Bipartite Matching: What to do in the Real World When Computing Assignment Costs Dominates Finding the Optimal Assignment

Tenindra Abeywickrama
Grab Holdings Inc.
tenindra.a@grab.com

Victor Liang
Grab Holdings Inc.
victor.liang@grab.com

Kian-Lee Tan
School of Computing
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

The Kuhn-Munkres (KM) algorithm is a classical combinatorial optimization algorithm that is widely used for minimum cost bipartite matching in many real-world applications, such as transportation. For example, a ride-hailing service may use it to find the optimal assignment of drivers to passengers to minimize the overall wait time. Typically, given two bipartite sets, this process involves computing the edge costs between all bipartite pairs and finding an optimal matching. However, existing works overlook the impact of edge cost computation on the overall running time. In reality, edge computation often significantly outweighs the computation of the optimal assignment itself, as in the case of assigning drivers to passengers which involves computation of expensive graph shortest paths. Following on from this, we also observe common real-world settings exhibit a useful property that allows us to incrementally compute edge costs only as required using an inexpensive lower-bound heuristic. This technique significantly reduces the overall cost of assignment compared to the original KM algorithm, as we demonstrate experimentally on multiple real-world data sets and workloads. Moreover, our algorithm is not limited to this domain and is potentially applicable in other settings where lower-bounding heuristics are available.

1. INTRODUCTION

The Kuhn-Munkres (KM) algorithm [13, 15], also known as the Hungarian Method, is a combinatorial optimization algorithm widely utilized to solve many real-world problems, particularly in transportation. The KM algorithm solves the *assignment problem*, also known as the *minimum-weight bipartite matching* problem, which involves finding an optimal pair-wise assignment of a set of *agents* to a set of *jobs*. Assigning an agent to a job is associated with some cost, thus the goal is to find an optimal assignment or *matching* of agent-job pairs, such that the overall cost is minimized (or maximized depending on the problem and desired outcome).

Assignment tasks are of particular importance in transportation problems, and the KM algorithm is widely used as a subroutine in many existing works [11, 8, 23, 21]. For example, it is used in ride-hailing services to optimally match drivers to passengers for maximum utilization of available vehicles. Other examples include computing mail delivery routes using Route Inspection, where minimum-weight bipartite matching is a subroutine or the order picking problem solved by using an approximate Traveling Salesman algorithm utilizing bipartite matching. The KM algorithm takes the assignment costs as input, hence these costs must be computed for each assignment task. However, we find that existing works overlook the significance of this step. Moreover, all of the aforementioned examples involve computing assignment costs based on computationally expensive graph shortest paths. For example, the cost to assign a car to a passenger is the wait time, which is commonly modeled by the travel time of the shortest path in a road network graph. As we discuss next, cost computation has significant implications for algorithm efficiency with increasingly expensive assignment cost metrics.

1.1 Motivation

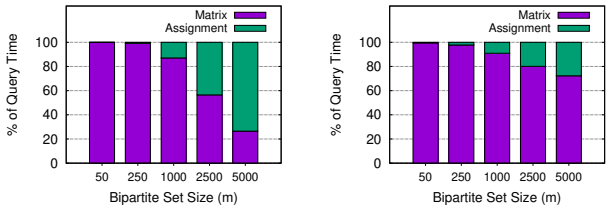
Let U be a set of agents and V be a set of jobs, both with size $m = |U| = |V|$, for which an optimal assignment is required. Also, let $c(u, v)$ be the cost of assigning agent $u \in U$ to job $v \in V$. Costs $c(u, v) \forall u \in U, v \in V$ are often conceptualized as an $m \times m$ matrix. To the best of our knowledge, all previous work utilizing KM to solve transport problems like ride-hailing assumes this matrix is provided to the KM algorithm or the cost of computing the matrix is not a bottleneck. However, in many real-world applications, computing the matrix is not only a non-trivial cost but also more computationally expensive than the assignment itself. Moreover, the matrix may need to be re-computed each time an assignment is required. Given the real-time nature of transportation problems, this may be quite frequent, which serves to only exacerbate the non-trivial cost of computing $c(u, v)$. For example, in a ride-hailing service, a new assignment is required as new cars become available and new passenger requests are received continuously in real-time. According to Fortune, popular ride-hailing services like Grab are reported to process 6 million ride requests a day, highlighting the scale of throughput required.

Our observation can be demonstrated using a simple ride-hailing assignment framework. Let us represent the cost of assigning a passenger (job) to a ride-hailing car (agent) as the travel-time of the shortest route from the car to the pas-

©Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. This is a minor revision of the paper entitled"Optimizing Bipartite Matching in Real-World Applications by Incremental Cost Computation", published in PVLDB, Vol. 14, No. 7, 2150-8097. DOI: https://doi.org/10.14778/3450980.3450983

https://fortune.com/longform/grab-gojek-super-apps/

senger. All costs for one car (agent) can be computed by performing a single Dijkstra's single-source multiple-destination (SSMD) shortest path query. The entire cost matrix can be populated by performing m such searches. Simple worst-case analysis based on Dijkstra using Fibonacci heaps suggests that this will cost $O(m|E| + m|N| \log |N|)$ time where $|N|$ and $|E|$ are the number of vertices and edges in the road network graph and $m = |U| = |V|$. Typical real-world scenarios would see this dominate the KM algorithm time complexity of $O(m^3)$. For example, in the Singapore road network $|N|$ is over $280,000$ while m might be 100 representing finding a matching for 100 ride-hailing cars to 100 passengers. We verify this intuition in practice for the Singapore road network for varying values of m in Figure 1a using Dijkstra's search as above. As expected, the time to compute the matrix dominates the time to compute the optimal assignment for increasing m, only being overtaken when m reaches 5000. In Figure 1b we show this is still true even if a fast modern point-to-point shortest path technique like Contraction Hierarchies is used.



(a) Dijkstra (b) Contraction Hierarchies

Figure 1: Proportion of running time spent on matrix computation and finding optimal assignment on Singapore road network for varying m

1.2 Contributions

We have seen that computing the cost matrix often dominates computing the optimal assignment itself. Moreover, the cost matrix must be computed from scratch for each assignment problem and may need to be performed frequently in real-world applications such as ride-hailing. In attempting to address the scalability and throughput concerns that arise as a result, it begs the question of whether all assignment costs are even necessary to compute an optimal solution as first observed by [14]. We observe that this is also not necessarily the case due to a property exhibited by optimal assignments in typical real-world scenarios. For example, in a ride-hailing service for a particular geographic region, such as Singapore, typically passengers and drivers will be distributed in various parts of the region. It is unlikely that a driver $u \in U$ will be assigned to some passenger $v \in V$ a significant distance away. We say that such problems exhibit high *spatial locality of matching*. Using this intuition we propose a minimum-weight bipartite matching algorithm based on the KM algorithm that incrementally computes costs that are most likely to be in the optimal matching. We develop novel refinement rules using inexpensive lower-bounding heuristics to only compute costs when necessary. Notably, our technique still computes the optimal matching, but does so while computing far fewer expensive pair-wise exact assignment costs, significantly reducing the overall running time. Moreover, our technique is a drop-in replacement for the KM algorithm in any technique or framework that uses the KM as a subroutine. Our contributions can be summarized as follows:

- We identify that computing assignment costs such as graph shortest paths are more computationally expensive than finding the optimal assignment itself for workloads for real-world problems such as ride-hailing.
- We present a minimum-weight bipartite matching algorithm based on the Kuhn-Munkres algorithm that incrementally computes the exact assignment costs required for an assignment only when it is necessary according to novel pruning rules utilizing inexpensive lower-bounding heuristics.
- We implement a specialized lower-bounding heuristic for use in ride-hailing services, where the assignment cost is represented by the travel-time of the shortest path in a road network graph, adapting landmark-based lower-bounds and graph search techniques.
- Our extensive experimental investigation using large-scale real-world data sets and workloads demonstrates the significant improvement achieved by our proposed solutions with highly favorable implications for real-world scalability and throughput.

2. PRELIMINARIES

The assignment problem is often formulated as the minimum weight bipartite matching problem. Then, we are given a bipartite graph $B = (U \cup V, E_B)$ where U and V are the bipartite sets of vertices. E_B is the set of edges, and contains an edge $(u, v) \forall u \in U, v \in V$. The weight $c(u, v)$ of an edge represents the cost of assigning u to v. The assignment problem finds a *perfect matching*, where every object in U is assigned to exactly one object in V (and vice versa), such that the sum of weights over all assigned pairs is minimized. For simpler exposition, we consider equally sized sets, i.e., $m = |U| = |V|$, which in practice can be simulated by adding dummy vertices to the smaller set. Next, we describe the preliminaries for the applied setting for which our techniques are designed to be deployed.

Road Network: In the case of a ride-hailing service, the bipartite sets consist of the locations of passengers and drivers to be matched. The cost of assigning a passenger to a driver is commonly considered as the minimum travel-time for the driver to reach the passenger's location. These costs can be computed by first considering the road network $G = (V_G, E)$, where V_G is the set of vertices and E is the set of edges. Each edge $(x, y) \in E$ represents the road segments connecting junction vertices x and y with weight $w(x, y)$ representing the travel-time to traverse the edge. Note that other real positive metrics, such as physical length, can also be considered. In our context travel-time, and hence the waiting time for passengers, is most relevant. The *network distance* $d(s, t)$ between a source vertex s and destination

Note that the number of edges $|E|$ on road network graphs observes $|E| = O(|N|)$

While faster techniques for point-to-point shortest path search are available, Dijkstra is typically faster for SSMD search when many destinations are involved. This is because for increasing m the number of point-to-point queries increases by its square, explaining why the matrix computation cost percentage is still high for large values of m in Figure 1b for CH.

vertex t is the minimum sum of weights connecting vertices s and t, i.e., by the shortest path in G. Note that we consider passenger and driver locations that occur on vertices for simpler exposition and implementation, but our techniques can be extended for when this is not the case. In relation to the assignment problem, $c(u, v) = d(u, v)$.

Landmark Lower-Bounds (LLBs): Our proposed technique leverages the idea of computing an inexpensive lower-bound on the assignment cost $c(u, v)$ that is as accurate as possible. In the case of network distance as assignment cost, Landmark Lower-Bounds (LLBs) [10] are an effective lower-bound for shortest paths in graphs and can be computed cheaply. LLBs involve selecting k "landmark" vertices and computing network distances to each vertex in V from each landmark in an offline pre-processing step. During the online query phase, a lower-bound distance between any two vertices s and t may be computed using the distances to any landmark vertex l and the triangle inequality as in (1). A surprisingly accurate lower-bound can be computed by considering lower-bounds over all k landmarks as in (2), even for small values of k. Consequently, we utilize LLBs as the lower-bound on assignment cost $c(u, v)$.

$$LB_l(q, p) = |d(l, q) - d(l, p)| \leq d(q, p) \tag{1}$$

$$LB_{max}(q, p) = \max_{l \in L}(|d(l, q) - d(l, p)|) \tag{2}$$

Kuhn-Munkres Algorithm: We use the Kuhn-Munkres (KM) algorithm as the basis for our improved techniques. KM works by iteratively updating a set of labels l_u (resp. l_v) for bipartite set U (resp. V) that imply a *reduced cost* of each bipartite edge $(u, v) \in E_B$:

$$c_r(u, v) = c(u, v) - l_u - l_v \tag{3}$$

KM adjusts the labels to generate edges of zero reduced costs while maintaining the invariants below. If a *perfect matching* exists amongst these edges (referred to as the reduced graph), then this matching is the optimal solution to the minimum-weight bipartite matching problem [17].

INVARIANT 1. *The reduced cost of each edge must be non-negative, i.e.,* $c_r(u, v) \geq 0$

INVARIANT 2. *Each edge in* M *is "tight" in that it has reduced cost zero, i.e.,* $c_r(u, v) = 0$ *where* $(u, v) \in M$

KM uses augmenting paths [7] to find a perfect matching in the reduced graph. When one does not exist, the labels are adjusted by computing δ below, where $S \in U$ and $N(S) \in V$ are vertices visited by the search. We refer to [17, 7, 6] for details of these well-known techniques.

$$\delta := min\{c(u, v) - l_u - l_v : u \in S, v \notin N(S)\} \tag{4}$$

3. INCREMENTAL KUHN-MUNKRES

Recall the intuition of *spatial locality of matching*, that posits an optimal assignment for ride-hailing matching task is unlikely to assign drivers to passengers that are very far away. A simple approach to utilize this intuition might be to subdivide the region further and run the KM algorithm on each subregion separately. Naturally, this would reduce the size of m and hence the number of assignment costs that must be computed. However, this approach would no longer provide a globally optimal assignment. For example, at borders between regions, suboptimal assignment is likely to occur. In this section, we propose methods to utilize the intuition and avoid computation of exact costs, while still returning the globally optimal result.

3.1 Lower-Bounding Module

Our technique is underpinned by the ability to compute lower-bounds on edge cost $c(u, v)$ during the KM algorithm iterations. We propose an abstract Lower-Bound Module that provides the ability to compute two different lower-bounds, defined as follows:

DEFINITION 1. *(Individual Lower-Bound Edge Cost) Given vertices* $u \in U$ *and* $v \in V$*, an individual lower-bound edge-cost* $LB(u, v)$ *is a lower-bound on the true edge-cost* $c(u, v)$*, i.e.,* $LB(u, v) \leq c(u, v)$*.*

DEFINITION 2. *(Group Lower-Bound Edge Cost) Given vertex* $u \in U$*, let* $Q_u \subseteq V$ *represent the set of vertices for which the true edge cost is not known (initially* $Q_u = V$*). A group lower-bound edge cost* $LB(Q_u)$ *is a lower-bound for all edge-costs* $c(u, v) \forall v \in Q_u$*, i.e.,* $LB(Q_u) \leq c(u, v) \forall v \in Q_u$*.*

The group lower-bound edge cost is best implemented as a minimum priority queue. This allows iterative extraction of candidates from Q_u, while maintaining the definition. Moreover, the queue can be lazily updated such that the definition is met, similar to the on-demand heaps in [2]. That is, Q_u is not required to contain individual lower-bounds for all vertices in V. Next, we show how to modify the KM algorithm to use individual and group lower-bound edge costs to avoid computation of exact costs $c(u, v)$ where possible .

Note that the solution is agnostic to the implementation of Q_u and the type of cost $c(u, v)$, and can be applied to any problem setting. However, we specify the implementation for costs based on shortest paths in road network graphs where significant benefits can be gained. This is because computation of shortest paths in road network graphs is a computationally intensive task and is often used in real-world applications such as ride-hailing services and the route inspection problem. Figure 2 depicts the components of the system. The priority queues for each vertex $u \in U$ are exposed to the KM algorithm module, as is a module to compute the true cost $c(u, v)$ (when deemed necessary) using a fast shortest path technique such as G-tree [24].

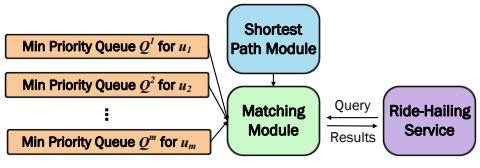


Figure 2: System Overview

3.2 Refinement Rules

We propose the Incremental Kuhn-Munkres (IKM) algorithm (Algorithm 1) that incrementally computes exact edge-costs only when necessary, utilizing the Lower-Bound Module in the process. In this section, we propose two novel

Algorithm 1 Optimized KM algorithm using refinement rules

```
1: function OptKuhnMunkres(U, V)
2:   M ← φ and initialize labels l_u = l_v = 0
3:   Initialize Lazy MPQ Q_u for each u ∈ U
4:   while M is not a perfect matching (i.e., |M|! = m) do
5:     while unmarked free u ∈ U & no augment. path do
6:       Call find-augmenting-path(u) and mark u
7:     if augmenting path P found then
8:       Augment M by P (increasing size of M by 1)
9:     else
10:      Call update-labels subroutine
11:  return minimum-weight matching M
```

refinement rules, which designate when an exact cost $c(u, v)$ must be computed during the incremental process.

Rule 1 - BFS Expansion: Let us first define *refinement* as extracting an element v from queue Q_u with the smallest individual lower-bound, and computing its cost $c(u, v)$, and then updating $LB(Q_u)$ such that Definition 2 is maintained. We can compute a lower-bound reduced cost for all vertices in Q_u based on (3), as we propose in Lemma 1.

LEMMA 1. $c_r(u, v) \geq LB_r(u, v) = LB(Q_u) - l_u - l_v \forall v \in Q_u$

PROOF. By Definition 2, we have $LB(Q_u) \leq c(u, v) \forall v \in Q_u$. By (3), $c_r(u, v) = c(u, v) - l_u - l_v$. Therefore, $c_r(u, v) \geq LB(Q_u) - l_u - l_v$. Substituting gives $c_r(u, v) \geq LB_r(u, v)$, thus completing the proof. □

The proof of Lemma 1 follows in a straight-forward manner given the definition of $LB(Q_u)$. During the BFS expansion in the augmenting path algorithm, the KM algorithm expands all "tight" edges, i.e., those with reduced cost zero by Invariant 2. To ensure correctness of this expansion in our algorithm, we first propose the following theorem:

THEOREM 1. *Given $v \in Q_u$, if $LB_r(u, v) > 0$ where $LB_r(u, v)$ computed by the definition in Lemma 1, then edge (u, v) cannot be a tight edge.*

PROOF. From Lemma 1, we have that $c_r(u, v) \geq LB_r(u, v)$. If $LB_r(u, v) > 0$, then it follows that $c_r(u, v) > 0$. Thus, $c_r(u, v) \neq 0$ and therefore edge (u, v) cannot be tight by Invariant 2. □

Theorem 1 implies the first refinement rule, which we incorporate into a modified augmenting path search as presented in Algorithm 2. If the BFS reaches vertex $v \in V$ from vertex $x \in U$ and $LB_r(x, v) \leq 0$, we iteratively refine Q_x by extracting the element in Q_x with the smallest lower-bound and updating $LB(Q_x)$ (and therefore $LB_r(x, v)$) for the vertices remaining in Q_x. This loop terminates when either (a) $LB_r(x, v) > 0$ and by Theorem 1, edge (x, v) is not tight and need not be expanded or (b) element v is extracted from Q_x. If the extracted element is not v then we save it in excess set E, which we make sure to re-insert into the queue after the loop ends, to ensure $LB(Q_x)$ remains accurate for other $v \in V$ while ensuring we only compute necessary edge costs. Note, LB_x remains correct for v even when we remove $e \neq v$ from Q_x by the definition $LB(Q_x)$.

Rule 2 - δ Computation: Exact edge costs may also be required to determine δ by (4). Let $\alpha := max(l_v) : v \notin N(S)$, i.e., the maximum label value for vertices not in set $N(S)$ defined in Section 2 (vertices in V visited by the augmenting

Algorithm 2 Find augmenting paths given Rule 1

```
1: function find-augmenting-path(u)
2:   Initialize new queue PQ by inserting u
3:   while PQ is not empty do
4:     Extract candidate x from PQ
5:     for each neighbor v ∈ V of x do
6:       if c(x, v) unknown & LB_r(x, v) ≤ 0 by Lemma (1) then
7:         while LB_r(x, v) ≤ 0 and c(x, v) not yet computed do
8:           Extract minimum element e ∈ V from Q_x
9:           if e = v then
10:            Compute c(x, v) and break loop
11:          else
12:            Add e to set E and update LB_r(x, v)
13:        Re-insert all e ∈ E back to Q_x by LB(x, e)
14:      if c(x, v) was calculated and c_r(x, v) = 0 then
15:        if v is a free vertex (i..e., not covered by M) then
16:          return path P from u to v as augmenting path
17:        else
18:          Add neighbors u ∈ U of v where (u, v) ∈ M to Q
```

path search). We propose an iterative process as in Algorithm 3 to refine and update δ until its final value is attained. We first propose Lemma 2 to define a lower-bound on the smallest reduced cost for any edge (u, v) where $v \in Q_u$:

LEMMA 2. *Let $LB_r(u) = LB(Q_u) - l_u - \alpha$. Then $LB_r(u) \leq c_r(u, v)$ for all $v \in Q_u \setminus N(S)$.*

PROOF. We prove Lemma 2 by contradiction. Let us assume there exists $LB_r(u) > c_r(u, v)$ for some $v \in Q_u$. Since $c_r(u, v) = c(u, v) - l_u - l_v$ and by the definition of $LB(Q_u)$, we have $c_r(u, v) \geq LB(Q_u) - l_u - l_v$. Given our assumption and $\alpha \geq l_v$, $c_r(u, v) \geq LB(Q_u) - l_u - \alpha$. I.e., $c_r(u, v) \geq LB_r(u, v)$, contradicting our assumption. □

Now, given the definition of $LB_r(u)$ we can propose Theorem 2 to identify when to refine a Q_u.

THEOREM 2. *Let $\delta_{cand} = c_r(x, y)$ be a potential δ by (4) for $x \in S, y \notin N(S)$. Given some $u \in S$, if $\delta_{cand} < LB_r(u)$, then $\delta_{cand} < c_r(u, v) \forall v \in Q_u$.*

PROOF. By Lemma 2, we have $LB_r(u) \leq c_r(u, v) \forall v \in Q_u$. Therefore, if $\delta_{cand} < LB_r(u)$ then $\delta_{cand} < c_r(u, v) \forall v \in Q_u$. Thus completing the proof. □

Using Theorem 2, Algorithm 3 can iteratively refine queues until converging to the correct δ. δ_{cand} is the candidate value of δ that we will iteratively update until it is correct. We initialize δ_{cand} with the minimum reduced cost $c_r(u, v)$ amongst $u \in S$ and $v \notin N(S)$ for which $c(u, v)$ has been already calculated and infinity otherwise. Given Q_u where $u \in S$, we compute lower-bound $LB_r(u)$ using Lemma 2. While $LB_r(u) < \delta_{cand}$, we extract the minimum element from Q_u. If it is in $N(S)$ we add to an excess set E, otherwise, we try to filter it by computing an individual lower-bound using the Lower-Bounding Module according to Definition 1, thus potentially avoiding computing an expensive exact cost. Otherwise, we compute the exact cost of the edge and update δ_{cand} if it improves it. Once Q_u is sufficiently refined (i.e., $LB_r(u) \geq \delta_{cand}$), we repeat the procedure for all $u \in S$. $\delta = \delta_{cand}$ upon termination.

The incremental computation of exact costs, adjudicated by the refinement rules, ensures that no other possible δ can be lower than the one computed by Algorithm 3. Similar to the modified augmenting path search in Algorithm 2, this is

Algorithm 3 Updated labels based on Rule 2

```
1: function UPDATE-LABELS
2:   Let S ⊂ U & N(S) ⊂ V be vertices visited by FIND-AUGMENTING-PATH
3:   Set δ to min c_r(u, v) for u ∈ S and v ∉ N(S) where c(u, v) has been computed
4:   for each u ∈ S do
5:     while LB_r(u) < δ_cand with LB_r(u) by Lemma (2) do
6:       Extract minimum element e ∈ V from Q_u
7:       if e ∉ N(S) then
8:         Compute individual LB(u, e) by LB Module
9:         Set LB_r(u, e) = LB(u, e) − l_u − l_e
10:        if LB_r(u, e) < δ_cand then
11:          Compute c(u, e) and c_r(u, e)
12:          if c_r(u, e) < δ_cand then
13:            Set δ_cand = c_r(u, e)
14:        else
15:          Add e to set E and update LB_r(u)
16:      else
17:        Add e to set E and update LB_r(u)
18:    Re-insert all e ∈ E back to Q_u by LB(u, e)
19:  for each u ∈ S do
20:    Increase l_u by δ
21:  for each v ∈ N(S) do
22:    Decrease l_v by δ
```

done in a greedy heuristic way, such that we only refine (and thus compute exact costs) for edges when it is necessary. We propose Theorem 3 to show that our refinement rules still produce the same assignment as the original KM algorithm.

THEOREM 3. *The matching produced by Algorithm 1 is identical to the matching produced by the original Kuhn-Munkres algorithm using the augmenting path search method.*

Proof Sketch: To prove Theorem 3 it is sufficient to show that (a) the modified-BFS and (b) the calculated delta is the same as the original. First, (a) follows simply as Algorithm 1 applies Theorem 1 to all edges originating from $u \in U$, so no tight edges are missed during the U to V expansion. For (b), Algorithm 1 iteratively applies Theorem 2 to each $u \in S$. As such no $c_r(u, v) \forall u \in S, v \notin N(S)$ can be smaller than δ_{cand} at termination.

3.3 IKM Variants

While we proposed our techniques in a way that is agnostic to the implementations and problem setting, the efficacy of our improvement will depend highly on these factors. The accuracy of the lower-bounds (i.e., how close they are to the true edge cost) will determine how effective the filtering steps are. The net gain in performance will be determined by the overhead added by our modifications versus the time saved avoiding exact computations. We propose two variants of our IKM technique to investigate the interplay between filtering efficiency versus overhead as described below:

IKM-DIJK: In this variant, we utilize the priority queue used by Dijkstra's search from each $u \in U$ to implement Q_u. Both individual and group lower-bounds provided by the Lower-Bounding Module utilize the minimum key in the priority queue. The traditional KM implementation would simply conduct a Dijkstra search from each $u \in U$, whereas our incremental approach stops and restarts the search as necessary, potentially terminating earlier. IKM-DIJK will provide an interesting point of comparison as it essentially adds no overhead to the original KM algorithm that utilizes Dijkstra to populate the whole distance matrix.

IKM-CAG: Many road network graph query processing studies have identified the potential benefit of using off-line preprocessing to increase online query performance. As a result, many techniques to compute shortest paths, lower-bounds, and retrieve nearest objects have been proposed that utilize indexing to improve performance. For our second variant, we implement Q_u using COLT [3], which is a state-of-art-technique technique to retrieve objects by minimum lower-bounds. We utilize the ALT index [10] to provide accurate but inexpensive lower-bound computations on shortest path distances in graphs. Lastly, we utilize G-tree [24] to efficiently compute shortest path distances with a reasonable memory footprint. Both the ALT and G-tree indexes are built in an offline pre-processing step, whereas COLT is unique to the current assignment query and built online at query time. All query time overheads are included in the running times reported in all of our experiments.

Approximate KM: [18, 14] proposed an approach similar to ours in goal, from which we develop an approximate algorithm inspired by their minimum-cost flow approach and our lower-bounding heuristic. Please refer to the full paper [4] for discussion of its experimental performance.

Name	Region	# Vertices	# Edges
SIN	Singapore	289,918	632,243
E	Eastern US	3,598,623	8,708,058

Table 1: Road Network Datasets

4. EXPERIMENTS

We conduct a detailed experimental study on the performance of the Incremental Kuhn-Munkres (IKM) algorithm. First, we investigate the likely real-world impact of IKM using actual production datasets provided by Grab . Then in the second section, we study scalability and conduct sensitivity analysis using publicly available real-world datasets and carefully generated synthetic workloads. Further details of the datasets will be provided in each section, while we describe the experimental settings below.

Environment: We run experiments on a MacBook Pro running OS X (64-bit) with a 6-core Intel Core i7 2.6 CPU and 16GB memory for the production datasets, and a Ubuntu 64-bit PC with a 16-core AMD Ryzen 3700X CPU and 32GB for the public datasets. All experiments were conducted using memory-resident indexes for fast querying. We implemented all techniques in single-threaded C++ and compiled by g++ v5.4 with O3 flag, sharing subroutines and basic data structures to ensure fairness.

Techniques: We include the two variants of our IKM technique described in Section 3.3, IKM-DIJK, and IKM-GAC. We compare our techniques against variants of the traditional KM algorithm where the cost matrix is fully computed before the matching is found. These non-incremental KM variants only differ in the technique used to compute the matrix. One variant, *Dijkstra* uses a single-source multi-destination Dijkstra search from each vertex in U to populate the matrix. *G-tree* and *CH* uses point-to-point shortest path distance queries using the G-tree [24] and Contraction Hierarchies (CH) [9] indexes, respectively. The Dijkstra

https://www.grab.com/

Method	Running Time (ms)			Matrix Computations (%)			Max. Throughput (m)		
	W=15s	W=30s	W=60s	W=15s	W=30s	W=60s	W=15s	W=30s	W=60s
Dijkstra	2876ms	4595ms	9749ms	100.0%	100.0%	100.0%	m=575	m=1050	m=1675
CH	661ms	1512ms	6605ms	100.0%	100.0%	100.0%	m=575	m=800	m=1150
G-tree	280ms	599ms	2942ms	100.0%	100.0%	100.0%	m=900	m=1200	m=1650
IKM-DIJK	65ms	110ms	407ms	2.7%	3.7%	4.5%	m=1400	m=1750	m=2275
IKM-GAC	12ms	31ms	255ms	2.9%	3.5%	3.2%	m=1425	m=1775	m=2250

Table 2: Performance metrics for a real-world ride-hailing workload for the city of Singapore. Time window $\boldsymbol{W}$ is the period that ride-hailing requests are batched for which bipartite matching is then used to compute an optimal matching

and G-tree variants allow an apples-to-apples comparison of each of our improved techniques with their corresponding non-incremental counterparts. For example, the difference in running time between G-tree and IKM-GAC will show us how much efficiency is gained from fewer distance computations, while taking into account the overhead added by object retrieval and lower-bound computations.

4.1 Real-World Performance

Given the importance of the real-world applications, we evaluate techniques on real-world data sets provided by Grab for the city of Singapore in several ways as we describe next.

4.1.1 Ride-Hailing Performance

We first evaluate the performance of our techniques on a real-world ride-hailing workload for the city of Singapore.

Datasets: The dataset consists of the road network graph G for Singapore as listed in Table 1 and workload B consisting of hundreds of thousands of anonymized ride-hailing booking records completed in a 1-week period from December 2018. Each booking in set B contains the time of the booking, the driver's location, and the user's location. Both datasets are provided by Grab and originate from real-world data generated in a production setting.

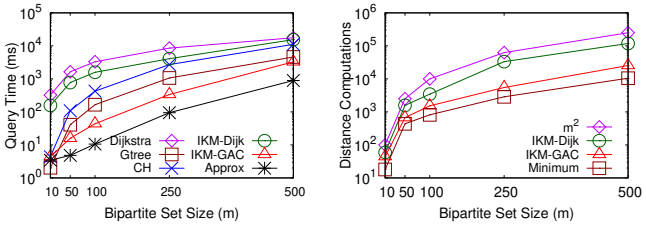
Methodology: To accurately evaluate bipartite matching performance in ride-hailing, we implement a simple batching framework based on public descriptions of real-world matching for ride-hailing applications [1]. Given a time-window W and a start time t, we select all bookings made in the time range $[t, t+W)$ from the booking set B. We then create two bipartite sets using the locations of drivers and users, respectively, from the selected bookings. We use each technique to find an optimal matching on these bipartite sets, reporting the running time and the percentage of the full cost matrix that is computed. We investigate windows W of 15, 30, and 60 seconds, and average the reported results over several randomly selected start times to reduce variability.

Running Time and Efficiency: The running times and matrix computations for each technique over all windows are listed in Table 2. The running times of our techniques, IKM-DIJK and IKM-GAC, are more than an order of magnitude less than their direct counterparts, Dijkstra and G-tree, for all values of W. The reason for this is seen in the percentage of the cost matrix that is computed by our techniques. Naturally, the original KM variants compute 100% of the cost matrix. Notably, the impressive results for IKM-GAC show that the overhead added in computing lower-bounds and retrieving objects is significantly outweighed by the time saved from reduced computations. The magnitude of improvement decreases slightly for the large window W. With a larger window, the density of driver and user locations increases, making lower-bounds less accurate. Nonetheless, the degradation is only slight, and running time is still over a magnitude better than the original KM algorithms.

Maximum Throughput: Due to the commercially-sensitive nature of the data, which is subject to non-disclosure agreements, we are not able to divulge details on the sizes of workload, particularly the average m for each window w. However, in place of this, we report the maximum throughput for each technique in Table 2. Maximum throughput is the largest possible m for which a technique can compute an optimal matching within the time window W. In real-world terms, it is the largest number of bookings that can be batched using each technique for window W, before the next batch must be computed. This is a particularly useful metric, as it will test the ability of each technique to scale to larger cities such as Jakarta and New York, which are likely to generate a far larger workload of bookings. The bipartite sets are again generated from the real-world booking set B as before, except we test increasing values of m by choosing additional bookings (in time order) until the running time is W. Table 2 shows IKM-DIJK and IKM-GAC again leads the way, reporting the highest supported throughput. Note that Dijkstra-based techniques perform relatively better here. This is because Dijkstra's running time grows linearithmically given its asymptotic complexity, while the running time of point-to-point shortest path techniques grows quadratically as it issues one query for each cell in the cost matrix. While modern techniques such as G-tree and CH have significantly improved on Dijkstra's for point-to-point shortest paths, this shows Dijkstra still works well for multi-target shortest paths.

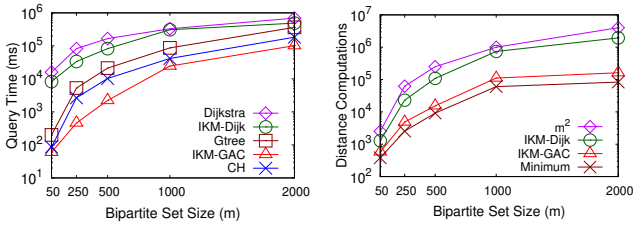
4.1.2 Sensitivity Analysis

The performance on increasing m evaluates the ability of techniques to handle increasingly large batches of ride-hailing requests. We use synthetic driver and user locations to conduct sensitivity analysis into the effect of the size of bipartite sets m. These locations are generated by selecting road network vertices uniformly at random for a given value of m. Road network vertices are more densely located in urban areas, so the coordinates of chosen vertices are more likely to be in such areas, which generally reflects booking requests. Figure 3a show that IKM-GAC improves significantly over G-tree in running time for most values of m. The exception for small values of m is due to the overhead added by IKM-GAC (such as initializing the priority queues Q_u and computing the COLT index). This overhead represents a higher proportion of running time for smaller m where the number of distance computations is small (and as such the savings are also small). In Figure 3b, we compare the number of distance computations computed by each method. Note that both Dijkstra and G-tree compute the same number of distance computations (i.e., for all pairs



(a) Running Time (b) Distance Computations

Figure 3: Singapore Dataset Performance on Varying m



(a) Running Time (b) Distance Computations

Figure 4: Eastern US Dataset Performance on Varying m

of locations), and this is represented in the m^2 line. The improvement shown on the synthetic dataset appears to be smaller than the real-world dataset. This is likely due to *spatial location of matching* being less prominent in the synthetic dataset, which we confirm experimentally. For each pair in the optimal matching, we found that objects in U were on average assigned to the 2nd to 3rd nearest object for the production dataset, experimentally confirming the presence of *spatial location of matching* in real-world datasets. On the other hand, objects were matched to increasingly further objects with increasing m (e.g., 10th nearest object for $m = 250$) for the synthetic datasets. Thus, the synthetic datasets are more challenging, and the still sizeable improvement demonstrates the robustness of our techniques. *Minimum* is an estimate on the theoretical minimum number of costs required to find the optimal matching (its derivation is outlined in [4]). IKM-GAC closeness to the minimum shows the benefit of using an accurate lower-bound [3].

4.2 Scalability Analysis

While the Singapore dataset used in the previous section provides valuable insight into the real-world performance of the techniques, we use additional publicly available datasets for further evaluation. In particular, Singapore has a relatively smaller road network and the size of the road network has a large impact on shortest path computation. Using publicly available datasets will also provide more reproducible results. To study the scalability of the techniques we study their performance on a larger road network dataset, namely, the Eastern (E) US dataset obtained from the 9th DIMACS Challenge with 3.5 million vertices. While a ride-hailing batching operation may not be performed on such a large region, road networks for big congested cities such as Jakarta have similar numbers of vertices and edges. Synthetic bipartite sets for these road networks are generated as in Section 4.1.2, however, we use larger values of m to scale with the increased road network size. We refer to the original paper [4] for experiments on more road networks.

Eastern US Dataset: Figure 4 reports the running time and number of distance computations of each technique for increasing values of m, which corresponds to having more objects to match. We largely see similar trends to the Singapore dataset. For running time (Figure 4a), the gap between IKM-GAC and IKM-DIJK is larger than for Singapore. This can be explained by the linearithmic time complexity of Dijkstra of $O(|E|+|V_G|\log|V_G|)$. Larger numbers of road network vertices $|V_G|$ for the Eastern US road network compared to Singapore (Table 1) results in more costly distance computations. Nonetheless, the improvement of IKM-DIJK is essentially free, as IKM-DIJK introduces no overhead compared to plain Dijkstra, as it uses the same priority queue as the Dijkstra search. Furthermore, the relative improvement of IKM-GAC is higher than IKM-DIJK over its orignal KM counterpart. Given that paths are more costly to compute in a bigger (or denser) road network, this shows that it is worthwhile to pre-process data offline to accelerate online shortest path queries, with fast shortest path distance techniques like G-tree scaling better with increasing size of the road network than Dijkstra. Moreover, it suggests that even the overhead added at query time by IKM-GAC (e.g., construction of the COLT index), which is included in running times reported in all figures, is worthwhile. We also verify the observation made for maximum throughput in Section 4.1.1, with running time of techniques beginning to converge with increasing m as the time to find an optimal assignment begins to dominate cost computation time.

5. RELATED WORK

Given their popularity, real-world ride-hailing apps have spawned a growing body of research. In particular, the Kuhn-Munkres (KM) algorithm is widely used as a subroutine in real-world ride-hailing systems. For example, ride-hailing service Didi reportedly uses KM in the driver dispatch framework [21]. Similarly, Uber frame driver-rider matching as a combinatorial optimization problem to minimize the overall wait time, which is typically solved by KM [1]. The assignment problem, bipartite matching, and Kuhn-Munkres are utilized in many ride-hailing and taxi studies [23, 8, 11]. Our techniques can potentially improve running time in these frameworks as a drop-in replacement for the KM algorithm. Other work has focused on improving different aspects of ride-hailing performance, such as predictive algorithms to increase the likelihood of the driver accepting the allocated job [22]. Such considerations are likely orthogonal to our work, as the cost of allocating a passenger to a driver will still incorporate travel cost.

Since the advent of the Kuhn-Munkres algorithm [13, 15], the time complexity for the general assignment problem has not been significantly improved after [6, 19] improved the original $O(n^4)$ time to $O(n^3)$. Further improvements have come primarily in the form of specialized domains such as considering bounded integer weights [16] or improvements through clever heuristics that work extremely well in practice [12]. Other approaches have attempted to find approximate solutions that trade running time for accuracy [5]. These works are complementary to our technique because they increase the relative running time of computing the cost

http://www.dis.uniroma1.it/%7Echallenge9/

matrix. For example in Figure 1, these techniques would decrease the time taken up by the assignment for increasing values of m, thus making it even more necessary to reduce computations. [4] discusses further related work.

An "incremental" variant of the assignment problem has also been proposed [20], but their definition of incremental involves updating an optimal assignment based on new objects. Some techniques [14, 18] improve the efficiency of the minimum cost flow algorithm by attempting to compute only a partial bipartite graph and terminate early. While utilizing a similar strategy to us, our techniques also attempt to terminate early by computing a partial bipartite graph *and* attempt do so while only computing lower-bounds on the edges we do compute, wherever possible. This is necessary in our problem domain as, for example, road network shortest paths are significantly more expensive to compute than the Euclidean distance costs in [14]. However, it suggests a possible future avenue for research in that we may be able to also optimize the running time of the assignment, which would be helpful when that running time exceeds the cost matrix computation, e.g., for very large values of m.

6. CONCLUSION

The computation of assignment costs is a significant contributor to the overall running time of finding a solution to the assignment problem. However, our techniques show that by utilizing lower-bound costs and pruning rules, it is possible to terminate sooner while computing fewer expensive exact costs. Our experiments show this is particularly effective in the case of driver-passenger matching in ride-hailing services, and applicable to a wide range of frameworks and applications that use the Kuhn-Munkres algorithm as a component. Moreover, the paradigm we present is generalizable and can be potentially applied to other real-world problem settings for similar benefits.

7. ACKNOWLEDGMENTS

This work was primarily conducted while Tenindra Abeywickrama was with the Grab-NUS AI Lab in the Institute of Data Science at the National University of Singapore.

8. REFERENCES

[1] How does Uber match riders with drivers? `https://www.uber.com/us/en/marketplace/matching/`.

[2] T. Abeywickrama, M. A. Cheema, and A. Khan. K-spin: Efficiently processing spatial keyword queries on road networks. *IEEE Trans. Knowl. Data Eng.*, 32(5):983–997, 2019.

[3] T. Abeywickrama, M. A. Cheema, and S. Storandt. Hierarchical graph traversal for aggregate k nearest neighbors search in road networks. In *ICAPS*, pages 2–10, 2020.

[4] T. Abeywickrama, V. Liang, and K.-L. Tan. Optimizing bipartite matching in real-world applications by incremental cost computation. *PVLDB*, 14(7):1150–1158.

[5] P. K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. In *STOC*, pages 555–564, 2014.

[6] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

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

[8] G. Gao, M. Xiao, and Z. Zhao. Optimal multi-taxi dispatch for mobile taxi-hailing systems. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 294–303, 2016.

[9] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[10] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[11] Y. Guo, Y. Zhang, J. Yu, and X. Shen. A spatiotemporal thermo guidance based real-time online ride-hailing dispatch framework. *IEEE Access*, 8:115063–115077, 2020.

[12] R. Jonker and T. Volgenant. Improving the hungarian assignment algorithm. *Oper. Res. Lett.*, 5(4):171–175, 1986.

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

[14] H. U. Leong, K. Mouratidis, M. L. Yiu, and N. Mamoulis. Optimal matching between spatial datasets under capacity constraints. *ACM TODS*, 35(2), 2010.

[15] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[16] L. Ramshaw and R. E. Tarjan. A weight-scaling algorithm for min-cost imperfect matchings in bipartite graphs. In *FOCS*, pages 581–590, 2012.

[17] T. Roughgarden. Cs261: A second course in algorithms, lecture #5: Minimum-cost bipartite matching, January 2016.

[18] Y. Tang, L. H. U, Y. Cai, N. Mamoulis, and R. Cheng. Earth mover's distance based similarity search at scale. *PVLDB*, 7(4):313–324, 2013.

[19] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2):173–194, 1971.

[20] I. H. Toroslu and G. íçoluk. Incremental assignment problem. *Inf. Sci.*, 177(6):1523–1529, 2007.

[21] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *SIGKDD*, pages 905–913, 2018.

[22] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, and J. Ye. A taxi order dispatch model based on combinatorial optimization. In *SIGKDD*, pages 2151–2159, 2017.

[23] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 11(8):853–865, 2018.

[24] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2175–2189, 2015.