

On Task Assignment in Spatial Crowdsourcing

ABSTRACT

Abstract

1. INTRODUCTION

Introduction

2. PRELIMINARIES

In this section, we define the terminologies used in the paper and give a formal definition of the problem under consideration. Also, we analyze the complexity of the problem and its hardness.

2.1 Problem Definition

In this section, we define a set of terminologies in order to define the task assignment problem in spatial crowdsourcing.

DEFINITION 1 (SPATIAL TASK). A spatial task t is a task to be performed at location $t.l$ with geographical coordinates, i.e. latitude and longitude. The task becomes available at $t.r$ (release time) and expires at $t.d$ (deadline). Also, $t.v$ is the obtained reward after completing t .

It should be pointed out that in a spatial crowdsourcing environment, a spatial task t can be executed only if a worker is at location $t.l$. For example, if the query is to report the traffic situation at a specific location, someone has to actually be present at the location to be able to report the traffic. From here on, whenever we use *task* we are actually referring to a spatial task. Now, we formally define a worker.

DEFINITION 2 (WORKER). A worker w is any entity willing to perform spatial tasks. We show the current location of the worker by $w.l$. Each worker has a list of tasks assigned to him, $w.T$, and a maximum number of tasks willing to perform, $w.max$. Also $w.s$ and $w.e$ show the availability of the worker such that the worker is available during the time interval $(w.s, w.e]$.

We assume once a task is assigned to a worker, the worker has no option but to perform the task. Hence, the server will assign a

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China. *Proceedings of the VLDB Endowment*, Vol. 7, No. 3. Copyright 2013 VLDB Endowment 2150-8097/13/11.

Notation	Description
$t.l$	location of task t
$t.r$	release time of task t
$t.d$	deadline of task t
$t.v$	value of task t
$w.l$	location of worker w
$w.s$	the time worker w becomes available
$w.e$	the time worker w becomes unavailable
$w.T$	list of tasks assigned to worker w
$w.max$	maximum number of tasks worker w performs
$w.PTS$	list of all potential task subsets worker w can perform

Table 1: List of notations

task to a worker only if it is possible for the worker to complete the current task and any other task already assigned to him within all the temporal constraints. More specifically, the worker has to be able to complete every task within his availability interval before the tasks' deadlines expire.

DEFINITION 3 (MATCHING). Assuming we have a set of workers W and a set of tasks T , we call $M \subset W \times T$ a matching if for each $t \in T$ there is at most one $w \in W$ such that $(w, t) \in M$. We call $(w, t) \in M$ an assignment and say t has been assigned to w . For each matching M , we define the value (benefit) of M as:

$$Value(M) = \sum_{(w,t) \in M} t.v$$

Now we can formally define the Task Assignment in Spatial Crowdsourcing (TASC) as follows:

DEFINITION 4 (TASK ASSIGNMENT IN SPATIAL CROWDSOURCING). Given a set of workers W , a set of spatial tasks T and a cost function $l : \mathbb{R}^2 \times T \cup (T \times T) \rightarrow \mathbb{R}$ where $l(\langle a, b \rangle)$ is the distance between a and b , the goal of the TASC $\langle W, T, l \rangle$ problem is to find a matching M with maximum value.

The equivalent decision problem for TASC is to decide if there exists a matching M with value K and is shown as TASK $\langle W, T, l, K \rangle$. In the case where every task has a value of 1, the TASC problem become similar to the MTA problem defined in [1]. Section 2.1 lists the notations we frequently use in this paper.

2.2 Complexity Analysis

In this section we use a slightly modified version of the well known Hamiltonian Path Problem. We call it the Minimum Length Hamiltonian Path Problem (Min-Ham-Path) and define it as follows:

DEFINITION 5 (MIN-HAM-PATH). *On a directed graph $G(V, E)$ where each edge $e \in E$ is assigned a length $l : E \rightarrow \mathbb{R}$, a source node s and a length $L \in \mathbb{R}$, the Min-Ham-Path problem $\langle G, l, s, L \rangle$ is to decide whether there exists a path in G that starts from s , visits every other node exactly once and has a length of at most L .*

THEOREM 1. *The Min-Ham-Path problem is NP-Hard.*

PROOF. In order to prove NP-Hardness of Min-Ham-Path we show $\text{Ham-Path} \leq_p \text{Min-Ham-Path}$. The Hamiltonian Path problem asks the following question: Given a directed graph $G(V, E)$ does there exist a path that goes through every node exactly once?

Given an instance of the Ham-Path problem $\langle G \rangle$ we modify graph $G(V, E)$ and generate a new graph $G'(V', E')$ where $V' = V \cup \{o\}$ and $E' = E \cup \{(o, v) : v \in V\}$. Also, for every $e \in E'$ we Assume $l(e) = 1$.

Now we show that $\text{Ham-Path}\langle G' \rangle$ is true iff $\text{Min-Ham-Path}\langle G', l, o, n \rangle$ is true where n is the number of vertices in G . If Min-Ham-Path returns a path of length n , we can remove the first edge from the path which will result in a Hamiltonian Path for the Ham-Path $\langle G \rangle$ problem. On the other hand every Hamiltonian Path on graph G will have length $n - 1$. By adding vertex o and connecting it to the starting vertex, we end up with a Hamiltonian Path of length n on G' . \square

THEOREM 2. *The TASC problem is NP-Complete.*

PROOF. We start the proof by showing that the decision problem of TASC is NP. Given a matching M , we can check that no task is assigned to more than one worker in polynomial time. Also, we can find the value of M with adding the value of every task in M .

Now we prove TASC is NP-Hard by showing reduction Min-Ham-Path \leq_p TASC. Given an instance of the Min-Ham-Path problem $\langle G(V, E), l, o, K \rangle$ we reduce it to an instance of the TASC $\langle W, T, l' \rangle$, where $W = \{o\}$, $T = V \setminus \{o\}$. For every task t we set $t.v = 1$, $t.r = 0$ and $t.d = K$ (We assume workers travel one unit of length with every unit of time). Also for every $e \in E$, $l'(e) = l(e)$. In addition for every $e' \in (W \times T) \cup (T \times T)$ where $e' \notin E$ we set $l'(e') = \infty$.

Finally we show the result of $\text{Min-Ham-Path}\langle G, l, o, K \rangle$ is true if $\text{TASC}\langle W, T, l', n - 1 \rangle$ is true where n is the number of vertices in G . Considering that $|T| = n - 1$ and $t.v = 1$ for every $t \in T$, if there exists a matching with size $n - 1$ it means every task has been assigned to the single worker. Also, since we set the deadline of every task to K this means the worker visits every task no later than K . Therefore, the path that the worker traverses starts at o and goes through every other vertex $v \in V \setminus \{o\}$ where the length of the path is no more than K . \square

3. RELATED WORK

Related Work

4. EXACT CLAIRVOYANT ALGORITHM

In order to solve the TASC problem, the server requires a global knowledge of all the current and incoming spatial tasks and workers. In such case, the server can assign tasks to workers such that

the value of the matching is maximized. However, the server does not have such global knowledge. The server will become aware of a task (worker) and all its corresponding information once the task (worker) becomes available. Due to this lack of future knowledge, finding a globally optimal solution is not feasible.


In this section, we assume there exists a clairvoyant server which has a global knowledge of tasks and workers. The result of this algorithm can later serve as an upper bound to evaluate how well the server assigns tasks to workers. We propose a two step algorithm which finds the optimal solution to TASC problem. In the first step we find all potential subsets of tasks that a single worker is able to complete. Having the output of step 1, in the next step we find a matching with maximum value. In the following sections, we describe different steps of the algorithm and later discuss the complexity of the algorithm. At the end, we show how to adopt the algorithm to address a more general spatial crowdsourcing framework. This adoption does not deteriorate the performance of the algorithm, and in most cases will even improve it.


4.1 Discovering Potential Task Subsets

In the first step of the algorithm we focus on finding all task subsets that a worker w will be able to complete. We can define a Potential Task Subset as:

DEFINITION 6 (POTENTIAL TASK SUBSET). *We call $PTS \subset T$ a potential task subset for worker w iff there exists a route that starts from $w.l$ and for every $t \in PTS$ the path visits $t.l$ at time δ such that $t.r \leq \delta < t.d$. Also the path should not start earlier than $w.s$ and end before $w.e$. We define the value of PTS as:*

$$\text{Value}(PTS) = \sum_{t \in PTS} t.v$$

For each worker w , we define $w.PTS$ as the  of all such potential task subsets.

In order to find all PTSs for a single worker, a straightforward method is to go through all subsets of T and check whether it's a PTS for the worker or not. It's trivial to see that such approach  require $O(n!)$ permutations which makes it computationally expensive. Therefore we will adopt a branch and bound algorithm similar to the one introduced in [2]. We use the following propositions for pruning purposes in our branch and bound algorithm.

PROPOSITION 1. *For every $t \in T$ and $w \in W$ if $(t.r, t.d] \cap (w.s, w.e] = \emptyset$ then for every $PTS \in w.PTS$ we have $t \notin PTS$.*

PROPOSITION 2. *For any $w \in W$ for every $PTS \in w.PTS$ we have $|PTS| \leq w.max$.*

PROPOSITION 3. *For every $w \in W$ if $PTS \in w.PTS$ for every $pts \subset PTS$ we will have $pts \in w.pts$.*

We model the entire solution space as a tree (Fig. 2) and use a depth-first approach to search the solution space. By Proposition 2 we know that we only have to extend the tree up to depth $w.max$ levels for worker w . Each node of the tree corresponds to a subset of T . Therefore, for any node n of the tree, if the corresponding subset of n is not in $w.PTS$, according to Proposition 3, then we can prune the entire sub-tree rooted at n .

Algorithm 2 finds all the PTSs under a specific branch of the search space. The current subset (pts) is expanded with a new task (t) if w temporally overlaps with t (lines 4-8). If the new subset s is a PTS itself we continue the search for more PTSs in the sub-tree

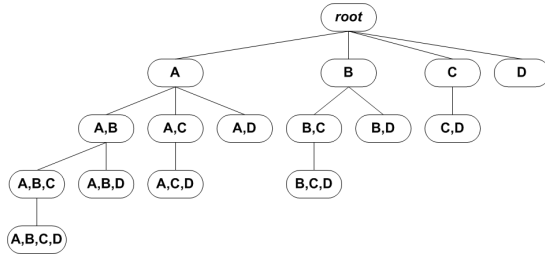


Figure 1: PTS search space for $|T| = 4$

Algorithm 1 FindPTSs(T, W)

Input: T is the set of all tasks and W is the set of all workers
Output: $PTSs[]$ is the list containing all potential task subsets for different workers.

```

1: for  $w$  in  $W$  do
2:    $PTSs[w] = \text{SearchBranch}(\emptyset, T, w)$ 
3: end for
4: return  $PTSs$ 

```

Algorithm 2 SearchBranch(pts, T, w)

Input: pts is the current subset, T is the list of remaining tasks and w is the worker

Output: $PTSs$ is the set of all potential task subsets for w

```

1:  $PTSs = \emptyset$ 
2:  $T_{copy} = T$ 
3: for  $t$  in  $T$  do
4:    $T_{copy} = T_{copy} \setminus t$ 
5:   if  $t$  doesn't temporally overlap with  $w$  then
6:     continue
7:   end if
8:    $s = pts \cup t$ 
9:   if IsPotentialSubset( $s, w$ ) then
10:     $PTSs = PTSs \cup s$ 
11:    if  $|s| < w.max$  and  $T_{copy} \neq \emptyset$  then
12:       $S = \text{Search\_Branch}(s, T_{copy}, w)$ 
13:       $PTSs = PTSs \cup S$ 
14:    end if
15:  end if
16: end for
17: return  $PTSs$ 

```

rooted at s (lines 9-15). Algorithm 3 checks if a subset s can be a PTS for w . According to Definition 6, the desired path should start from the current location of w and no sooner than the time the worker becomes available (lines 1-2). We consider all permutations of s and check if a permutation satisfies all the temporal constraints of Definition 6 (lines 5-14). The subset s will be a PTS only if a permutation is found to satisfy all constraints. Finally in Algorithm 1 for each worker we search for PTSs starting the search at the root of the tree.

4.2 Finding the Maximal Matching

Having found all PTSs for each worker in step 1, the next step of the algorithm focuses on finding the matching with the maximum value. For this purpose, we construct a graph such that each PTS from step 1 is a vertex in the graph. We call this graph the *PTS-Graph*. Then we show that the *maximum weighted clique* in the PTS-Graph corresponds to the matching with the maximum value.

The PTS-Graph is a multi-layered graph such that for each worker we add a layer to the graph. Withing each layer l , for each PTS in-

Algorithm 3 IsPotentialSubset(s, w)

Input: s is the input set

Output: *true* if s is a PTS for w and *false* otherwise

```

1: for  $P$  in PermutationsOf( $s$ ) do
2:    $possible = true$ 
3:    $loc = w.l$ 
4:    $time = w.s$ 
5:   for  $i = 1$  to  $|P|$  do
6:      $nextTime = \max(P[i].r, time + \text{dist}(w.l, P[i].l))$ 
7:     if  $nextTime < P[i].d$  and  $nextTime < w.e$  then
8:        $loc = P[i].l$ 
9:        $time = nextTime$ 
10:    else
11:       $possible = false$ 
12:      break
13:    end if
14:  end for
15:  if  $possible$  then
16:    continue
17:  else
18:    return true
19:  end if
20: end for
21: return false

```

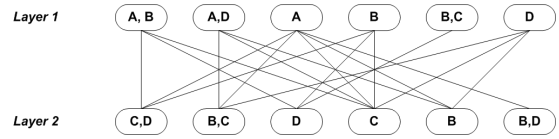


Figure 2: A sample two layered PTS-Graph

side $w_l.PTS$ we add a node to layer l . The weight of this node is equal to the value of the corresponding PTS. We use an arbitrary ordering to name the nodes within each layer of the PTS-Graph. Assuming layer l contains m nodes, we name them n_{l1} through n_{lm} . Also, we define the set of edges of the PTS-Graph as:

$$E = \{(n_{ai}, n_{bj}) | a \neq b \wedge (PTS_{ai} \cap PTS_{bj} = \emptyset)\} \quad (1)$$

where PTS_{lm} refers to the corresponding PTS of node n_{lm} . In other words, there will be an edge between two nodes in different layers if the corresponding PTSs of the two nodes don't contain a common task.

Given the PTS-Graph $G(V, E)$, we associate an assignment of tasks to workers with a set of vertices $N \subset V$ as follows: if $n_{lm} \in N$, then every task $t \in PTS_{lm}$ is assigned to w_l .

DEFINITION 7 (CLIQUE). For undirected graph $G=(V,E)$, a Clique in G is a subset $S \subset V$ of vertices, such that $\{x, y\} \in E$ for all distinct $x, y \in S$.

A clique is said to be *maximal* if its vertices are not a subset of the vertices of a larger clique, and *maximum* if there are no larger cliques in the graph.

DEFINITION 8 (MAXIMUM WEIGHT CLIQUE). For undirected graph $G=(V,E)$ where each $v \in V$ is assigned a value $w(v)$, the Maximum Weight Clique is the clique for which the sum over the weight of all vertices in the clique, is larger than any other clique in the graph.

It's worth noting that a maximum weight clique is not necessarily a maximum clique but it definitely is a maximal clique.

THEOREM 3. *If C^* is a maximum weighted clique in the PTS-Graph, then the assignment of tasks to workers corresponding to C^* , M^* , will be a maximum value matching for the TASC problem.*

PROOF. Based on the way we generated the PTS graph, no two vertices in C^* can contain the same task in their corresponding PTS. Therefore, any task t will be assigned to at most one worker. Next we need to show no other matching M can have a larger value than M^* . If such matching M existed, then the clique C corresponding to M , will have a larger weight than C^* which conflicts the fact that C^* is the maximum weighted clique. \square

There exists a vast body of literature on solving the maximum weight clique problem [3]. Here we briefly explain one of these algorithms and refer the interested readers to [4].

In Algorithm 4 we start with an arbitrary node ordering. [4] suggests a node ordering where v_1 is the node with the largest total sum of the weights of its adjacent vertices and so on. Algorithm 4 computes a value $C(k)$, $1 \leq k \leq n$ which tells the largest weight of a clique, only considering nodes $\{v_k, v_{k+1}, \dots, v_n\}$. If $w(i)$ denotes the weight of v_i , then we will have $C(n) = w(n)$ and other values for $C()$ are computed in a backtrack search. For $1 \leq k \leq n-1$, $C(k) > C(k+1)$ only when a corresponding clique exists and contains v_k . If such clique does not exist then $C(k) = C(k+1)$. In order to find such $C(k)$ clique, a branch and bound approach is used to search all possible cliques. Details of the branch and bound algorithm can be found in [4].

Algorithm 4 MaximumWeightClique(V, E)

Input: V is an ordering of the nodes and E is the set of edges in the graph

Output: C a subset of V as the maximum weighted clique

```

1:  $C = \{v_n\}$ 
2:  $max = w(n)$ 
3: for  $k : n-1$  down to 1 do
4:    $\langle C_k, w \rangle = \text{FindMaxClique}(V, E, v_k)$ 
5:   if  $w > max$  then
6:      $C = C_k$ 
7:      $max = w$ 
8:   end if
9: end for
10: return  $C$ 
```

4.3 Complexity Analysis

Here we analyze the time complexity of our proposed exact algorithm. For simplicity we assume $|T| = n$, $|W| = m$ and $|w.max| = p$. We start with Algorithm 3. The size of set s is at most equal to p , hence the number of permutations of s is $O(p^p)$. Also the for loop in line 5 will run at most p times which will make the overall complexity of Algorithm 3 $O(p^{p+1})$. For Algorithm 2 we can assume that we are running the IsPotentialPTS() method for each node of the search space tree (Fig. 2) once. At level i of the search space tree we will have at most $C(n, i)$. Also for each node in level i the size of the task subsets will be i . Hence, the IsPotentialSubset() method will run in $O(i^{i+1})$. Therefore the amortized time complexity of Algorithm 2 will be:

$$\sum_{i=1}^p \binom{n}{i} O(i^{i+1}) = O(n^{n-p} p^{p+1})$$

Since we run the SearchBranch method for each worker once in, we can conclude the the time complexity of 1 is $O(mn^{n-p} p^{p+1})$.

In Algorithm 4 we implement the FindMaxClique() method, we use a branch and bound algorithm described in [4]. The time complexity of this algorithm will be $O(n^n)$ for a graph of size n . Therefore, for a graph of size n , the overall time complexity of Algorithm 4 will be $O(n^{n+1})$.

We end the section with a discussion on how generalizing the spatial crowdsourcing framework can affect the algorithm introduced in this section. In [5] it's assumed that each task requires a certain level of *confidence* and only workers with a *trust* level higher than that will be able to perform the task. It's also seen that in some spatial crowdsourcing frameworks, workers define a spatial range and only perform task within this range. Constraints like this will only remove a subset of tasks from the list of tasks a certain worker is able to execute. As a result, the search space of Algorithm 1 will get reduced and we end up with few number of PTSs per worker. This in turn will reduce the size of the PTS-Graph in Section 4.2.

5. APPROXIMATE CLAIRVOYANT ALGORITHMS

The algorithm presented in Section 4 is computationally expensive to run. In this section we present a greedy algorithm for solving the TASC problem. We use two rather simple heuristics; (1) We give priority to tasks with higher values and (2) among different workers that can execute task t , we select the worker which is closest to t . The logic behind heuristic (1) is to assign tasks with larger benefit gains first in order to maximize the overall benefit. We can see similar heuristics in many job scheduling problems [6]. As for the second heuristic, we try to assign tasks to nearby workers so the amount of time spent to execute a specific task is minimized. This in turn will leave the worker with more time to execute other tasks.

Algorithm 5 MostValuableFirst(W, T)

Input: W is the set of workers and T is the set of tasks

Output: M is a set of assignments between tasks and workers

```

1:  $M = \emptyset$ 
2:  $T_{sorted} = \text{Sort } T \text{ based on } t.v$ 
3: while  $T_{sorted} \neq \emptyset$  do
4:    $W_t = W$ 
5:   while  $W_t \neq \emptyset$  do
6:      $w = w \in W_t$  with minimum distance to  $t$ 
7:     if IsPotentialSubset( $w.T \cup t, w$ ) then
8:       assign  $t$  to  $w$ .
9:        $M = M \cup \langle w, t \rangle$ 
10:      if  $|w.T| = w.max$  then
11:         $W = W \setminus w$ 
12:      end if
13:    else
14:       $W_t = W_t \setminus w$ 
15:    end if
16:  end while
17: end while
18: return  $M$ 
```

Algorithm 5 outlines the implementation of these heuristics. In line 2, we sort the tasks based on their values in accordance with heuristic (1). For the sort method, we break ties by giving a higher

priority to tasks with smaller distance to their nearest worker. This means that if two task have equal values, the one with smaller distance to its nearest worker will have a higher priority. In this way, the MostValuableFirst() method will work perfectly fine in the case all tasks have similar value and the goal is to only maximize the number of assigned tasks.

In line 7 of Algorithm 5 we use the IsPotentialSubset() method described in Algorithm 3. In Section 4.3 we showed that the complexity of Algorithm 3 is $O(p^{p+1})$. This can make Algorithm 5 computationally expensive to run which contradicts the goal of implementing a fast heuristic algorithm. In the case of larger values of p , instead of the exact solution proposed in Algorithm 3, we can use several approximate algorithms that have been proposed for vehicle routing problems. Depending on the desired accuracy we seek to achieve, these algorithm can run as fast as $O(p)$ [7].

6. NON-CLAIRVOYANT ALGORITHM

In previous sections, we were assuming there existed a clairvoyant server which had a global knowledge of future tasks and workers. However, in a real-world spatial crowdsourcing environment, you cannot make this assumption. The server will know about different attributes of a task at the same time the task is released. Also, at each point of time, the server knows only about the workers that are available at that time. In such environment, once a task gets released the server has to make a decision whether to assign the task to a worker or not. We assume that the server's decision is irrevocable. Clearly it is not possible for the server to make these decision such that we end up with an optimal assignment in the end.

In this section we propose a greedy algorithm for the server to assign a task to a worker once the task arrives. Since this algorithm makes the assignments as soon as a task arrives and on the fly, we call it the *OnlineTASC* algorithm. The general idea of OnlineTASC is to distribute the workers within the entire region such that the spatial distribution of *currently available* workers (S_W) is as close as possible to the *overall* spatial distribution of tasks (S_T). In other words, the server should attempt to put more available workers in regions where there's a higher chance for a task to show up. Therefore, the server needs to know S_T . For this, it can either start with a uniform distribution and update it as new tasks arrive or we can assume the server has S_T as a priori. As for the spatial distributions, we assume we have a grid on the entire region. For each cell i in the grid, we assign the probability $P(i)$ to this cell as the likeliness of a spatial entity (task or worker) being located in cell i .

We can summarize the general heuristic of OnlineTASC as follows: Upon arrival of task t , among all the workers *eligible* to perform t , we assign t to the worker w such that moving w from $w.l$ to $t.l$ will result in minimum difference between S_W and S_T . This suggests that we need to have a measure which tells us how different two spatial distributions are. With the way we model our spatial distributions, we can think of them as *discrete probability distributions*. This allows us to use some concepts widely used in the field of information theory to measure the distance between two probability distributions.

DEFINITION 9 (KULLBACK-LEIBLER DIVERGENCE). Let P_1 and P_2 be two discrete probability distributions. The KL divergence [8] of P_2 from P_1 is defined as:

$$D_{KL}(P_1 \parallel P_2) = \sum_i P_1(i) \log \frac{P_1(i)}{P_2(i)}$$

The advantage of D_{KL} is that it is simple to compute which is necessary in our scenario where we want close to real-time decisions. On the other hand D_{KL} is a non-symmetric measure meaning that $D_{KL}(P_1 \parallel P_2) \neq D_{KL}(P_2 \parallel P_1)$. In addition, $D_{KL}(P_1 \parallel P_2)$ is only defined when we have $P_2(i) = 0 \implies P_1(i) = 0$. However, this is not the case with S_W and S_T . Therefore, we need a measure that overcomes this problem yet remains simple to compute.

DEFINITION 10 (JENSEN-SHANNON DIVERGENCE). Let P_1 and P_2 be two discrete probability distributions and $\pi_1, \pi_2 \geq 0, \pi_1 + \pi_2 = 1$ be the relative weights for distributions $P_1()$ and P_2 , respectively. The JS divergence [9] of P_2 from P_1 is defined as:


$$D_{JS}(P_1 \parallel P_2) = H(\pi_1 P_1 + \pi_2 P_2) - \pi_1 H(P_1) - \pi_2 H(P_2)$$

where $H()$ is the Shannon entropy [10].

For two distributions of similar importance, we set $\pi_1 = \pi_2 = \frac{1}{2}$. Then we will have:

$$D_{JS}(P_1 \parallel P_2) = \frac{1}{2} D_{KL}(P_1 \parallel Q) + \frac{1}{2} D_{KL}(P_2 \parallel Q)$$

where $Q = \frac{1}{2}(P_1 + P_2)$.

Algorithm 6 outlines the process of assigning a task t to a worker once t arrives. It is important to note that, when computing the  al distribution of the workers (line 3), we are not just considering about the presence of a worker in a specific region. What we really care about is the availability of the workers in different regions. Therefore, a worker that is willing to accept 2 more tasks, is twice as important as a worker who can take only 1 more task. Therefore, for each worker w in a specific region, we need to consider the value $m = w.max - |w.T|$. Also, in line 4, we sort the workers based on their distance to t . The reason is that if we assume the current distribution of workers is relatively close to S_T , we can claim that a rather small movement in the location of a single worker will most probably still keep S_W close to S_T . In this way, we can minimize the number of times the algorithm passes the condition of the if clause on line 7.

Algorithm 6 OnlineTASC(W, S_T, t)

Input: W is the set of currently available workers, S_T is the spatial distribution of tasks and t is a task the has just released

Output: Either $w \in W$ as the worker task t should be assigned to or *null* if no worker is selected

```

1:  $w_{selected} = null$ 
2:  $min = \infty$ 
3:  $S_W =$  Spatial Distribution of current workers
4:  $W_{sorted} =$  Sort  $W$  based their distance to  $t$ .
5: for  $w \in W_{sorted}$  do
6:    $Q_w =$  Modify  $S_W$  by moving  $w$  from  $w.l$  to  $t.l$ 
7:   if  $D_{JS}(Q_w \parallel S_T) < min$  then
8:     if IsPotentialSubset( $w.T \cup t, w$ ) then
9:        $min = \delta$ 
10:       $w_{selected} = w$ 
11:   end if
12: end if
13: end for
14: return  $w_{selected}$ 
```

7. EXPERIMENTS

Experiments

8. CONCLUSION AND FUTURE WORK

Conclusion and Future Work

9. REFERENCES

- [1] L. Kazemi and C. Shahabi, "Geocrowd: Enabling query answering with spatial crowdsourcing," in *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL '12. New York, NY, USA: ACM, 2012, pp. 189–198. [Online]. Available: <http://doi.acm.org/10.1145/2424321.2424346>
- [2] D. Deng, C. Shahabi, and U. Demiryurek, "Maximizing the number of worker's self-selected tasks in spatial crowdsourcing," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL'13. New York, NY, USA: ACM, 2013, pp. 324–333. [Online]. Available: <http://doi.acm.org/10.1145/2525314.2525370>
- [3] M. Y. Kovalyov, C. Ng, and T. E. Cheng, "Fixed interval scheduling: Models, applications, computational complexity and algorithms," *European Journal of Operational Research*, vol. 178, no. 2, pp. 331 – 342, 2007. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0377221706003559>
- [4] P. R. J. Östergård, "A new algorithm for the maximum-weight clique problem," *Nordic J. of Computing*, vol. 8, no. 4, pp. 424–436, Dec. 2001. [Online]. Available: <http://dl.acm.org/citation.cfm?id=766502.766504>
- [5] L. Kazemi, C. Shahabi, and L. Chen, "Geotrucrowd: Trustworthy query answering with spatial crowdsourcing," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL'13. New York, NY, USA: ACM, 2013, pp. 314–323. [Online]. Available: <http://doi.acm.org/10.1145/2525314.2525346>
- [6] A. W. J. Kolen, J. K. Lenstra, C. H. Papadimitriou, and F. C. R. Spieksma, "Interval scheduling : a survey," *Naval Research Logistics*, vol. 54, pp. 530–543, 2007.
- [7] G. Laporte, M. Gendreau, J.-Y. Potvin, and F. Semet, "Classical and modern heuristics for the vehicle routing problem," *International Transactions in Operational Research*, vol. 7, no. 4-5, pp. 285 – 300, 2000. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0969601600000034>
- [8] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.
- [9] J. Lin, "Divergence measures based on the shannon entropy," *Information Theory, IEEE Transactions on*, vol. 37, no. 1, pp. 145–151, Jan 1991.
- [10] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal, The*, vol. 27, no. 3, pp. 379–423, July 1948.