Maximizing the Total Utility of Requesters in Crowdsourcing

Xiao Chen
Department of Computer Science, Texas State University, San Marcos, TX 78666
Email: xc10@txstate.edu

***Abstract*—Crowdsourcing coordinates a large group of workers online to do small tasks published by requesters on a crowdsourcing platform. In the literature, the model used by many papers assumes that one task is given to and completed by one worker only. In this paper, we consider a model that extends this model in space and time. Based on our model, we formulate an optimization problem from the perspective of the requesters that maximizes the utility of all the requesters subject to the constraint that the total workload given to a worker should not exceed his capability. We then provide a solution to the problem and design distributed algorithms for the requesters and the workers to interact with each other in multiple rounds. After that, we give a concrete example to explain the solution and the algorithms. Then, we discuss the convergence speed of our algorithms using different methods. Finally, we conduct simulations to compare these convergence methods and draw conclusions.**

***Index Terms*—convergence, crowdsourcing, optimization, utility, workload**

I. Introduction

Crowdsourcing [2] has gained popularity in recent years because it allows requesters to find a group of workers online to work on small tasks that an individual or organization cannot easily do. There are three basic components in crowdsourcing, as shown in Fig. 1: requesters (R_1, R_2, $\cdots$) who publish tasks on a platform, workers (W_1, W_2, $\cdots$) who carry out the tasks, and a platform such as Amazon Mechanical Turk [1] that matches requesters and workers.

In the literature, the model used by many papers [11], [13], [18] assumes that one task is given to and finished by one worker. In this paper, we consider a model that is an extension of this model. We assume that a task from a requester may be processed by multiple workers in a pipeline fashion. For example, in Fig. 1, a task from requester R_3 needs to be processed by workers W_1, W_3, and W_5 sequentially. Similar to the existing model, a requester can have multiple tasks and a worker can get tasks from different requesters. For instance, requester R_3 has another task that needs to be finished by worker W_4, and worker W_1 participates in tasks from requesters R_1, R_2, and R_3.

In our model, we not only extend the one-task-by-one-worker model to a one-task-by-multiple-worker model (which we call space extension), but also extend the model in time. We assume that a requester can generate some amount of the same task continuously over time and distribute it to the same workers to finish. With time extension, we are able to model the interaction between the requester and workers over time. That is, in each round, the requester can decide how much workload of tasks should be assigned to the workers based on the prices charged by them. When a worker receives the workload from the requester, he can decide the price he will charge the requester next time. Once the prices are changed by the workers, the requester can update the workload of the task to send to the workers and so on. For the requester, the higher the prices, the lower the amount of work he is willing to give to the workers. And for the workers, the more workload they can get from the requester, the higher the prices they can charge the requester. The requester and workers will interact with each other like this for many rounds. We also assume that, in this whole process, at any time, the workload obtained by each worker should not exceed his capability. If we set the number of workers to finish a task to one and the number of rounds to one in our model, then our model is exactly the same as the one in the literature.

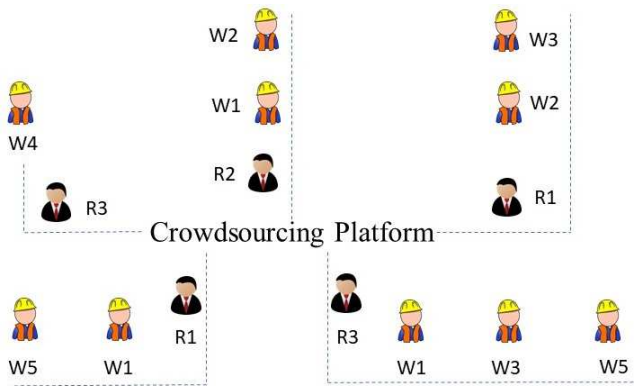


Fig. 1. A crowdsourcing model

One important question to make crowdsourcing practical is how to motivate workers to contribute to the tasks. This has been extensively studied [6], [9], [16], [21], [23]. In this paper, we shift gears to look at the problems on the side of the requesters, which have not been discussed as much. Based on our defined model, we first formulate an optimization problem to maximize the total utility of all the requesters, subject to the constraint that the total workload assigned to a worker should not exceed his capability. Then, we provide a solution to the optimization problem and propose distributed algorithms for the requesters and the workers to interact with each other in multiple rounds. After that, we give a concrete example to explain the solution and the algorithms. We then discuss the convergence speed of our algorithms using different methods.

Finally, we conduct simulations to compare these methods and draw conclusions.

The differences between our work and others, as well as the key contributions of our work, are as follows:

- We extend the existing task-worker model in both space and time.
- We define an optimization problem from the requesters' perspective based on the extended model.
- We provide a general solution to the problem and propose distributed algorithms for requesters and workers.
- We instantiate the solution with a concrete example to explain how requesters and workers interact.
- We discuss the convergence speed of our algorithms using various methods.
- We conduct simulations to compare these methods and draw conclusions.

The rest of the paper is organized as follows: Section II references the related work; Section III defines the problem; Section IV provides the solution and algorithms; Section V explains the solution using a concrete example; Section VI discusses the convergence speed of the algorithms using different methods; Section VII presents the simulations to compare these methods; and the conclusion is in Section VIII.

II. Related Work

In this section, we reference the related work and point out the differences between our effort and theirs.

Utility Function: A utility function represents the level of preference in microeconomics. The definitions of utility functions vary depending on the research problem. In [7], the utility function was calculated by the sum of contexts and whether the sensing task had been performed or not. In [17], it was related to the sensing incentive in a certain area. In this paper, the utility function of a requester reflects his satisfaction. The more workload is completed, the more satisfied the requester will be. However, the increase in satisfaction diminishes with each additional amount of workload [3].

Supply and Demand: it is an economic model of price determination in a market [4]. The higher the price, the lower the demand, and vice versa. If no one conducts a task in crowdsourcing, the incentive will increase to promote participation [17]. In our model, the requesters are the consumers; the higher the prices the workers charge, the lower the demand to have the work done. The workers are the suppliers; the higher the demand, the higher the price they can ask for.

Crowdsourcing Model: Many papers [11], [13], [18] use a model which assumes that one task is given to and completed by one worker. Here, we assume that a task from a requester may be processed by multiple workers in a pipeline fashion.

Perspective: The problems for the workers, for example, how to motivate them to contribute to the tasks, have been extensively studied [6], [9], [16], [21], [23]. In this paper, we shift gears to look at the problems on the side of the requesters, which have not been discussed as much.

Interaction of Players: A model that can describe the interplay of two sides is the Stackelberg game [14], where there are two players: the leader and the follower. The follower will give a response to the leader's strategy, and the leader will consider the response in its next strategy. The papers that utilized this model in various ways in crowdsourcing are [19], [20], [22]. To make the Stackelberg game work, the leader needs to know the information of the follower so that he can calculate the optimal action from the follower. In our model, a worker is a leader who initiates the interaction process by charging a price, but he is not able to know the information of all the other workers used by the follower (the requester) since we allow a task to be processed by multiple workers. Therefore, it is not appropriate to use the Stackelberg game to model the interaction of the requesters and the workers here. The interaction in our model will be described in Section IV.

III. Problem Formulation

In this section, we formulate the problem. Suppose there are n requesters and m workers. For a requester R_i, each of his tasks may be processed by multiple workers in a pipeline way. A worker W_j can get tasks from multiple requesters, but he has a capability of only processing c_j workload at any time. Each worker asks a price based on his workload given by the requesters and each requester decides how much workload to send to the workers based on the prices the workers charge. The workload from the requesters will in turn decide the price charged by a worker. Our goal is to reflect the interaction between the requesters and the workers in multiple rounds.

We use the utility function $U_i(x_i)$ to represent the satisfaction or benefit of requester R_i related to the workload x_i given to the workers. The more work is done, the more satisfied a requester becomes, so it should be an increasing function. It should also be concave to reflect the principle of diminishing marginal returns [3]. That is, the incremental satisfaction of a requester drops with each additional amount of workload x_i. Our overall goal is to maximize the total utility of all the requesters. We introduce a binary-valued indicator matrix A, so that $A_{ji} = 1$ if requester R_i sends workload to worker W_j, and $A_{ji} = 0$ otherwise. Combining our goal and conditions, we formulate an optimization problem as follows.

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & \sum_i U_i(x_i) \\ \text{subject to} \quad & \sum_i A_{ji} x_i \leq c_j, \forall j \end{aligned} \tag{1}$$

This problem is to maximize the total utility of all the requesters. The constraint states that for any worker W_j, the total workload of the tasks he receives from the requesters should not exceed his capability to finish the tasks.

IV. The Solution and Algorithms

In this section, we provide a solution to our defined problem and propose algorithms for the requesters and workers.

In our defined problem, the utility functions are concave, the constraints are linear, and the objective is to maximize. Thus, the problem is convex. We can first turn the constrained optimization problem into an unconstrained one by writing down the Lagrangian [5]. We add a Lagrange multiplier p_j

($p_j \geq 0$) to each constraint. We name the multiplier p_j because it represents the price charged by worker W_j to the requesters he works for. Now, the Lagrangian of the original problem (1) has two variables: x and p. That is:

$$L(x,p) = \sum_i U_i(x_i) + \sum_j p_j(c_j - \sum_i A_{ji}x_i). \quad (2)$$

For the two unknown parameters x and p, we first fix p. Given p, we want to find x^* that maximizes $L(x,p)$. Since p_j and $c_j - \sum_i A_{ji}x_i^*$ are non-negative, the following is true:

$$\begin{aligned} max_x L(x,p) &= \sum_i U_i(x_i^*) + \sum_j p_j(c_j - \sum_i A_{ji}x_i^*) \\ &\geq \sum_i U_i(x_i^*) \end{aligned} \quad (3)$$

This means that given p, $max_x L(x,p)$ is an upper bound of the original objective $\sum_i U_i(x_i)$. The x^* that can maximize $L(x,p)$ can be calculated as follows: we take the partial derive of $L(x,p)$ with respect to x_i and make it zero. We have:

$$\frac{\partial L(x,p)}{\partial x_i} = U_i'(x_i) - \sum_j p_j A_{ji} = 0.$$

Here, $\sum_j p_j A_{ji}$ can be rewritten as $\sum_{j \in W(i)} p_j$, where the set $W(i)$ represents all the workers used by requester R_i in a task. Thus,

$$U_i'(x_i) = \sum_{j \in W(i)} p_j. \quad (4)$$

If the utility function is given, we can solve equation (4) to get the optimal x^* that maximizes $L(x,p)$.

Then, we can tighten the upper bound by minimizing it over p and have the Lagrange dual problem as follows. Since the original problem (1) is convex, the answer to the dual problem is exactly the answer to the original problem [5].

$$\begin{aligned} &minimize_p L(x^*,p) \\ &= minimize_p \{\sum_i U_i(x_i^*) + \sum_j p_j(c_j - \sum_i A_{ji}x_i^*)\}. \end{aligned} \quad (5)$$

For this dual problem, we can use the gradient method [5]. The gradient is the partial derivative of $L(x^*,p)$ with respective to p_j. That is,

$$\frac{\partial L(x^*,p)}{\partial p_j} = c_j - \sum_i A_{ji}x_i^*$$

We can rewrite $\sum_i A_{ji}x_i^*$ as $\sum_{i \in R(i)} x_i^*$, where the set $R(i)$ contains all the requesters that worker W_j works for. We derive the following to find the optimal solution to problem (5).

$$p_j[t] = max\{p_j[t-1] - \gamma(c_j - \sum_{i \in R(i)} x_i^*),\ 0\} \quad (6)$$

In this equation, t is the time stamp or round number and γ is the step size. We assign the maximum of $p_j[t-1] - \gamma(c_j - \sum_{i \in R(i)} x_i^*)$ and 0 to $p_j[t]$ to ensure that it is non-negative.

Once the solution to the problem is ready, we can write a *workload decision algorithm* (WLD) for each requester R_i to determine the workload to send to his workers and a *price update algorithm* (PU) for each worker W_j to update the price based on the workload he receives. These are distributed algorithms- the requesters and the workers update the workload and price independently based on the information available to them at time t. At the same time, these algorithms are interrelated. Once the workers change their prices, the requesters will change their workload. And once the requesters change their workload, the workers' prices will be updated. The process will continue until it converges. Finally, we have the solution to the dual (5) and the original (1) problems.

Algorithm WLD: Workload Decision by Requester R_i

1: **Inputs:** the price charge from the workers
2: **Output:** the workload decided by requester R_i at time t.
3: /* At time t, each requester R_i calculates the total price (denoted by $P_i[t]$) charged by the workers to finish a task */
4: $P_i[t] = \sum_{j \in W(i)} p_j[t-1]$
5: obtain $x_i^*[t]$ by solving equation (4)

Fig. 2. The algorithm used by requester R_i to decide the workload

Algorithm PU: Price Update by Worker W_j

1: **Inputs:** the workload from the requesters
2: **Output:** the updated price
3: /* At time t, each worker W_j sums up the total workload (denoted by $L_j[t]$) from the requesters */
4: $L_j[t] = \sum_{i \in R(i)} x_i[t]$
5: /* update the price using equation (6) */
6: $p_j[t] = max\{p_j[t-1] - \gamma(c_j - \sum_{i \in R(i)} x_i^*),\ 0\}$

Fig. 3. The algorithm used by worker W_j to update his price

V. An Example

In this section, we use a concrete example to explain the proposed distributed algorithms. We show how the requesters and workers interact with each other in the process.

For convenience's sake, we scale the numbers to the range of [0, 1]. Suppose there are 3 requesters and 4 workers. We choose $\ln x_i$ as the utility function for requester R_i, as it is increasing, continuous, and differentiable. It is also concave to reflect the principle of diminishing marginal returns.

Matrix A is shown in Fig. 4. It indicates that requester R_1 has a task that requires the pipeline of workers W_1, W_2, and W_4, while requester R_2 has a task that will be processed by workers W_2 and W_3, etc. The workers' capability vector $C = [c_1, c_2, c_3, c_4] = [1, 0.6, 0.8, 0.7]$ and the step size γ is set to 1. Initially, each worker charges a price of 1. The requesters and workers will update their workload and price in the following loop until the process converges.

At time $t = 0$:

The charge for each requester is:

$$P_1[1] = p_1[0] + p_2[0] + p_4[0] = 1 + 1 + 1 = 3$$
$$P_2[1] = p_2[0] + p_3[0] = 1 + 1 = 2$$
$$P_3[1] = p_3[0] + p_4[0] = 1 + 1 = 2$$

$$
\begin{array}{cc}
Workers \backslash Requesters & \begin{array}{ccc} R_1 & R_2 & R_3 \end{array} \\
\begin{array}{c} W_1 \\ W_2 \\ W_3 \\ W_4 \end{array} & \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{array}\right]
\end{array}
$$

Fig. 4. Matrix A

Based on the charge, each requester decides the workload to send to the workers by solving equation (4). The utility function of requester R_i is $\ln x_i$, and the derivative of $U_i'(x_i)$ with respect to x_i is $\frac{1}{x_i}$. Therefore, each requester R_i sends the following the workload to the workers.

$$
\begin{aligned}
x_1[1] &= \frac{1}{P_1[1]} = \frac{1}{3} = 0.333 \\
x_2[1] &= \frac{1}{P_2[1]} = \frac{1}{2} = 0.5 \\
x_3[1] &= \frac{1}{P_3[1]} = \frac{1}{2} = 0.5
\end{aligned}
$$

Next, the workers add up their workload from the requesters.

$$
\begin{aligned}
L_1[1] &= x_1[1] = 0.333 \\
L_2[1] &= x_1[1] + x_2[1] = 0.333 + 0.5 = 0.833 \\
L_3[1] &= x_2[1] + x_3[1] = 0.5 + 0.5 = 1 \\
L_4[1] &= x_1[1] + x_3[1] = 0.333 + 0.5 = 0.833
\end{aligned}
$$

And then the workers update their prices according to the workload they receive from the requesters.

$$
\begin{aligned}
p_1[1] &= max(p_1[0] + L_1[1] - c_1, 0) = 0.333 \\
p_2[1] &= max(p_2[0] + L_2[1] - c_2, 0) = 1.233 \\
p_3[1] &= max(p_3[0] + L_3[1] - c_3, 0) = 1.2 \\
p_4[1] &= max(p_4[0] + L_4[1] - c_4, 0) = 1.133
\end{aligned}
$$

These steps will be repeated at $t = 2, 3, \cdots$ until the process converges. Finally, requester R_1 will send a workload of 0.25, R_2 will send a workload of 0.35, and R_3 will send a workload of 0.45, respectively, to the workers after the convergence.

VI. Convergence Discussion

In the above example, we can see that if the problem is convex, the iteration process will eventually converge. In this section, we are exploring various methods to accelerate the convergence process. The speed of the convergence is controlled by Step 6 in Algorithm PU, which uses the gradient method. Therefore, we can explore various methods to make the gradient method converge faster. We will try the popular AdaGrad [15], RMSprop [8], Momentum [12], and Adam [10] methods and embed them in Algorithm PU.

A. AdaGrad

This method improves the convergence process by considering the history of the squared gradient. The idea is translated into the following three steps to replace Step 6 in Algorithm PU. In formula (7), the variable $curr$ is the gradient value at the current x_i^*. The variable $hist_j$ records the sum of all the previous $curr^2$. In updating $p_j[t]$, γ is still the step size and ϵ is a very small number to prevent the denominator from becoming zero.

$$
\begin{aligned}
curr &= c_j - \sum_{i \in R(i)} x_i^*; \\
hist_j[t] &= hist_j[t-1] + curr^2; \\
p_j[t] &= max(p_j[t-1] - \gamma \frac{curr}{\sqrt{hist_j[t]} + \epsilon}, 0);
\end{aligned} \tag{7}
$$

B. RMSprop

The RMSprop method is similar to AdaGrad, but it adds weight $\beta \in [0, 1]$ to the history and $1 - \beta$ to the current gradient's square.

$$
\begin{aligned}
curr &= c_j - \sum_{i \in R(i)} x_i^*; \\
hist_j[t] &= \beta * hist_j[t-1] + (1-\beta) * curr^2; \\
p_j[t] &= max(p_j[t-1] - \gamma \frac{curr}{\sqrt{hist_j[t]} + \epsilon}, 0);
\end{aligned} \tag{8}
$$

C. Momentum

Gradient descent with momentum remembers the solution update ($v_j[t]$) at each iteration and determines the next update as a linear combination of the gradient and the previous update, with the weight β being in the range of $[0, 1]$. The steps are as follows:

$$
\begin{aligned}
curr &= c_j - \sum_{i \in R(i)} x_i^*; \\
v_j[t] &= \beta * v_j[t-1] + (1-\beta) * curr; \\
p_j[t] &= max(p_j[t-1] - \gamma * v_j[t], 0);
\end{aligned} \tag{9}
$$

D. Adam

The Adam method is a combination of the AdaGrad and Momentum methods. The steps are as follows: the parameters β_1 and β_2 are weights between 0 and 1.

$$
\begin{aligned}
curr &= c_j - \sum_{i \in R(i)} x_i^*; \\
v_j[t] &= \beta_1 * v_j[t-1] + (1-\beta_1) * curr; \\
hist_j[t] &= \beta_2 * hist_j[t-1] + (1-\beta_2) * curr^2; \\
p_j[t] &= max(p_j[t-1] - \gamma \frac{v_j[t]}{\sqrt{hist_j[t]} + \epsilon}, 0);
\end{aligned} \tag{10}
$$

VII. Simulations

In this section, we conduct simulations to compare the convergence processes of the various methods we discussed above using Matlab. The example we use is the one in Section V. We compare the following algorithms:

1) The Original Method (Org): Algorithms WLD and PU
2) The AdaGrad Method (AdaGrad): replacing Step 6 of PU by steps in (7)
3) The RMSprop Method (RMSprop): replacing Step 6 of PU by steps in (8)
4) The Moment Method (Moment): replacing Step 6 of PU by steps in (9)

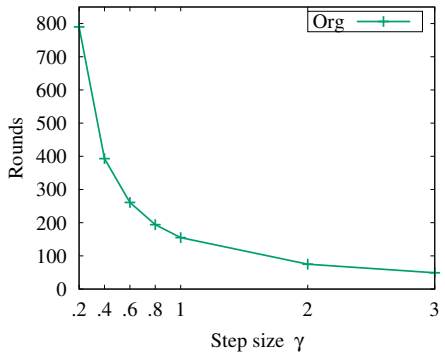


Fig. 5. The Original Method: γ and rounds to converge to the optimal solution

5) The Adam Method (Adam): replacing Step 6 of PU by steps in (10)

A. The Original Method (Org)

In Org, the step size is fixed. We tried γ values from 0.2 to 1 with a step of 0.2. In this γ range, the larger the step size, the faster the convergence speed. We also tried γ values of 2 and 3. The process converged even faster. However, when we set γ to 4 and above, the process would no longer converge to the optimal solution because the step size was too large. Fig. 5 shows the relationship between γ and the number of rounds for the process to converge.

B. The AdaGrad Method (AdaGrad)

In AdaGrad, first we set ϵ to a very small number, 1.4901×10^{-8}. Second, the step size γ is no longer a constant; with the increase of the rounds, it becomes smaller and smaller. We initialized γ from 0.02 to 0.1 with a step of 0.02. The results are shown in Fig. 6(a). We found that none of the γs in this range was able to converge to the exact optimal solution; instead, it oscillated in a very small area (from 0.2% to 2%) around the optimal solution. We added a small circle around a data point in the figure to represent that the process only converges to the neighborhood of the optimal solution. We will use this convention for all the figures below. In the γ range from 0.02 to 0.1, the larger the γ, the faster the process converged to the neighborhood of the optimal solution. Comparing with Org, AdaGrad got to the neighborhood faster.

C. The RMSprop Method (RMSprop)

The RMSprop method is similar to AdaGrad, but with an additional weight β to balance the history and the current squared gradient. We tried $\beta = 0.3, 0.5, 0.7$, respectively. We still set γ from 0.02 to 0.1 with a step of 0.02. The results are in Fig. 6(b). Like AdaGrad, none of the γs was able to converge to the exact optimal solution. The final results oscillated in the 0.6% to 5% neighborhood of the optimal solution. Smaller β values needed more rounds to converge than the larger ones. This means putting a higher weight on the current squared gradient rather than the history makes the process converge faster. Comparing with AdaGrad, for the same γ, the process could converge faster.

D. The Momentum Method (Momentum)

The Momentum method uses the β value to balance the history and the current gradient. We tried $\beta = 0.3, 0.5, 0.7$, respectively. We set γ from 1 to 6 with a step of 1. The results are in Fig. 7(a). This time, treating the history and the current gradient with the same weight resulted in most rounds converging, compared to treating the two differently. Also, when $\beta = 0.3$ and 0.5, the process could converge to the exact optimal solution. But when $\beta = 0.7$, the process could only converge to the 1% neighborhood of the exact optimal solution.

E. The Adam Method (Adam)

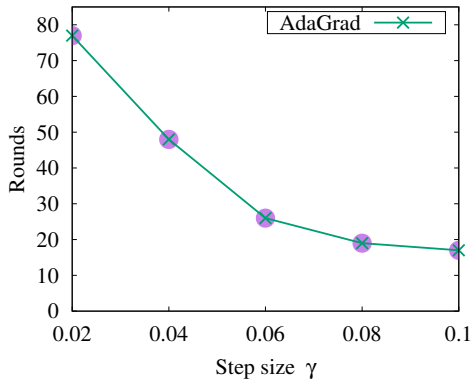
The Adam method is a combination of the AdaGrad and the Momentum methods, with two β variables. We tried $\beta_1 = \beta_2 = 0.3, 0.5, 0.7$, respectively, and set γ from 0.02 to 0.1 with a step of 0.02. The results are shown in Fig. 7(b). Again, none of the γs was able to converge to the exact optimal solution; the final results oscillated in the 0.2% to 3% neighborhood of the optimal solution. Interestingly, smaller β values converged in fewer rounds than the larger ones, unlike RMSprop.

F. Comparison Summary

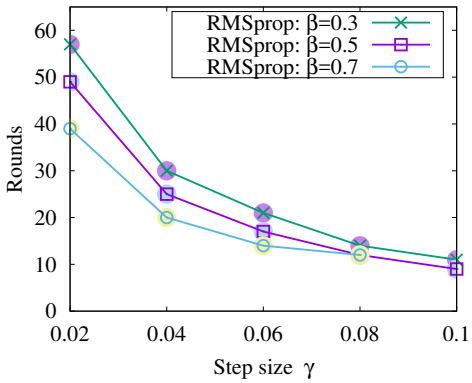
We summarize the simulation results as follows. Out of the five methods we compared, Org can converge to the exact optimal solution, while the others may only converge to the neighborhood of the optimal solution. The order of the convergence speed, from the fastest to the slowest, is: RMSprop, AdaGrad, Adam, Momentum, and Org. This means that adding the history and the current gradient to the update does improve the convergence speed in Org. It is also obvious that considering the current squared gradient (RMSprop, AdaGrad, Adam) performs better than just considering the current gradient (Momentum). In addition, putting weights to the history and the current squared gradient (RMSprop) outperforms the one without weights (AdaGrad). Finally, how much weight should be given to each factor to outperform does not have a consistent answer (RMSprop, Momentum, Adam).

VIII. Conclusion

In this paper, we have considered a crowdsourcing model that extends the current model in space and time. Based on our model, we have formulated an optimization problem from the perspective of the requesters that maximizes the total utility of all the requesters, subject to the constraint that the total workload given to a worker should not exceed his capability. We have provided a solution and designed distributed algorithms for the requesters and the workers to interact with each other in multiple rounds. We have also given a concrete example to explain the solution and the algorithms. We have then discussed the convergence speed of our algorithms using different methods. Finally, we have conducted simulations to compare these convergence methods and drawn conclusions.

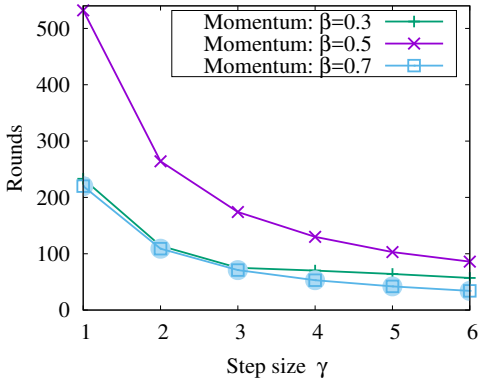


(a) AdaGrad: rounds to converge to the neighborhood of optimal

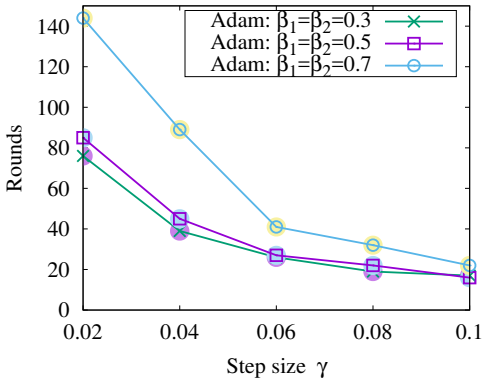


(b) RMSprop: rounds to converge to the neighborhood of optimal

Fig. 6. The AdaGrad and RMSprop Methods



(a) Momentum: rounds to converge to the neighborhood of optimal



(b) Adam: rounds to converge to the neighborhood of optimal

Fig. 7. The Momentum and Adam Methods

REFERENCES

[1] Amazon mechanical turk. http://mturk.com.

[2] Crowdsourcing. https://en.wikipedia.org/wiki/Crowdsourcing.

[3] Marginal utility. https://en.wikipedia.org/wiki/Marginal_utility.

[4] Supply and demand. https://en.wikipedia.org/wiki/Supply_and_demand.

[5] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[6] Z. Feng, Y. Zhu, Q. Zhang, L. M. Ni, and A. V. Vasilakos. TRAC: Truthful Auction for Location-Aware Collaboration Sensing in Mobile Crowdsourcing. In *IEEE INFOCOM*, 2014.

[7] Y. Han, Y. Zhu, and J. Yu. Utility-maximizing data collection in crowd sensing: An optimal scheduling approach. In *IEEE SECON*, 2015.

[8] G. Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2020.

[9] L. G. Jaimes, I. Vergara-Laurens, and M. A. Labrador. A location-based incentive mechanism for participatory sensing systems with budget constraints. In *IEEE PERCOM*, 2012.

[10] D. Kingma and J. Ba. Adam: A method for schochastic optimization. https://arxiv.org/abs/1412.6980, 2014.

[11] W. Liu, E. Wang, Y. Yang, and J. Wu. Worker selection towards data completion for online sparse crowdsensing. In *INFOCOM*, 2022.

[12] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, (12):145–151, 1999.

[13] G. Samore, J. Bates, and X. Chen. Improving satisfaction in crowdsourcing platforms. In *IEEE International Conference on Intelligent Data and Security*, 2021.

[14] M. Simaan and J. B. Cruz. On the stackelberg strategy in nonzero-sum games. *Journal of Optimization Theory and Applications*, (11):533–555, 1973.

[15] J. Duchi; E. Hazan; Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, (12):2121–2159, 2011.

[16] J. Sun and H. Ma. A behavior-based incentive mechanism for crowd sensing with budget constraints. In *IEEE ICC*, 2014.

[17] T. Tsujimori, N. Thepvilojanapong, Y. Ohta, Y. Zhao, and Y. Tobe. History-based incentive for crowd sensing. In *ACM IWWISS*, 2014.

[18] H. Wang, E. Wang, Y. Yang, J. Wu, and F. Dressler. Privacy-preserving online task assignment in spatial crowdsourcing: A graph-based approach. In *INFOCOM*, 2022.

[19] R. Wang, F. Zeng, L. Yao, and J. Wu. Game-Theoretic Algorithm Designs and Analysis for Interactions Among Contributors in Mobile Crowdsourcing With Word of Mouth. *IEEE Internet of Things Journal*, 7(9), 2020.

[20] Y. Xu, M. Xiao, J. Wu, S. Zhang, and G. Gao. Incentive mechanism for spatial crowdsourcing with unknown social-aware workers: A three-stage stackelberg game approach. *IEEE Transactions on Mobile Computing*, 2022.

[21] D. J. Yang, G. L. Xue, X. Fang, and J. Tang. Crowdsourcing to smartphones: incentive mechanism design for mobile phone sensing. In *ACM MOBICOM*, 2012.

[22] X. Yang, J. Zhang, J. Peng, and L. Lei. Incentive mechanism based on stackelberg game under reputation constraint for mobile crowdsensing. *International Journal of Distributed Sensor Networks*, 17(6), 2021.

[23] D. Zhao, X.-Y. Li, and H. Ma. How to crowdsource tasks truthfully without sacrificing utility: Online incentive mechanisms with budget constraint. In *IEEE INFOCOM*, 2014.