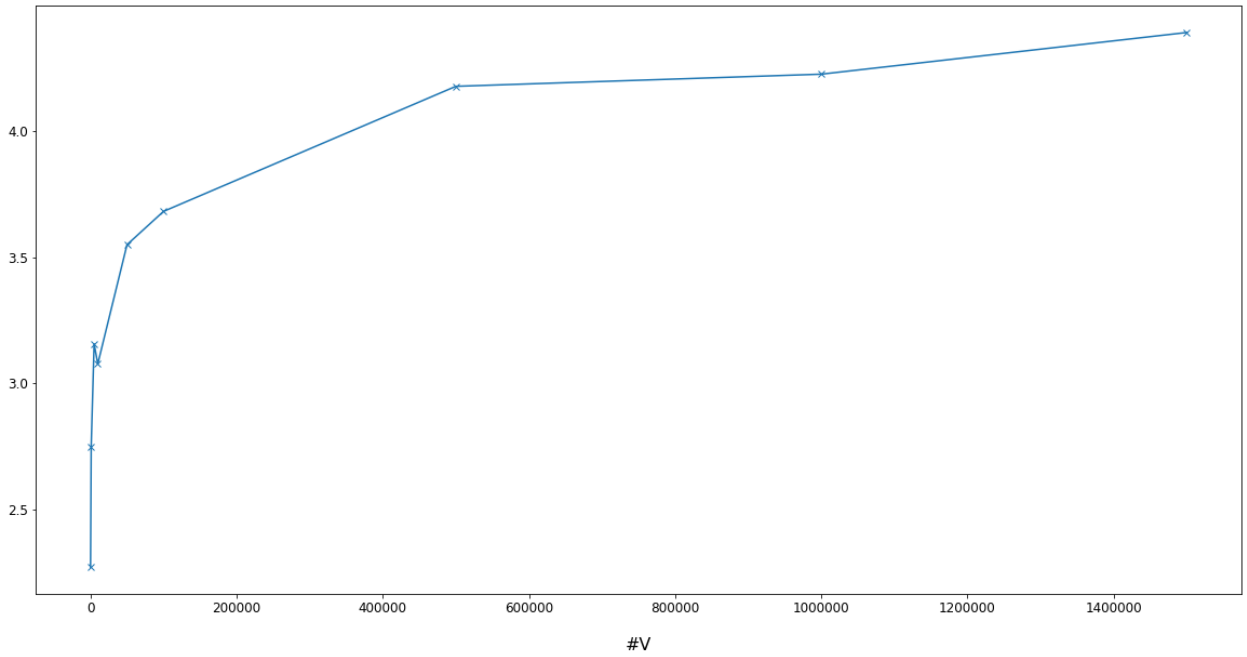


Figure 3.3: $\frac{\sqrt{n}}{iteration(n)}$

# Chapter 4

# The FAST-MATCH Algorithm

In 2019, Lahn and Raghvendra presented a new approach to find a maximum cardinality matching in an arbitrary bipartite graph [16]. Let us refer to this new algorithm as the FAST-MATCH algorithm. The FAST-MATCH algorithm takes as input a weighted bipartite graph $G(V, E)$ with edge weights of 0 or 1, where $V = A \cup B$. We use $w(u, v)$ to denote the weight of the edge between $u$ and $v$. Let $\omega \leq n$ be the upper bound on the weight of any matching in $G$. Consider all connected subgraphs containing only edges of weight 0. We refer to each such connected subgraph as a *piece*. If each piece has $O(r)$ vertices and $O(\frac{mr}{n})$ edges, the FAST-MATCH algorithm can compute a maximum cardinality matching in $G$ in $\tilde{O}(m(\sqrt{\omega} + \sqrt{r} + \frac{\omega r}{n}))$ time.

When all edges have weights of 0 or all edges have weights of 1, the algorithm will be equivalent to the HK-Algorithm [11], and run in $O(m\sqrt{n})$ time. But when the graph $G(A \cup B, E)$ has an easily computable balanced vertex separator [18] of size $|V'|^\lambda$, for every subgraph $G'(V', E')$ where $\lambda \in [1/2, 1)$, the algorithm can compute a maximum matching in $O(mn^{\frac{\lambda}{\lambda+1}})$ time improving upon the $O(m\sqrt{n})$ time taken by the Hopcroft-Karp's algorithm. In addition, the algorithm can compute an $\epsilon$-approximate bottleneck matching of $A, B \subseteq \mathbb{R}^2$ in $O(n^{4/3}/\epsilon^4)$ time, and all previous algorithms takes $\Omega(n^{3/2})$ time. We describe this algorithm next. The original algorithm relied on dual weights. However, we present a slightly simplified description of their algorithm that does not make use of dual weights.

## 4.1 Algorithm Description

In this section, we define an arbitrary weighted bipartite graph $G(V, E)$, where $V = A \cup B$, and $E \subseteq A \times B$. We also use $M$ to denote the current matching maintained by the algorithm. Let $A_F$ and $B_F$ represents the set of free vertices in $A$ and $B$ correspondingly. For an edge $e = (u, v)$, let $w(u, v) = w(e)$ denote the weight of the edge $e \in E$, and $w(e) \in \{0, 1\}$. The length of an augmenting path is defined to be $\sum_{e \in P} w(e)$. Initially, $M = \emptyset$, $A_F = A$, and $B_F = B$.

The FAST-MATCH algorithm has two steps. The first step is a pre-processing step. In the pre-processing step, we run Hopcroft-Karp's algorithm to compute the maximum cardinality matching in each piece separately. The second step of the algorithm is executed in phases. For any $k$, the $k^{th}$ phase consists of two stages described below.

**First stage:** In this stage, we first construct the residual graph $G'$ for the matching $M$. We assign a weight of 0 for all edges directed from $s$ to $a \in A_F$, and those from $b \in B_F$ to $t$. Then we run 0/1 BFS starting from $s$ and compute the shortest path from $s$ to every node in the residual network. Details of the 0/1 BFS can be found in Section 4.3. Let $l_v$ be the shortest distance from $s$ to $v$ in the residual network $G'$. Let

$$l = \min_{v \in B_F} l_v$$

**Second stage:** In this stage, we first construct the admissible graph $G_M$ according to $G'$. In the HK-algorithm, an edge $e(v, u)$ exists in an admissible graph only if $e$ is in the residual graph and $l_u = l_v + 1$, but in the FAST-MATCH algorithm, an edge $e(v, u)$ exists in the admissible graph if $e$ is in the residual graph and $l_u = l_v + w(u, v)$. For each $a \in A_F$, similar to the HK-Algorithm, we initiate a DFS to search for an augmenting path in the residual

network. Suppose the DFS finds an augmenting path $P$, the let every piece that contains at least one edge of $P$ be an *affected piece*. For any edge just being explored by the DFS, we delete the edge if it is not in an affected piece and these deleted edge will not be used for the rest part of this phase. Remaining edges, including those that were visited by the DFS but belonged to an affected piece can still be used in the current phase. In a single DFS, any edge will be explored at most once, but in one phase, an edge with weight 0 may be visited multiple times by different DFS. This is unlike the HK-Algorithm where, within a phase, every edge is visited at most once.

**Algorithm 5** The new algorithm

```
    M = ∅
 2: Split G to pieces
    Execute Hopcroft-Karp's algorithm in each piece
 4: while True do
        Construct the residual graph G' according to M
 6:     Run 0/1 BFS in G' starting with s, and get l
        if l doesn't exist then
 8:         Finish the algorithm
        else
10:         Construct the admissible graph G_M
            for a_i ∈ A_F do
12:             Run DFS in G_M starting with a_i
                if DFS find an shortest augmenting path P then
14:                 Augment the path P
                end if
16:             Delete Edges
            end for
18:     end if
    end while
```

## 4.2 Algorithm Complexity

As shown by Lahn and Raghvendra [16], the time complexity of the FAST-MATCH algorithm is $\tilde{O}(m(\sqrt{\omega}+\sqrt{r}+\frac{\omega r}{n}))$, and it can be split into three parts: $O(m\sqrt{r})$, $O(m\sqrt{\omega})$ and $\tilde{O}(m\frac{\omega r}{n})$. In this section, we discuss each part separately.

### 4.2.1 Pre-processing

The pre-processing step takes $O(m\sqrt{r})$ time. For any such piece $\mathbb{P}$, recollect that it has $O(r)$ vertices. Furthermore, let $m_{\mathbb{P}}$ denote the number of edges in $\mathbb{P}$. The time taken to pre-process $\mathbb{P}$ is $O(m_{\mathbb{P}}\sqrt{r})$. Since all pieces are vertex-disjoint, each edge appears in at most one piece. Therefore, the total time taken to pre-process all pieces is $O(m\sqrt{r})$.

### 4.2.2 Iteration

Lahn and Raghvendra [16] also bounded the number of phases by $O(\sqrt{\omega})$, where $\omega$ is the upper bound of the cost of any matching in the graph. In each phase, if we ignore potential revisits of edges, then the algorithm will explore at most $m$ edges. Therefore, the time complexity of excluding revisits will be $O(m\sqrt{\omega})$. In order to bound the total number of revisits across all phases, Lahn and Raghvendra [16] bound the total number of affected pieces which we describe next.

### 4.2.3 Affected Pieces

To bound the number of edges that are visited multiple times within the same phase, Lahn and Raghvendra [16] observe that an edge that is visited a second time must have been in an

affected piece when it was visited the first time. Using this observation, they upper-bound the number of visits by simply bounding the total number of edges in all the affected pieces ( the edges of a piece that is affected in $t$ different augmentations is counted $t$ times) across all augmentations. They show that the number of affected pieces is $O(\omega \log \omega)$, and each piece contains $O(\frac{mr}{n})$ edges. Therefore the total number of revisited edges is $O(\frac{mr}{n} \times \omega \log \omega) = \tilde{O}(m\frac{\omega r}{n})$.

## 4.3 Implementation

In order to be able to utilize FAST-MATCH for bottleneck matching, we need to first describe a method to assign weights of 0 and 1 to the edges of the $\delta$-disc graphs. In the following, we describe this weight assignment.

### 4.3.1 Weight Assignment

Based on the analysis of the FAST-MATCH algorithm, a very small piece size could make $\omega$ be very high and therefore increase the number of phases. On the other hand, choosing large piece size will not only increase the preprocessing time but also increase the number of revisits. Therefore, the piece size has to be chosen carefully to balance the various parameters. In [16], Lahn and Raghvendra discuss an approach to assign weights for approximate bottleneck matching in $d$-dimensional space. Their algorithm worked for any point set but produced only an approximate solution. Recollect, however, that we are interested in identifying the exact bottleneck matching for the case where $A$ and $B$ are chosen uniformly at random from a unit square.

We use a strategy that is very similar to the one used by Lahn and Raghvendra for assigning

weights for any $\delta$-disc graph that our algorithm encounters. The intuitive idea is to use a grid to split the unit square that contains $A$ and $B$ into $n/r$ squares. This can be done by setting up a grid with multiple equally spaced vertical and horizontal lines as boundaries of the grid cells. The distance between two successive vertical and horizontal lines is $\sqrt{r/n}$. Since each square is of the same area, the expected number of points in each of the $n/r$ squares is $r$. We set the weights of all edges that are completely contained within the same square to 0. Any edge between two points that lie in different squares receive a weight of 1. Therefore, the set of all edges of the $\delta$-disc graph with both the end points inside a given square would form a single piece. Note that, unlike in Lahn and Raghvendra [16], the $n/r$ pieces generated here may not form a connected component. For our experiments, we choose $n/r$ to be $n^{1/3}$. In expectation, each piece has $r = n^{2/3}$ points.

### 4.3.2 0/1 BFS

In the HK-algorithm, because the graph is unweighted, we can find the augmenting path with shortest length using the standard BFS. However, for the FAST-MATCH algorithm, the edges of the graph have 0/1 weights. The standard approach to find the shortest path between two vertices in a weighted graph is to use Dijkstra's algorithm, which has a time complexity of $O(m + n \log n)$. However, because the graph has only edges of weight 0 and weight 1, we can compute the augmenting path with shortest length in $O(m + n)$ if we slightly modify the standard BFS, which we refer to as the *0/1 BFS*. In the standard BFS, we maintain a queue. We enqueue new vertices at the end of the queue, and dequeue from the head. In 0/1 BFS, we also maintain a queue, and dequeue vertices from the head. The difference is how we add a vertex to the queue. Suppose we are adding a vertex $v$ to the queue, and $v$ comes from the edge $e(u, v)$. If the edge has the weight of 0, we add $v$ to the head, otherwise, we add it to the end. The distance of a vertex is updated only when it

is dequeued from the queue. The time complexity of 0/1 BFS is as same as BFS, which is $O(m+n)$.

**Algorithm 6** 0/1 BFS

```
    for v ∈ V do
        if v ∈ A_F then
3:          Dist[v] = 0
        else
            Dist[v] = ∞
6:      end if
    end for

9:  Let Q be a queue, and Q = ∅
    Add vertices in A_F to Q
    while Q is not empty do
12:     v = Q.dequeue()
        for u_i ∈ {u_1, u_2, ...} s.t. edge(v, u_i) exists do
            if Dist[u_i] > Dist[v] + w(edge(v, u_i)) then
15:             Dist[u_i] ← Dist[v] + w(edge(v, u_i))
                if w(edge(v, u_i)) == 0 then
                    add u_i to the head of Q
18:             else
                    add u_i to the tail of Q
                end if
21:         end if
        end for
    end while
```

### 4.3.3 Edge Deletion

In each phase, the FAST-MATCH algorithm runs DFS multiple times. After each DFS, some of the edges that are explored in this DFS will be deleted and these edges will not be used anymore in the rest DFS's in this phase.

In Hopcroft-Karp's algorithm, we delete vertices instead of edges, which only takes $O(n)$ memory to store information about if a vertex is deleted or not. However, it requires $O(n^2)$

memory if we are going to use an adjacency matrix to store information about if edges are deleted or not, which is memory-inefficient. It will also be time-inefficient to maintain an adjacency list storing edges that are deleted, because it takes not-constant time to delete an edge from an adjacency list.

Therefore, for each vertex, which has an adjacency list storing its edges, I use three pointers on the list to define if an edge is deleted, just explored, or still unexplored. This approach requires an additional constant memory and constant time per edge and can handle deletions efficiently.

When I generate edges, I sort them by 0/1 weights such that the edges of weight 1 is in front of the edges of weight 0. Three pointers are defined for each vertex. The first pointer is a fixed pointer called "one-zero" pointer, and it is the index of the first edge of weight 0 in the list. The second and third pointer are dynamic. The second pointer is called "deleted" pointer, and all edges before this pointer are deleted and won't be used in the current phase. The third pointer is called "explored" pointer. Edges between "deleted" pointer and "explored" pointer are edges that are just explored in the current DFS, and edges after "explored" pointer are unexplored edges.

In the beginning of each stage, we reset "deleted" and "explored" pointers to 0. Every time DFS explores an edge, we move the "explored pointer" one step forward. If the "explored pointer" has reached the end of the list, it means we have explored all edges out of the current vertex.

After each DFS, for arbitrary explored vertex $v$, if $v$ is not in an affected piece, we delete all explored edges out of $v$, and no matter if the weight of edges is 0 or 1. We do this by setting "deleted" pointer to be "explored" pointer.

Otherwise, if the vertex is in the affected piece. If "explored" pointer is smaller than "one-

zero" pointer, we also set "deleted" pointer to be "explored" pointer. The reason for this is we will only use edges of weight 1 once in each stage, and since all explored edges have the weight of 1, it is safe to delete all of them.

If "explored" pointer is larger than "one-zero" pointer, we set "deleted" pointer to be max("deleted" pointer, "one-zero" pointer), and set "explored" pointer to be "deleted" pointer. This is because any edge of weight 0 means it doesn't cross pieces, so we keep all edges going out of the vertex with weight 0 and delete all edges with weight 1.

**Algorithm 7** Delete Edges

```
    Input = v ∈ V
    Suppose v contains the list U = {u_1, u_2, ...} that stores edges going out of v
    U has three pointers: deleted, zero-one, explored. Each one is an index of U
4:  if v is not in an affected piece or explored < zero-one then
        deleted ← explored
    else
        if deleted < zero-one then
8:          deleted ← zero-one
        end if
        explored ← deleted
    end if
```

For example, suppose we have a vertex $v$ with 8 edges, and they are stored in a sorted list: the first 4 edges have the weight of 1, and the last 4 edges have the weight of 0 (Figure 4.1). In the Figure 4.1, $D$, $E$, and $Z$ represent "deleted", "explored" and "zero-one" pointer. All of the edges haven't yet been explored in the current phase.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E<br>D 1 | 1 | 1 | 1 | Z<br>0 | 0 | 0 | 0 |

Figure 4.1: Initial setting

Suppose in the first DFS search, we explored the first two edges. The pointer $E$ is now

pointing the third position (Figure 4.2). Because any edge with the weight of 1 can't be in an affected piece, the first two explored edges can be deleted safely, so we also move the pointer $D$ to the third position (figure 4.3), which deletes two edges implicitly.

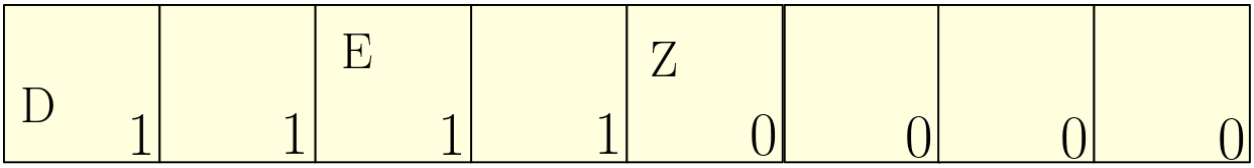


Figure 4.2: After exploring two edges with the weight of 1

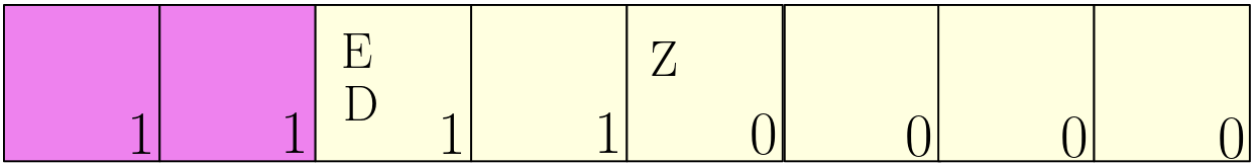


Figure 4.3: Delete two edges

Suppose in the second DFS search, three edges are explored. The first two edges have the weight of 1, and the third one has the weight of 0 (figure 4.4). If the vertex $v$ is in an affected piece, its edges with the weight of 0 are also in an affected piece. Therefore, we only need to delete the first two edges (figure 4.5). However, if $v$ is not in an affected piece, all explored edges are deleted (figure 4.6).

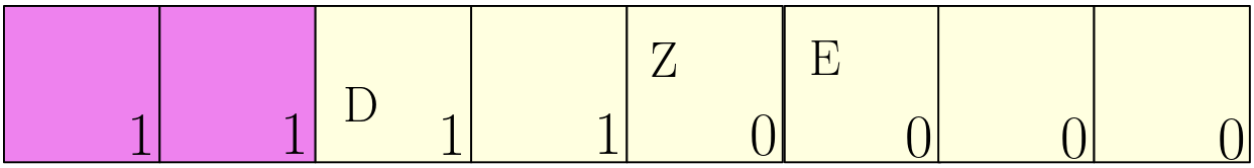


Figure 4.4: Explore another three edges

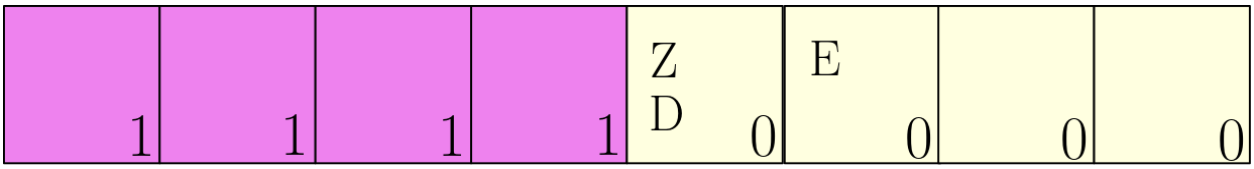


Figure 4.5: If this vertex is in an affected piece

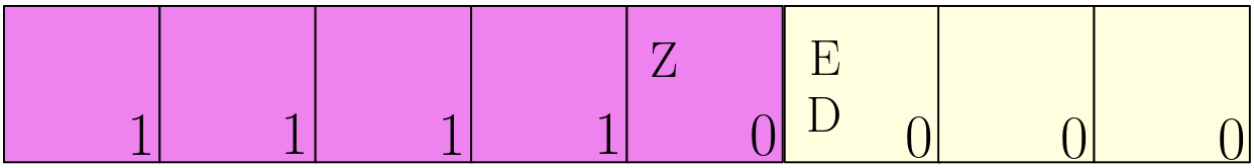


Figure 4.6: If this vertex is not in an affected piece

# Chapter 5

# Experiments

## 5.1 Bottleneck Search

Since we construct the graph explicitly, a large bottleneck guess can take a lot of time to generate edges, especially when the input point set is large. Therefore, we intend to start with a small initial bottleneck guess. In my implementation, I use $\frac{S}{\sqrt{n}}$ as the initial bottleneck guess, where $S$ is the side length of the entire bounding square and $n$ is the number of points. The reason is, if all points are chosen uniformly, in the expectation, the bounding square can be split into $n$ smaller squares with equal sizes of $\frac{S}{\sqrt{n}} \times \frac{S}{\sqrt{n}}$, and each small square contains (in expectation) one point. If the guess does not give us a perfect matching, we double the bottleneck guess until we find a perfect matching. Otherwise, if this guess of $\delta = \frac{S}{\sqrt{n}}$ returns a bottleneck matching, we reduce our guess by half and continue. Once we find a perfect matching, we run the binary search between the last two bottleneck guesses, and stop until it achieves our desired accuracy. More specifically, using the binary search, we find an interval $[o, (1+\epsilon)o]$ that contains the bottleneck distance; here $\epsilon$ is a sufficiently small error parameter of our choice. For $\delta = (1+\epsilon)o$, we generate and sort (in increasing order of their Euclidean lengths) all the edges of the $\delta$-disc graph that lie within the interval $[o, (1+\epsilon)o]$. We can identify the exact bottleneck distance by conducting a binary search on this sorted order.

An additional optimization that one may apply is the following. When the current bottleneck

guess is larger than the previous one, we can reuse the matching computed for the smaller guess as a starting point for the larger one.

However, we do not apply this optimization for our implementation. This is because, doing so could affect the number of phases executed by our algorithm as well as the HK-Algorithm. In order to ensure that the statistics collected are accurate, we choose to run from scratch for every guess of the bottleneck distance.

## 5.2 Experiment Setup

In the experiment, we test the datasets containing 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 1500000 points. For each number of points, we generate 10 datasets where points are uniformly chosen in a $128 \times 128$ bounding squares. Then for each dataset, we run both the Hopcroft-Karp and the FAST-MATCH algorithm once to find the exact bottleneck. Experiments are run on a server running CentOS Linux 7, with 12 Intel E5-2683v4 cores with 128GB of RAM. Experiments are run in sequence instead of in parallel. We would like to acknowledge, Advanced Research Computing (ARC) at Virginia Tech that provided us with the computational resources that have contributed to the results reported within this thesis.

## 5.3 Experiments

There are two objectives for our experiments: (1) compare the performance of Hopcraft-Karp's algorithm with the FAST-MATCH algorithm for the bottleneck matching problem; (2) understand the empirical behavior of the FAST-MATCH algorithm.

### 5.3.1 Running Time

Figure 5.1 compares the actual running time of searching for the exact bottleneck of both algorithms. For datasets with more than $10^6$ points, the FAST-MATCH algorithm finds the exact bottleneck in approximately half of the running time of the Hopcroft-Karp's algorithm.

The actual running times can be affected by several factors including the exact implementation details of the two algorithms and the constants associated with the implementation. We observe that both the FAST-MATCH algorithm as well as the HK-Algorithm are executing variants of BFS and DFS. Therefore, the number of steps executed by these algorithms can also be measured by the total number of edges visited by these search procedures.

Figure 5.2 compares the total number of visited edges of the two algorithms. It seems that the number of visited edges also increase in a pattern similar to the increase in the execution time. The gap between two curves also seems to increase as the number of points increases. This suggests the time complexity ratio between two algorithms is a function of $n$.

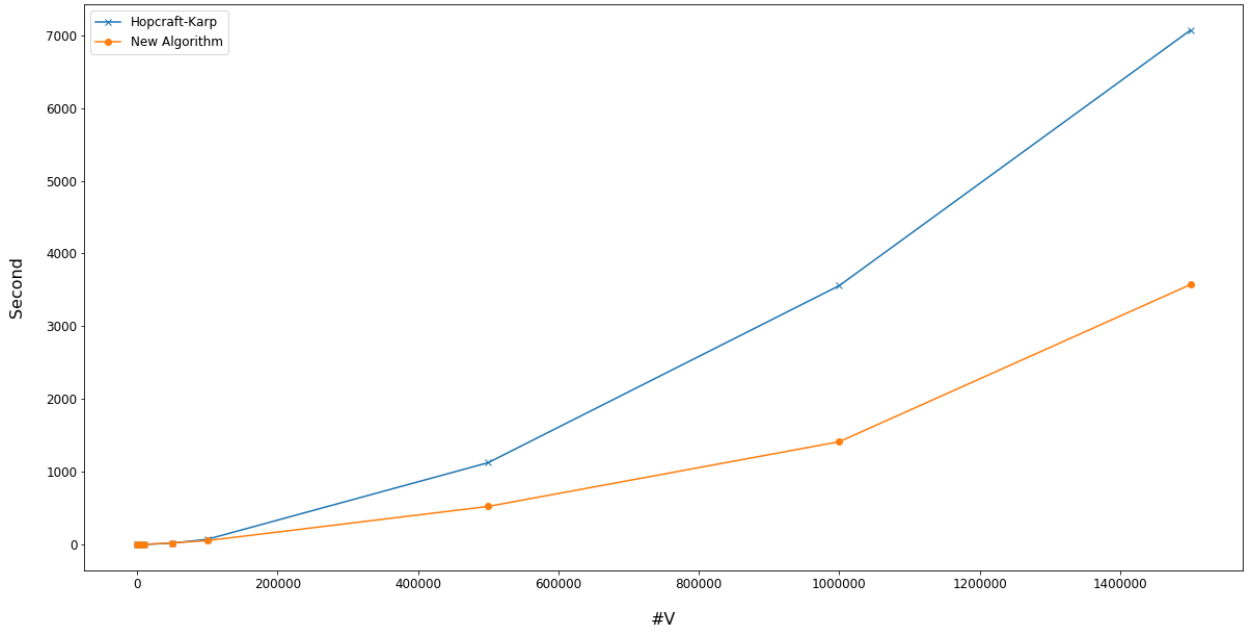


Figure 5.1: Running time

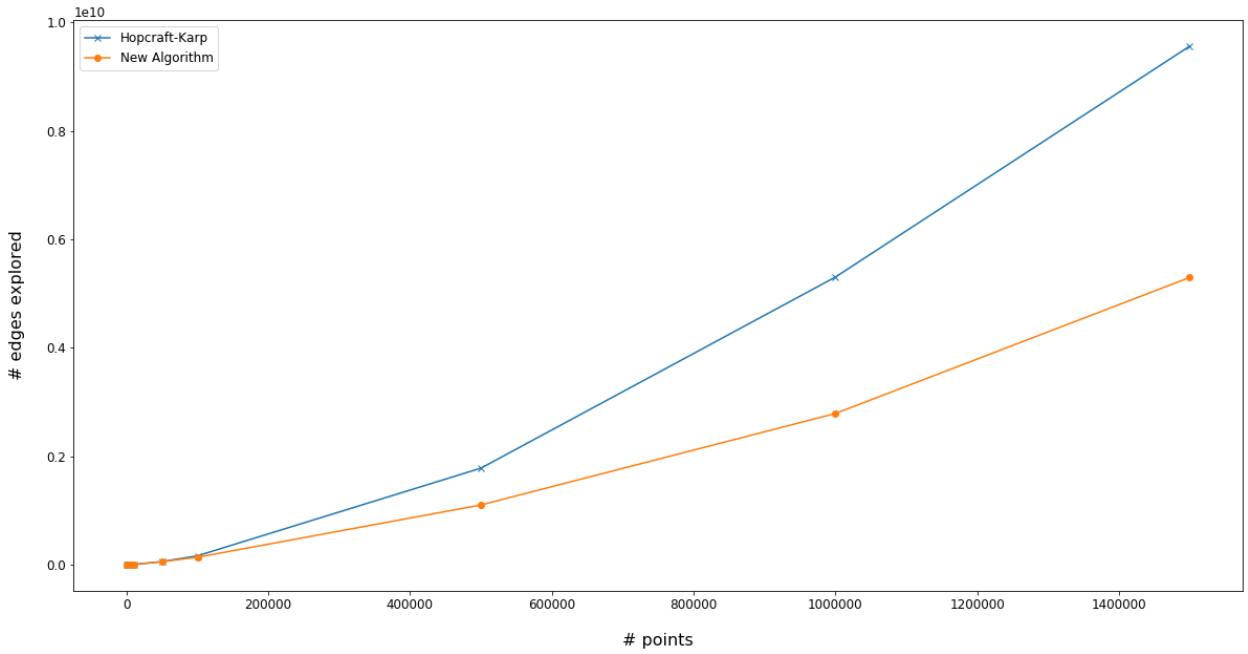


Figure 5.2: Number of edges explored by two algorithms

Besides the running time of the entire search process, we are also interested in analyzing the performance of the FAST-MATCH algorithm for different $\delta$ values. In the Figure 5.3, we compare the performances of the FAST-MATCH Algorithm with the HK-Algorithm. The plot shows the relationship between the total number of visited edges and the value of $\delta$ for both algorithms. Each point is one experiment where we run the HK-algorithm (orange points) or the FAST-MATCH algorithm (blue points) on a $\delta$-disc graph with the fixed value of $\delta$. The $x$-axis is $\frac{\delta(the\ \ Bottleneck\ \ Guess)}{the\ \ Bottleneck\ \ Distance}$, and the $y$-axis represents $\frac{\#edge(\delta)}{\#edge(the\ \ Bottleneck\ \ Distance)}$ (Let $h$ be a guess of the bottleneck distance, and $\#edge(h)$ represents the total number of edges being explored by the algorithm on a $\delta$-disc graph when $\delta = h$). As we can see, the FAST-MATCH algorithm visits significantly fewer edges than the HK-algorithm, especially near the bottleneck distance. Figure 5.4 shows the same set of data, but we exclude points from small datasets. We find when datasets have large sizes, in our experiments, the FAST-MATCH algorithm always outperformed Hopcroft-Karp's algorithm near the bottleneck distance.

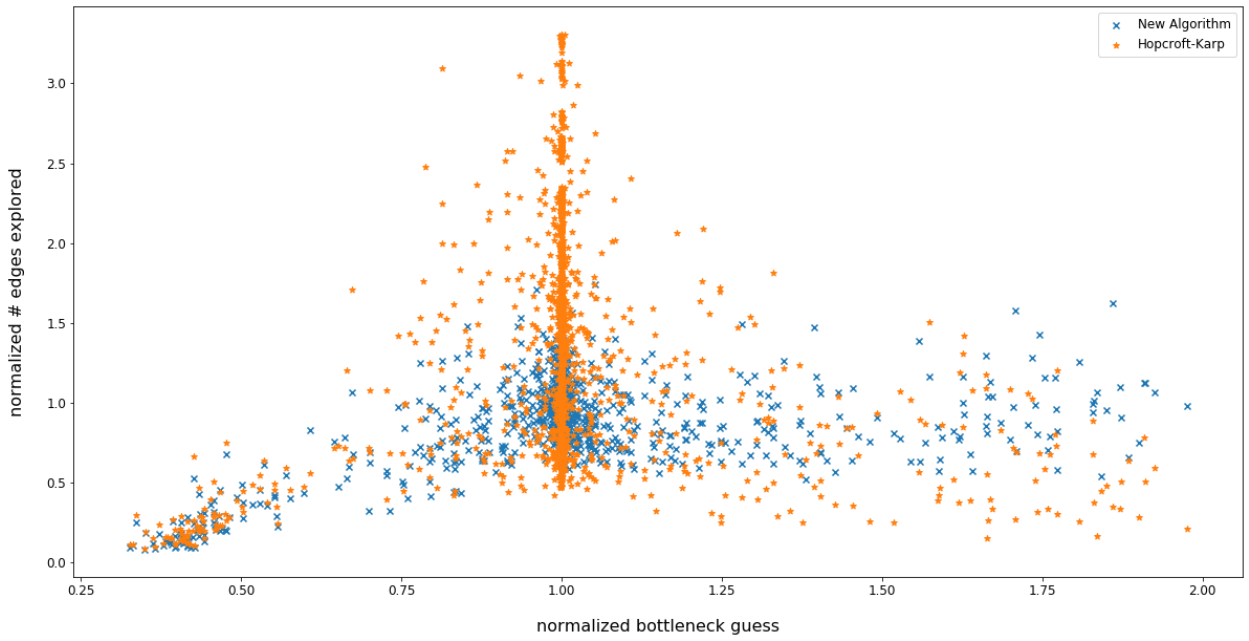


Figure 5.3: Normalized Bottleneck Guess v.s. Normalized number of visited edges (All datasets)

Figure 5.5 shows the normalized value of number of phases that the FAST-MATCH algorithm executes for different $\delta$ values. Similar to the empirical performance of HK-Algorithm (shown in Chapter 3), the FAST-MATCH algorithm also seems to exhibit its worst case in the number of iterations when $\delta$ is close to the bottleneck distance.

In Figure 5.6, we plot the number of phases taken by the FAST-MATCH algorithm for different values of $n$ (For this experiment, we use 10 different data sets for each value of $n$ and, for each data set we choose $\delta$ to be the bottleneck distance). We compare the number of phases to $n^{1/3}$[1]. Figure 5.7 plots $\frac{n^{1/3}}{iteration(n)}$, and it seems, in our experiments, the ratio between them converges to 5 as $n$ increases suggesting that $n^{1/3}$ may be a good estimate for the number of phases executed by FAST-MATCH Algorithm. Note that the number of phases executed by the FAST-MATCH algorithm is $\sqrt{\omega}$. Therefore, we suspect that $\omega$ is

[1]The algorithm of Lahn and Raghvendra uses a weight assignment similar to that of ours but for the approximate bottleneck matching problem. They prove that the number of phases with this weight assignment is $n^{1/3}$. For this reason, we choose to compare the number of phases with $n^{1/3}$

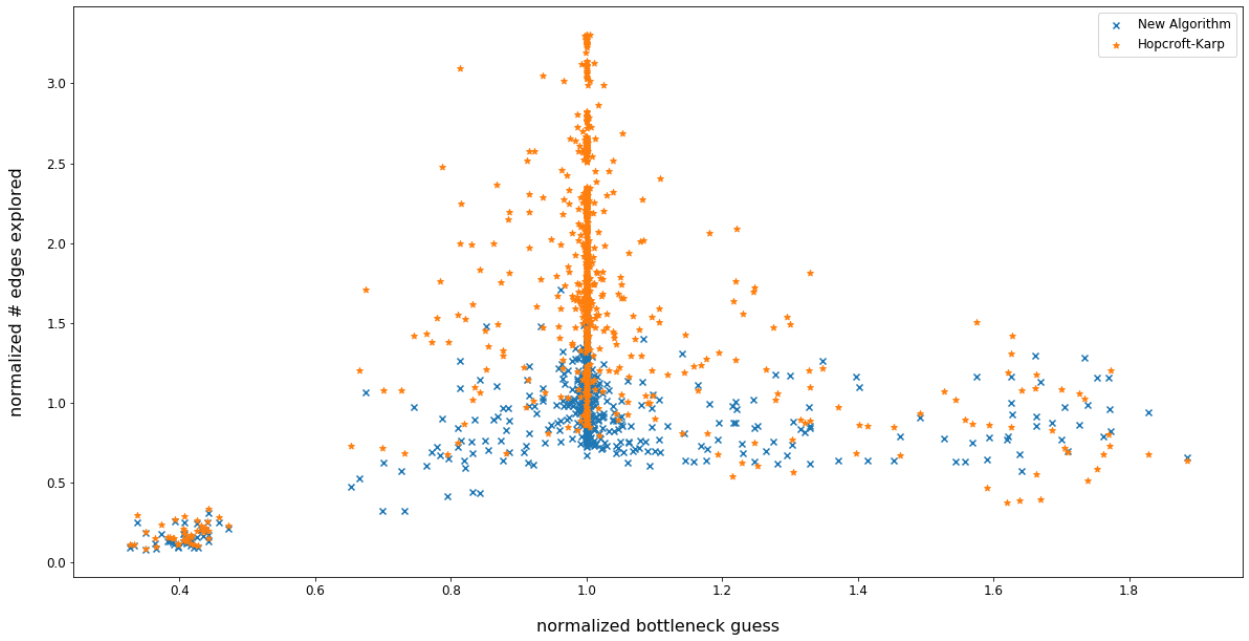


Figure 5.4: Normalized Bottleneck Guess v.s. Normalized number of visited edges (Exclude small datasets)

approximately $n^{2/3}$.

The Figure 5.8 compares the numbers of phases that two algorithms take. As can be noted, the number of phases of FAST-MATCH is significantly smaller than the number of phases taken by the HK-Algorithm.

In order to obtain a theoretical bound, we need to know the values of $r$ and $\omega$. By our choice, the value of $r$ is $n^{2/3}$. However, we do not have a provable bound on $\omega$. We therefore, perform an empirical analysis of the algorithm.

In the paper [16], the time complexity of the FAST-MATCH algorithm consists of three parts. Prepossessing part takes $O(m\sqrt{r})$ times. Edges that are only visited once in each iteration take $O(m\sqrt{\omega})$ time. The time taken due to revisited edges is $O(\frac{m\omega r}{n}\log(\omega))$.

In the Figure 5.9, we plot the number of edges visited by the "pre-processing" part ($O(m\sqrt{r}$ time), the "visitOnce" part ($O(m\sqrt{\omega}$ time) and the "revisited" part ($O(\frac{mr\omega}{n}\log\omega)$. The

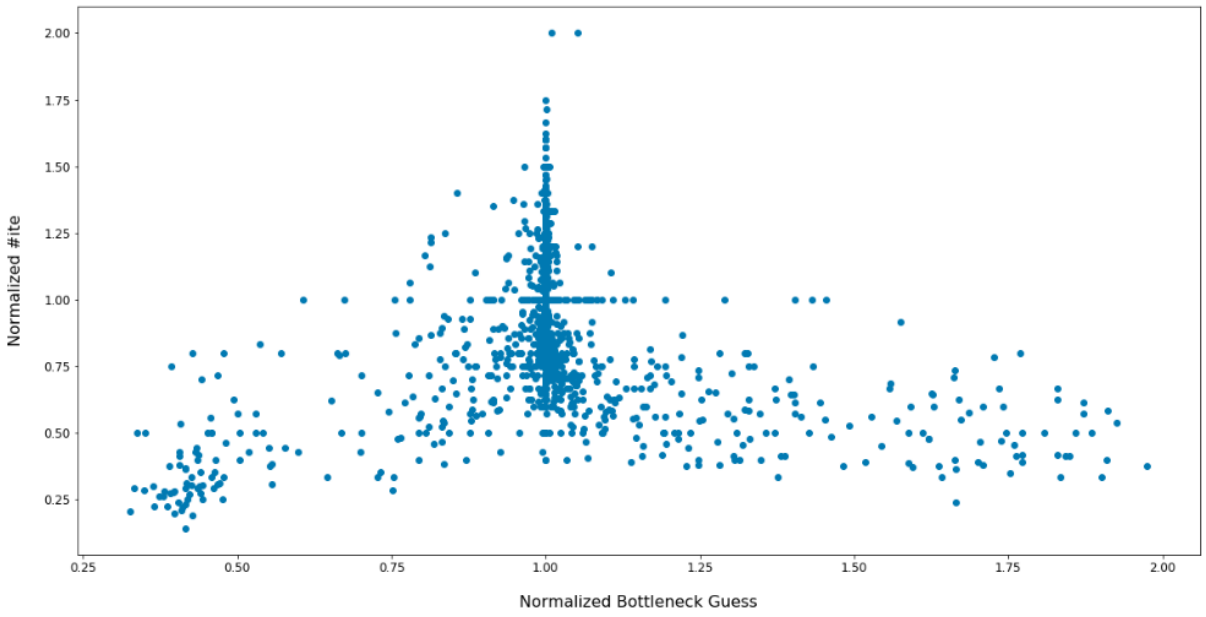


Figure 5.5: Normalized Bottleneck Guess v.s. Normalized number of ite

number of revisited edges seems to dominate the total number of visited edges. and for the other two parts, the number of edges that are only visited once grows a little faster than the pre-processing part. Based on our prior discussion, we suspect $\omega$ and $r$ are approximately $n^{2/3}$. In order to understand the behavior of revisited edges, we compare $O(\frac{mr\omega}{n}\log\omega) = O(mn^{1/3}\log n)$ to the number of revisited edges. In Figure 5.10, we plot the ratio of the number of revisited edges to $mn^{1/3}\log n$. From the Figure 5.10, it seems that the ratio stabilizes as $n$ increases. Based on our experiments, we believe that $O(mn^{1/3}\log n)$ may be an upper bound on the number of revisited edges.

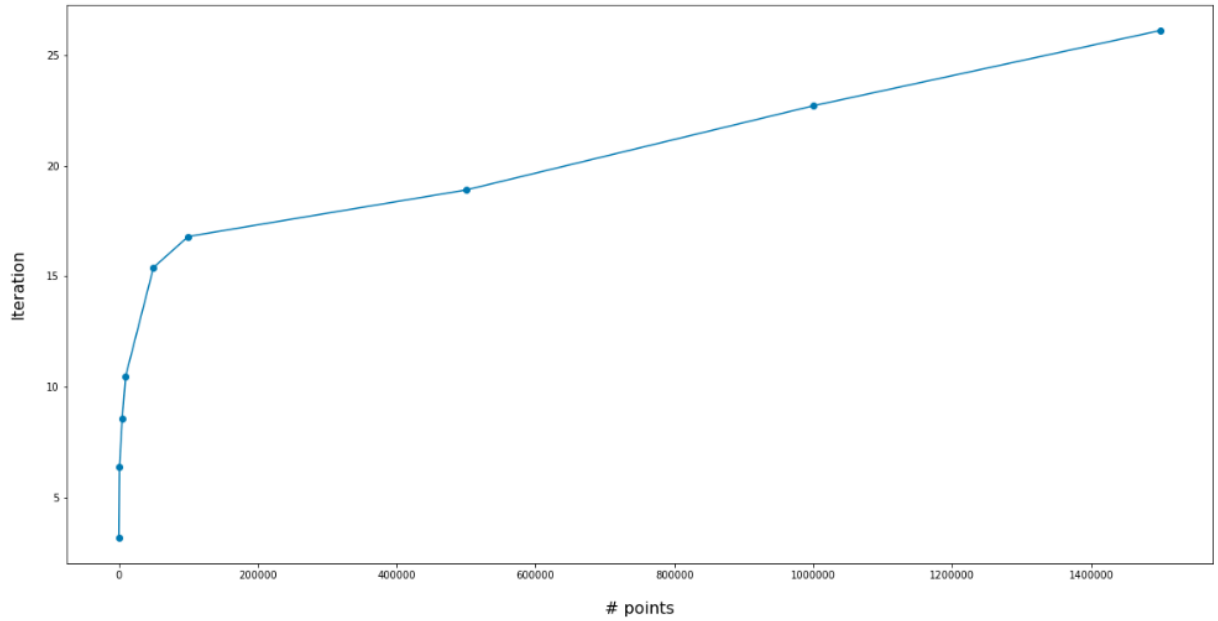


Figure 5.6: Number of iterations for the FAST-MATCH algorithm

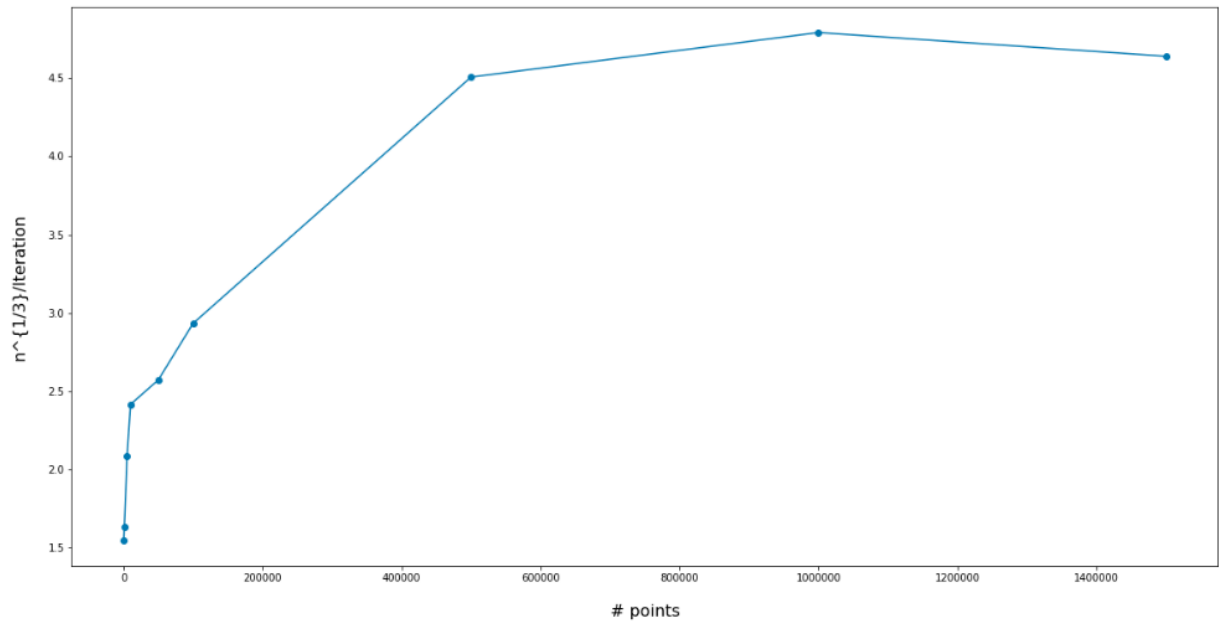


Figure 5.7: $\frac{n^{1/3}}{iteration(n)}$

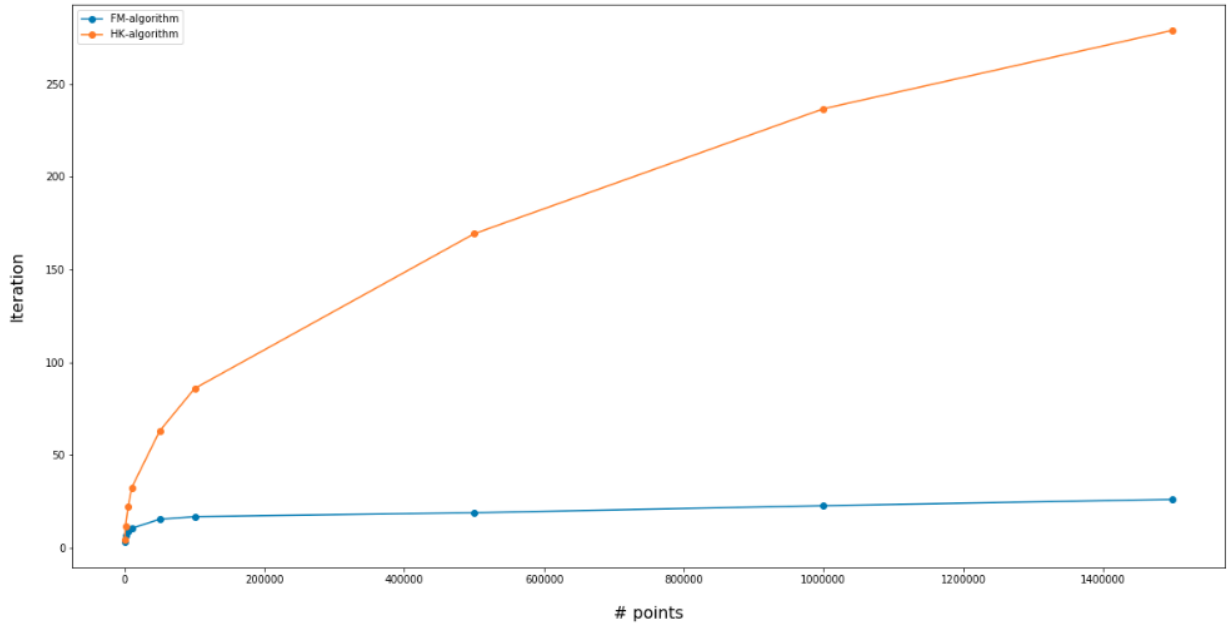


Figure 5.8: Compare the numbers of iterations of both algorithms

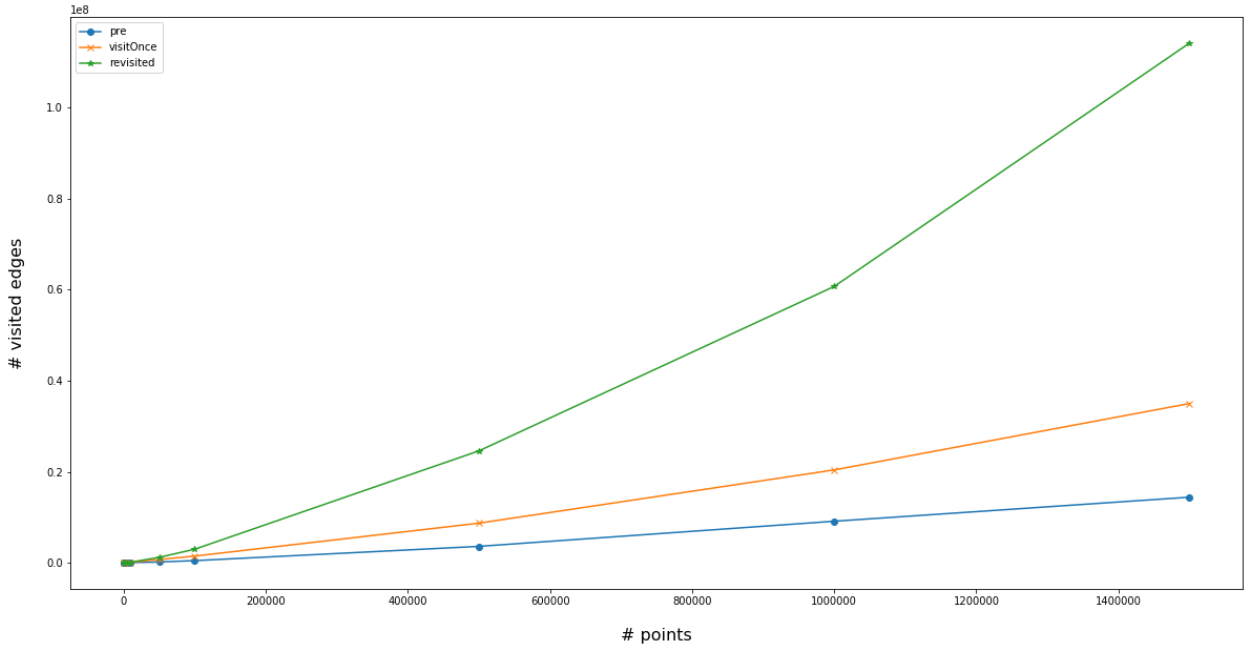


Figure 5.9: Number of edges each part explored

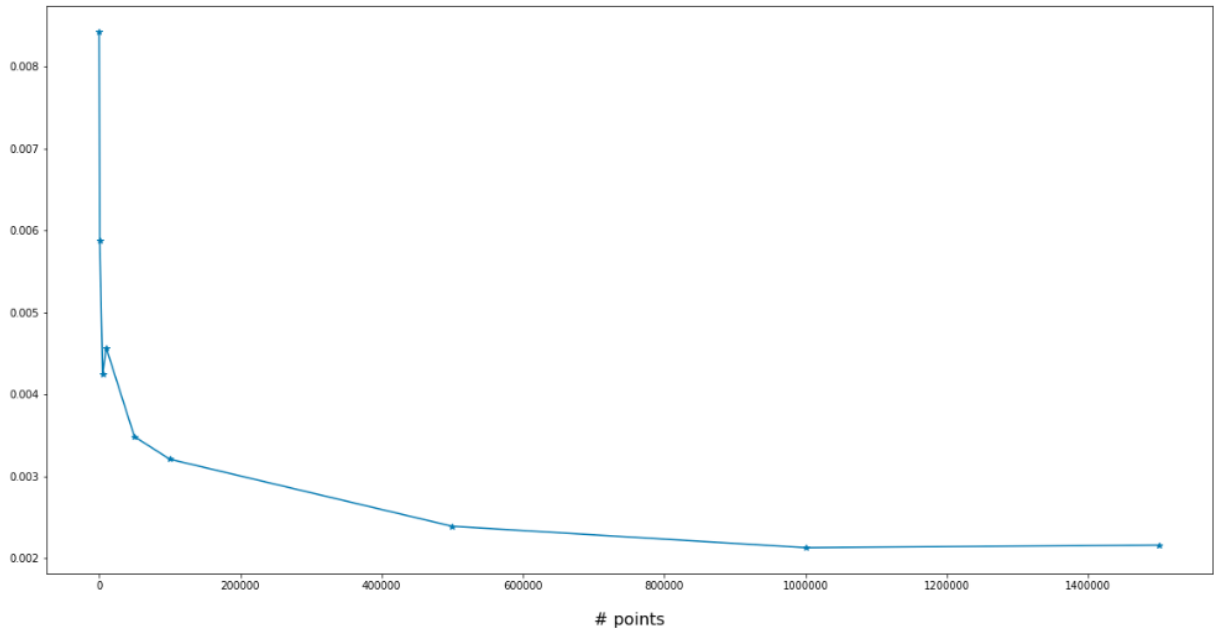


Figure 5.10: The ratio of the number of revisited edges to $mn^{1/3} \log n$

## 5.4 Conclusion and Future Work

In this thesis, I implement a new algorithm, called the FAST-MATCH algorithm, that solves maximum cardinality matching problem in a bipartite graph. I run experiments to compare the FAST-MATCH algorithm and the Hopcroft-Karp's algorithm's performances on the bottleneck matching problem. Experiments are run on the datasets where points are generated randomly in an unit square. Experiment results show the FAST-MATCH algorithm outperforms the Hopcroft-Karp's algorithm. When the dataset has more than $10^6$ points, the FAST-MATCH algorithm is 1.5 times faster than the Hopcroft-Karp's algorithm. It also seems the gap between two algorithms becomes larger as the size of the dataset increases. In addition, I also found that when the Hopcroft-Karp's algorithm is executed in a $\delta$-disc graph where $\delta$ is close to the bottleneck distance, the Hopcroft-Karp Algorithm seems to exhibit its worst case behaviour of executing $\approx \sqrt{n}$ iterations, where $n$ is the number of points in the graph.

So far, I have only conducted an empirical analysis of the FAST-MATCH algorithm for the exact bottleneck matching problem. An important future work is to give a theoretical bound on the execution time of this algorithm.

Besides, the two approaches for bottleneck matching are only compared on randomly generated datasets. It is still unknown if the FAST-MATCH algorithm can still maintain its advantages in other kinds of datasets. For example, all datasets I used contain points on the 2-D space, and it is unknown how will the algorithms behave differently on datasets containing points with a higher dimension. In addition, we are also interested in the FAST-MATCH algorithm's performance in real applications, where data are collected from the real world instead of generated. Therefore, one important future work is to test the FAST-MATCH algorithm on more types of datasets.

Another important future work is to optimize the FAST-MATCH algorithm further. Before the execution of the FAST-MATCH algorithm, we need to assign weights for edges in the graph. A good weight assignment helps the algorithm. Currently, we use an intuitive strategy to assign weights, and it is still unclear if there is a more efficient way to assign weights. In addition, the performance of the FAST-MATCH algorithm is also related to the piece size. Therefore, another way to optimize the FAST-MATCH algorithm is to find the optimal piece size. Currently, in my implementation, each piece contains approximate $n^{2/3}$ points, but we do not know how far away it is from the optimal piece size. In addition, for different kinds of datasets, it is possible that the optimal piece sizes are different, and more experiments need to be done to explore that.

# Bibliography


[1] Alon Itai Alon Efrat and Matthew J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[3] Mudabir Kabir Asathulla, Sanjeev Khanna, Nathaniel Lahn, and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite perfect matching in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 457–476, USA, 2018. Society for Industrial and Applied Mathematics.

[4] Erik D Demaine, Sándor P Fekete, Phillip Keldenich, Henk Meijer, and Christian Scheffer. Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch. *SIAM Journal on Computing*, 48(6):1727–1762, 2019.

[5] Yago Diez, Mario A Lopez, and J Antoni Sellares. Noisy road network matching. In *International Conference on Geographic Information Science*, pages 38–54. Springer, 2008.

[6] Oranit Dror, Ruth Nussinov, and Haim Wolfson. Arts: alignment of rna tertiary structures. *Bioinformatics*, 21(suppl_2):ii47–ii53, 2005.

[7] David Duran, Vera Sacristán, and Rodrigo I Silveira. Map construction algorithms: a local evaluation through hiking data. *GeoInformatica*, pages 1–49, 2020.

[8] P. Erdos and A. Rényi. On random graphs i. *Publicationes Mathematicae*, 6:290–297, 1959.

[9] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[10] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[11] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):1973–231, 1905.

[12] Michael Kerber, Dmitriy Morozov, and Arnur Nigmetov. Geometry helps to compare persistence diagrams. *Journal of Experimental Algorithmics (JEA)*, 22:1–20, 2017.

[13] Andrey Boris Khesin, Aleksandar Nikolov, and Dmitry Paramonov. Preconditioning for the geometric transportation problem. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*, volume 129 of *LIPIcs*, pages 15:1–15:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[14] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[15] Nathaniel Lahn and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite matching in minor-free graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 569–588. SIAM, 2019.

[16] Nathaniel Lahn and Sharath Raghvendra. A weighted approach to the maximum cardinality bipartite matching problem with applications in geometric settings. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational*

*Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*, volume 129 of *LIPIcs*, pages 48:1–48:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[17] Rajeev Motwani. Average-case analysis of algorithms for matchings and related problems. *J. ACM*, 41(6):1329–1356, November 1994.

[18] Arnold L Rosenberg and Lenwood S Heath. *Graph separators, with applications*. Springer Science & Business Media, 2001.

[19] R. Sharathkumar. A sub-quadratic algorithm for bipartite matching of planar points with bounded integer coordinates. In *Proceedings of the Twenty-Ninth Annual Symposium on Computational Geometry*, SoCG '13, page 9–16, New York, NY, USA, 2013. Association for Computing Machinery.

[20] R. Sharathkumar and Pankaj K. Agarwal. A near-linear time -approximation algorithm for geometric bipartite matching. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 385–394, New York, NY, USA, 2012. Association for Computing Machinery.