An Improved Approach on the Extremal-weight Tripartite Assignment Problem by Optimization Algorithms

Beining Zhou[1]

[1]Advisor: Edward Valitutto
[1]Saint Mark's School, Southborough, MA, USA

September 8, 2019

Contents

Abstract

The bipartite matching problem is a classical problem in graph theory that can be solved in polynomial time. However, in contrast, the extremal-weight tripartite assignment problem is considered to be NP-hard. Since this problem has various applications, a fast and accurate optimization method is needed. In this article, we look into three gradient-based algorithms, including two versions of the hill-climbing methods and a simulated-annealing algorithm in order to gain a faster and more accurate optimization to the problem than approximating using a bottleneck matching method. In addition, we analyze the parameters in these algorithms to achieve better performance.

Keywords: NP-hardness, bipartite matching, tripartite matching, gradient-based algorithms

1 Introduction

In this paper, we examine the tripartite matching problem and present several algorithms that provide accurate approximations to the problem. However, in order to analyze the maximum-weight matching in the case of tripartite graphs, we will first take a look at the more simple case, the maximum-weight matching in bipartite graphs.

1.1 Maximum-weight Matching in Bipartite Graphs

To start with, the maximum-weight matching problem is a problem on weighted graphs. A graph $G(V, E)$ is called weighted if for each edge $e \in E$, a weight $W_e \in \mathbb{R}^+$ is prescribed to e. Now, we introduce the definition for bipartite graphs and bipartite matching.

Definition 1.1 (bipartite graph) *A graph $G(V, E)$ is called **bipartite** if V can be partitioned into two disjoint subsets $V = V_1 \cup V_2$ such that for all edges $e \in E$, e is in the form of $e = (a, b)$ where $a \in V_1$ and $b \in V_2$.*

Definition 1.2 (matching, bipartite) *For a bipartite graph $G(V, E)$, a subset $M \subseteq E$ is called a **matching** with respect to G if for each vertex $v \in V$, there is at most one edge that is incident to v.*
A vertex $v \in V$ is matched by matching M if there exist an edge $e \in M$ such that e is incident to v.

Next, we introduce the definitions of complete bipartite graphs and perfect bipartite matchings. The motivations of such definitions will be explained later.

Definition 1.3 (complete bipartite graph) *A bipartite graph $G(V, E)$ is called complete if $E = V_1 \times V_2$.*

Definition 1.4 (perfect matching) *For a bipartite graph $G(V, E)$ and a matching M with respect to G, M is called **perfect** if for each vertex $v \in V$, there exist an edge $e \in M$ such that v is incident to e.*

The maximum-weigth bipartite matching problem (BMP) is a classic problem in graph theory that finds the maximum-weighted matching in bipartite graphs. Let $G(V, E)$ be a bipartite graph with $|V_1| = |V_2| = n$ and $|E| = m$. The BMP seeks to find the maximum-weight matching M^* such that:

$$\sum_{e \in M} W_e$$

is maximized.

Note that a general bipartite graph $G_1(V, E_1)$ can be expressed as a complete bipartite graph $G_2(V, E_2)$ using the following method: for all $e \in E_1$, let the weight of the corresponding edge in E_2 be the same as w_e; for all $e \in E_2$ such that $e \notin E_1$, let $w_e = 0$. Then, a matching in G_1 will correspond to another matching in G_2 with the result of the objective function unchanged. A general graph with $|V1| \neq |V2|$ could also be turned into a graph with $|V1| = |V2|$ by the same token, adding 0-weighted edges. In addition, note that since all weights are non-negative, the maximum-weight matching has to be a perfect matching in a complete bipartite graph. Therefore, we only consider the case of perfect matchings in complete bipartite graphs.

The BMP could be solved with polynomial time algorithm by the hungarian algorithm and multiple improved version[1, 3]. Specifically, the Hopcroft-Karp algorithm solves this problem with running time of $O(\sqrt{n}m)$ or $O(n^5/2)$.[2]

1.2 Extremal-weight Assignment in Tripartite Graphs

In this paper, we examine the extremal-weight tripartite assignment problem (TAP), which is an extension of the previously stated BMP. Now that we have seen the BMP, we make generalizations for for the TAP.

Definition 1.5 (tripartite graph) *A graph $G(V, E)$ is called **tripartite** if all vertices V can be partitioned into three disjoint subsets $V = V_1 \cup V_2 \cup V_3$ and all edges E can be partitioned into two disjoint subsets $E = X \cup Y$ such that for all edges $e_X \in X$, e_X connects a vertex in V_1 to a vertex in V_2; for all edges $e_Y \in Y$, e_Y connects a vertex in V_2 to a vertex in V_3.*

Definition 1.6 (complete tripartite graph) *A tripartite graph $G(V, E)$ is called **complete** if $X = V_1 \times V_2$ and $Y = V_2 \times V_3$.*

The definition of perfect matching remain the same.

Definition 1.7 (path) *Let p be a **path** if $v_1, v_2, v_3 \in p, v_1 \in V_1, v_2 \in V_2, v_3 \in V_3$ and $p_X = (v_1, v_2), p_Y = (v_2, v_3) \in p$. Then, the path weight $W_p = \min(W_{p_X}, W_{p_Y})$.*

Definition 1.8 (assignment, tripartite) *Let $G(V, E)$ be a tripartite graph. Let F_X be the set of all perfect matchings on the bipartite graph $G_X(V_X, X)$. Respectively, let F_Y be the set of all perfect matchings on the bipartite graph $G_Y(V_Y, Y)$. Let $M_X \in F_X$, $M_Y \in F_Y$, and let P be the set of all paths from a vertex in V_1 to another vertex in V_3 in $M_X \cup M_Y$. Then, the set of edges in P is called a **assignment** in respect to G.*

Note that the tripartite graph assignment under our definition is not a matching under the definition of general graphs since there exist vertices (vertices in V_2) such that two edges are adjacent to each vertex. In that case, we cannot apply the matching algorithm of general graphs to this problem.

Similar to the bipartite case, any matching in a general tripartite graph will correspond to a matching in a complete tripartite graph. Also, since the weights of edges are all non-negative, it suffices to only consider perfect matchings in complete tripartite graphs.

Therefore, the TAP asks for the perfect matchings ${M_X}^*$, ${M_Y}^*$ such that:

$$\sum_{p \in P} W_p$$

is maximized.

However, in contrast to the BMP which has a polynomial-time solution, the TAP is believed to be NP-hard. We cannot simply identify the maximum-weight matching for the bipartite graphs G_X and G_Y to obtain the maximum-weight matching for the larger tripartite graph, G. This is because the left and right optimal matchings ${M_X}^*$, ${M_Y}^*$ are dependent on each other. Below is an example illustrating this.

Consider the tripartite graph in Figure 1.1:

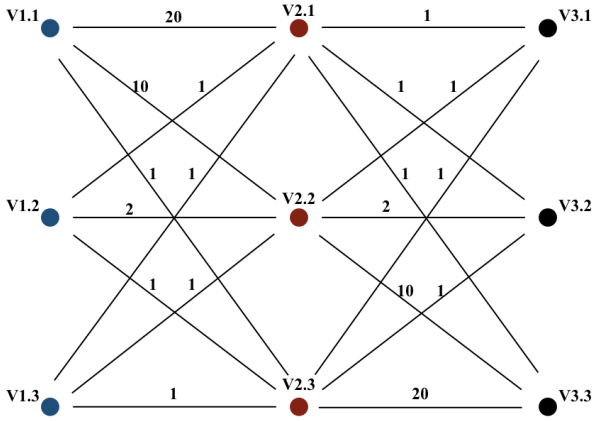

Figure 1.1: Tripartite graph

The left and right maximum-weight matching are presented in Figure 1.2 and Figure 1.3 and the union of the two matchings in Figure 1.4. Matchings are shown by bolded edges.

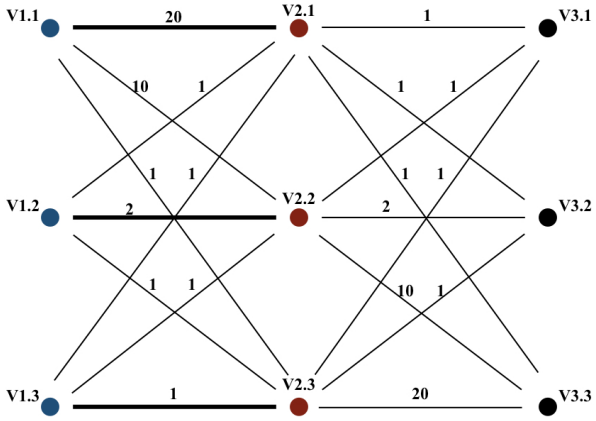

Figure 1.2: Left maximum-weight matching, M_X

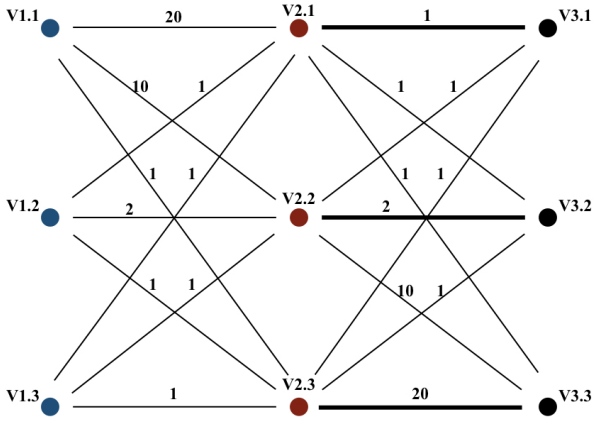

Figure 1.3: Right maximum-weight matching, M_Y

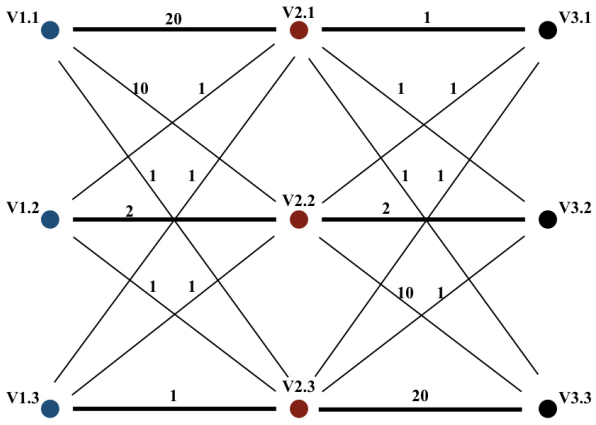

Figure 1.4: $M_X \cup M_Y$

The flow of $M_X \cup M_Y$ is $1+2+1 = 4$. However, this is not the extremal-weight tripartite assignment. The optimal matching has flow of $10+1+1 = 12$ presented in Figure 1.5.

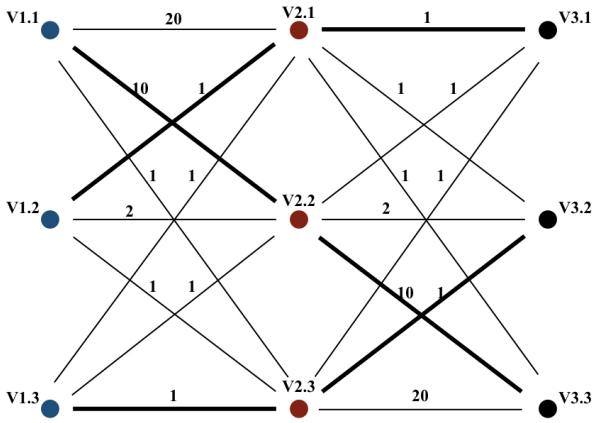

Figure 1.5: Extremal-weight tripartite assignment, M

This problem could be estimated through the bipartite bottleneck matching as explained in section 4. In this paper, we propose several gradient-based algorithms to improve on finding the optimal matching or a close-to-optimal matching.

2 The General Method

We approach the problem considering two separate matchings, M_X and M_Y. If M_X is fixed, the problem could be easily turned into a BMP by the following process. With a known M_X, we could generate an updated bipartite graph, $G'(V2, V3, Y')$ by updating the weights of edges in Y using:

$$W_{e=(u,v)} = \min(W_{e_X}, W_{e_Y})$$

for all $(u, v), u \in V_2, v \in V_3$, where $e_X \in M_X$ is the edge incident to u and $e_Y \in Y$ is the edge incident to both u and v.

Using the polynomial-time BMP algorithm on the graph G', we can obtain the optimal result. Therefore, the problem becomes searching for the matching M_X in the sample space of F_X that yields the optimal result. In

order to do this, we examine the structure of the set of matchings F_X. Since $|V_1| = |V_2|$, we can consider the set of vertices V_1 and V_2 as the same set. Then, for a matching $M_X \in F_X$, we consider M_X as a permutation of the set of vertices V_1, with the set of matchings F_X as the set of permutations. Note that F_X is a non-abelian group since the order of permutation matters. To Further this concept, we define k-swaps.

Definition 2.1 (k-swap) *For $k \in \mathbb{Z}$, a permutation ϕ on a matching M is called a k-swap if $\phi(M) \cup M$ contains n-k edges, and $M/(\phi(M) \cup M)$ contains k edges.*

Below is an example illustrating that k-swaps are non-abelian. Given a matching M as presented in Figure 3.1:

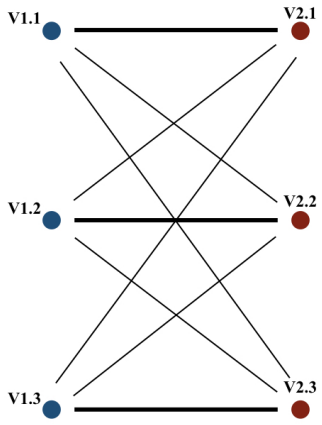

Figure 2.1: Original Matching, M

We define two 2-swaps, ϕ and τ. ϕ swaps the two edges in the matching that are incident to $V1.1$ and $V1.2$. τ swaps the two edges in the matching that are incident to $V1.2$ and $V1.3$. $\phi(M)$ is shown in Figure 3.2 and $\tau(M)$ in Figure 3.3:

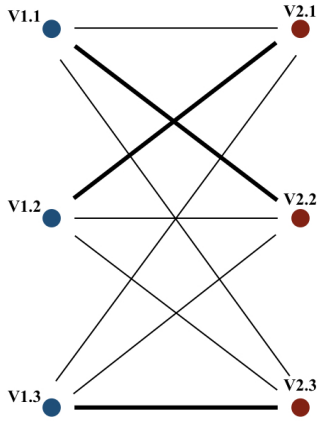

Figure 2.2: $\phi(M)$

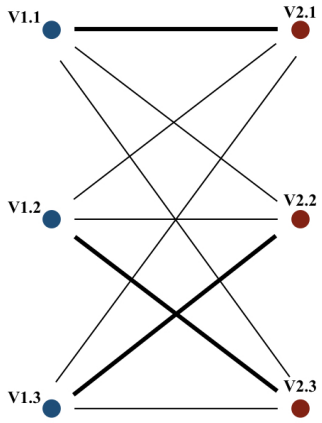

Figure 2.3: $\tau(M)$

Figure 3.4 and Figure 3.5 shows $\tau(\phi(M))$ and $\phi(\tau(M))$ respectively. Notice these two are not isomorphic if the edges have different weights assigned to them. Therefore, the group of k-swaps are non-abelian.

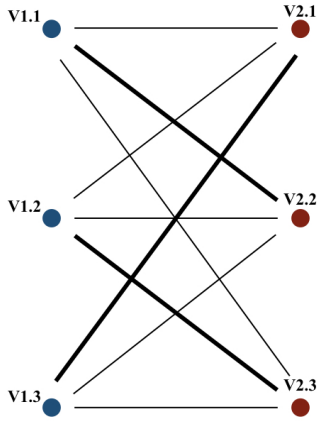

Figure 2.4: $\tau(\phi(M))$

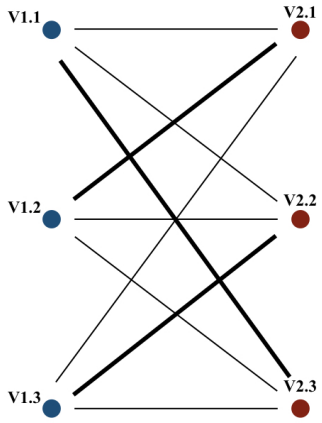

Figure 2.5: $\phi(\tau(M))$

In order to search for the optimal matching in Fx, we attempted several gradient-based methods with modifications to this specific problem, including

the hill-climbing method and the simulated annealing algorithm. The details are explained in the next section.

3 Three Optimization Algorithms

In this section, we introduce three optimization algorithms for solving the TAP.

3.1 Hill-Climbing Algorithm

Hill-climbing (HC) is an optimization method that starts with a random point in the sample space, F. Then, it makes increments to adjacent neighbors to achieve better results until no improvements can be made. First, we define neighboring matchings.

Definition 3.1 (k-neighbor) *Let M be a matching of bipartite graph $G(V, E)$. Then a matching M' is a k-neighbor of M if there a k-swap ϕ such that $\phi(M) = M'$.*

The swap size, k is a parameter in the algorithm. Below is the general structure:

Hill-climbing(G, k)
begin
    choose a random matching M_X
    output hill(G, k, M_X)

Function hill(G, k, M_X)
begin
    $f(M_X) \leftarrow$ maximum flow with the left matching M_X
    $N \leftarrow \{M | M$ is a k-neighbor of $M_X\}$
    foreach $M \in N$ with order of choice random **do**
        calculate $f(M)$
        if $f(M) > f(M_X)$ **then**
            output hill(G, k, M_X)
    else output $f(M_X)$

3.2 Modified Hill-Climbing Algorithm

The HC in the last section could be inefficient in some cases since it only accepts one M that yields a better result than M_X. Since it only accepts matchings of larger results, it could reach a point of local maxima, where for each of its neighbors $n \in N$, $f(n) <= f(M_X)$, but M_X is not optimal. Also, since it takes in the first matching that exceeds the previous results and does not take into account of other neighbors, this could lead to less accuracy.

Given this condition, we examine a modified version of the hill-climbing algorithm (HC2) that accounts for other neighbors. Below is the structure of the algorithm:

Modified Hill-climbing(G, k)
begin
choose a random matching M_X
output hill2(G, k, M_X)

Function hill2(G, k, M_X)
begin
$f(M_X) \leftarrow$ maximum flow with the left matching M_X
$N \leftarrow \{M | M$ is a k-neighbor of $M_X\}$
bestNeighbor $\leftarrow M \in N$ such that $f(M)$ is maximal
if f(bestNeighbor) $> f(M_X)$ **then**
output hill2$(G, k,$ bestNeighbor)
else output $f(M_X)$

3.3 Simulated Annealing Algorithm

Simulated Annealing (SA) is a more complicated algorithm used for optimization that simulates the heat dissipation of a material. We define neighboring matchings in the same way as we did in the hill-climbing methods. Then, we give the algorithm a temperature $T, T \in \mathbb{Q}$ and $0 < T < 1$. The algorithm always accepts neighbors whose value is larger than the value of the current matching, but it also accept neighbors whose value is smaller. The chance of accepting worse neighbors, P_w is determined by the current temperature, where a higher temperature gives a larger P_w. This takes into consideration

the previous problem of reaching a local maxima and not able to accept other matchings. Below is the general structure:

Simulated Annealing(G, k)

begin

Initialize T_s, T_f
$T \leftarrow T_s$
accepted $\leftarrow 0$
$\delta E_{avg} \leftarrow 0.0$
choose a random matching M_X
repeat for c cycles
$f(M_X) \leftarrow$ maximum flow with the left matching M_X
$N \leftarrow \{M | M$ is a k-neighbor of $M_X\}$
choose random matching $M \in N$
$\delta E \leftarrow |f(M) - f(M_X)|$
if $f(M) < f(M_X)$ **then**
if in first cycle **then**: $\delta E_{avg} \leftarrow \delta E$
$P_w \leftarrow e^{-\delta E / \delta E_{avg} \cdot T}$
choose random number p, $0 < p < 1$
if $p < P_w$ **then** accept $\leftarrow$ **True**
else accept $\leftarrow$ **False**
else accept $\leftarrow$ **True**
if accept = **True then**
$M_x \leftarrow M$
accepted $\leftarrow$ accepted + 1
$\delta E_{avg} \leftarrow (\delta E_{avg} \cdot (\text{ accepted } -1) + \delta E) / \text{ accepted}$
$T \leftarrow T \cdot (T_f / T_s)^{1.0/(c-1.0)}$
output $f(M_X)$

In this algorithm, there are two parameters. The size of the swap, k, is still a parameter since it determines the neighborhood. Also, the number of cycles, c, determines how many times the algorithm repeats itself. Analysis on the parameters will be provided in the next section.

4 Lower Bound Estimation from the Maximum-Cardinality Bottleneck Bipartite Matching Problem

In order to gain a better understanding of this problem, we take a look at the related maximum-cardinality bottleneck bipartite matching problem (BBMP).

Definition 4.1 (bottleneck bipartite matching) *For a bipartite graph $G(V, E)$, let F be the set of all perfect matchings M on G. A matching M° is called the* ***maximum bottleneck matching*** *on the bipartite graph G if:*

$$\min_{e \in M^\circ}\{W_e\} = \max_{M \in F} \min_{e \in M}\{W_e\}$$

We denote the minimum-weighted edge in M° as e°.

In other words, the BBMP finds the perfect matching in a bipartite graph M° such that M° maximizes the minimum edge in the matching. This problem could be solved by a polynomial-time algorithm presented by Punnen and Nair with the complexity of $O(n\sqrt{nm})$ or $O(n^5/2)$. [4]

The BBMP can provide a coarse estimation to our TAP problem. First, we have the following lemma:

Lemma 1 *Let $G(V,)$ be a tripartite graph with maximum-weight matching M^*. Let ${M_X}^\circ$ and ${M_Y}^\circ$ be the maximum bottleneck matchings of bipartite graphs $G(V_X, X)$ and $G(V_Y, Y)$, respectively. Let ${e_X}^\circ \in {M_X}^\circ$ and ${e_Y}^\circ \in {M_Y}^\circ$ be the minimum-weighted edges in the bottleneck matchings. Then,*

$$\begin{aligned}\sum_{p \in M^*} W_p &\geq n \cdot ({e_X}^\circ + {e_Y}^\circ) \\ &\geq 2n \cdot \min({e_X}^\circ, {e_Y}^\circ)\end{aligned}$$

Proof Consider in the flow of the maximum-weight matching M^*. We have:

$$\begin{aligned}\sum_{p\in M^*} W_p &\geq \sum_{p\in {M_X}^\circ \cup {M_Y}^\circ} W_p \\ &= \sum_{e_X\in {M_X}^\circ} W_{e_X} + \sum_{e_Y\in {M_Y}^\circ} W_{e_Y} \\ &\geq n\cdot({e_X}^\circ + {e_Y}^\circ) \\ &\geq n\cdot \min({e_X}^\circ, {e_Y}^\circ)\cdot 2\end{aligned}$$

where p denotes paths from V_1 to V_3. The equal sign is achieved if and only if ${M_X}^\circ \cup {M_Y}^\circ$ is a maximum-weight matching in the tripartite graph G and all edges in the matching have equal weights. One example that satisfies this condition is a tripartite graph (V_1, V_2, V_3, X, Y), $|V_1| = |V_2| = |V_3|$ and all edges have the same weight.

Therefore, the BBMP algorithm provides an lower-bound for the TAP. We will use this as a standard of examination for the other algorithms presented in this paper.

5 Improving Performance of the Algorithms

In this section, we examine the parameters in each algorithm and look for ways to improve both the accuracy and the runtime of our proposed algorithms. In addition, we analyze different edge weight distributions to see if different algorithms perform differently on various distributions. For all of the data presented in this section, more detailed data tables are provided in Appendix B.

5.1 Hill-Climbing Algorithms: HC and HC2

The major parameter in the hill-climbing algorithms is the size of the swap, k. For starters, increasing the parameter k will result in a larger number of permutations since:

$$\#\text{ of permutations} = \binom{n}{k}$$

increases exponentially as k increases. In the HC2 method, this will bring a longer runtime since the HC2 does calculation for every neighbor and chooses the best out of those neighbors. However, in the HC algorithm, it does not necessarily increase the runtime by a significant amount. In addition, a higher value of k does not necessarily mean better accuracy for the algorithms. Now, let's take a look at some data.

We design our testing as such: we test $k = 2, 3, 4$ on multiple different graphs. We randomly generate three graphs for each vertex number, $n = 5, 10, 15, 20$. For the sake of generality, we use edge weight of random distribution. We perform the two algorithms for 10 trials on each graph, with each trial starting with a different randomized matching. We measure the runtime in terms of seconds and we note down the flow of the matchings.

Table 1: Average Runtime of the HC with Parameter k

	n=5	n=10	n=15	n=20
k=2	0.03	1.54	16.2	78.27
k=3	0.04	1.58	17.24	83.24
k=4	0.03	1.55	16.63	85.87

Table 2: Average Runtime of the HC2 with Parameter k

	n=5	n=10	n=15	n=20
k=2	0.04	3.47	57.43	437.74
k=3	0.04	4.09	80.66	605.44
k=4	0.04	5.8	130.88	1426.86

Table 1 and Table 2 are tables on the average runtime of the HC and HC2. These results follows our hypothesis. For the HC2, the runtime increases drastically as k increases. The exception is n=5, since:

$$\binom{5}{2} = \binom{5}{3} > \binom{5}{4}$$

The case of HC, however, appeared to be different. Comparing by columns, there is not a lot of fluctuation as k increases. The runtime remains mostly

constant for small graphs and increases slowly in comparison to the HC2 case.

After examining the runtime, we examine the accuracy. Since enumerating every matching in order to obtain the optimal result is extremely time-inefficient for sufficiently large graphs, we are not able to use the enumeration method. Therefore, we set the maximum achieve over all trials to be the standard maximal result. matchings with 97% range of the maximal result will be considered as "accurate".

Table 3: Accuracy of HC with Parameter k

	n=5	n=10	n=15	n=20
k=2	0.9	0.83	0.86	0.83
k=3	0.93	0.83	0.9	0.86
k=4	0.87	0.77	0.86	0.86

Table 4: Accuracy of HC2 with Parameter k

	n=5	n=10	n=15	n=20
k=2	0.66	0.33	0.5	0.6
k=3	0.63	0.5	0.33	0.63
k=4	0.4	0.57	0.3	0.53

Table 3 and Table 4 are two tables that shows the 97% accuracy of the two hill-climbing algorithms. In table 3, we can see that for smaller graphs, a smaller choice of k yields better results. However, when it comes to performing the algorithm on larger graphs, an increase in the selection of k could increase the accuracy. Therefore, the optimal selection of k depends on the size of the graph.

In table 4, we notice that the selection of k does not have a particular effect on the accuracy. In fact, it may differ based on different graphs. Taking into consideration that a larger k would increase the runtime of the HC2 significantly, a smaller choice of k would be ideal.

Comparing Table 3 and Table 4, we noticed that the HC2 is significantly less accurate than the HC. This might be because selecting the "best neighbor" in each iteration prevents the algorithm from climbing up a less "steep"

hill and exploring more choices as it improves the result slower. In addition, taking into consideration that the HC2 takes a much longer runtime than the HC, the HC algorithm performs better than the HC2 as a whole.

5.2 Simulated Annealing

The number of cycles, c, is another parameter that is unique to the simulated annealing approach. Since larger graphs would require larger number of iterations to cover, we hypothesize that the optimal selection of the parameter, c, depends on the size of the graph. Therefore, perform the algorithm testing $c = 25, 35, 45, 55, 65, 75, 85, 95, 105, 115$on graphs of size $n = 5, 10, 15, 20$ to examine the best selection of c. For the sake of generality, we use edge weights of random distribution. We perform the two algorithms for 10 trials on each graph, with each trial starting with a different randomized matching. We measure the runtime in terms of seconds and we note down the flow of the matchings.

Table 5: Average Runtime of the SA

	n=5	n=10	n=15	n=20
c=25	0.11	0.76	0.47	0.7
c=35	0.17	0.30	0.59	3.97
c=45	0.19	0.48	0.54	5.60
c=55	0.24	0.56	0.56	5.47
c=65	0.27	0.47	0.58	3.23
c=75	0.31	0.43	0.65	5.23
c=85	0.33	0.65	0.61	6.37
c=95	0.38	0.51	1.52	3.25
c=105	0.42	0.38	2.59	4.84
c=115	0.45	0.58	4.02	6.51

Table 5 shows the average runtime for the SA in our testing, in seconds. From the table, we can see that the runtime increases with the increase of cycles. However, unlike with the parameter k, this increase in linear based on the nature of the algorithm.

Table 6: Accuracy of SA

	n=5	n=10	n=15	n=20
c=25	0.17	0.10	0.00	0.03
c=35	0.13	0.07	0.03	0.03
c=45	0.17	0.07	0.03	0.00
c=55	0.00	0.03	0.07	0.03
c=65	0.00	0.67	0.10	0.10
c=75	0.00	0.33	0.07	0.13
c=85	0.00	0.00	0.07	0.10
c=95	0.00	0.07	0.10	0.10
c=105	0.00	0.07	0.07	0.10
c=115	0.00	0.07	0.07	0.10

Table 6 shows the 97% accuracy of the SA. From this table, we can see that smaller graphs(n=5 and n=10) might need a smaller choice of c to get better accuracy, while in larger graphs, increasing the number of cycles will yield higher accuracy.

The number of swaps, k is also a parameter in this algorithm. Since the definition of neighbors are the same across all three algorithms, the theoretical analysis for the parameter k would be the same. Since the SA does not calculate all neighbors and will choose the first better neighbor, the runtimes does not increase drastically as in the HC2. When the size of the graph is large, a comparatively larger k would yield better results.

5.3 Cross Comparison of the Algorithms with Distribution Analysis

First, using the data presented in Table 1, Table 2, and Table 6, we examine the runtime of the different algorithms. The HC2 has the longest runtime of the three algorithms, since it runs through all neighbors for each matching that is considered. For the SA, the runtime depends on the number of cycles that it is executed for. Therefore, SA is the most flexible, since it can compute approximation matchings in shorter or longer runtime, depending on the computation needs. The HC is also a fast algorithm that can be used for larger graph assignments.

For comparing the algorithms in term of accuracy, we take into account different kinds of weight distributions. We consider random, normal and uniform distribution, which could appear in potential real-world applications of the extremal-weight tripartite matching problem. Here, we restrict the graph to $n = 15$ and the parameters to $k = 2$ and $c = 95$. Below is a table of the results. In the normal distribution, we set the mean of the edge weights as 10 and variance as 1. In the uniform distribution, we set upper bound as 10 and lower bound as 1.

By a straight forward comparison in Table B.4, we can see clearly that the HC outperforms the SA in every distribution in each graph. Therefore, we need a new measure in comparing the result accuracy. We look at the percentage accuracy, where we add up all numbers in a testing group and divide it by the maximum obtained $\times$ the number of data in the group.

Table 7: Average ercentage Accuracy of the SA and HC for Different Weight Distributions

	random	normal	uniform
SA	0.67	0.91	0.70
HC	0.98	0.99	0.98
Bottleneck	0.51	0.89	0.52

Table 7 is a table that shows the averaged percentage accuracy for each case. In this table, we see that the SA and HC both outperforms the bottleneck matching method. SA performs well under normal distribution but not as well under random and uniform distribution. However, HC is accurate across all three distributions. Therefore, when a fast running time is required under a normal distribution, the SA could be applied. In other cases, especially cases where accuracy is more valued, the HC is a better choice.

6 Conclusion and Future Directions

In conclusion, the hill-climbing and simulated-annealing methods outperforms the bottleneck matching method and improves the accuracy for the estimation of the extremal-weight tripartite assignment problem. The HC method has the highest accuracy and a relatively short runtime. The HC2

needs a long runtime and is not as accurate as the HC. The parameter k in both the HC and HC2 should be chosen according to the size of the graph, in the way that smaller values of k works better for smaller graphs, and vice versa. The SA is the most flexible, since it has short runtime and the runtime can be controlled through adjusting the parameter, cycles.

The algorithm in this paper could be extended in several ways. First, the algorithm could be future improved for an extremal-matching problem with integer edge weights using similar ideas presented in [5]. Furthermore, this problem could be extended to assignments on 3-uniform hypergraphs, since the problem in this paper is a special case of its corresponding problem on 3-uniform hypergraph assignment.

In addition, this problem has multiple applications. This problem was originally derived from a scenario of signal processing in electric engineering. Let $G(V, E)$ be a tripartite graph, the the problem could be formulated as signal transferring from V_1 to V_2 to V_3. V_1 sends out signals, V_2 transfers, and V_3 receives. Our algorithm provides estimation to the optimal way of transferring signals. Furthermore, this problem has a wide range of applications, including image retrieval [6] and studies of molecular structures [7].

References

[1] Kuhn, Harold W. *The Hungarian method for the assignment problem.* Naval research logistics quarterly 2, no. 1-2 (1955): 83-97.

[2] Hopcroft, John E., and Richard M. Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.* SIAM Journal on computing 2, no. 4 (1973): 225-231.

[3] Derigs, U. *The shortest augmenting path method for solving assignment problems—motivation and computational experience.* Annals of Operations Research 4, no. 1 (1985): 57-102.

[4] Punnen, Abraham P., and K. P. K. Nair. *Improved complexity bound for the maximum cardinality bottleneck bipartite matching problem.* Discrete Applied Mathematics 55, no. 1 (1994): 91-93.

[5] Gabow, Harold N., and Robert E. Tarjan. *Algorithms for two bottleneck optimization problems.* Journal of Algorithms 9, no. 3 (1988): 411-417.

[6] Pulla, Chandrika, and C. V. Jawahar. *Tripartite graph models for multi modal image retrieval.* In Proceedings of the British Machine Vision Conference, pp. 1-11. 2010.

[7] Leong, Samuel, Ming-Yang Kao, Tak-Wah Lam, Wing-Kin Sung, and Siu-Ming Yiu. *Predicting RNA secondary structures with arbitrary pseudoknots by maximizing the number of stacking pairs.* Journal of Computational biology 10, no. 6 (2003): 981-995.

A Appendix: Example Code

Below is a set of example codes in Python 3.

A.1 The Vertex Class

```
class vertex:
    def __init__(self, name=None):
        """
        @param name: int, used for printing out graphs or matchings.
        """

        self.name = name

    def __eq__(self, other_vtx):
        """
        @param other_vtx: vertex, vertex that is compared to
        @return: boolean, whether 'self' and 'otherVtx' are the same
        """
        if isinstance(other_vtx, self.__class__):
            return hash(id(self)) == hash(id(other_vtx))
        else:
            return False

    def __str__(self):
        """
        @return: str, name of vertex
        """
        return self.name


# Below are some methods related to vertices
def vertex_list(vtx_number):
    """
    Makes a list of vertices of certain size
    @param vtx_number: int, number of vertex in the list
    @return: list, vertices with names 0, 1, ..., vtx_number-1
    """
```

```
    vtx_list = []

    for i in range(0, vtx_number):
        vtx_list.append(vertex(str(i)))
    return vtx_list


def get_vertex_list(vtx_ist):
    """
    Returns a list with the names of all vertices
    @param vtx_list: the list of all vertices to print
    @return: the list of names of all vertices
    """
    return_list = []

    for vtx in vtx_ist:
        return_list.append(str(vtx))
    return return_list
```

A.2 The Edge Class

```
class edge:
    # v is a list of vertices
    # w is int: the weight of the edge
    def __init__(self, u, v, w=0):
        """
        @param u: vertex, starting vertex
        @param v: vertex, ending vertex
        @param w: int or float, weight of edge
        """
        self.u = u
        self.v = v
        self.w = w

    def __eq__(self, other_edge):
        """
        @param other_edge: edge, edge that is compared to
        @return: boolean, whether 'self' and 'other_edge' are the
            same
        """
        if isinstance(other_edge, self.__class__):
            return hash(id(self)) == hash(id(other_edge))
        else:
            return False

    def get_edge(self):
        """
        when printing , print from_vtx, to_vtx and weight
        """
        return [str(self.u), str(self.v), self.w]

    def get_from_vtx(self):
        return self.u

    def get_to_vtx(self):
        return self.v

    def get_weight(self):
        return self.w
```

```
def get_edge_list(edge_list):
    """
    print out the edges in the list
    @param edge_list: list to be printed
    @return: list, name of edges
    """
    return_list = []
    for e in edge_list:
        return_list.append(e.get_edge())
    return return_list
```

A.3 The Matching Class

```
class matching:
    """
    a matching is a list of edges without weights
    """

    def __init__(self, edge_list):
        """
        @param edge_list: list, edges in matching without weights
        """
        self.edge_list = edge_list

    def __str__(self):
        """
        print out the matching
        """
        return_list = []
        for e in self.edge_list:
            return_list.append((str(e.get_from_vtx()),
                str(e.get_to_vtx())))
        return str(return_list)

    def get_edge_list(self):
        """
        @return: list, edge_list
        """
        return self.edge_list

    def get_flow(self):
        """
        @return: int or float, the flow of the matching
        """
        flow = 0
        for edge in self.edge_list:
            flow += edge.get_weight()

    def get_weights(self):
        weight_list = []
        for e in self.edge_list:
```

```
            weight_list.append(e.get_weight())
        return weight_list

    def is_matched(self, start_vtx, end_vtx):
        """
        @param start_vtx: starting vertex
        @param end_vtx: ending vertex
        """
        tup = (start_vtx, end_vtx)
        for element in self.edge_list:
            if element == tup:
                return True
            return False

    def bigOplus(self, path):
        """
        Calculate the symmetrical difference
        @param path: list, of edges in path
        @return: matching, after the symmetrical difference
        """
        result = []
        edge_list = self.edge_list
        for e in edge_list:
            if e not in path:
                result.append(e)
        for e in path:
            if e not in edge_list:
                result.append(e)
        return matching(result)
```

A.4 The Graph Class

```
class graph:
    def __init__(self, V, E):
        """
        @param V: list, vertices
        @param E: list, edges
        """
        self.V = V
        self.E = E

    def get_weight(self, start_vtx, end_vtx):
        """
        list the weight of the edge with starting vertex u and
            ending vertex v
        @param start_vtx: vertex, starting vertex
        @param end_vtx: vertex, ending vertex
        @return: int or float, the weight of the edge (start_vtx,
            end_vtx)
        """
        edge_list = self.E
        for e in edge_list:
            if (e.get_from_vtx() == start_vtx) and (e.get_to_vtx()
                == end_vtx):
                return e.get_weight()
```

A.5 The Bipartite Graph Subclass

```
class bipartite(graph):
    def __init__(self, V1=[], V2=[], E=[]):
        """
        @param V1: list, first disjoint set of vertices
        @param V2: list, second disjoint set of vertices
        @param E: list, edges
        """
        self.V1 = V1
        self.V2 = V2
        self.E = E
        self.V = V1 + V2

    def __str__(self):
        """
        Printing the bipartite graph
        @return: list, names of vertices in v1 and v2 and list of
            edges.
        """
        V1_list = get_vertex_list(self.V1)
        V2_list = get_vertex_list(self.V2)
        E_list = get_edge_list(self.E)
        return str([V1_list, V2_list, E_list])

    def __len__(self):
        """
        @return: int, size of one set of vertices
        """
        return len(self.V1)

    def get_matrix(self):
        """
        Putting the graph into Matrix form
        @return: matrix, [u][v]th element denoting the weight of
            the edge(u, v)
        """
        matrix = []
        vnum = len(self)
        for i in range(vnum): # build matrix
```

```
            matrix.append([])

        for i in range(vnum): # put 0 for edge weights
            for j in range(vnum):
                matrix[i].append(0)

        for i in range(vnum):
            for j in range(vnum):
                u = self.V1[i]
                v = self.V2[j]
                matrix[i][j] = self.get_weight(u, v)
        for i in range(len(matrix)):
            for j in range(len(matrix[i])):
                if matrix[i][j] is None:
                    matrix[i][j] = 0
        return matrix

    def munks(self):
        """
        Applying the Hungarian algorithm.
        @return: int, flow of the maximum matching
        """
        matrix = self.get_matrix()
        neg = copy.deepcopy(matrix)
        # negative: all the Munkres() method is for minimum-weight
            matching.
        for i in range(len(neg)):
            for j in range(len(neg[i])):
                neg[i][j] = -neg[i][j]
        m = Munkres()
        indices = m.compute(neg)
        # calculates the total flow
        total = 0
        for row, column in indices:
            value = matrix[row][column]
            total += value
        return total

    def assign_weight(self, weightlist):
        """
```

```
    Assigns weights to edges in a bipartite graph
    @param weightlist: list, integers for each weight
    @return: None
    """
    A = self.V1
    B = self.V2
    E = self.E
    i = 0
    for a in A:
        for b in B:
            E.append(edge(a, b, weightlist[i]))
            i = i+1

def edge_with_vtx(self, vtx):
    """
    list the edges in graph that contains a vertex
    @param vtx: vertex, the vertex that edges contains
    @return: list, edges
    """
    return_list = []
    for e in self.E:
        if e.get_from_vtx() == vtx or e.get_to_vtx() == vtx:
            return_list.append(e)
    return return_list

def agmPath(self, start_vtx, end_vtx, Mx):
    """
    Searches for augmenting path regarding a bipartite graph
    @param start_index: index of starting vertex; index is
        original index
    @param end_index: index of ending vertex
    @param Mx: matching to apply augmenting path on
    @return: list, edges in augmenting path
    """
    visited = [] # list of visited vertices
    queue = [] # queue to draw out vertices to check
    # innitialize
    path = []
    status = "matched" # search for matched or unmatched edge
        next
```

```
queue.append((start_vtx, path))
while queue:
    (current, path) = queue.pop(0)
    if current not in visited:
        if current == end_vtx:
            return path
        visited.append(current)
        # Search for matched and unmatched edges
            alternatingly
        if status == "unmatched":
            for e in self.edge_with_vtx(current):
                vtx = e.get_from_vtx()
                queue.append((vtx, path+[e]))
            status == "matched"
        else:
            for e in self.edge_with_vtx(current):
                vtx = e.get_from_vtx()
                if Mx.is_matched(current, vtx):
                    queue.append((vtx, path+[e]))
                    status = "unmatch"
    return None
```

A.6 The Tripartite Graph Subclass

```
# Some iteration methods for listing neighbors
def choose_iter(elements, length):
    """
    yields all permutations of length k
    @param elements: elements to choose from for permutation
    @param length: length of desired permutation
    @return: list of permutations in form of tuples
    """
    for i in range(len(elements)):
        if length == 1:
            yield (elements[i],)
        else:
            for next in choose_iter(elements[i+1:len(elements)],
                length-1):
                yield (elements[i],) + next


def choose(n, k):
    """
    Generate a list of permutations of length k
    @param n: integer
    @param k: integer, length of permutation
    @return: the desired list
    """
    numbers_list = []
    for i in range(n):
        numbers_list.append(i)
    return list(choose_iter(numbers_list, k))


class tripartite(graph):
    def __init__(self, V1, V2, V3, X, Y):
        """
        @param V1, V2, V3: list of disjoint vertices
        @param X, Y: list of edges in tripartite graph
        """
        self.V1 = V1
        self.V2 = V2
```

```
        self.V3 = V3
        self.X = X
        self.Y = Y
        self.V = V1 + V2 + V3
        self.E = X + Y

    def __str__(self):
        """
        Printing the tripartite graph
        """
        V1_list = get_vertex_list(self.V1)
        V2_list = get_vertex_list(self.V2)
        V3_list = get_vertex_list(self.V3)

        X_list = get_edge_list(self.X)
        Y_list = get_edge_list(self.Y)
        return str([V1_list, V2_list, V3_list, X_list, Y_list])

    def __len__(self):
        """
        @return: int, size of one set of vertices
        """
        return len(self.V1)

    def update_Y(self, Mx):
        """
        Produces a bipartite graph of equivalence maximum flow.
        @param Mx: matching, the left matching of the graph on X
        @return: bipartite graph
        """
        vnum = len(self)
        result_bipartite = bipartite(self.V2, self.V3, [])
        for e in Mx.get_edge_list():
            vtx_from = e.get_from_vtx()
            vtx_to = e.get_to_vtx()
            weight = self.get_weight(vtx_from, vtx_to)
            for index in range(vnum):
                start_vtx = vtx_to
                end_vtx = self.V3[index]
                original_weight = self.get_weight(start_vtx, end_vtx)
```

```
                weight = min(original_weight, weight)
                result_bipartite.E.append(edge(start_vtx, end_vtx,
                    weight))
        return result_bipartite

    def get_max_Y(self, Mx):
        """
        Calculates the maximum flow givin the left matching Mx
        @param Mx: matching, the left matching of the graph on X
        @return: flow of the maximum-weight matching
        """
        bpY = self.update_Y(Mx)
        return bpY.munks()

    def get_neighbor(self, Mx, k):
        """
        @param Mx: current matching
        @param k: integer, swap size
        @return: list of matching neighbors
        """
        vnum = len(self)
        numbers_list = choose(vnum, k)
        matching_list = []
        for element in numbers_list:
            edge_list = []
            for e in Mx.get_edge_list():
                vtx_from = e.get_from_vtx()
                vtx_to = e.get_to_vtx()
                weight = copy.copy(self.get_weight(vtx_from, vtx_to))
                edge_list.append(edge(vtx_from, vtx_to, weight))
            for i in range(len(element)):
                edge_list[element[i-1]].v =
                    Mx.edge_list[element[i]].v
            matching_list.append(matching(edge_list))
        return matching_list
```

A.7 The Generator Method

```
def generator(vnum, distribution, param):
    """
    @param vnum: int, number of vertices in each group
    @param distribution: string, either "random", "normal" or
        "uniform"
    @param param:
        in random: largest edge weight
        in nomal: list of mu and sigma (mean and variance)
        in uniform: list of range (high and low)
    """
    A = vertex_list(vnum)
    B = vertex_list(vnum)
    C = vertex_list(vnum)
    X = []
    Y = []

    if distribution == "random":
        wx = np.random.randint(param, size=np.square(vnum))
        wy = np.random.randint(param, size=np.square(vnum))

    elif distribution == "normal":
        mu = param[0]
        sigma = param[1]
        w = np.random.normal(mu, sigma, 2*np.square(vnum))
        for i in range(len(w)):
            w[i] = int(w[i])
        wx = w[:np.square(vnum)]
        wy = w[np.square(vnum):]

    elif distribution == "uniform":
        low = param[0]
        high = param[1]
        w = np.random.uniform(low, high, 2*np.square(vnum))
        for i in range(len(w)):
            w[i] = int(w[i])
        wx = w[:np.square(vnum)]
        wy = w[np.square(vnum):]
```

```
    wx = list(wx)
    wx = list(wx)
    bipartite(A, B, X).assign_weight(wx)
    bipartite(B, C, Y).assign_weight(wy)
    return tripartite(A, B, C, X, Y)
```

A.8 The HC Method

```
def hill(graph, Mx, k=2):
    # as explained in section 4.1
    maxnum = graph.get_max_Y(Mx)
    maxmatch = Mx
    neighbors = graph.get_neighbor(Mx, k)
    num = len(neighbors)
    while neighbors != []:
        index = np.random.randint(0, num)
        index = int(index)
        matching = neighbors.pop(index)
        num = num - 1
        result = graph.get_max_Y(matching)
        if result > maxnum:
            maxnum = result
            maxmatch = matching
            return hill(graph, maxmatch)
    return maxnum
```

A.9 The HC2 Method

```
def hill2(graph, Mx, k=2):
    # as explained in section 4.2
    maxnum = graph.get_max_Y(Mx)
    neighbors = graph.get_neighbor(Mx, k)
    bestNeighbor = None
    bestNeighborNum = 0
    for matching in neighbors:
        result = graph.get_max_Y(matching)
        boolean = False
        if (result > maxnum) and (bestNeighborNum == 0):
            boolean = True
        elif (bestNeighborNum != 0) and (result > bestNeighborNum):
            boolean = True
        if boolean:
            maxnum = result
            currentNeighbor = {}
            for matching in neighbors:
                currentNeighbor[graph.get_max_Y(matching)] = matching
            bestNeighborNum = max(currentNeighbor.keys())
            if bestNeighborNum > maxnum:
                bestNeighbor = currentNeighbor.get(bestNeighborNum)
    if bestNeighbor is None:
        return maxnum
    else:
        return hill2(graph, bestNeighbor)
```

A.10 The SA Method

```
def anneal(graph, cycles, Mx, k=2):
    # as explained in section 4.3
    acceptedSol = 0.0
    pWorseStart = 0.7
    pWorseEnd = 0.001
    t1 = -1.0/math.log(pWorseStart)
    tFinal = -1.0/math.log(pWorseEnd)
    frac = (tFinal/t1)**(1.0/(cycles-1.0)) # reduction of T
    currentValue = graph.get_max_Y(Mx)
    acceptedSol = acceptedSol + 1.0
    tCurrent = t1
    deltaE_avg = 0.0
    for i in range(cycles):
        print("Cycle: "+str(i)+" with Temperature: "+str(tCurrent))
        neighborList = graph.get_neighbor(Mx, k)
        neighborNum = len(neighborList)
        index = np.random.randint(neighborNum)
        matching = neighborList[index]
        value = graph.get_max_Y(matching)
        print("value: "+str(value)+" current value:
            "+str(currentValue))
        deltaE = abs(value - currentValue)
        if (value < currentValue):
            if (i == 0):
                deltaE_avg = deltaE
            p = math.exp(-deltaE/(deltaE_avg * tCurrent))
            if (random.random() < p):
                accept = True
            else:
                accept = False
        else:
            accept = True
        if accept:
            currentValue = graph.get_max_Y(matching)
            acceptedSol = acceptedSol + 1.0
            deltaE_avg = (deltaE_avg*(acceptedSol-1.0) + deltaE) /
                acceptedSol
        tCurrent = frac * tCurrent
```

```
    return currentValue
```

A.11 The Bottleneck Estimation Method

```
def bottle(graph):
    """
    Calculate the bottleneck matching of the bipartite graph
    @param graph: bipartite graph
    @return: matching, bottleneck matchign
    """
    vnum = len(graph)
    edge_list = []
    for i in range(vnum):
        weight = graph.get_weight(graph.V1[i], graph.V2[i])
        e = edge(graph.V1[i], graph.V2[i], weight)
        edge_list.append(e)
    Mx = matching(edge_list)

    path = []
    while path is not None:
        weight_list = []
        for e in Mx.edge_list:
            weight_list.append(e.get_weight())
            min_weight = min(weight_list)
            min_index = weight_list.index(min_weight)
            min_edge = Mx.edge_list[min_index]

        for e in graph.E:
            if e == min_edge or e.get_weight() < min_weight:
                graph.E.remove(e)

        start = min_edge.get_from_vtx()
        end = min_edge.get_to_vtx()
        Mx.edge_list.pop(min_index)
        path = graph.agmPath(start, end, Mx)
        if path is None:
            return Mx
        Mx.edge_list = Mx.bigOplus(path)
    return Mx


def bottleneck(graph):
```

```
"""
Calculate the bottleneck estimation
@param graph: tripartite graph
@return: estimation
"""
self_X = bipartite(graph.V1, graph.V2, graph.X) # no change
self_Y = bipartite(graph.V2, graph.V3, graph.Y) # no change
matching_X = bottle(self_X)
matching_Y = bottle(self_Y)
weights_X = matching_X.get_weights()
weights_Y = matching_Y.get_weights()
x = min(weights_X)
y = min(weights_Y)
minimum = min(x, y)
vnum = len(graph)
return minimum * vnum
```

A.12 The Ennumeration Method

```
def ennumerator(graph):
    """
    Ennumerates the matchinigs to calculate the maximum-weight
        matching
    @param graph: tripartite graph
    @return: tuple of flow of maximum-weight matching and the
        matching
    """
    vnum = len(graph)
    vtx_list = []
    for i in range(vnum):
        vtx_list.append(i)

    permuteList = list(itertools.permutations(vtx_list))
    result = 0
    maxMatch = None

    for permutation in permuteList:
        matchlist = []
        for i in range(vnum):
            matchlist.append((i, permutation[i]))
        Mx = matching(matchlist)
        current = graph.get_max_Y(Mx)[0]
        if current > result:
            result = current
            maxMatch = Mx
    return result, maxMatch.edgelist
```

B Appendix: Tables for Running Results

B.1 Running Results for HC

B.2 Running Results for HC2

B.3 Running Results for SA

B.4 Cross Comparison and Distribution Analysis

Table B.1.1: HC Runtime

Time (s)	n=5			10			15			20		
	k=2	3	4	2	3	4	2	3	4	2	3	4
Graph 1	0.05	0.03	0.01	1.38	1.48	1.46	12.62	12.17	12.89	78.20	83.24	132.10
	0.03	0.04	0.05	1.60	1.83	0.94	18.74	12.89	23.88	91.11	89.85	83.72
	0.05	0.04	0.03	1.67	2.48	1.70	13.28	17.19	11.70	52.60	63.82	103.70
	0.04	0.03	0.04	1.47	1.55	1.55	20.07	22.07	17.95	76.32	94.14	113.38
	0.04	0.07	0.02	1.94	1.36	1.53	15.45	17.62	12.86	74.73	43.52	67.62
	0.02	0.02	0.01	0.97	1.37	1.03	19.58	16.25	27.88	62.52	122.67	71.75
	0.02	0.02	0.01	1.19	0.98	1.33	21.41	12.41	22.34	82.08	68.44	117.78
	0.04	0.04	0.06	0.91	1.54	1.37	16.08	16.61	15.88	65.15	72.79	92.26
	0.06	0.07	0.11	2.90	1.41	2.66	22.38	19.45	14.68	80.57	95.60	118.88
	0.02	0.03	0.06	1.91	1.52	2.27	18.80	19.20	15.63	82.37	42.33	92.68
Graph 2	0.03	0.03	0.02	1.79	2.15	1.13	17.77	18.54	22.60	95.68	124.56	75.02
	0.03	0.08	0.04	1.87	2.05	3.10	9.79	18.06	16.97	109.07	97.63	79.70
	0.04	0.04	0.04	1.55	1.27	1.44	10.40	19.73	16.27	107.48	98.02	97.38
	0.05	0.05	0.04	1.70	1.42	2.42	18.35	20.47	11.17	69.38	95.67	56.84
	0.03	0.03	0.04	2.74	1.74	1.88	7.96	15.63	18.29	101.62	71.22	78.12
	0.05	0.04	0.01	1.01	1.23	2.08	14.02	23.28	15.71	44.49	78.16	106.61
	0.06	0.05	0.06	1.17	2.71	0.98	21.64	21.78	20.02	81.27	170.55	64.01
	0.03	0.03	0.04	1.17	1.62	1.42	16.30	16.33	20.37	75.42	44.79	68.64
	0.03	0.05	0.02	0.89	0.87	1.30	14.13	20.16	17.20	100.99	97.71	91.08
	0.02	0.05	0.04	1.20	1.25	1.46	23.04	17.79	15.66	97.61	84.50	98.05
Graph 3	0.03	0.03	0.03	1.65	1.33	2.59	18.26	16.96	21.98	70.81	84.32	69.86
	0.02	0.01	0.01	1.02	2.12	0.97	13.76	13.37	12.02	97.82	71.93	72.00
	0.01	0.02	0.03	1.51	1.91	0.95	13.39	13.61	11.09	66.22	109.28	90.10
	0.03	0.05	0.04	0.89	1.97	1.06	11.77	20.43	17.32	58.52	67.73	42.62
	0.02	0.01	0.01	1.84	0.96	0.81	24.10	17.60	12.68	73.78	83.45	76.95
	0.02	0.02	0.03	2.14	1.52	1.15	14.08	11.70	14.45	61.97	71.57	96.63
	0.03	0.03	0.02	1.66	1.16	1.54	16.13	22.79	15.06	62.05	56.81	63.94
	0.03	0.05	0.01	1.20	2.03	0.88	15.49	12.30	13.01	72.07	61.18	118.35
	0.01	0.01	0.02	1.46	1.05	1.30	17.08	15.31	12.35	77.69	56.96	85.06
	0.02	0.03	0.03	1.46	1.58	2.02	10.17	15.60	19.10	78.58	94.77	51.38
Average	0.03	0.04	0.03	1.53	1.58	1.54	16.20	17.24	16.63	78.27	83.24	85.87

Table B.1.2: HC Matching Results

	n=5			10			15			20		
	k=2	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**
Graph 1	36.86	36.86	36.86	85.21	85.21	82.08	129.17	128.30	131.99	173.63	178.71	175.36
	36.86	36.86	36.86	84.51	85.21	81.24	128.86	129.81	126.45	176.36	177.99	175.07
	36.86	36.86	36.86	85.21	85.21	81.24	129.34	126.44	130.20	173.02	178.34	178.20
	36.86	36.86	36.86	85.21	85.21	84.51	130.18	131.96	129.84	174.33	175.17	174.60
	36.86	36.86	36.86	85.21	80.66	84.51	130.25	129.86	126.52	176.73	174.81	176.85
	36.86	36.86	36.86	81.75	85.21	83.73	129.75	128.57	130.09	176.44	176.47	175.06
	36.86	30.91	36.86	85.21	83.73	85.21	131.22	130.18	130.66	177.05	174.30	177.11
	36.86	36.86	36.86	85.21	82.98	84.51	128.57	130.18	129.51	175.47	170.76	176.37
	36.86	36.86	36.86	84.51	84.51	84.51	129.05	130.80	130.29	178.60	176.56	177.17
	36.86	36.86	36.86	82.08	81.75	85.21	130.92	127.04	128.42	176.80	176.12	177.72
Graph 2	37.03	37.03	37.03	85.14	85.32	83.16	129.93	131.46	129.90	175.37	177.89	177.82
	37.03	37.03	37.03	79.85	82.70	81.88	127.48	131.96	128.71	179.27	178.04	175.68
	37.03	37.03	37.03	84.51	84.51	83.82	128.56	131.13	130.66	174.71	171.94	175.83
	37.03	37.03	37.03	85.25	83.16	83.62	128.87	130.13	127.52	177.97	176.79	179.26
	37.03	37.03	37.03	85.14	83.62	83.94	127.38	128.39	131.96	179.12	176.82	172.95
	37.03	37.03	32.67	84.97	83.62	84.52	127.04	129.93	131.25	172.71	175.58	176.17
	37.03	37.03	37.03	83.94	83.16	83.62	130.73	131.99	130.73	176.27	174.81	175.02
	37.03	37.03	37.03	85.25	84.97	84.52	131.13	127.97	129.76	174.11	168.43	171.97
	37.03	37.03	37.03	83.62	80.87	83.62	129.35	130.02	129.71	173.01	175.56	175.83
	37.03	37.03	37.03	85.42	83.62	84.52	132.09	131.45	130.04	177.99	170.98	178.39
Graph 3	31.55	31.55	24.24	86.35	86.35	86.12	124.76	125.14	122.97	172.12	173.58	170.67
	31.55	31.11	31.11	86.12	86.12	81.78	122.34	121.06	120.18	176.08	174.04	172.96
	29.81	31.55	31.55	86.12	81.84	84.93	121.66	121.53	123.58	175.77	174.92	175.81
	29.81	31.55	31.55	84.93	84.93	81.84	120.70	121.46	122.57	170.82	171.67	174.21
	31.55	31.11	31.11	81.84	82.11	86.35	121.43	124.47	123.19	170.76	174.44	168.54
	29.81	31.55	30.59	86.12	86.12	86.12	119.25	120.49	123.29	173.19	175.98	176.02
	31.55	31.55	29.81	86.12	86.35	86.12	123.60	124.86	122.80	174.24	173.50	175.87
	31.55	31.55	25.89	80.28	86.12	79.84	122.08	121.57	122.38	177.83	175.18	175.66
	30.59	31.55	31.55	86.12	81.84	86.35	120.79	124.85	121.59	173.97	177.34	177.54
	30.59	29.81	31.55	86.35	86.35	86.12	122.23	125.14	121.83	172.81	176.35	174.98

Table B.2.1: HC2 Runtime

Time(s)	n=5			10			15			20		
	k=2	3	4	2	3	4	2	3	4	2	3	4
Graph 1	0.03	0.01	0.01	3.13	3.53	4.10	45.56	62.50	117.13	397.78	554.56	1412.80
	0.04	0.02	0.01	3.14	3.46	4.17	81.44	75.77	132.77	116.73	313.60	1116.87
	0.04	0.04	0.02	3.01	4.15	7.97	44.69	62.12	101.94	240.14	513.79	1437.23
	0.05	0.06	0.07	4.53	4.91	8.17	59.35	75.68	107.71	529.56	518.63	1524.70
	0.03	0.04	0.01	6.30	4.35	7.63	75.57	84.16	126.54	563.30	756.63	1241.46
	0.06	0.04	0.01	1.58	2.80	4.67	96.85	100.35	162.21	198.99	680.82	1159.83
	0.07	0.06	0.05	3.77	4.18	8.12	73.56	90.72	138.94	513.83	513.39	1402.43
	0.06	0.02	0.03	2.89	4.10	6.38	52.57	69.38	117.89	155.92	314.09	1156.03
	0.09	0.07	0.01	2.24	3.56	6.11	75.20	68.39	154.67	238.28	770.92	1593.09
	0.03	0.02	0.01	4.54	1.89	5.68	59.14	75.89	138.47	582.37	719.38	1264.28
Graph 2	0.02	0.02	0.01	4.57	4.99	5.68	30.23	46.65	109.67	135.46	321.79	1113.33
	0.05	0.05	0.03	2.61	3.02	5.64	75.46	62.40	116.92	266.89	505.36	1745.46
	0.02	0.06	0.01	2.23	5.00	4.09	21.91	61.63	124.51	504.16	483.24	1591.86
	0.03	0.02	0.02	0.75	2.69	4.89	84.01	75.72	152.74	326.87	816.60	1635.40
	0.02	0.02	0.01	3.08	3.44	4.93	51.16	75.22	154.44	608.76	728.81	1573.00
	0.06	0.05	0.08	4.47	4.18	5.74	14.77	61.66	101.84	530.51	597.12	1571.02
	0.02	0.05	0.04	3.43	4.25	5.04	67.12	91.65	124.64	523.05	710.66	1532.57
	0.02	0.02	0.01	4.53	5.76	5.55	74.83	93.05	154.73	296.41	456.03	1307.50
	0.03	0.04	0.02	2.32	3.54	6.12	53.09	69.41	142.91	361.97	526.42	1447.20
	0.02	0.03	0.01	4.46	4.17	6.18	29.18	106.20	162.37	660.94	754.15	1743.11
Graph 3	0.08	0.06	0.07	2.94	3.36	4.76	83.17	107.04	131.86	471.61	662.78	1465.63
	0.03	0.07	0.04	2.99	3.44	4.12	59.36	77.20	125.01	577.70	700.08	1603.17
	0.05	0.04	0.05	4.50	4.96	7.24	67.50	84.50	102.24	490.58	741.04	1517.22
	0.05	0.05	0.01	5.27	5.78	6.41	7.31	98.40	131.91	658.91	690.90	1538.55
	0.04	0.04	0.01	3.70	5.66	4.80	44.98	92.84	154.30	391.81	512.73	1064.24
	0.05	0.07	0.01	3.81	3.51	7.98	52.37	114.64	124.59	425.67	619.57	1534.67
	0.02	0.06	0.04	2.38	3.67	5.14	22.01	69.18	117.25	589.66	566.86	1250.21
	0.06	0.04	0.03	2.91	4.11	4.83	84.72	116.71	138.74	587.84	659.80	1451.98
	0.07	0.06	0.01	4.06	4.51	4.77	69.02	67.43	112.47	514.20	664.12	1317.81
	0.04	0.04	0.03	3.95	5.66	7.20	66.86	83.20	144.87	672.35	789.29	1493.19
Average	0.04	0.04	0.03	3.47	4.09	5.80	57.43	80.66	130.88	437.74	605.44	1426.86

Table B.2.2: HC2 Matching Results

	n=5			10			15			20		
	k=2	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**
Graph 1	33.91	32.33	32.33	77.53	77.53	76.15	125.06	125.06	121.30	166.99	166.99	168.71
	37.98	33.91	32.33	76.26	78.46	76.26	126.25	124.60	122.11	147.50	147.87	158.86
	35.62	36.82	36.82	74.98	73.30	77.63	126.51	123.26	112.07	160.98	167.22	175.63
	37.98	37.98	37.98	73.33	75.72	79.46	125.06	123.32	114.88	171.14	161.96	172.07
	33.91	37.98	37.98	79.43	77.11	77.04	129.33	125.52	119.19	173.86	174.69	164.62
	37.98	36.82	30.62	76.08	76.25	79.46	129.48	129.03	129.48	153.88	173.63	164.10
	36.82	35.02	36.82	75.10	75.10	78.96	123.70	123.70	124.55	175.89	169.86	174.45
	37.98	35.58	35.62	71.03	73.72	76.60	128.81	124.37	122.24	158.26	158.26	165.97
	37.98	36.82	27.77	71.67	77.22	78.94	125.16	127.41	123.76	162.71	173.89	164.10
	37.98	36.82	35.62	79.43	50.49	79.43	128.36	120.91	126.80	172.26	172.87	166.37
Graph 2	36.87	36.02	36.92	75.31	76.82	74.23	116.35	107.89	114.26	134.04	139.83	142.14
	36.87	36.87	36.87	74.66	74.66	75.27	129.17	121.67	121.67	164.49	165.89	172.15
	31.02	36.87	34.35	73.86	76.38	70.62	116.68	124.56	124.70	169.86	166.02	173.07
	36.02	36.87	36.87	72.72	73.76	77.18	129.01	121.51	127.36	171.82	171.82	165.37
	36.57	36.87	31.71	74.37	73.76	75.17	123.89	125.81	128.69	173.34	173.34	173.34
	36.92	36.87	36.87	76.48	73.20	76.28	117.51	125.38	121.52	174.29	170.06	175.53
	33.57	36.87	36.02	75.12	74.74	74.24	130.08	127.48	123.10	161.43	169.12	171.71
	26.05	29.56	31.10	75.31	76.31	74.28	130.20	129.85	130.20	169.21	171.27	170.42
	36.86	36.92	32.91	70.99	70.09	75.69	129.88	129.88	128.68	167.20	163.02	166.64
	36.87	36.86	36.92	74.48	71.31	74.20	114.67	129.48	129.48	168.87	167.30	172.48
Graph 3	38.35	38.35	38.35	71.44	69.17	72.99	126.63	130.42	117.36	166.02	169.10	169.10
	31.55	38.35	38.35	73.20	73.20	75.67	126.41	126.41	123.45	170.73	171.61	174.21
	38.35	37.35	38.35	75.69	77.02	78.79	123.59	123.59	113.91	170.07	172.28	162.92
	38.35	38.01	29.84	78.63	78.63	77.69	98.70	128.60	126.88	172.61	171.13	171.13
	36.17	31.53	31.53	74.34	77.56	75.04	113.32	124.67	128.60	167.40	167.40	164.66
	38.35	38.35	26.22	76.77	75.11	78.79	102.10	119.79	119.27	173.17	169.77	169.77
	37.35	37.35	36.17	76.13	75.69	75.47	121.21	125.34	118.21	170.17	170.34	148.81
	31.55	38.35	38.35	74.23	74.15	74.15	127.10	125.73	125.08	172.48	169.63	171.47
	38.35	38.35	36.17	79.45	79.45	79.45	127.42	121.74	117.34	173.01	171.40	174.19
	37.35	38.35	33.11	78.75	77.02	78.75	130.42	122.60	125.28	174.31	171.67	168.58

Table B.3.1: SA Runtime

Time(s)	n=5									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	0.08	0.13	0.16	0.20	0.25	0.33	0.38	0.34	0.38	0.42
	0.09	0.14	0.17	0.21	0.23	0.28	0.29	0.34	0.36	0.44
	0.10	0.16	0.21	0.21	0.24	0.42	0.35	0.46	0.41	0.50
	0.10	0.13	0.17	0.20	0.23	0.28	0.36	0.36	0.37	0.40
	0.10	0.16	0.18	0.22	0.26	0.25	0.31	0.33	0.36	0.41
	0.09	0.13	0.17	0.20	0.22	0.29	0.31	0.33	0.38	0.40
	0.16	0.20	0.16	0.24	0.24	0.29	0.30	0.33	0.36	0.40
	0.11	0.18	0.19	0.26	0.22	0.28	0.33	0.32	0.42	0.48
	0.11	0.17	0.23	0.23	0.25	0.31	0.29	0.41	0.45	0.47
	0.11	0.12	0.18	0.21	0.29	0.28	0.37	0.43	0.45	0.46
Graph 2	0.11	0.24	0.27	0.23	0.22	0.28	0.35	0.42	0.49	0.45
	0.10	0.24	0.19	0.25	0.31	0.30	0.29	0.32	0.36	0.47
	0.09	0.15	0.24	0.29	0.36	0.54	0.38	0.35	0.39	0.42
	0.10	0.15	0.17	0.22	0.24	0.25	0.32	0.33	0.37	0.40
	0.09	0.13	0.18	0.19	0.26	0.33	0.31	0.35	0.44	0.48
	0.19	0.29	0.20	0.29	0.27	0.28	0.31	0.36	0.40	0.44
	0.14	0.18	0.21	0.23	0.26	0.31	0.31	0.36	0.43	0.52
	0.14	0.17	0.20	0.25	0.24	0.40	0.31	0.43	0.46	0.48
	0.11	0.17	0.24	0.24	0.30	0.31	0.35	0.39	0.41	0.45
	0.12	0.21	0.17	0.27	0.26	0.31	0.32	0.33	0.42	0.51
Graph 3	0.12	0.19	0.25	0.31	0.29	0.30	0.31	0.33	0.47	0.47
	0.11	0.15	0.16	0.22	0.27	0.29	0.36	0.43	0.52	0.45
	0.10	0.13	0.17	0.20	0.28	0.31	0.45	0.50	0.53	0.48
	0.10	0.13	0.16	0.22	0.31	0.35	0.34	0.34	0.41	0.45
	0.10	0.15	0.17	0.24	0.31	0.28	0.32	0.39	0.44	0.42
	0.11	0.16	0.18	0.25	0.25	0.31	0.32	0.38	0.41	0.44
	0.10	0.16	0.20	0.29	0.32	0.29	0.37	0.63	0.54	0.45
	0.09	0.16	0.22	0.23	0.30	0.38	0.36	0.43	0.41	0.51
	0.11	0.20	0.21	0.27	0.27	0.31	0.30	0.41	0.37	0.42
	0.11	0.19	0.17	0.25	0.27	0.29	0.33	0.39	0.45	0.52
Average	0.11	0.17	0.19	0.24	0.27	0.31	0.33	0.38	0.42	0.45

Table B.3.1: SA Runtime[Continued]

Time(s)	n=10									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	0.80	0.33	0.52	0.53	0.98	0.37	0.62	0.22	0.17	0.40
	0.99	0.05	0.74	0.05	0.90	0.46	0.79	0.15	0.20	0.48
	0.06	0.40	0.48	0.97	0.91	0.34	0.92	0.95	0.13	0.48
	0.93	0.36	0.53	0.08	0.97	0.32	0.43	0.83	0.53	0.63
	0.96	0.39	0.64	0.03	0.06	0.44	0.57	0.22	0.07	0.72
	0.94	0.27	0.66	0.74	0.03	0.33	0.79	0.15	0.37	0.52
	0.85	0.23	0.43	0.82	0.98	0.57	0.73	0.15	0.15	0.47
	0.84	0.15	0.58	0.73	0.08	0.24	0.32	0.89	0.05	0.43
	0.84	0.05	0.41	0.84	0.14	0.38	0.88	0.43	0.13	0.78
	0.87	0.43	0.46	0.87	0.33	0.85	0.81	0.16	0.46	0.97
Graph 2	0.92	0.08	0.37	0.23	0.04	0.50	0.90	0.90	0.71	0.74
	0.86	0.17	0.92	0.01	0.92	0.17	0.58	0.80	0.48	0.99
	0.80	0.04	0.39	0.68	0.26	0.27	0.07	0.68	0.73	0.72
	0.89	0.08	0.31	0.61	0.02	0.68	0.08	0.43	0.26	0.60
	0.77	0.07	0.43	0.69	0.07	0.23	0.43	0.69	0.54	0.46
	0.84	0.43	0.96	0.76	0.55	0.54	0.78	0.18	0.55	0.72
	0.77	0.46	0.46	0.85	0.13	0.52	0.89	0.12	0.55	0.58
	0.07	0.49	0.64	0.88	0.43	0.16	0.87	0.28	0.17	0.45
	0.71	0.98	0.31	0.67	0.52	0.43	0.57	0.86	0.14	0.62
	0.69	0.03	0.31	0.61	0.61	0.11	0.89	0.91	0.27	0.24
Graph 3	0.82	0.02	0.42	0.01	0.32	0.55	0.87	0.71	0.31	0.52
	0.79	0.03	0.28	0.68	0.75	0.48	0.58	0.15	0.54	0.63
	0.73	0.02	0.35	0.63	0.54	0.24	0.50	0.05	0.10	0.94
	0.73	0.94	0.48	0.71	0.26	0.62	0.83	0.90	0.22	0.57
	0.73	0.05	0.43	0.52	0.88	0.45	0.01	0.85	0.34	0.43
	0.79	0.97	0.43	0.74	0.17	0.59	0.89	0.69	0.80	0.47
	0.28	0.58	0.32	0.55	0.07	0.33	0.80	0.36	0.58	0.50
	0.84	0.38	0.35	0.72	0.92	0.87	0.72	0.34	0.14	0.82
	0.78	0.31	0.26	0.60	0.79	0.37	0.72	0.43	0.66	0.23
	0.86	0.26	0.54	0.02	0.30	0.54	0.62	0.67	0.94	0.18
Average	0.76	0.30	0.48	0.56	0.47	0.43	0.65	0.51	0.38	0.58

Table B.3.1: SA Runtime[Continued]

Time(s)	n=15									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	0.62	0.91	0.89	0.82	0.95	0.43	1.53	1.09	3.35	9.16
	0.21	0.56	0.71	0.06	0.73	0.88	0.84	2.17	3.86	5.51
	0.63	0.93	0.57	0.63	0.75	2.50	0.94	1.46	3.36	5.34
	0.20	0.67	0.29	0.20	0.44	0.66	0.43	2.51	3.61	7.17
	0.12	0.77	0.97	0.79	0.36	2.02	2.91	4.08	6.07	4.70
	0.24	0.17	0.57	0.91	0.36	0.89	1.63	3.56	3.29	3.81
	0.64	0.76	0.23	0.40	0.37	0.06	0.84	0.87	2.65	2.97
	0.08	0.35	0.41	0.78	0.98	0.51	0.03	1.14	2.29	4.41
	0.18	0.54	0.82	0.82	1.00	0.12	0.66	0.52	2.33	3.85
	0.58	0.46	0.15	0.19	0.24	0.68	0.01	1.92	2.17	3.15
Graph 2	0.30	0.24	0.55	0.42	0.10	0.45	0.44	1.51	2.32	3.40
	0.51	0.82	0.56	0.27	0.88	0.43	0.33	1.36	3.40	4.20
	0.86	0.56	0.80	0.80	0.99	0.52	0.43	0.97	2.32	3.42
	0.59	0.83	0.82	0.09	0.73	0.99	0.95	1.17	1.75	2.58
	0.49	0.47	0.68	0.84	0.88	0.13	0.24	1.52	1.75	3.99
	0.18	0.67	0.48	0.06	0.81	0.75	0.13	1.68	2.49	4.06
	0.37	0.86	0.11	0.56	0.02	0.11	0.20	1.45	1.89	3.55
	0.73	0.73	0.66	0.71	0.75	0.41	0.28	1.42	2.46	2.89
	0.30	0.06	0.44	0.73	0.16	0.35	0.66	1.65	2.48	3.52
	0.18	0.34	0.67	0.52	0.82	0.92	0.71	1.49	2.77	3.64
Graph 3	0.99	0.80	0.17	0.75	0.25	0.44	0.10	1.97	2.12	3.22
	0.82	0.97	0.23	0.56	0.32	0.08	0.93	1.08	3.26	3.95
	0.04	0.26	0.28	0.74	0.39	0.65	0.04	0.90	2.29	3.60
	0.11	0.05	0.89	0.93	0.69	0.40	0.82	1.01	2.18	3.37
	0.89	0.43	0.38	0.66	0.58	0.60	0.31	0.62	1.85	3.61
	0.78	0.95	0.36	0.62	0.74	0.29	0.14	0.96	1.44	2.86
	0.16	0.09	0.61	0.58	0.29	0.93	0.36	1.45	2.16	3.44
	0.81	0.52	0.63	0.44	0.49	0.58	0.27	1.51	2.00	3.98
	0.81	1.00	0.86	0.40	0.91	0.92	0.03	1.05	2.10	3.68
	0.73	0.90	0.56	0.36	0.38	0.16	0.98	1.45	1.78	3.48
Average	0.47	0.59	0.54	0.56	0.58	0.63	0.61	1.52	2.59	4.02

Table B.3.1: SA Runtime[Continued]

Time(s)	n=20									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	0.71	2.90	5.77	9.46	2.12	4.44	7.85	4.31	8.05	6.43
	0.08	2.20	5.51	7.85	9.26	5.27	7.91	2.10	5.64	5.48
	0.94	3.42	5.53	8.71	3.59	4.33	7.55	0.27	0.98	0.51
	0.15	1.44	5.55	9.77	2.68	7.14	6.41	4.61	7.01	4.28
	0.25	1.04	5.08	7.45	0.97	1.96	1.69	4.20	7.15	9.44
	0.34	1.84	5.07	8.47	2.90	6.16	8.01	0.68	3.50	9.70
	0.45	0.66	3.80	9.71	5.48	5.87	9.96	7.16	6.68	0.96
	0.60	3.24	5.04	8.30	0.11	9.13	7.83	8.06	6.63	7.67
	0.52	2.39	5.10	9.62	2.65	5.94	9.72	0.18	5.23	7.73
	1.00	2.94	4.99	7.94	1.37	6.28	9.63	2.00	4.91	9.81
Graph 2	0.43	3.74	6.57	9.64	3.54	5.86	1.16	4.10	6.61	1.72
	0.27	2.31	5.16	8.76	2.01	5.36	8.41	9.44	6.48	7.02
	0.79	2.14	7.53	7.64	2.24	6.39	7.73	4.51	6.47	5.96
	0.12	3.55	6.14	2.83	1.59	5.31	7.78	0.50	5.04	3.67
	0.05	4.49	8.92	2.01	2.14	3.81	1.25	0.45	7.03	8.97
	0.43	4.98	8.44	1.70	0.49	5.16	8.54	7.35	5.28	6.19
	0.59	3.45	7.22	1.27	0.07	4.17	7.39	2.23	3.65	9.60
	0.18	4.14	8.01	2.87	0.73	3.46	8.85	1.06	6.14	7.38
	0.07	3.36	7.70	2.88	2.97	6.70	9.16	2.65	5.84	6.27
	0.36	4.25	8.06	0.56	3.98	7.54	7.67	2.33	5.25	9.77
Graph 3	0.53	4.61	8.25	2.48	1.14	5.39	8.19	0.53	3.48	7.55
	0.96	3.93	7.85	1.99	1.03	5.42	8.87	1.38	2.13	8.89
	0.56	3.08	7.09	1.34	2.01	5.57	7.56	5.05	4.39	1.89
	0.97	5.29	0.06	2.71	9.76	1.78	5.40	5.06	5.29	5.50
	0.64	9.79	0.49	9.71	0.97	4.81	0.22	1.71	5.61	9.13
	5.20	6.13	0.07	3.22	7.90	6.81	3.48	0.88	3.17	9.15
	0.67	4.65	0.17	0.67	6.17	0.15	0.60	5.83	1.68	7.36
	1.30	7.95	7.69	4.46	5.79	3.44	3.60	1.12	3.15	1.70
	0.74	8.14	1.51	6.80	5.77	9.37	3.10	3.47	0.95	9.98
	1.11	6.97	9.62	3.38	5.36	3.84	5.56	4.20	1.80	5.55
Average	0.70	3.97	5.60	5.47	3.23	5.23	6.37	3.25	4.84	6.51

Table B.3.2: SA Matching Results

	n=5									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	28.68	28.68	28.68	28.68	23.95	23.95	23.95	23.95	23.95	23.95
	30.72	30.72	30.72	30.72	26.29	26.29	26.29	26.29	26.29	26.29
	27.08	27.08	27.08	27.08	30.33	30.33	30.33	30.33	30.33	30.33
	26.55	31.48	31.48	31.48	26.29	26.29	26.04	26.29	26.29	26.29
	28.68	28.66	28.68	28.66	28.04	28.04	28.04	28.04	28.04	28.04
	29.24	29.24	29.24	29.24	26.04	26.04	25.99	26.04	26.04	26.04
	33.75	33.75	33.75	33.75	26.25	26.25	26.25	26.25	26.25	26.25
	26.07	26.07	24.87	24.87	26.97	26.97	26.97	26.97	26.97	26.97
	32.42	32.42	32.42	30.72	26.25	26.25	26.25	26.25	26.25	26.25
	28.66	29.15	28.66	29.15	23.56	23.56	23.56	23.56	23.56	23.56
Graph 2	31.75	31.87	31.87	31.75	27.67	27.67	27.67	27.67	27.67	27.67
	39.60	39.60	39.60	39.60	29.82	29.56	29.82	29.82	29.82	29.82
	34.90	34.90	34.90	34.90	32.52	31.85	31.85	32.52	32.52	32.52
	34.90	34.90	34.90	34.90	29.87	29.87	29.87	29.87	29.87	29.87
	30.35	30.35	30.35	30.35	36.14	36.14	36.14	36.14	36.14	36.14
	39.60	39.60	39.60	39.60	30.88	30.88	30.88	30.88	30.88	30.88
	34.90	34.93	34.93	34.90	36.14	36.14	36.14	36.14	36.14	36.14
	39.80	39.80	38.29	39.80	32.52	32.52	32.52	32.52	29.09	29.09
	27.96	27.96	27.96	27.96	27.34	27.07	27.07	27.34	27.07	27.07
	34.25	34.25	34.25	34.25	36.14	36.14	36.14	36.14	36.14	36.14
Graph 3	40.21	40.21	40.21	40.21	26.13	30.79	30.79	30.79	30.79	30.79
	34.99	35.05	35.93	35.93	25.12	25.12	25.12	25.12	25.12	25.12
	35.55	35.55	35.55	35.55	25.12	25.12	25.12	25.12	25.12	25.12
	33.30	33.30	33.30	33.30	33.58	33.58	33.58	33.58	33.58	33.58
	35.93	35.93	35.93	35.93	33.58	33.58	33.58	33.58	33.58	33.58
	36.41	36.41	36.41	36.41	32.05	32.05	32.05	32.05	32.05	32.05
	34.85	34.85	36.41	36.41	33.58	33.58	33.58	33.58	33.58	33.58
	34.85	34.65	34.85	34.85	34.82	34.82	34.82	34.82	34.82	34.82
	35.41	35.41	35.41	34.45	30.46	30.46	30.46	30.46	30.46	30.46
	34.85	34.85	34.85	34.85	34.82	34.82	34.82	34.82	33.68	33.68

Table B.3.2: SA Matching Results [Continued]

	n=10									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	56.10	55.86	55.86	55.86	56.96	56.72	56.96	56.72	56.96	56.96
	72.83	68.79	69.77	69.86	68.67	70.32	70.32	70.32	68.67	68.67
	52.72	50.67	51.81	52.72	67.71	65.11	65.92	68.53	65.92	65.92
	61.93	60.98	59.00	61.93	52.14	52.26	56.75	56.75	56.75	56.75
	55.13	52.38	50.07	55.13	59.10	58.01	57.54	57.54	59.10	59.10
	60.14	60.14	58.22	60.26	66.20	59.16	66.20	59.16	66.20	66.20
	59.76	59.98	59.98	59.98	53.28	53.28	50.67	53.28	53.28	53.28
	51.07	55.20	55.20	55.20	74.79	66.35	66.35	74.79	74.79	74.79
	62.25	59.05	58.02	58.02	70.37	70.37	69.08	68.88	68.87	68.87
	58.90	60.03	60.03	57.09	56.13	58.52	56.13	58.33	58.33	58.33
Graph 2	53.09	53.09	47.94	50.95	69.67	70.87	68.96	70.87	69.67	69.67
	52.66	51.17	52.44	54.27	57.98	57.98	57.98	57.98	54.81	54.81
	71.83	71.83	71.83	64.05	46.79	42.72	47.35	47.35	47.35	47.35
	60.82	62.60	62.20	64.51	65.73	65.53	65.73	65.53	65.53	65.53
	57.14	58.99	66.21	63.39	65.13	64.11	65.13	65.13	62.82	62.82
	51.90	55.76	55.73	57.70	60.33	60.86	60.86	61.79	61.42	61.42
	50.70	53.32	54.15	50.70	57.97	58.53	58.54	60.30	60.30	60.30
	56.44	59.88	55.14	61.13	66.01	61.50	67.37	67.37	67.37	67.37
	65.47	65.46	65.46	65.47	59.55	59.55	62.84	59.55	62.84	62.84
	53.08	53.08	55.27	56.05	66.13	62.59	62.59	66.13	66.13	66.13
Graph 3	47.07	48.21	48.19	48.21	59.92	54.85	59.92	59.92	59.92	59.92
	56.75	60.13	60.13	60.13	60.31	60.85	60.31	60.31	60.85	60.85
	43.31	48.05	48.05	46.29	69.25	69.75	68.19	69.75	69.75	69.75
	60.44	58.40	60.44	60.44	67.75	65.65	67.75	63.30	67.75	67.75
	50.09	50.09	49.92	48.98	62.67	62.67	62.67	62.67	62.67	62.67
	54.31	52.46	52.97	52.97	50.20	47.61	49.10	50.20	49.10	49.10
	53.77	55.27	58.08	57.25	62.74	62.74	63.89	63.89	63.89	63.89
	55.79	57.00	60.51	55.85	51.54	50.10	51.54	51.54	51.54	51.54
	72.67	73.28	73.28	73.28	51.32	53.84	53.84	53.84	53.84	53.84
	57.62	52.79	57.62	57.62	47.48	51.40	51.84	51.84	51.40	51.40

Table B.3.2: SA Matching Results [Continued]

	n=15									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	75.63	79.93	74.90	69.24	84.82	79.70	82.85	88.38	84.13	84.13
	93.96	86.22	86.53	89.23	109.67	108.83	111.30	111.30	111.30	111.30
	77.58	80.96	74.29	82.98	69.13	72.94	72.94	72.94	70.21	70.21
	84.10	80.43	82.19	85.37	89.92	90.82	90.82	90.78	90.78	90.78
	67.42	70.86	74.01	69.06	96.32	92.02	92.20	96.32	93.91	93.91
	96.78	97.81	97.45	96.75	99.81	99.81	98.92	98.92	99.81	99.81
	72.59	71.03	78.49	78.49	73.20	73.20	72.37	71.70	70.54	70.54
	67.15	69.43	67.40	73.01	72.01	67.39	72.01	66.98	69.94	69.94
	73.72	72.24	73.72	78.67	96.54	97.95	92.79	96.54	97.95	97.95
	54.14	54.14	54.14	57.59	76.97	74.40	76.97	74.64	76.97	76.97
Graph 2	98.91	92.09	98.91	92.99	95.37	91.31	92.28	94.99	95.37	95.37
	75.05	70.13	72.33	75.05	103.20	98.75	101.87	102.09	99.95	99.95
	78.26	79.02	79.02	70.60	96.02	96.03	96.02	96.03	96.03	96.03
	90.85	85.90	84.31	87.86	84.18	85.35	84.45	84.30	86.22	86.22
	91.55	80.49	86.92	91.55	83.05	90.57	89.28	89.28	87.29	87.29
	95.64	95.64	95.64	88.56	71.12	74.50	71.46	71.26	74.90	74.90
	88.26	84.84	87.24	86.41	104.36	101.60	99.56	104.36	104.36	104.36
	100.40	93.70	99.98	100.40	62.91	64.51	64.51	64.51	62.91	62.91
	77.40	76.54	71.32	80.14	96.44	96.04	96.44	96.04	96.04	96.04
	95.15	89.20	94.94	94.94	87.96	88.55	89.60	88.65	87.96	87.96
Graph 3	100.51	105.67	105.67	105.67	85.70	91.19	91.19	87.37	91.19	91.19
	80.14	81.32	86.37	80.22	93.97	93.97	95.46	95.46	93.71	93.71
	95.02	98.74	93.56	90.56	77.84	76.38	76.38	78.08	75.36	75.36
	96.16	95.57	94.59	103.04	80.67	80.67	79.68	80.67	78.91	78.91
	93.75	93.05	94.61	100.06	82.19	86.33	85.34	82.68	86.33	86.33
	93.46	88.31	87.77	93.46	101.95	94.73	101.95	101.95	101.95	101.95
	101.94	92.46	101.94	97.54	79.54	83.07	82.70	80.50	82.70	82.70
	84.22	87.15	88.04	88.04	72.60	76.31	72.60	72.60	72.60	72.60
	86.25	90.00	83.34	87.79	82.54	82.54	84.59	84.85	84.85	84.85
	94.47	91.35	97.09	96.94	75.09	75.09	70.30	74.81	73.84	73.84

Table B.3.2: SA Matching Results [Continued]

	n=20									
	k=25	**35**	**45**	**55**	**65**	**75**	**85**	**95**	**105**	**115**
Graph1	99.60	95.27	97.80	94.71	113.59	113.59	113.59	111.38	113.59	113.59
	108.87	108.79	108.85	110.70	113.05	113.05	113.05	111.08	112.04	112.04
	101.88	96.92	101.88	101.93	98.10	98.55	103.01	98.55	103.01	103.01
	122.92	119.85	121.50	126.81	126.51	122.09	122.09	124.59	122.45	122.45
	88.41	88.97	87.86	91.22	111.87	113.33	116.31	116.82	114.67	114.67
	120.75	124.27	124.27	122.63	116.12	111.48	118.46	116.12	118.46	118.46
	125.35	131.40	127.66	128.83	84.60	85.46	87.36	83.99	87.41	87.41
	89.95	89.03	94.65	89.10	136.11	139.25	136.86	136.86	137.43	137.43
	105.44	106.61	107.81	105.44	112.05	117.53	111.79	114.27	114.43	114.43
	86.61	87.48	87.69	86.49	106.93	106.93	107.67	106.93	107.67	107.67
Graph 2	116.06	112.41	112.04	116.06	101.24	105.26	103.67	107.18	103.66	103.66
	102.75	103.40	102.15	106.29	116.95	116.95	115.56	118.44	115.61	115.61
	93.91	97.29	100.43	97.43	98.74	103.19	103.19	102.23	102.23	102.23
	126.47	120.32	118.56	126.47	123.99	121.80	119.34	123.99	125.22	125.22
	98.42	96.66	92.72	96.66	116.83	115.97	116.40	118.59	121.60	121.60
	91.53	92.99	91.76	85.45	130.25	131.78	130.25	131.78	131.78	131.78
	98.93	97.36	101.43	100.06	132.03	134.13	134.13	129.45	129.45	129.45
	111.30	109.93	115.73	107.97	109.13	114.70	114.70	111.79	112.17	112.17
	113.96	112.32	117.67	117.67	84.21	85.90	79.68	88.34	82.21	82.21
	100.06	102.22	104.55	104.55	110.26	114.65	114.26	111.94	116.51	116.51
Graph 3	131.25	128.73	131.54	132.11	137.24	140.10	137.12	139.04	139.04	139.04
	102.50	102.50	95.62	99.25	119.67	125.09	117.53	118.02	119.03	119.03
	136.21	141.61	133.38	141.61	123.45	123.45	127.96	127.96	124.47	124.47
	92.74	84.77	95.11	88.27	102.55	102.91	105.75	107.19	101.18	101.18
	100.47	102.25	94.72	102.25	91.90	88.02	89.76	88.02	91.90	91.90
	106.08	105.06	112.11	108.76	125.35	123.33	125.16	125.35	125.35	125.35
	107.50	118.76	116.15	119.93	122.48	118.27	120.99	120.97	120.97	120.97
	135.61	131.93	130.67	132.98	107.46	108.94	108.94	108.94	108.94	108.94
	94.18	91.40	94.18	90.37	112.10	112.10	112.10	113.20	113.20	113.20
	138.74	133.85	132.29	135.51	119.65	122.02	124.34	124.15	124.34	124.34

Table B.4.1: Distribution Analysis Results

	SA			HC		
	random	**normal**	**uniform**	**random**	**normal**	**uniform**
Graph 1	80.21	155.48	93.374	130.88	170.14	125.15
	99.189	156.21	91.228	132.45	169.97	126.489
	74.886	157.72	94.856	130.542	170.8	128.524
	80.797	157.08	92.578	130.878	169.6	124.805
	93.71	153.98	83.89	128.175	171.51	126.84
	89.576	153.95	79.481	130.323	168.06	126.888
	104.54	154.09	79.981	131.744	167.82	126.444
	86.256	156.33	102.02	131.762	170.74	128.116
	88.705	150.96	93.128	128.704	170.74	125.377
	89.166	157.83	85.784	132.32	170.05	126.98
Graph 2	81.489	158.6	91.962	131.17	168.63	132.60
	74.267	156.26	83.247	132.138	167.99	126.608
	92.195	156.7	105.57	130.577	168.99	131.888
	85.753	154.07	77.764	128.994	168.22	130.052
	106.5	152.52	93.442	128.341	168.99	128.216
	85.01	154.24	97.289	132.455	167.34	131.166
	83.771	154.05	97.645	132.117	167.88	131.613
	95.938	153.12	87.537	130.774	168.02	130.248
	97.429	150.86	92.66	130.498	168.8	130.238
	90.134	153.14	86.575	128.34	168.07	131.467
Graph 3	90.788	153.25	95.714	134.05	167.14	128.50
	96.617	149.32	107.99	132.915	166.27	127.595
	81.662	151.64	88.989	136.178	165.31	128.329
	108.86	157.26	95.565	132.208	167.14	128.557
	63.637	150.13	89.095	133.847	166.4	130.064
	79.83	148.89	89.1	134.573	166.3	128.791
	92.397	148.64	87.823	133.896	165.98	128.782
	100.75	157.01	94.134	131.019	165.46	129.417
	90.209	155.5	93.199	134.457	166.18	127.516
	94.924	151.56	97.717	133.72	167.03	124.412

Table B.4.2: Bottleneck Matching Results

	random	**normal**	**uniform**
Graph 1	66.76	154.65	62.09
Graph 2	52.55	151.28	77.44
Graph 3	83.45	144.59	62.78