CONCORDIA UNIVERSITY

DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

PROJECT REPORT

COMBINATORIAL ALGORITHMS - COMP 6661

WINTER 2020

Submitted to:

HOVHANNES HARUTYUNYAN

Submitted by:

HUA WANG

40011373

SUTHAKHAR PONNAMBALAM

40091123

TABLE OF CONTENTS

1. ABSTRACT

A bipartite graph is one wherein we have two different partitions of vertices and we have edges connecting the two partitions. The two vertex sets are disjoint and independent of each other. There is no edge between vertices in a set. A complete bipartite graph is one wherein we have all the vertices in one partition connected to all the other vertices in the opposite partition. The bipartite graph has growing applications these days starting from ride-sharing platforms to medical applications and even went on to influence advertisement campaigns online. Here in our project, we have considered four different algorithms that operate in different conditions and tabulated our findings to better critic and potentially develop trivial solutions that don't tend to affect the overlying architecture of the algorithm. One of the key reasons to analyze how online algorithms work with a bipartite graph is because of its growing popularity. Almost any problem that we tend to model can be represented as a graph. The advantage with a graph is because a graphical representation can help us understand the different variables in play and how the components in a graph are connected. We also have means of traversing a graph at ease compared with other data structures. A graphical representation opens doors to many possibilities and hence bipartite graph is closely studied and implemented. We will look at all the algorithms with reference to different ride-sharing scenarios. The main aim of this paper is to compare all the algorithms and come up with the most desirable solution that can directly influence the behaviour of an end-user.

2. BACKGROUND

A Bipartite can be expressed as graph G=(U, V, E) where U, V are two disjoint and independent vertex sets. The advantage of using a bipartite graph is one wherein we have a "m" set of service providers and "n" number of users and we have to match them efficiently with minimal costs. The matching between producers and consumers is our area of study. Is there a specific way to match them? Can we match them randomly? On what basis are we trying to match them? These are some of the questions that a developer/ designer has to answer before implementing a bipartite system. Now let us look at a scenario wherein a user is at the airport trying to book a cab to get to his destination. He opens his local ride-sharing application and books a cab instantly. But what is the waiting time? The waiting times can be as low as 30 seconds and as high as 20 minutes. But why is there such a big disparity? Uber is one of the big players in this industry. The company was founded in 2008 and has been expanding ever since. With some basic calculations, one can easily conclude that compared to 2008 we have more cars and more Uber partners in 2020 but still, the waiting time is not reduced. Increased demand can be attributed to increased waiting times but is that the only factor governing the ways the ride-sharing companies schedule their services. The way these companies tend to operate has to be understood in order to answer this question [1]. One recent study conducted in New York showed that the waiting times range anywhere between 20 seconds and a few minutes. The regional manager for Uber East Coast found that to be pretty quick for a city of the size of New York. The same cannot be attributed to other districts in the US or any country for that matter. In reality, a ride-sharing platform can be modeled into a computing problem.

We have “m” users and “n” service providers and we have to match them in an online environment. The word online here refers to dynamic matching because at any instant of time any number of users may be active on the platform and demand may, in turn, affect the matching algorithm. As developers, we have to find the shortest time to match the user and provider. This is an optimization problem as we try to perform matching as quickly and efficiently as possible.

3. PREVIOUS WORK

The classic example for this problem is the Hungarian algorithm.

Input: A bipartite graph, $\{V, U, E\}$ (where $|V|=|U|=n$) and an $n\times n$ matrix of edge costs C

Output: A complete matching, M

1. Perform initialization.

(A)Begin with an empty matching, $M_0=\varnothing$.

(B)Assign feasible values to the dual variables αi and βj as follows:

$$\forall\ v_i \in V,\ \alpha_i = 0 \text{ —–-Equation (a)}$$

$$\forall\ u_j \in U,\ \beta_j = m_i\ n_i\ (c_{ij}) \text{ —–-Equation(b)}$$

2. Perform n stages of the algorithm, each given by the routine stage.

3. Output the matching after the nth stage : $M=M_n$

This the traditional Hungarian algorithm. The problem with this algorithm is that this problem cannot be applied to a dynamic setting. If the input is fixed then this problem can be easily applicable. But in everyday applications the number of hosts tends to keep changing at any instant of time, hence this cannot be used for such cases. Thus we will discuss about the four dynamic solutions that we have that can solve the problem in a dynamic environment.

a) *Greedy approach*

We will first look into a greedy algorithm that solves the problem but does not provide us with an optimal solution. We will look at the bounds that are derived from this problem then we look at the disadvantages this approach has to offer. As the name suggests the approach is greedy and hence the decisions made are local. Here local refers to the best local optimum we derive at that point of time. This can be bad in certain ways because future inputs can provide us with much better results if we had waited, but the algorithm will force us to make a decision at that point with no knowledge about future.

b) *Batch based Approach*

The disadvantage that we tend to encounter in most ride sharing platforms is large waiting times. The reason for this can be large data sets. One way to limit this problem is to implement a solution where in we divide the input into batches and then try to find matching between the variables inside a pair. In this approach we can derive a much faster algorithm as once we have set our input it becomes an offline solution as no new nodes will be added. Thus we can design a much faster algorithm. But we are missing a crucial point here. We are not interested in designing a faster algorithm that improves the execution speed but we tend to design an algorithm that runs faster in terms of its data structure organization and implementation.

But this approach has a cache. We have to come up a batch size so that it's not too big nor too small. But how do we come up with such a value. As we were discussing the bipartite environment is dynamic and volatile and the loads may differ at different point of time. In order to come up with a value that encompasses all the properties we can refer to historical data and change the batch value for different time units depending on the circumstances. This threshold value is not perfect but can mimic the environment to some extent.

c) *TGOA Algorithm*

This is the third approach that we will discuss. In this approach we essentially combine the greedy and the batch based approach to formulate a new working schema that respects the conditions. If the input is not too big we can go for greedy approach. But if the input is too dynamic we can switch to batch based approach and perform matching in such cases. The time we decide to make a switch is really important. Because each approach has its own advantages and disadvantages. We essentially should not try to create something for betterment and in-turn make it more complicated. The switch will based on number of active nodes in the system at any particular instant of time. In batch based strategy we only care about the load factor but in this factor we focus on load as well as the time in which loads vary. By broadening our spectrum we can provide better experience by reducing waiting times at the user end.

d) *Restricted Q learning approach*

Machine learning is our final strategy to tackle this problem. We can use reinforcement learning technique like restricted Q learning to essentially train the program to act accordingly at different situations. According to one source, “Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.” [Source: Wikipedia]. We will look into this algorithm and analyze how AI can solve this problem. It follows different strategies compared to traditional problem solving techniques and we will compare and contrast all the algorithms.

4. ANALYSIS OF ALGORITHMS:

a) Greedy approach

Consider the pseudocode as follows [2].

```
Input : A dynamic bipartite graph $B$
Output : A matching allocation $M$
1. $M \leftarrow \varnothing$;
2. for each new arrival node $u$ in $B$ do
3.  if $u \in L$ then
4.  $V \leftarrow \{ v \mid v \in R, w(u, v) > 0\}$;
5. if $V /= \varnothing$ then
6. $v' \leftarrow \arg\max_{v \in V} w(u, v)$;
7. Remove $u$, $v'$ from $B$;
8. $M \leftarrow M \cup \{(u, v')\}$;
9. else
10. Symmetrically matching $u \in R$;
```

11. Return *M*

In this algorithm, whenever we insert a new vertex we compare it's associated cost with the set of vertices and we tend to serve the vertex that has the highest cost. Here the authors have presented a solution where in the vertex with maximum cost gets served first. But imagine a situation where in worst case a new vertex that is inserted always has the highest cost. This can lead to other nodes in the system to starve. Hence this approach is not the widely accepted. The baseline in this approach is one where in a node will be matched with neighbour with highest cost and leave out a vertex if it has no neighbours.

Next the discussion has to be in terms of complexity ratio. The algorithm performs poorly in terms of adverbial arguments because we can insert a node v opposite to u with a weight potentially higher than u, and as soon as *u* gets matched greedily to some other node with the weight say ϵ. If ϵ is set to an infinitely small value, then the competitive ratio is close to 0.

The complexity ratio is expressed as $CR = \min_B U(B,M) / Opt(B)$

Based on this condition we found some of the results to be interesting and intriguing.

For any dynamic bipartite graph we can have an upper bound on C to be 2, C >= 2. This is in accordance with the equation:

$$\mathrm{Min}_B = U_E(B, S_B) / Opt(B) = 1/C - 1 \qquad \text{—Equation (1)}$$

b) Batch Based Strategy

In batch based strategy we can start matching the vertices only after a batch is full. There are many considerations that we have to govern.

1. Matching cannot start unless and until a batch is full.

2. Once a batch is set no new vertices can be added to the batch.
3. There should be heuristics defined to manage the batch size.
4. Once batch size is fixed we cannot change the size in the middle of execution.

The notion behind the batch based strategy is:

1. If a node is allowed for fixed period of time rather than getting matched immediately then it has a higher probability of finding a suitable match.
2. The batch size should be adjusted according to the dynamic bipartite graph in consideration.

We set a binary function b(i, j) = 1, if (i − 1, j] is a batch; otherwise, b(i, j) = 0. Based on this condition we can come up with the following conclusions:

1. If all nodes inside a batch has value 1, then they are ready to be matched once the batch is full.
2. The authors then proceed to finding optimal matching inside each batch to verify the correctness of the algorithm.
3. The authors also have a pointer to make sure that unmatched nodes in a batch are removed or whether they tend to get carried away to the next batch in declaration.

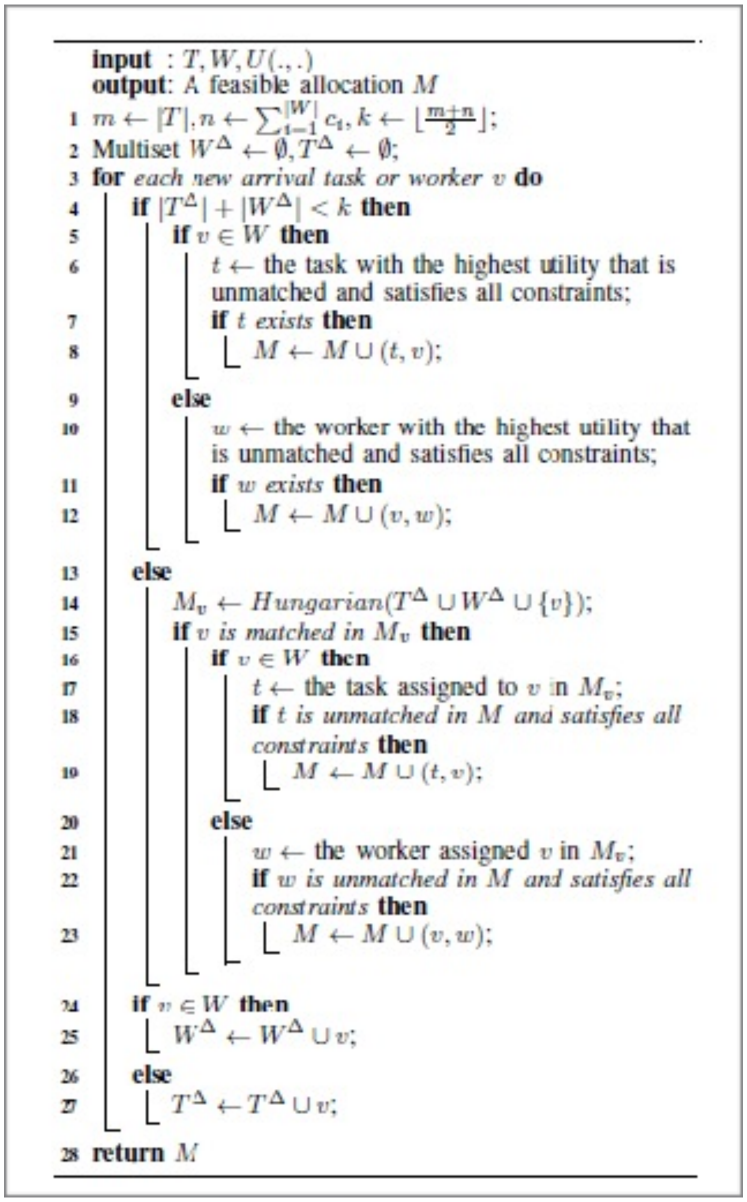
For any given dynamic bipartite graph B with an upper bound of duration $C \geq 3$, we have

$$\mathbf{1/C - 1 \leq \min_B UR(B, S_b) / Opt(B) < 2/C - 2.} \qquad \text{— Equation(2)}$$

The proof for this claim is quite straight forward. If C=2 then it becomes an optimal allocation. To get an upper bound we set n= (C-1)/2 enclosed in a floor function. Plugging the values for the equation (2) we get a $(1+(n-1)\,\epsilon)/n$ which provides with the expected upper bound. Thus the relation is holding well. This competitive ratio is applicable to batch based strategy in a expiration model. This is a model where in after a period of time batch size is set based on the input and the matching starts to take place in the batch. This strategy also has a constant competitive ratio according to [2].

c) TGOA Algorithm

Before discuss about this algorithm let us take a closer look at the pseudo code for this algorithm.



```
input : $T, W, U(.,.)$
output: A feasible allocation $M$
$m \leftarrow |T|, n \leftarrow \sum_{i=1}^{|W|} c_i, k \leftarrow \lfloor \frac{m+n}{2} \rfloor$;
Multiset $W^{\Delta} \leftarrow \emptyset, T^{\Delta} \leftarrow \emptyset$;
for each new arrival task or worker $v$ do
    if $|T^{\Delta}| + |W^{\Delta}| < k$ then
        if $v \in W$ then
            $t \leftarrow$ the task with the highest utility that is unmatched and satisfies all constraints;
            if $t$ exists then
                $M \leftarrow M \cup (t, v)$;
        else
            $w \leftarrow$ the worker with the highest utility that is unmatched and satisfies all constraints;
            if $w$ exists then
                $M \leftarrow M \cup (v, w)$;
    else
        $M_v \leftarrow Hungarian(T^{\Delta} \cup W^{\Delta} \cup \{v\})$;
        if $v$ is matched in $M_v$ then
            if $v \in W$ then
                $t \leftarrow$ the task assigned to $v$ in $M_v$;
                if $t$ is unmatched in $M$ and satisfies all constraints then
                    $M \leftarrow M \cup (t, v)$;
            else
                $w \leftarrow$ the worker assigned $v$ in $M_v$;
                if $w$ is unmatched in $M$ and satisfies all constraints then
                    $M \leftarrow M \cup (v, w)$;
    if $v \in W$ then
        $W^{\Delta} \leftarrow W^{\Delta} \cup v$;
    else
        $T^{\Delta} \leftarrow T^{\Delta} \cup v$;
return $M$
```

Fig. 1. TGOA Algorithm

The analysis involves establish a competitive ratio for the algorithm. In TGOA algorithm we can divide the operation of the algorithm to operate in two different phases. The initial operating conditions is quite similar to that of greedy strategy. Then once a threshold is reached the algorithm will start to mimic batch based approach for better efficiency. Thus the TGOA algorithm is able to maintain a fairly better competitive ratio compared to other algorithms that we have discussed.

The switch between two phases takes place based on a threshold value that is an indicator of the load on the network. If the load increases exponentially then a batch based strategy seems to be a better alternative compared to the traditional greedy approach. The reason being in extreme load conditions the greedy approach will not provide optimal matching as the greedy approach makes local decisions.

Consider the theorem, according to [3].

$$
\begin{aligned}
&\mathbb{E}[MaxSum(M)] = \frac{1}{2}\mathbb{E}[\sum_{v=1}^{m+n} U_v] \\
&\geq \frac{1}{2} \times \frac{1}{2} \sum_{v=\lceil \frac{m+n}{\epsilon} \rceil}^{m+n} (\frac{\lfloor \frac{m+n}{2\epsilon} \rfloor}{|W^{\Delta}|-1} \frac{|T^{\Delta}|MaxSum(OPT)}{mn} + \\
&\frac{\lfloor \frac{m+n}{2\epsilon} \rfloor}{|T^{\Delta}|-1} \frac{|W^{\Delta}|MaxSum(OPT)}{mn}) \\
&\geq \frac{1}{2} \sum_{i=\lceil \frac{m+n}{\epsilon} \rceil}^{m+n} \frac{\frac{m+n}{2\epsilon}}{mn} MaxSum(OPT) \\
&= \frac{\lfloor \frac{m+n}{2\epsilon} \rfloor}{2mn}(m+n-\lceil \frac{m+n}{\epsilon} \rceil + 1) MaxSum(OPT) \\
&\geq \frac{1}{\epsilon}(1-\frac{1}{\epsilon}) MaxSum(OPT)
\end{aligned}
$$

In order to maximize the $\frac{1}{\epsilon}(1-\frac{1}{\epsilon})$, we can easily know that

$$\frac{1}{\epsilon}(1-\frac{1}{\epsilon}) \leq (\frac{\frac{1}{\epsilon}+1-\frac{1}{\epsilon}}{2})^2 = \frac{1}{4}$$

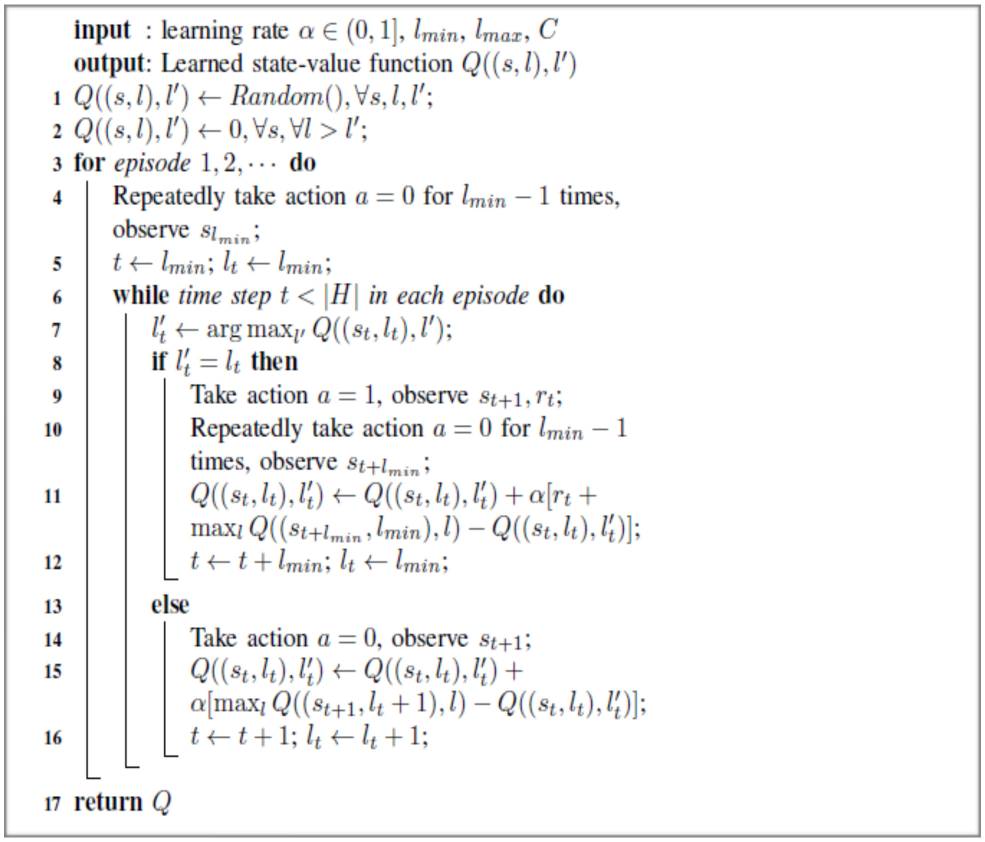
Therefore, $\mathbb{E}[MaxSum(M)] \geq \frac{1}{4}MaxSum(OPT)$. ■

This is because in random order model, an arbitrary order must have corresponding symmetrical order. In an arbitrary order, if a new arrival item v to a task, in the corresponding symmetrical order, there must be an item with the same arrival order that is a worker. Thus this ratio.

The advantage with this approach is its ability to perform switch between two approaches based on the load on the network. Thus it is able to achieve a better competitive ratio compared to the rival algorithms. Since this switch is advantageous the algorithm has better complexity ratio in terms of efficiency.

d) *Restricted Q learning approach*

This is the final approach we will be discussing. In this approach we implement a reinforcement learning based solution that has better efficiency in the long run and is relatively efficient and provides us with insightful data.



```
   input  : learning rate $\alpha \in (0,1]$, $l_{min}$, $l_{max}$, $C$
   output: Learned state-value function $Q((s,l),l')$
1  $Q((s,l),l') \leftarrow Random(), \forall s,l,l'$;
2  $Q((s,l),l') \leftarrow 0, \forall s, \forall l > l'$;
3  for episode $1,2,\cdots$ do
4      Repeatedly take action $a = 0$ for $l_{min} - 1$ times,
       observe $s_{l_{min}}$;
5      $t \leftarrow l_{min}$; $l_t \leftarrow l_{min}$;
6      while time step $t < |H|$ in each episode do
7          $l'_t \leftarrow \arg\max_{l'} Q((s_t,l_t),l')$;
8          if $l'_t = l_t$ then
9              Take action $a = 1$, observe $s_{t+1}, r_t$;
10             Repeatedly take action $a = 0$ for $l_{min} - 1$
               times, observe $s_{t+l_{min}}$;
11             $Q((s_t,l_t),l'_t) \leftarrow Q((s_t,l_t),l'_t) + \alpha[r_t +$
               $\max_l Q((s_{t+l_{min}}, l_{min}), l) - Q((s_t,l_t),l'_t)]$;
12             $t \leftarrow t + l_{min}$; $l_t \leftarrow l_{min}$;
13         else
14             Take action $a = 0$, observe $s_{t+1}$;
15             $Q((s_t,l_t),l'_t) \leftarrow Q((s_t,l_t),l'_t) +$
               $\alpha[\max_l Q((s_{t+1}, l_t + 1), l) - Q((s_t,l_t),l'_t)]$;
16             $t \leftarrow t + 1$; $l_t \leftarrow l_t + 1$;
17 return $Q$
```

Fig. 2 . Restricted Q Learning

The restricted strategy space of S_b by an interval Len is denoted by $S_{len}B$. The strategies in S_{len} B satisfy that ∀ b(i, j) = 1, j−i ∈ Len. This restricted space allows for the algorithm to limit strategies when making decisions. Thus more time will not be wasted for computations and decisions can be made quickly. Based on the algorithm we can say that this algorithm will use less space compared to other algorithms as it makes optimized decisions.

In The R-QL algorithm every action is modelled as a state. And all the states are tabulated in a table. The state table has a few limitations though:

The size of our state space is infinite meaning it is not possible to record all actions in the table.

The table also has sparsity problem which means we cannot come across same state twice. This can be harmful as two states that are similar have same functionality and the former can be used as a reference for the later.

When we discuss about complexity of an ML program we have to consider the space and time complexities. The space complexity of an ML program is $O(C^2(l_{max}-l_{min})^2)$ where l_{max} and l_{min} are the boundaries that we set initially to prevent the algorithm from processing unwanted data. The time complexity is of the order O(H(C)) where H(C) is the complexity of the Hungarian algorithm. This is because once the limit is set and values are filled the problem becomes an offline problem and it can be implemented using Hungarian algorithm.

5. CONCLUSION AND FUTURE WORK

From this comparative study we can conclude that,

1. A greedy solution can provide optimal solution only for small data sets.
2. A greedy solution is not particularly helpful in large volumes of data as local optimum is not be optimal solution when considering future inputs.

3. A batch based strategy can be useful to get optimal solution by converting an online problem into an offline problem based on the batch size.
4. Identifying batch size is crucial as it can have positive or negative impacts in our solution.
5. The batch size cannot be fixed before hand as it is not static. It can be fixed based on historical data that closely resemble the environment in which future data may operate upon.
6. The TGOA algorithm overcomes the pitfalls of both greedy and batch based strategy and presents a solution where in the algorithm has the potential to operate in two different phases.
7. The threshold value for making the switch in the TGOA algorithm depends on the load on the network and also the time during which the application is being used.
8. The RQL algorithm has the best potential to solve the problem in limited amount of time and space complexities. IT uses reinforcement learning technique to make decisions in real time environment. It uses an upper and lower bound to limit the execution within the interval and anything that is outside the interval will be neglected.

The future scope in this domain is open and remains to be explored. A few areas where we believe improvements can be made are,

1. In the greedy approach we have a situation where in we always serve the node with maximum weight. In some applications this need not be the case. One particular example would be targeted advertisements where in we always don’t target the frequent buyers. We should look to get new customers to try our product. So in this example users with average weights are considered then the customers with max weights.

2. In batch based strategy fixing an appropriate batch size is open for exploration. This algorithm core is on fixing the batch size and hence future studies on this area is expected to be promising.

3. In ML approach, the authors have only given us an overview of using reinforcement learning. Machine learning is a big area and has the resources under supervised, unsupervised and rote learning techniques that can provide better results. This area looks even more promising and intriguing in many aspects.

6. ACKNOWLEDGEMENTS

We sincerely thank our Professor Mr. Hovhannes Harutyunyan for all his valuable inputs in helping us achieve our goals in this project. His inputs were crucial in decision making and helped us understand the concepts with better frame of mind. We also would like to extend our gratitude to the course TA Anne-Laure Ehresmann for her swift response whenever we had a query regrading the project. We again wish to thank you for all your efforts in helping us.

References

[1] R. M. Karp, U. V. Vazirani, and V. V. Vazirani, “An optimal algorithm for on-line bipartite matching” in *STOC*, 1990, pp. 352–358.

[2] Y. Wang, Y. Tong, C. Long, P. Xu, K. Xu and W. Lv, "Adaptive Dynamic Bipartite Graph Matching: A Reinforcement Learning Approach," 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, Macao, 2019, pp. 1478-1489.

[3] Y. Tong, J. She, B. Ding, L. Wang and L. Chen, "Online mobile Micro-Task Allocation in spatial crowdsourcing," 2016 IEEE 32nd International Conference on Data Engineering (ICDE), Helsinki, 2016, pp. 49-60.

[4] Z. Huang, N. Kang, Z. G. Tang, X. Wu, Y. Zhang, and X. Zhu, “How to match when all vertices arrive online,” in *STOC*, 2018, pp. 17–29.

[5] H. Ting and X. Xiang, “Near optimal algorithms for online maximum edge-weighted b-matching and two-sided vertex-weighted b-matching,” *Theor. Comput. Sci.*, vol. 607, pp. 247–256, 2015.

[6] C.-J. Ho and J. W. Vaughan, “Online task assignment in crowdsourcing markets,” in AAAI 2012.

[7] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[8] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction*, ser. Adaptive computation and machine learning. MIT Press, 1998.