Online Team Formation with Outsourcing

Aris Anagnostopoulos
Sapienza University of Rome
aris@dis.uniroma1.it

Carlos Castillo
Eurecat
chato@acm.org

Adriano Fazzone
Sapienza University of Rome
fazzone@dis.uniroma1.it

Stefano Leonardi
Sapienza University of Rome
leonardi@dis.uniroma1.it

Evimaria Terzi
Boston University
evimaria@cs.bu.edu

ABSTRACT

Online labor market places (e.g., UpWork) allowed us to observe the practical benefits of *crowdsourcing*, the practice of outsourcing a task to a large number of workers. At the same time, there is clear evidence that tasks can benefit from expert collaboration. This observation has led to research and practices of *team formation*. In this paper, we propose a new model that incorporates elements of crowdsourcing and team formation. We call this model *team formation with outsourcing*.

In our model, tasks arrive in an *online fashion*; i.e., the number and the composition of the tasks is not known a-priori. At any point in time, there is a core team of hired workers. This team is dynamic: new members can be hired and existing members can be fired. Additionally, some parts of the arriving jobs can be outsourced and thus completed by non-team members. The key contribution of our paper is an efficient online cost-minimizing algorithm for hiring, firing and outsourcing. Moreover, using a primal-dual scheme, we are able to prove that our algorithm has logarithmic competitive-approximation ratio. Our experiments with data from three large online labor marketplaces demonstrate the efficiency and the efficacy of our algorithms in practice.

ACM Reference format:
Aris Anagnostopoulos, Carlos Castillo, Adriano Fazzone, Stefano Leonardi, and Evimaria Terzi. 2017. Online Team Formation with Outsourcing. In *Proceedings of 23rd SIGKDD Conference on Knowledge Discovery and Data Mining, Halifax, Nova Scotia - Canada, August 2017 (KDD'17)*, 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Self-employment is an increasing trend, for instance, between 10% and 20% of workers in OECD countries are self-employed [21]. This is due in part to business downsizing and employee dissatisfaction, as well as to the existence of online labor markets (e.g., Upwork, Amazon Mechanical Turk). This trend has enabled independent experts to work remotely in specialized tasks. Observations from online labor markets have allowed researchers and practitioners to explore the benefits of outsourcing and *crowdsourcing* [11, 12, 19, 25]. Essentially, crowdsourcing is driven by the assumption that problems can be decomposed into parts that can be addressed by distributed, independent workers.

A key problem in crowdsourcing environments is to split an input task into smaller separate sub-tasks that can be addressed by independent workers. Consistent with these assumptions, recent work in computer science has focused on developing advanced tools for crowdsourcing that break complex projects into parts and facilitate their recombination in a manner that maximizes output quality [14, 22]. Although there are clear benefits from crowdsourcing, recent work suggests that crowdsourcing results can be improved by introducing some degree of collaboration between workers [17, 23]. From the computer-science perspective, the idea of combining collaboration with crowdsourcing, led to multiple definitions of the *team formation* problem [1–3, 6, 7, 9, 13, 15, 16, 18, 24]. In that problem, the goal is to identify a group of workers that can perform together a given task or a sequence of tasks while incurring small *communication cost* between them. That is, team-formation problems focus on finding teams whose members can collaborate effectively.

In this paper, we propose a new model for team formation that is a middle ground between pure crowdsourcing[1] and pure team formation. We call this model *team formation with outsourcing*. In this model, tasks arrive *online* and at any point in time there is a team of hired workers. This team is dynamic: new members can be hired and existing members can be fired. Additionally, some parts of the incoming tasks can be completed by non-team members, who are outsourced. We consider the problem of finding an online cost-minimizing algorithm for hiring, firing and outsourcing. We call this general problem the TFO problem (Team Formation with Outsourcing).

To the best of our knowledge, we are are the first to consider this problem and study some of its variants. In order to solve it, we use the *primal-dual* scheme in order to design efficient online algorithms [4]. Our problem turns out to be an original combination of *online set cover* and *online ski-rental*, two of the most paradigmatic online problems. In fact TFO has elements that make it more complex. First, there are two options for covering the skills of a task: either by hired or by outsourced workers. Secondly, one has to decide when it is a good time for a hired worker to be fired. As in the previous problems, all these decisions need to be made in an online fashion.

[1] This definition refers to central cases of crowdsourcing where peer visibility and accessibility to peer contributions is low or non-existing [8], leaving aside cases such as *Wikis*, that are also considered crowdsourcing by some authors, but where contributions are not independent.

KDD'17, Halifax, Nova Scotia - Canada

DOI: 10.1145/nnnnnnn.nnnnnnn

Our analysis results in polynomial-time algorithms that have logarithmic competitive approximation ratios. This means that despite the fact that our algorithms work in an online fashion and they do not have any knowledge of the number and the composition of future tasks, we can guarantee that the cost they will incur will be –*at any point* – only a logarithmic factor worse than the cost incurred by an optimal algorithm that knows the set of requests a-priori. Our experimental results with real data from three large online labor marketplaces demonstrates the efficiency and the efficacy of our methods in practice.

2 PRELIMINARIES

In this section we describe our setting, our problem and also give some necessary background.

2.1 Notation and Setting

The high-level goal is to design algorithms that are able to complete tasks requiring some particular skills by teams of workers who possess the required skills. Let us now define these notions.

Skills: We consider a set S of skills with $|S| = m$. Skills can be any kind of qualification a worker can have or a task may require. For example, *video editing*, *technical writing*, *Python programming*, or even *acting* and *nursing*, may be examples of skills.

Tasks: We consider a set of T^* tasks (or jobs), $\mathcal{J} = \{J^t;\ t = 1, 2, \ldots, T^*\}$, which arrive one-by-one in a streaming fashion; J^t is the t-th task that arrives. Every task $J \in \mathcal{J}$ requires a set of skills from S, therefore, $J \subseteq S$. We use J^t to refer to both the task and the skills that it requires.

Workers: Throughout we assume that we have a set $\mathcal{W}$ of n workers: $\mathcal{W} = \{W^r;\ r = 1, \ldots, n\}$. Every worker r possesses a set of skills ($W^r \subseteq S$). Similarly to the tasks, we use W^r to characterize both the worker and his skills. Finally, we define P_ℓ to be the subset of workers possessing a given skill ℓ: $P_\ell = \{r;\ \ell \in W^r\}$.

We partition the set of available workers $\mathcal{W}$ into the set of workers that are *hired* at time t, denoted by $\mathcal{H}^t$, and the set of workers that are *not hired*, denoted by $\mathcal{F}^t$ (we sometimes refer to these workers as "freelancers"), so that $\mathcal{H}^t \cap \mathcal{F}^t = \emptyset$ and $\mathcal{W} = \mathcal{H}^t \cup \mathcal{F}^t$. The hired workers can be thought of as the workers that belong to the workforce of a company. On the other hand, the unhired workers or freelancers do not have any obligations to a specific company; any company can outsource tasks to them at some—usually high—cost.

Costs: Every worker W^r is associated with the following non-negative fees paid to him: (i) the *hiring fee* (C_r), (ii) the *outsourcing fee* λ_r and (iii) the worker's *salary* σ_r. The hiring fee is worker-specific and in practice it refers to expenses associated with hiring (or firing) a worker, such as signup bonuses (or severance payments). We assume that hiring fees are paid at the moment of hiring.

Workers in $\mathcal{F}^t$ are workers that are not hired when task J^t arrives. Therefore, these workers can be *outsourced* for J^t. For worker $W^r \in \mathcal{F}^t$ we use $\lambda_r \geq 0$ to denote the payment required by this worker when a task is outsourced to him. Note that λ_r depends on the worker but does not depend on the task. A running assumption will be that $\lambda_r < C_r$.

Once a worker r is hired, he is paid a recurring salary salary $\sigma_r \geq 0$, which recurs for every t that the worker is hired. A running assumption we make is that $\sigma_r < \lambda_r$; that is, once a worker is hired then the cost of his work is lower than the his outsourcing cost, due to reduced transaction costs – the very basis of why firms are created [5].

Remark: In order to avoid making the model overly complicated, we assume that the salary periods are defined by the arriving tasks; i.e., we have one task per salary period.

The above notation is summarized in Table 1.

Table 1: Notation

S	Set of skills, size m.
$\mathcal{J}$	Set of tasks, size T^*.
T	Number of tasks till current time.
J^t	The t-th task arriving.
	$J^t_\ell = 1$ if task t requires skill ℓ, 0 otherwise.
$\mathcal{W}$	Set of workers, size n.
	$W^r_\ell = 1$ if worker r possess skill ℓ, 0 otherwise.
P_ℓ	Subset of workers possessing skill ℓ.
C_r	Hiring fee, paid when worker r is hired.
λ_r	Outsourcing fee, paid every time r performs a task.
σ_r	Salary paid to a hired worker r.

Completion of tasks: Whenever task $J^t \subseteq S$ arrives an algorithm has to create a *team of workers* that covers it. We say that J^t can be *completed* or *covered* by a set (team) of workers $Q \subseteq \mathcal{W}$ if for every skill required by J^t, there exists at least one worker in Q that has this skill. Formally: $J^t \subseteq \cup_{W \in Q} W$. We assume that the problem instance is such that each input taks can be covered by some set of workers, that is, for each $J \in \mathcal{J}$ we have that $J \subseteq \cup_{W \in \mathcal{W}} W$.

2.2 Problem Definition

We now define the problem that we study:

Problem 1 (TFO). *There exists a set of skills S. We have a pool of workers $\mathcal{W}$, where each worker W^r is characterized by (1) the subset of skills $W^r \subseteq S$, (2) a hiring cost $C_r \in \mathbb{R}_{\geq 0}$, (3) a salary cost $\sigma_r \in \mathbb{R}_{\geq 0}$, and (4) an outsourcing cost $\lambda_r \in \mathbb{R}_{\geq 0}$. Given a set of tasks $\mathcal{J} = \{J^1, J^2, \ldots, J^{T^*}\}$, with $J^t \subseteq S$, which arrive in a streaming fashion, the goal is to design an algorithm that when task J^t arrives decides which workers to hire (paying cost $C_r + \sigma_r$), keep hired (paying cost σ_r), and outsource (paying cost λ_r), such that all the tasks are covered by the workers that are hired or outsourced and the total cost paid over all the tasks is minimized.*

Online-algorithm design goals: Note that the TFO—as defined above—is an *online problem.* That is, the total number of tasks T^* as well as the skill-composition of the tasks that will arrive are unknown. Therefore, our goal in terms of algorithm design is to provide a guarantee that for *any* number of tasks $T = 1, 2, \ldots, T$ the total cost of our algorithm is only a small factor greater than the total cost of the *optimal* algorithm that knows the number and the skill-composition of tasks apriori. Note that the above statement should be true for *any* T as the input stream may terminate after any task. This factor is called the *competitive ratio* of the algorithm. Formally the competitive ratio of an algorithm is defined as:

$$\max_{\mathcal{J}} \frac{\text{Cost of an alg. that does not know } \mathcal{J} \text{ in advance}}{\text{Cost of an optimal alg. that knows } \mathcal{J} \text{ in advance}}.$$

We solve the TFO problem in Section 4. Because the algorithm and the analysis are complicated, we introduce them gradually by first solving a simplified version of TFO, which we describe and solve in Section 3.

2.3 Background Problems

Two very simple cases of our problem are the *set cover* and the *ski-rental* problems. We present them here as we need to refer to them in subsequent sections.

SetCover: **The single-task, multiple-skill case.** The set-cover problem is an instance of our problem when there is a single task $J \subseteq S$ and for each worker W^r, $C_r = \infty$. Then, as soon as the task J arrives the algorithm nees to cover all skills in J by selecting a set of workers $Q \subseteq \mathcal{W}$ such that Q covers J and $\sum_{r \in Q} \lambda_r$ is minimized. In this case, our problem can be solved using the greedy algorithm for the set-cover problem (see [26, Chapter 2]). :

SkiRental: **the single-skill, single-worker case.** The ski-rental problem is an instance of our problem when the sequence of tasks $\mathcal{J}$ consists of a repetition of the same single-skill task J and the workforce $\mathcal{W}$ consists of a single worker W^r that posesses the same one skill, and has $\sigma_r = 0$ and some C_r, λ_r. In this ski-rental version of our problem [20], the question is the following: without knowledge of the total number of tasks that will arrive, when should worker W be hired so that the total cost paid to him in outsourcing plus hiring fees is minimized?

A folklore algorithm for solving this problem is the following: for every instance of J^t that arrives outsource J^t to worker W^r as long as: $\sum_{t'=1}^{t} \lambda_r < C_r$. Then, hire the worker when $\sum_{t'=1}^{t} \lambda_r \geq C_r$. The above algorithm, achieves a *competitive ratio* of 2.

3 THE LUMPSUM PROBLEM

In this section we define and solve a simplified version of the TFO problem, where for every worker W^r, $\sigma_r = 0$. We call this problem the LumpSum problem. In the LumpSum problem, a hired worker W^r is only paid hiring cost C_r the moment the worker is hired. Thus, we assume thus cost covers all future work done by the worker. Instead, when a worker W^r is outsourced, he is paid λ_r every time she performs a task.

3.1 The `LumpSum-Heuristic` algorithm

A natural algorithm for solving the LumpSum problem is to combine ideas from SetCover and SkiRental. Such an algorithm, works as follows: first, it starts with no worker being hired. Additionally, each worker W^r is associated with variable δ_r initially set to 0.

For any $T \in \{1, \ldots T^*\}$, when task J^T arrives the algorithm proceeds as follows: first, it identifies $J^T_{\mathcal{F}}$ to be the set of skills of J^T that cannot be covered by already-hired workers. Then, it covers the skills in $J^T_{\mathcal{F}}$ using the greedy algorithm for set cover. This way it finds $Q^T \subseteq \mathcal{W}$ such that $\sum_{W^r \in Q} \lambda_r$ is minimized. Finally, for each worker $W^r \in Q^T$, it sets $\delta_r = \delta_r + \lambda_r$. Worker W^r is hired if $\delta_r \geq C_r$. Clearly, since there are no salaries there is no motivation to fire a worker once he is hired.

The above algorithm, which we call `LumpSum-Heuristic`, is a combination of existing algorithms for SetCover and SkiRental (see Section 2.3). The running time of this algorithm per task J is dominated by running the greedy set-cover algorithm and is thus $O(Rn|J|)$, where R is the number of iterations of greedy and thus $R = \min\{|J|, n\}$.

Although our experiments demonstrate that `LumpSum-Heuristic` performs quite well in practice, one can show that its competitive ratio can be arbitrarily bad. Consider an example where $\mathcal{W} = \{W^1, W^2\}$ where both workers have the same skill $W^1 = W^2 = \{\ell\}$. Further assume that $\lambda_1 = 10$, $\lambda_2 = 10 + \epsilon$ and $C_1 = M$, $C_2 = 11$, where M is a really large value. For a sequence of tasks $J^1 = J^2 = \ldots = J^{T^*} = \{\ell\}$, it is clear that `LumpSum-Heuristic` will always outsource to W^1 until he is hired and will incur worse-case cost $2M$ while the optimal algorithm pays 11.

3.2 A primal-dual algorithm

In order to avoid worst-cases cases like the above we need to design an algorithm with bounded competitive ratio. For this we deploy a primal-dual scheme, which drives our algorithm design.

The linear program: The first step of the primal–dual approach, is to define an integer formulation for the problem. We assume that the current task is the Tth task and we use the following variables:

- $x_r = 1$ if worker W^r is hired when task J^T arrives; otherwise and $x_r = 0$.
- $f_{rt} = 1$ if worker W^r is outsourced for performing task J^t; otherwise and $f_{rt} = 0$.

Using this notation, the LumpSum can be expressed as follows:

Linear program for LumpSum:

$$\min \sum_{r=1}^{n} \left(C_r x_r + \lambda_r \sum_{t=1}^{T} f_{rt} \right)$$

subject to: $\forall t = 1, \ldots, T, \ell \in J^t$:

$$\sum_{W^r \in P_\ell} (x_r + f_{rt}) \quad \geq \quad 1 \tag{1}$$

$\forall t = 1, \ldots, T, r = 1, \ldots, n$:

$$x_r, f_{rt} \quad \geq \quad 0$$

The above, in addition to the integrality constraint $x_r, f_{rt} \in \mathbb{N}$ form the integer program from LumpSum.

In the above formulation, the objective function sums over all workers the hiring costs (paid if the corresponding worker has been hired by time t) and the outsourcing cost for the tasks for which the worker has been outsourced. This is the total cost of the solution until the current task J^T.

Note that in this formulation of the problem there is no motivation for a worker that is hired to be fired. Therefore, once x_r is set to 1, it does not change its value to become 0 again.

The first constraint is the covering constraint: it simply enforces that for every skill required for each task, there exists a hired or outsourced worker that has this skill. This guarantees that the team

selected for each task J^t covers all the required skills. The nonnegativity and the integrality constraints, ensure that the solutions that we obtain from the integer-program formulation can be transformed to a solution to our problem: eventually, every variable will take the value 0 or 1.[2]

To apply the online primal–dual approach, we first consider the linear relaxation of the integer program, which simply drops the integrality constraint $x_r, f_{rt} \in \mathbb{N}$. A solution to this linear program (LP) gives a solution in which each variable takes values in $[0, 1]$. Given this LP, we can write its dual as follows:

The dual version of the above LP is the following:

The dual of the linear program for LumpSum:

$$\max \sum_{t=1}^{T} \sum_{\ell \in J^t} u_{\ell t}$$

subject to: $\forall r = 1, \ldots, n$:

$$\sum_{t=1}^{T} \sum_{\ell \in J^t \cap W^r} u_{\ell t} \quad \leq \quad C_r \tag{2}$$

$\forall t = 1, \ldots, T, r = 1, \ldots, n$:

$$\sum_{\ell \in J^t \cap W^r} u_{\ell t} \quad \leq \quad \lambda_r \tag{3}$$

$\forall t = 1, \ldots, T, \ell \in J^t$:

$$u_{\ell t} \quad \geq \quad 0,$$

Note that for every time point $t \in \{1, \ldots, T\}$. we have such a pair of primal–dual formulations for each time point $t \in \{1, \ldots, T\}$. We are now going to use these two formulations for designing and analyzing our algorithm.

The LumpSum algorithm: In this paragraph we present the LumpSum algorithm, which is designed and analyzed using the primal and the dual linear programs.

For the presentation of the algorithm, we assume that task J^T for $T \in \{1, \ldots, T^*\}$ has just arrived and the algorithm must act before task $T + 1$ arrives (or the stream finishes if $T = T^*$). Note all the variables used in our algorithm are initialized to 0 before the arrival of the first task. Thus, when task J^T arrives the algorithm proceeds as follows:

(1) Let $\mathcal{F}^T$ and $\mathcal{H}^T$ represent the workers that are not hired and hired, respectively, at the time that J^T arrives. Clearly, when the first task arrives ($T = 1$), then $\mathcal{F}^T = \mathcal{W}$ and $\mathcal{H}^T = \emptyset$. For $T > 1$, the values of $\mathcal{H}^T$ and $\mathcal{F}^T$ are updated in the last step (step 11) of the previous round.

(2) Let $J^T_{\mathcal{H}} = J^T \cap \cup_{W^r \in \mathcal{H}^T} W^r$ be the skills from J^T that are covered by already-hired workers and $J^T_{\mathcal{F}} = J^T \setminus J^T_{\mathcal{H}}$.

(3) For every skill $\ell \in J^T_{\mathcal{F}}$ let $P^{\mathcal{F}}_{\ell} = P_\ell \cap \mathcal{F}^T$ be the set of workers in $\mathcal{F}^T$ such that every worker in $P^{\mathcal{F}}_{\ell}$ has skill ℓ. Also let

$$P^{\mathcal{F}}_{T} = \cup_{\ell \in J^T_{\mathcal{F}}} P^{\mathcal{F}}_{\ell}$$

be the set of unhired workers that possesses at least one skill required.

(4) **foreach** $W^r \in P^{\mathcal{F}}_{\ell}$: Set $\tilde{x}'_r \leftarrow \tilde{x}_r$

(5) **for each** skill $\ell \in J^T_{\mathcal{F}}$:

while $\sum_{W^r \in P_\ell} \left(\tilde{x}_r + \tilde{f}_{rT}\right) < 1$:

$u_{\ell t} \leftarrow u_{\ell t} + 1$

for each $W^r \in P_\ell$: $\tilde{x}_r \leftarrow \tilde{x}_r \left(1 + \frac{1}{C_r}\right) + \frac{1}{nC_r}$

for each $W^r \in P_\ell$: $\tilde{f}_{rT} \leftarrow \tilde{f}_{rT} \left(1 + \frac{1}{\lambda_r}\right) + \frac{1}{n\lambda_r}$

(6) **for each** $W^r \in P^{\mathcal{F}}_{\ell}$: Set $\Delta\tilde{x}_r \leftarrow \tilde{x}_r - \tilde{x}'_r$

(7) Set $\mathcal{H}' \leftarrow \emptyset$.

(8) **repeat** ρ times:

for each $W^r \in P^{\mathcal{F}}_{T}$

with probability $\Delta\tilde{x}_r$:

hire worker W^r (set $x_r \leftarrow 1, \mathcal{H}' \leftarrow \mathcal{H}' \cup \{r\}$)

with probability $\tilde{f}_{rT}$:

outsource worker W^r (set $f_{rT} \leftarrow 1$)

(9) **for each** skill $\ell \in J^T_{\mathcal{F}}$:

if skill ℓ is not covered:

hire worker $W^r \in P^{\mathcal{F}}_{\ell}$ with minimum cost C_r

(set $x_r \leftarrow 1, \mathcal{H}' \leftarrow \mathcal{H}' \cup \{r\}$)

(10) $\mathcal{H}^{T+1} \leftarrow \mathcal{H}^T \cup \mathcal{H}'$, $\mathcal{F}^{T+1} \leftarrow \mathcal{W} \setminus \mathcal{H}^{T+1}$.

For $T = 1$, the LumpSum starts with no worker being hired. Intuitively, as tasks arrive, the algorithm tries to gauge two things: (a) the usefulness of every worker for the task at hand J^T and (b) the overall usefulness of the worker for tasks $J^1, \ldots, J^T$. This is done in step (5) and via variables $\tilde{f}_{rT}$ and $\tilde{x}_r$ respectively. The more useful the worker proves for this task and over time, the larger the values of these variables. Subsequently, in step (8) every worker is outsourced or hired based on the increase in the values of $\tilde{f}_{rT}$ and $\tilde{x}_r$ observed in step (5). [3] Finally, for every skill that remains uncovered after step (8) (which is randomized), LumpSum hires worker W^r with the minimum C_r that covers the skill. Note that the increase of the variables $u_{\ell T}$ in step (5) is not required for solving the LumpSum, but it is used in our analysis and thus we leave it in the description above.

Our analysis requires to set the value of ρ in step (8) to

$$\rho = \ln m + \ln C^*,$$

where $C^* = \max_{W^r \in \mathcal{W}} C_r$.

Note that although one may think that an additive update of variables in step (5) would seem more natural, such an update would introduce an $O(m)$ factor in the competitive ratio. On the other hand, the multiplicative update we adopt enables us to state Theorem 3.1 below.

Analysis: We have the following result for LumpSum.

Theorem 3.1. LumpSum *is an* $O(\log n (\log m + \log C^*))$*- competitive algorithm for the* LumpSum *problem, where* $C^* = \max_{W^r \in \mathcal{W}} C_r$.

Our analysis and proof follows the approach in [4] and it consists in three steps: (1) First, we show the feasibility of the primal solution after the arrival of J^T and the assignment of values to the variables according to LumpSum. (2) Then, we show that after the completion of J^T, the expected value of the obective function of the primal LP (which we denote by P^T) is at most $3(\ln m + \ln C^*)$ times the expected value of the objective function of the dual LP (denoted by

[2] A solution in which some variables take values greater than 1, can be transformed to another feasible solution with lower cost by setting these variables to 1.

[3] Note that the larger the initial $\tilde{x}'_r$, the larger the increase $\Delta\tilde{x}$ will be observed in step (5).

D^T). In particular we show that $\mathbf{E}\left[P^T\right] \leq 3(\ln m + \ln C^*)\mathbf{E}\left[D^T\right] + 1$. This expectation is taken over the random choices of the algorithm when rounding the fractional solution. (3) Finally, we show that the dual solution is almost feasible, in particular, we show that each dual constraint is violated by a factor of at most $2 \log n$.

Then, as shown in [4], it follows that the algorithm maintains a feasible solution at an expected cost that is at most a factor $[3(\ln m + \ln C^*) \cdot 2 \log n$ higher than the optimal offline solution. Due to space constraints, we defer the analysis in the full version of the paper. [4]

Running time: The running time of `LumpSum` per task is dominated by the execution of step (5). Using binary search, one can determine in $O(\log C^*)$ steps the minimum increase of u_{lt} that makes false the condition of the while loop for at least one uncovered skill ℓ. Therefore, the running time of this step is $O\left(\left|J^T\right| (\log C^* + n)\right)$.

4 THE TFO PROBLEM

In this section, we provide an algorithm for the general version of the TFO problem (Problem 1). The only difference from the setting we considered in the LumpSum problem is that now a hired worker W^r is paid salary $\sigma_r \geq 0$. This complicates the problem significantly as it may now be beneficial for hired workers to be fired.

The linear program: Below we provide the linear program (LP) for TFO. A key notion introduced in this new LP is the notion of intervals, which ar required to model the fact that workers can be hired, then fired and potentially hired again. For worker W^r, we introduce the intervals $\mathcal{I}_r$. These are the time intervals for which W^r was hired.[5]. Therefore, the new LP uses the following variables:

- $x(r, I)$ such that for $I \in \mathcal{I}_r$, $x(r, I) = 1$ if worker W^r is hired during the entire interval I; otherwise $x(r, I) = 0$.
- g_{rt} such that $g_{rt} = 1$ if worker W^r receives salary when task J^t arrives; otherwise $g_{rt} = 0$. Note that W^r receives salary if and only if r is hired.
- f_{rt} such that $f_{rt} = 1$ if worker W^r is outsourced for performing J^t; otherwise $f_{rt} = 0$.

Hence, we formulate the following LP for TFO:

Linear program for TFO:

$$\min \sum_{r=1}^{n} \left[\sum_{I \in \mathcal{I}_r} C_r x(r, I) + \sum_{t=1}^{T} \lambda_r f_{rt} + \sum_{t=1}^{T} \sigma_r g_{rt} \right]$$

subject to

$\forall t = 1 \ldots T, \ell \in J^t$:

$$\sum_{W^r \in P_\ell} \left(f_{rt} + \sum_{I \in \mathcal{I}_r : t \in I} x(r, I) \right) \geq 1. \tag{4}$$

$\forall t = 1 \ldots T, r = 1 \ldots n$:

$$\sum_{I \in \mathcal{I}_r : t \in I} x(r, I) \leq g_{rt} \tag{5}$$

$\forall t = 1 \ldots T, r = 1 \ldots n, I \in \mathcal{I}_r$:

$$x(r, I), f_{rt}, g_{rt} \geq 0$$

[4] An appendix can be found at: cs-people.bu.edu/evimaria/supplementary.pdf

[5] As before time units are defined by the arrivals of tasks.

It turns out that it is hard to design an approximation algorithm using directly this program, mostly because it is hard to keep track of the costs being paid for every worker when the intervals of him being hired, outsourced and idle are of variable length. Therefore, we resort to a different approach: We consider an *alternative problem*, which we call Alt-TFO. In Alt-TFO, we restrict our solution to allow each worker to be hired for intervals of fixed (worker-specific) lengths. Not only is Alt-TFO easier to solve, every solution to Alt-TFO is also a feasible solution to TFO.

Overall strategy: Therefore, our overall strategy is the following: (i) First, we define the Alt-TFO problem. (ii) Then, we design an algorithm for this problem and prove that this algorithm has good competitive ratio. (ii) Finally, we prove that any solution of the TFO problem can be transformed to a feasible solution of the Alt-TFO problem that is only a factor of at most 3 times higher. We proceed with these three steps in the next three sections.

4.1 The Alt-TFO problem

The only difference between Alt-TFO and TFO is that we restrict the solutions of the former to have a specific structure; whenever worker W^r is hired he is then fired after $\eta_r \triangleq \lceil C_r/\sigma_r \rceil$ time units—independently of whether he is used or not in tasks within these η_r time units.

In this case, every worker W^r is associated with a new hiring cost $\widehat{C}_r$, which is the summation of his original hiring cost C_r plus the salaries paid to him for the η_r time units he is hired. Obseve that $C_r + \eta_r \cdot \sigma_r \leq (2 + \sigma_r/C_r) \cdot C_r \leq 3C_r$. Therefore, throughout we will use: $\widehat{C}_r \triangleq 3 \cdot C_r$.

We can now write the LP for Alt-TFO. For each worker W^r, we denote by $\mathcal{I}_r$ the set of intervals for which W^r was hired. Recall that all these intervals are of fixed length η_r. The generic such interval that starts at time t is also denoted by I_r^t. Worker W^r has $x(r, I) = 1$ if and only if the worker is hired during the *entire* interval I. Hence, the LP is the following:

Linear program for Alt-TFO:

$$\min \sum_{r=1}^{n} \left[\sum_{I \in \mathcal{I}_r} \widehat{C}_r x(r, I) + \sum_{t=1}^{T} \lambda_r f_{rt} \right]$$

subject to

$\forall t = 1 \ldots T, \ell \in J^t$:

$$\sum_{W^r \in P_\ell} \left(f_{rt} + \sum_{I \in \mathcal{I}_r : t \in I} x(r, I) \right) \geq 1. \tag{6}$$

$\forall t = 1 \ldots T, r = 1 \ldots n, I \in \mathcal{I}_r$:

$$x(r, I), f_{rt} \geq 0$$

Note that this LP is very similar to the LP we showed for the LumpSum problem.

4.2 Solving the Alt-TFO problem

In this section we design and analyze an algorithm for the Alt-TFO problem. The striking similarity between the LPs for Alt-TFO and LumpSum (Section 3) translates into a similarity in the algorithms (and their analysis) of the two problems.

The `Alt-TFO` algorithm: The `Alt-TFO` algorithm is a primal-dual algorithm designed using the LP for Alt-TFO and its dual (which we omit due to space constraints and because it is very similar to the dual for LumpSum). `Alt-TFO` has the exact same 11 steps as the `LumpSum`. The only differences are the following:

- In step (1) `Alt-TFO` additionally removes workers whose hiring interval finished in the previous step and thus sets:

$$\mathcal{F}' \leftarrow \{W^r \in \mathcal{W};\ x(r, I_r^{T-\eta_r}) = 1\}$$
$$\mathcal{H}^T \leftarrow \mathcal{H}^T \setminus \mathcal{F}', \quad \mathcal{F}^T \leftarrow \mathcal{W} \setminus \mathcal{H}^T$$
$$\textbf{for each } W^r \in \mathcal{F}': \quad \text{set } \tilde{x}_r \leftarrow 0$$

- In step (5), the update rule for varialbe $\tilde{x}_r$ uses $\widehat{C}_r$ instead of C_r and it thus becomes:

$$\textbf{for each } W^r \in P_\ell: \quad \tilde{x}_r \leftarrow \tilde{x}_r \left(1 + \frac{1}{\widehat{C}_r}\right) + \frac{1}{n\widehat{C}_r}$$

- In steps (8) and (9) variables $x(r, I_r^T)$, instead of x_r, are set to 1.
- Due to the modified step (1), step (10) becomes simply: $\mathcal{H}^{T+1} = \mathcal{H}^T \cup \mathcal{H}'$.

Our analysis requires to set

$$\rho = \ln m + \ln \widehat{C}^* + \ln T^*,$$

where $\widehat{C}^* = \max_{W^r \in \mathcal{W}} \widehat{C}_r$.

Analysis of `Alt-TFO`: Using methodology very similar to the one we used for analyzing `LumpSum`, we prove the following result for `Alt-TFO`.

Theorem 4.1. `Alt-TFO` *is an* $O\left((\log m + \log C^* + \log T^*)\log(n)\right)$*-competitive algorithm for the Alt-TFO problem.*

The proof of this theorem is very similar to the proof of Theorem 3.1 and due to space constraints we omit it from this version. [6]

4.3 Solving TFO using Alt-TFO

Note that any solution output by `Alt-TFO` can be transofmed into a feasible solution to the original TFO problem by setting $g_{rt} = 1$ for each $r, t \in I$ for which $x(r, I) = 1$, and $g_{rt} = 0$ otherwise. We call the algorithm that runs `Alt-TFO` and subsequently does this transformation a its final step, the `TFO` algorithm.

Now the question is whether `TFO` has a bounded competitive factor for the TFO problem. We answer this question affirmatively by showing that any solution for the TFO problem can be turned into a feasible solution to of Alt-TFO at the expense of a small loss in the approximation factor.

More specifically, we can prove the following:

Lemma 4.2. *If* P^{Alt} *is the cost of any feasible solution to the Alt-TFO and* P^{Tfo} *the cost of the same solution when transformed to a solution of TFO using the last step of* `TFO`*, then we have that:* $P^{Alt} \leq 3P^{Tfo}$.

The proof of this lemma is ommitted due to space constraints.

Combining Theorem 4.1 and Lemma 4.2 we obtain the following:

Theorem 4.3. `TFO` *is an* $O\left((\log m + \log C^* + \log T^*)\log(n)\right)$*- competitive algorithm for the TFO problem.*

[6] An appendix with the proofs can be found at: cs-people.bu.edu/evimaria/supplementary.pdf

Table 2: Characteristics of the three datasets used in our experiments. Numbers in italics correspond to the semi-synthetic workload generated for the Upwork dataset (D3), as explained in Section 5.1.

Dataset	D1 Freelancer	D2 Guru	D3 Upwork
Skills (m)	175	1,639	2,335
Workers (n)	1,211	6,119	18,000
Tasks (T)	992	3,194	*50,000*
... distinct	600	2,939	*50,000*
... avg. similarity (Jaccard)	0.045	0.018	*0.095*
Skills/worker			
.. average	1.45	13.07	6.29
.. median	1	10	6
Skills/task			
.. average	2.86	5.24	*41.88*
.. median	3	4	*41*

5 EXPERIMENTS

We perform extensive experiments on real data from online marketplaces for freelance work, as well as on semi-synthetic data that resembles it. Section 5.1 introduces our datasets, Section 5.2 the baselines, and Section 5.3 presents the experimental results.

5.1 Datasets

In this section we introduce our datasets of freelancers, the workloads that we use (both real and semi-synthetic), and our choice of cost parameters for experimentation.

Input data: We use datasets obtained from three large online marketplaces for outsourcing: Freelancer, Guru, and Upwork.[7] All three are in the top-30 of traffic in their category ("consulting marketplaces") according to data from Alexa;[8] indeed, Freelancer and Guru are respectively number 1 and number 3. In all these marketplaces, graphic design and web development are popular categories for freelance work, but they also include many other types of tasks that can be done remotely, ranging from data entry to accounting, virtual assistants, and legal consulting. In the following, we identify these three datasets as D1, D2, and D3; their characteristics are summarized on Table 2.

The input data we obtained contains anonymized profiles for people registered as freelancers in these marketplaces. This includes their self-declared sets of skills, as well as the average rate they charge for their services. We note there is a large variation in the number of skills and the number of skills per worker across datasets. Data was cleaned to remove skills that were not possessed by any worker, or skills that were never required by any task. The numbers in the table refer to the clean datasets.

Workloads: For Freelancer (D1) and Guru (D2) we also obtained a sample of tasks commissioned by buyers in the marketplace, they

[7] The authors are not associated with any of these services: https://www.freelancer.com/, https://www.guru.com/, https://www.upwork.com/, accessed Feb. 2017.
[8] http://www.alexa.com/topsites/category/Top/Business/Business_Services/Consulting/Marketplaces, accessed Feb. 2017.

are included as tasks on Table 2. In order to be able to generate an arbitrarily large workload, for these two datasets we sampled with replacement from these tasks when a larger number of tasks was required to determine the long-term behavior of some method.

In the case of Upwork, we generated a synthetic workload. The workload, denoted by D3 in Table 2, follows a method for data generation used by previous work [3]. It corresponds to setting aside a small number of workers (10% in our case), and then repeatedly sampling subsets of these workers in order to create tasks, by interpreting the union of the skills of the workers in the subset as the requirements of the task. These workers do not participate in the pool, i.e., cannot be selected to satisfy the task. Given the large number of different skills in the data (over 2,000), in general the created tasks tend to have disjoint sets of required skills.

Cost parameters: We have data about the rates charged by workers in each marketplace, which we directly interpret as their outsourcing costs λ_r. We have, however, no data regarding hiring costs or salary costs. Hence, we need to produce realistic estimates for these costs.

For hiring costs, which are characterized by $C_r > \lambda_r$, we set $C_r = \alpha_r \lambda_r$, in which α_r is drawn at random from a probability distribution. We use two probability distributions that reflect possible scenarios for α_r: high hiring cost, and low hiring cost. These scenarios correspond to interpreting the outsourcing cost as the cost of a work that would take one hour to perform, and setting the hiring cost as the equivalent of (i) between one and two days of work, i.e., 8 to 16 hours; and (ii) between one and two weeks of work, i.e., 40 to 80 hours. We then draw α_r from a uniform distribution in the range determined by the scenario (e.g., between 8 and 16 in the low-hiring-cost scenario). Due to space limitations we present detailed results for the low hiring cost case; in the long term, results with the high hiring cost are similar in terms of the relative performance of different methods, except that the number of tasks required before hiring a worker is larger.

For salary costs, which are characterized by $0 < \sigma_r < \lambda_r$, we considered a scenario where salary per time unit is 10% of the outsource cost.

5.2 Baselines and Implementation

We consider two baselines: `Always-Hire` and `Always-Outsource`. For every task `Always-Hire` finds the minimum cost set of workers that are required to cover un-covered skills and hires them. `Always-Outsource` never hires any worker and always outsources to workers that cover the required skills for the task, via solving a weighted set cover. In both cases, every incoming task is fulfilled. We also plot the performance of `LumpSum-Heuristic` and `TFO-Heuristic`, which is the naive variant for the Alt-TFO problem and is identical to `LumpSum-Heuristic`, except that it uses for every worker W^r the value of $\widehat{C}_r$ instead of C_r and it always fires every hired worker W^r every η_r intervals.

The implementation of the methods, done in Java, is a relatively straightforward mapping of the algorithm to simple counters. It takes about 5 to 8 seconds on average to process 10K incoming tasks using commodity hardware. We recall that although our formulation is a linear program, the method we propose does not involve solving this linear program with a generic linear program solver, but instead using the specific primal-dual method we have described and analyzed. All the code is available upon request.

5.3 Results

The main quantity of interest we measure is the total cost up to time t, this is, the total amount paid in outsourcing costs, hiring costs, and salaries by the time the t-th task arrives. All experiments in this section are averages of 100 runs with different permutations of the incoming tasks.

Experiments for LumpSum: Figure 1 summarizes our results for LumpSum. We can observe that across the different datasets and hiring cost scenarios, the behavior is the same. `Always-Outsource` has cost proportional to the number of tasks. The `Always-Hire` strategy in the long run has a smaller cost than `LumpSum` by a factor close to 2 in all the scenarios we tested. However, for short task sequences our algorithm has lower cost, sometimes by an order of magnitude. This matches the analysis and the intuitions derived from the SkiRental problem. Overall, in cases where the hiring cost is larger, the advantage for our strategy is sustained for a longer time period, i.e., for a larger number of tasks.

Experiments for TFO: In the implementation of `TFO`, we made some choices to reduce the cost of the final solution. We adopted a lazy approach for hiring and outsourcing workers and we also used the greedy set cover algorithm to reduce the cost required to satisfy the input task. For this, we collected all workers to hire and to outsource in two different sets. At the end of the rounding process (step (8) of the algorithm), we ran the greedy set cover algorithm to cover all skills of the input task, using workers in these two sets together with not hired workers not contained in these sets. The generated cover is used to select which worker in the hiring set to hire, which worker in the outsourcing set to outsource, and which not hired workers not in these sets to hire, in order to totally cover the input task (this last hiring part replaces step 10).

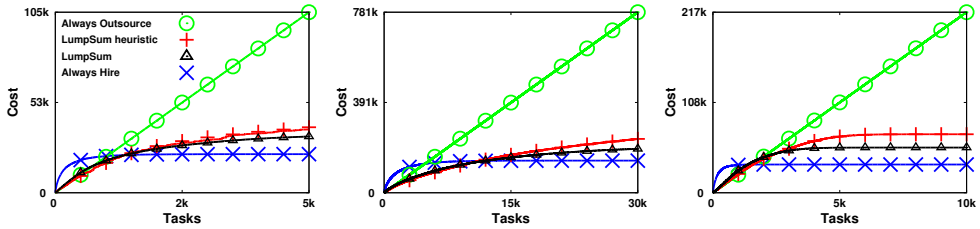
Results: Figure 2 summarizes the results. `Always-Hire` has the worst performance. This is expected as its cost is dominated by salary costs that accumulate over time. `TFO-Heuristic` is able to achieve a performance similar to `Always-Outsource`. In fact, `TFO-Heuristic` exhibits better performance than the theoretically justified `TFO` algorithm.

Experiments with high task locality: We note that in general the simple `Always-Outsource` strategy performs the best across all datasets. This is because of the diversity of workers and tasks in each platform, which does not make it economical to hire anyone. Instead, in this section we produce a stream of tasks for D1 and D2 in which the tasks change *locally*, i.e., a task is more likely to be similar to the next task than if we were sampling at random.

In order to achieve this, we run the following procedure:

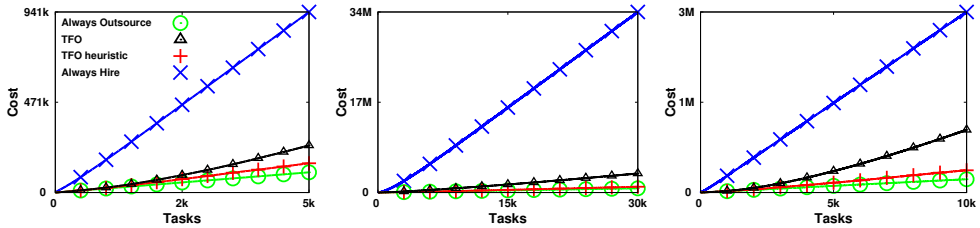
(1) Pick a random task as a pivot.
(2) With probability $1-p$, pick the next task within those that are similar enough to the pivot, i.e., those whose Jaccard similarity with the pivot is larger than a threshold θ.
(3) With probability p, pick another random task as a pivot.

We considered a pivot as eligible only if it had at least one other task that was similar enough to it. We experimented with various values of p and θ. Figure 3 shows results for $\theta = 0.5$, $p = 0.1, 0.01$. In general, we observe that with a smaller value of p, which yields



(a) D1. Freelancer (b) D2. Guru (c) D3. UpWork

Figure 1: Performance of primal-dual algorithm for LumpSum, compared to the `Always-Hire`, `Always-Outsource` and `LumpSum-Heuristic` baselines, averaged over 100 permutation of the input stream of tasks. The x-axis represents the task number, while the y-axis represents the total cost paid. For each worker, the considered hiring cost is at least eight and at most sixteen times the outsourcing cost: $8\lambda_r \leq C_r \leq 16\lambda_r$.



(a) D1. Freelancer (b) D2. Guru (c) D3. UpWork

Figure 2: Performance of primal-dual algorithm for TFO, compared to the `Always-Hire`, `Always-Outsource` and `TFO-Heuristic` baselines, averaged over 100 permutation of the input stream of tasks. The x-axis represents the task number, while the y-axis represents the total cost paid. For each worker, the considered hiring cost is at least eight and at most sixteen times the outsourcing cost ($8\lambda_r \leq C_r \leq 16\lambda_r$), while the salary cost is 10% of the outsourcing cost ($\sigma_r = 0.1\lambda_r$).

longer subsequences of tasks similar to a pivot, generates a workload in which hiring becomes more efficient than outsourcing.

6 RELATED WORK

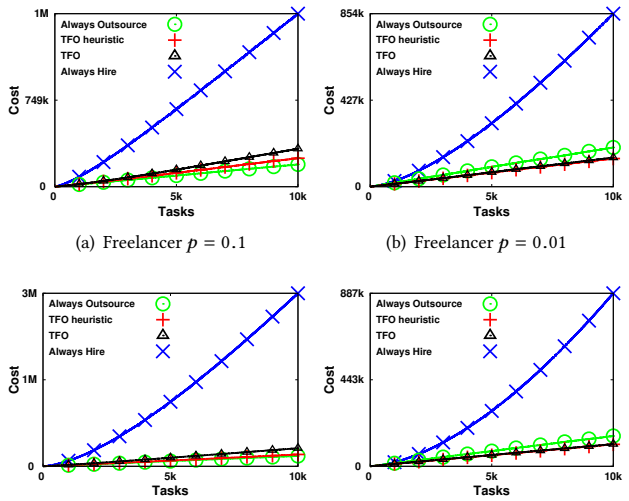
To the best of our knowledge, we are the first to introduce and solve the Team Formation with Outsourcing (TFO) problem. However, our work is related to existing work on crowdsourcing, team formation, and online algorithms design, which we outline next.

Crowdsourcing: Crowdsourcing is the process of completing a task by deploying a large number of workers that work independently, usually through a digital device and in a remote manner. Recent work in computer science has focused on developing advanced tools for crowdsourcing that break complex projects into small sub-tasks and facilitate the recombination of work in a manner that maximizes output quality and minimizes interaction requirements among collaborators [14, 22]. This previous work is orthogonal to ours as the objectives are different: we deal with the problem of assembling teams and hiring/firing workers, not with sub-task design or work integration.

In the crowdsourcing domain, probably the most related to ours is the work of Ho and Vaughan [10]. Their goal is to assign individual workers to tasks, based on the workers' skills. Although Ho and Vaughan also deploy the primal-dual technique in order to solve the task-assignment problem, the tasks they consider can be performed by individual workers and not by teams. As a result, both their problem and their algorithm is different from ours.

Team formation: A large body of work in team formation considers the following problem: given a social or a collaboration network among the workers and a set of skills that needed to be covered, select a team of experts that can collectively cover all the required skills, while minimizing the communication cost between the team members [1, 3, 6, 7, 13, 15, 16, 24]. Other variants of this problem have also considered optimizing the cost of recruiting promising candidates for a set of pre-defined jobs in an offline fashion [9] and minimizing the workload assigned to each individual team member [2, 18].

Although the concept of set-cover is common between our work and previous work, the framework we propose on this paper is different in multiple dimensions. First, we do not focus on optimizing the communication cost; in fact we do not assume any network among the individual workers. Our goal is to minimize the overall cost paid on hiring, outsourcing, and salary costs. This difference in the objectives leads to different (and new) optimization problems that we need to solve. Secondly, most of the work above focuses on the offline version of the team-formation problem, where the jobs to be completed are a-priori known to the algorithm. The exception is the work of Anagnostopoulos et al. [2, 3].However, in their setting they aim to distribute the workload as evenly as possible among



(a) Freelancer $p = 0.1$ (b) Freelancer $p = 0.01$

(c) Guru $p = 0.1$ (d) Guru $p = 0.01$

Figure 3: Results obtained using a stream of tasks with task locality, selecting tasks that are similar to the current pivot with probability $1-p$, and selecting another pivot with probability p. In all cases, the similarity threshold is $\theta = 0.5$.

the workers, while our objective is to minimize the overall cost of maintaining a team that can complete the arriving tasks. Moreover, the option of outsourcing that we propose is new with respect to the team formation literature. Finally, in the design of our online algorithms we use the primal-dual framework, which was not the case for previous work on online team formation.

Primal-dual algorithms for online problems: The algorithms we design for our problems use the primal-dual technique. A thorough analysis on the applicability of this technique for online problems can be found in the book by Buchbinder and Naor [4]. Probably the most closely related to problem are the *ski-rental* and the *caching* problems. We have already discussed the connection of TFO to ski-rental in Section 2. To draw the analogy with caching one can think that bringing a page to the main memory is analogous to hiring a person. The main differences are that in the typical caching problem we do not have covering constraints and there is a fixed limit on the number of pages we can insert in the cache.

7 CONCLUSIONS

In this paper, we introduced a new paradigm of team formation with outsourcing and studied online problems that appear within this paradigm.

In practice, we have shown that decisions regarding hiring (and firing) of workers can be taken by an online algorithm that takes several sources of cost into consideration: hiring/firing costs, outsourcing costs, and salaries. The algorithm is simple to describe and implement, and experimentally, leads to cost savings with respect to alternatives. These cost savings are more striking when hiring costs are low, because then hiring becomes an attractive option, and when the time horizon is long enough that we can find a core pool of workers to stay hired and satisfy a large fraction of the skills required by incoming jobs.

Technically, the problems we have analyzed on this paper involve embedding a set-cover problem in an online algorithm. In all the variants we have analyzed, the primal-dual framework has shown to be a powerful tool for describing and solving this problem. The main technical challenge in the analysis is to carefully consider the various intervals at which a worker can be hired or not hired, as they affect the salaries being paid to these workers.

Code and data. All of our code and data will be available for research purposes with the camera-ready version of this paper.

REFERENCES

[1] A. An, M. Kargar, and M. ZiHayat. Finding affordable and collaborative teams from a network of experts. In *SDM*, pages 587–595, 2013.
[2] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Power in unity: forming teams in large-scale community systems. In *ACM CIKM*, pages 599–608, 2010.
[3] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Online team formation in social networks. In *WWW*, pages 839–848, 2012.
[4] N. Buchbinder and J. Naor. The design of competitive online algorithms via a primal: dual approach. *Foundations and Trends in Theoretical Computer Science*, 3 (2–3):93–263, 2009.
[5] R. H. Coase. The nature of the firm. *economica*, 4(16):386–405, 1937.
[6] C. Dorn and S. Dustdar. Composing near-optimal expert teams: A trade-off between skills and connectivity. In *OTM Conferences (1)*, pages 472–489, 2010.
[7] A. Gajewar and A. D. Sarma. Multi-skill collaborative teams based on densest subgraphs. In *SDM*, pages 165–176, 2012.
[8] D. Geiger, S. Seedorf, T. Schulze, R. C. Nickerson, and M. Schader. Managing the crowd: Towards a taxonomy of crowdsourcing processes. In *AMCIS*, 2011.
[9] B. Golshan, T. Lappas, and E. Terzi. Profit-maximizing cluster hires. In *ACM SIGKDD*, pages 1196–1205, 2014.
[10] C.-J. Ho and J. W. Vaughan. Online task assignment in crowdsourcing markets. In *AAAI*, volume 12, pages 45–51, 2012.
[11] J. Howe. The rise of crowdsourcing. *WIRED (June)*, 2006.
[12] L. Jeppesen and K. Lakhani. Marginality and problem-solving effectiveness in broadcast search. *Organization Science*, 21(5):1016–1033, 2010.
[13] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, pages 985–994, 2011.
[14] A. Kittur, B. Smus, S. Khamkar, and R. Kraut. Crowdforge: Crowdsourcing complex work. In *Annual ACM Symposium on User Interface Software and Technology*, pages 43–52, 2011.
[15] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *ACM SIGKDD*, pages 467–476, 2009.
[16] C.-T. Li and M.-K. Shan. Team formation for generalized tasks in expertise social networks. In *SocialCom/PASSAT*, pages 9–16, 2010.
[17] A. Majchrzak and A. Malhotra. Towards an information systems perspective and research agenda on crowdsourcing for innovation. *The Journal of Strategic Information Systems*, 22(4):257–268, 2013.
[18] A. Majumder, S. Datta, and K. V. M. Naidu. Capacitated team formation problem on social networks. In *KDD*, pages 1005–1013, 2012.
[19] T. Malone, R. Laubacher, and C. Dellarocas. The collective intelligence genome. *MIT Sloan Management Review*, 51(3):21–31, 2010.
[20] M. S. Manasse. Ski rental problem. In *Encyclopedia of Algorithms*, pages 849–851. Springer, 2008.
[21] OECD. Organization for economic cooperation and development data on self-employment. https://data.oecd.org/emp/self-employment-rate.htm, 2016.
[22] D. Retelny, S. Robaszkiewicz, A. To, W. Lasecki, and J. Patel. Expert crowdsourcing with flash teams. In *ACM symposium on User interface software and technology*, pages 75–85, 2014.
[23] C. Riedl and A. W. Woolley. Teams vs. Crowds: Incentives, member ability, and collective intelligence in temporary online team organizations. 2016.
[24] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *ACM SIGKDD*, pages 939–948, 2010.
[25] J. Surowiecki. *The wisdom of crowds: Why the many are smarter than the few and how collective wisdom shapes business, economies, societies and nations.* Anchor Books, 2004.
[26] V. V. Vazirani. *Approximation algorithms.* Springer Science & Business Media, 2013.