Maximizing the Number of Worker's Self-Selected Tasks in Spatial Crowdsourcing

Dingxiong Deng
Dept. of Computer Science
University of Southern California, USA
dingxiong.deng@usc.edu

Cyrus Shahabi
Dept. of Computer Science
University of Southern California, USA
shahabi@usc.edu

Ugur Demiryurek
Dept. of Computer Science
University of Southern California, USA
demiryur@usc.edu

ABSTRACT

With the progress of mobile devices and wireless broadband, a new *eMarket* platform, termed *spatial crowdsourcing* is emerging, which enables workers (aka crowd) to perform a set of spatial tasks (i.e., tasks related to a geographical location and time) posted by a *requester*. In this paper, we study a version of the spatial crowdsourcing problem in which the workers autonomously select their tasks, called the *worker selected tasks* (WST) mode. Towards this end, given a worker, and a set of tasks each of which is associated with a location and an expiration time, we aim to find a schedule for the worker that maximizes the number of performed tasks. We first prove that this problem is NP-hard. Subsequently, for small number of tasks, we propose two exact algorithms based on dynamic programming and branch-and-bound strategies. Since the exact algorithms cannot scale for large number of tasks and/or limited amount of resources on mobile platforms, we also propose approximation and progressive algorithms. We conducted a thorough experimental evaluation on both real-world and synthetic data to compare the performance and accuracy of our proposed approaches.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms

Algorithms

Keywords

crowdsourcing, spatial crowdsourcing, spatial task assignment

1. INTRODUCTION

The ubiquity of mobile devices with high-fidelity sensors and recent decreases in the cost of ultra-broadband wireless networks (e.g., 4G) enable mobile users to easily sense, collect and transmit quality data from real-world locations. The main feature of these collected datasets is that they are tagged automatically with the time and location of their collection. In turn, such geo-tagged datasets can be used in many applications, such as location-aware image collection (e.g., Picasa [1]), road traffic monitoring (e.g., Waze [2]), and geographical data generation (e.g., OpenStreetMap [3]). One way to generalize and harness these capabilities is to develop a market for anyone to submit requests for real-world data collections tasks, tagged with time and location, and then distribute these tasks among people with smart phones at the vicinity of the tasks, who are willing to collect the required data. Such a *spatial crowdsourcing* market platform was first detailed in [15] where each *requester* submits a set of *spatial tasks* (tasks related to a location and time) to be performed by a set of *workers*. The workers must physically travel to the tasks' locations to perform time-sensitive spatial tasks.

In [15], Kazemi and Shahabi focused on the problem of optimally assigning tasks to workers, assuming that the server has the global knowledge about locations of all the workers and tasks, termed *Server Assigned Tasks* (SAT) mode. In this paper, however, we focus on the scenario where workers autonomously select their desired tasks from a list of published tasks posted by the spatial crowdsourcing server, termed *Worker Selected Tasks* (WST) in [15]. Consequently, our optimization objective is to maximize the number of performed tasks per worker. The extra complexity of our problem comes from the fact that we consider: 1) the variable cost of traveling from one task to the other, and 2) the expiration time of a task after which the task cannot be completed; neither of which were considered in [15]. Moreover, note that the WST mode is more privacy-friendly than the SAT mode.

In sum, given a set of spatial tasks and a worker, our goal is to find a schedule which maximizes the number of tasks that can be completed by the worker while both travel cost and expiration time of the tasks have been taken into consideration. We refer to this problem as the Maximum Task Scheduling (MTS) problem. To illustrate, consider a simple MTS problem in Figure 1 where there is one worker w with five available tasks, namely, A, B, C, D and E. Each task is associated with a location (x, y) and an expiration time d. For instance, Task A is located at $(4, 3)$ and will expire after 9 time units. The objective of the worker is to complete as many as tasks while conforming to the expiration time of the tasks. Obviously the worker needs to make a plan to finish these tasks. Assume that in Figure 1 the worker starts from time 0, the travel cost for one grid is one time unit and the distance between the tasks is Manhattan distance. The worker can finish four tasks by following the order $A \rightarrow E \rightarrow C \rightarrow D$, whereas, he can only complete one task if he chooses to start with task B. Therefore, the task schedule $A \rightarrow E \rightarrow C \rightarrow D$ is a better solution to our MTS problem than the schedule B.

At first glance MTS may look similar to the class of job scheduling problems [20]. In particular, given a single machine and a set of jobs with processing time, release time and expiration time, the objective of job scheduling problem is to find a schedule which allocates one time interval for each job on the machine and maximizes the number of jobs completed before their expiration times.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACM SIGSPATIAL '13, November 05-08, 2013. Orlando, FL, USA
Copyright © 2013 ACM ISBN 978-1-4503-2521-9/13/11...$15.00.

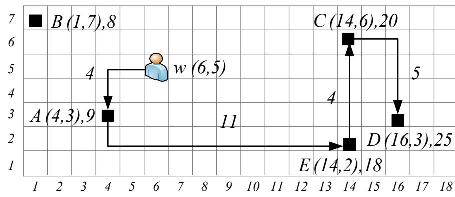


Figure 1: Running example

However, the processing time of each job is known in advance and is *independent of the schedule*. In contrast, with MTS the cost to travel from one task to the other depends on the order that the tasks are scheduled. Hence, the time to complete a task, which includes the travel time to the task, is not known a priori and depends on the task schedule itself, which renders MTS a new and more complex problem. Moreover, while other variations of the job scheduling problem may come close to MTS, with crowdsourcing we need an algorithm (running on a smart phone) to provide a schedule for the worker in milliseconds, which is different than solving a job-scheduling problem as a one time optimization problem (see Section 6 for more detailed explanation).

In this paper, we first prove that MTS is NP-hard by reduction from a specialized version of Traveling Salesman problem. Given that in real applications the number of available tasks for one worker may be relatively small, we propose two exact algorithms based on dynamic programming and branch-and-bound algorithm to find the optimal solution. Our dynamic programming approach reduces the search space by iteratively expanding the sets of tasks in the ascending order of the number of tasks. We also incorporate the Apriori principle [6] to our dynamic programming that further improves the performance by reducing the search space. With our branch-and-bound algorithm, we calculate the *candidate task set* for each branch which filters the unpromising tasks and then derive bounds for ordering and pruning. However, the running time of our dynamic programming and branch-and-bound algorithm grows exponentially as the number of tasks increases. Therefore, we propose three approximation algorithms to accomplish efficient processing of MTS with large number of tasks and/or limited amount of resources on mobile platforms. Our approximation algorithms are based on three different heuristics, namely Least Expiration Time Heuristic (LEH), Nearest Neighbor Heuristic (NNH) and Most Promising Heuristic (MPH). The main idea of LEH and NNH is to exploit expiration time and spatial proximity of the tasks to expedite response time and minimize the memory consumption. On the other hand, MPH takes advantage of branch-and-bound algorithm to greedily choose the tasks with the highest upper bound. Finally, we propose a class of progressive algorithms, where they first use one of the approximation algorithms to suggest a small set of initial tasks (e.g., 1 or 2 task(s)) and then while the worker is busy performing those initial tasks, the progressive algorithm chooses one of the optimal algorithms to provide the rest of the schedule.

We conducted extensive experiments with real-world and synthetic datasets to compare the performance of our various algorithms. Our results show that the exact algorithms are feasible for real-world applications only when the number of tasks is small (i.e., less than 20). Meanwhile, the response time of all the three approximation algorithms are within milliseconds, which makes them interactive and suitable for running on mobile platforms. However, their accuracy varies depending on the location of tasks and their expiration times. Finally, progressive algorithms have the best of the two worlds as they can achieve near-optimum results, while being as efficient and interactive as the approximate algorithms.

The remainder of this paper is organized as follows. In Section 2, we formally define our maximum task scheduling problem and study its computation complexity. In Section 3, we establish the theoretical foundation and present two exact algorithms to solve MTS. We propose our approximation and progressive algorithms in Section 4. Section 5 reports the results of our experiments. In Section 6, we review the related work and Section 7 concludes the paper and discusses some of our future directions.

2. PRELIMINARIES

In this section, we define the terminology used throughout the paper, and analyze the complexity of the MTS problem.

2.1 Problem Definition

Definition 1. A *spatial task* s is a query to be answered at location l_s with expiration d_s, where l_s is a point in the $2D$ space.

With spatial crowdsourcing, a spatial task s can be answered only if the worker is physically located at that location l_s. Besides, considering the expiration time, a spatial task s can be completed only if the worker arrives at l_s before its deadline d_s. For simplicity and without loss of generality, we assume that the processing time of each task is 0, which means that a worker will go to the next task upon finishing the current task.

Definition 2. A worker, denoted by w, is a person who volunteers to perform spatial tasks. A worker can be in an either online or offline mode. A worker is online when he is ready to accept tasks. An online worker is associated with location l_w, his current time instance t_w and a spatial region D_w.

In *worker selected tasks mode*, once a worker is online, he sends a task inquiry to the server which includes his spatial region D_w [1]. The server returns all available tasks in the worker's vicinity for him to perform. For instance, Figure 1 shows an example of one worker w and all available tasks A, B, C, D, E in his spatial region. The worker located at $(6, 5)$ starts from time zero; each task is associated a location and deadline: Task A located at $(4, 3)$ will expire after 9 time units. The worker can choose any subsets of the tasks to finish. Considering the travel cost and expiration time of the task, the worker needs to make a plan to finish these tasks. Next we define task sequence and arrival time of a task.

Definition 3. Given an online worker w and a set of n tasks S in D_w, $R = (s_1, s_2, ..., s_r)$ is a *task sequence* if and only if $r \leq n, s_i \in S, s_i \neq s_j$ for $i \neq j$. The *arrival time* at task s_i in R is defined as:

$$a(s_i) = \begin{cases} a(s_{i-1}) + c(s_{i-1}, s_i) & \text{if } i \neq 1 \\ t_w + c(w, s_1) & \text{if } i = 1 \end{cases}$$

where $c(a, b)$ is the travel cost from the location of a to the location of b.

Task sequence represents the sequence of how a worker finishes these tasks and determines the number of tasks a worker can complete. For example, in Figure 1 by following (A, E, C, D), worker w finishes 4 tasks, since $a(A) = 4, a(E) = 4 + 11 = 15, a(c) = 15+4 = 19$ and $a(D) = 19+5 = 24$ are less than their deadlines.

Definition 4. A *Valid Task Sequence* is a task sequence in which all of its tasks can be finished, i.e., $a(s_i) \leq d_{s_i}$ for each task $s_i \in R$. A *Maximum Valid Task Sequence* is a valid task sequence in which none of its super sequences is valid.

Definition 5. Given a worker w and a set of n tasks S in his vicinity, the **maximum task scheduling (MTS)** problem is to find the longest *maximum valid task sequence*.

[1] We note that D_w is an optional parameter that represents worker's preferred working area (e.g., city of Los Angeles). Our proposed algorithms are not restricted by D_w and they can schedule any number of tasks in the vicinity of the worker. We use D_w as a stop condition for our algorithms to stop scheduling more tasks. We could use other restrictions such as "available time" or "maximum number of tasks" as stop conditions instead.

In Figure 1, $(C, D, E), (A, E, D)$ and (A, E, C, D) are valid task sequences, but (A, E, C, D, B) is not a valid task sequence since task B cannot be finished on time. Notice that both (C, D, E) and (A, E, C, D) are maximal valid task sequences, however, only (A, E, C, D) is the optimum solution for the MTS problem since it contains 4 tasks instead of 3.

2.2 Problem Complexity

We prove that the MTS problem is NP-hard by reduction from a specialized version of Traveling Salesman Problem (TSP) called sTSP. First, we give the definition of sTSP and prove it as NP-C.

Definition 6. Given a complete graph $G(V, E)$ with weight function $c : V \times V \rightarrow Z$, a source vertex x and cost $k \in Z$, where $k \geq 2 \cdot c(x, y)$ for any $y \neq x$, the sTSP problem $< G, c, x, k >$ is to determine whether there exists a tour which visits each and every vertex exactly once and finishes at the source vertex x from which the tour has started with the cost of at most k.

THEOREM 1. *sTSP is NP-C.*

PROOF. The proof is shown in Appendix A. ■

THEOREM 2. *Given a worker w, a set of n tasks S and number z, deciding whether there exists a valid task sequence R, st. $|R| = z$ is NP-C. That is, the decision problem of MTS $< w, S, z >$ is NP-C.*

PROOF. The proof is shown in Appendix B. ■

Since we have proved that the decision version of MTS is NP-C, we can conclude that the MTS problem is NP-hard.

3. EXACT ALGORITHMS

Even though we proved that the MTS problem is NP-hard, if the number of tasks is relatively small, the exponential time complexity might still be affordable. A straightforward method is to use brute-force approach, which enumerates all permutations of a set of tasks, finds the permutation with the maximum number of completed tasks, and then returns the corresponding valid task sequence as the solution. It is trivial to see that this brute-force approach is computationally expensive since there are $O(n!)$ permutations in total. To address this issue, we design two types of exact algorithms based on dynamic programming strategy and branch-and-bound strategy.

3.1 Dynamic Programming

3.1.1 Naïve Dynamic Programming

Compared to the brute-force search, the superiority of dynamic programming is due to the fact that it ignores the order of task sequence and examines the sets of tasks. Specifically, it iteratively expands the sets of tasks in the ascending order of set size. For each task in one set, we consider the scenario that it is finished in the end, and find the maximum number of completed tasks by utilizing the best sets of its subsets. We present the details of the algorithm as follows.

Given a worker w, and a set of tasks $Q \subseteq S$ (for simplicity, we just use the index to denote a specific task, i.e., $S = \{1, 2, \cdots, n\}$), we define $\texttt{opt}(Q, j)$ as the maximum number of tasks completed by scheduling all the tasks in Q with constraints starting from w and ending at the task j, and R as the corresponding task sequence [2] to achieve this optimum value. We also use i to denote the second-to-last task before arriving at j in R, and R' to denote the corresponding task sequence for $\texttt{opt}(Q - \{j\}, i)$. Then the computation of $\texttt{opt}(Q, j)$ has the recursive solution shown in Equation 1.

$$\texttt{opt}(Q, j) = \begin{cases} 1 & \text{if } |Q| = 1 \\ \max\limits_{i \in Q, i \neq j} \{\texttt{opt}(Q - \{j\}, i) + \delta_{ij}\} & \text{otherwise} \end{cases} \quad (1)$$

[2] R contains $|Q|$ tasks, and is not necessary to be a valid task sequence.

Algorithm 1 MST_DP()

1: **for** each task i in {1, 2, ..., n} **do**
2: $\quad \texttt{opt}(\{i\}, i) \leftarrow 1$
3: $\quad \texttt{pre}(\{i\}, i) \leftarrow null$
4: **for** $len \leftarrow 2$ to n **do**
5: $\quad$ **for** all subsets of $Q \subseteq \{1, 2, ..., n\}$ of size len **do**
6: $\quad\quad$ **for** all $j \in Q$ **do**
7: $\quad\quad\quad \texttt{opt}(Q, j) \leftarrow \max\limits_{i \in Q, i \neq j} \{\texttt{opt}(Q - \{j\}, i) + \delta_{ij}\}$
8: $\quad\quad\quad \texttt{pre}(Q, j) \leftarrow \arg \max\limits_{i \in Q, i \neq j} \{\texttt{opt}(Q - \{j\}, i) + \delta_{ij}\}$
9: $|R^*|$=$\max\limits_{j} \texttt{opt}(\{1, 2, ..., n\}, j)$
10: compute R^* based on opt and pre
11: **return** R^*

$$\delta_{ij} = \begin{cases} 1 & \text{if } j \text{ can be finished after connecting } j \text{ in the end of } R' \\ 0 & \text{otherwise} \end{cases}$$

When Q contains only one task j, the problem is trivial; we can set $\texttt{opt}(\{j\}, j) = 1$ since we know the worker can succeed to finish any task starting from w ($c(w, j) \leq d(j)$). When $|Q| > 1$, assuming that we have known the second-to-last task i and R', then we can get the solution for $\texttt{opt}(Q, j)$ by concatenating j in the end of R', and then check whether j can be finished after R'. Obviously, the assumption that i is known does not hold, and hence we need to search through Q to examine all possibilities and find the particular i that achieves the optimum value of $\texttt{opt}(Q, j)$.

With Equation 1, now we can compute the longest maximum valid task sequence, which is presented in Algorithm 1. Note that we introduce another notation $\texttt{pre}(Q, j)$ for recording the last-to-second task i before achieving $\texttt{opt}(Q, j)$ to facilitate the reconstruction of the maximum valid task sequence R^*. It first initializes the optimum value when Q contains one task (lines 1–3). Subsequently, it generates and processes sets in the increasing order of their size from 2 to n (lines 4–5). For each task $j \in Q$, it computes $\texttt{opt}(Q, j)$ and $\texttt{pre}(Q, j)$ according to Equation 1 (lines 7–8). Note that in order to compute δ_{ij}, for each subproblem we also need to record the least travel time when the worker arrives at j by scheduling tasks in Q. In this way we can efficiently check whether task j can be finished after scheduling $Q - \{j\}$ ends with i. To save space, the procedure of constructing R^* from tables opt and pre is omitted here (lines 9–10).

PROPOSITION 1. *Algorithm 1 correctly computes the maximum valid task sequence R^* in $O(n^2 \cdot 2^n)$ time and $O(n \cdot 2^n)$ space.*

Proof (Sketch): The correctness is straightforward to derive from Equation 1. For the time complexity, there are at most $(\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n}) \cdot n = O(2^n \cdot n)$ subproblems, and each one takes linear time to solve, thus the total running time is $O(n^2 \cdot 2^n)$, which is much faster than $O(n!)$. For each subproblem $\texttt{opt}(Q, j)$, we record the previous task from which it comes, thus the space complexity is $O(n \cdot 2^n)$. ■

Example: In Figure 1, for the sets with one task, $\texttt{opt}(\{A\}, A) = 1, \cdots$, and $\texttt{opt}(\{E\}, E) = 1$. For all the sets with size from 2 to 5, we iteratively calculate the opt value. For instance, $\texttt{opt}(\{A, E\}, A) = 1$ since by following the task sequence (E, A), only E can be finished, but $\texttt{opt}(\{A, E\}, E) = 2$ because by following (A, E), both A and E can be finished. For set $\{A, C, E\}$ with end task C, the second-to-last task could be either A or E, thus,

$$\texttt{opt}(\{A, C, E\}, C) = \max \begin{cases} \texttt{opt}(\{A, E\}, E) + \delta_{EC} = 2 + 1 = 3, \\ \texttt{opt}(\{A, E\}, A) + \delta_{AC} = 1 + 0 = 1, \end{cases}$$
$$= 3$$

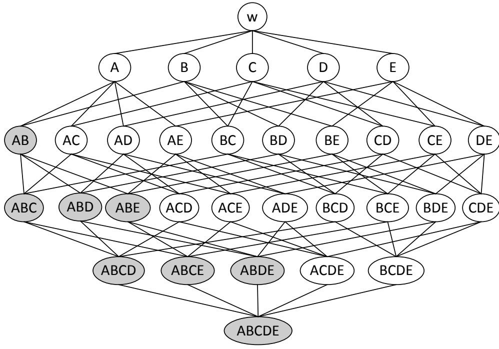


Figure 2: **Search space for the naïve dynamic programming**

Figure 2 shows the lattice of all the task sets that needs to be checked by naïve dynamic programming when dealing with the example of Figure 1. In the end $\texttt{opt}(\{A, B, C, D, E\}, D) = 4$ is one optimum answer with its corresponding valid task sequence (A, E, C, D).

3.1.2 Optimization of Naïve Dynamic Programming

Even though our proposed naïve dynamic programming is faster than the brute-force approach, it suffers from the issue that it needs to examine all the $2^n - 1$ sets of tasks in the ascending order of their sizes. As shown in Figure 2, when $S = \{A, B, C, D, E\}$, the naïve dynamic programming checks 31 sets in total. In the following, we alleviate this cost by adopting the Apirior principle [6]. Specifically, we introduce the definition of *invalid set* and further show that all supersets of any invalid set can be pruned safely. Thus, we avoid enumerating all of the sets, and hence improve the efficiency.

Definition 7. Given a worker w and a task set Q, Q is an *invalid set* if and only if none of its permutations is a valid task sequence; otherwise, Q is valid.

Let us use the same example in Figure 1 to elaborate further on *invalid set*. For instance, $\{A, B\}$ is invalid since neither (A, B) nor (B, A) are valid task sequences, whereas, $\{A, C\}$ is a valid set since (A, C) is a valid task sequence.

Based on this definition, we present the following lemma:

LEMMA 1. *If a task set is invalid, then all of its supersets must be invalid. Furthermore, all invalid sets can be safely pruned during sets generation (lines 5–6) of Algorithm 1.*

PROOF. Obvious from its definition. ■

Based on Lemma 1, we can trim the exponential search space. As illustrated in Figure 2, $\{A, B\}$ and all the supersets of $\{A, B\}$ (the shaded task sets) are invalid sets, thus can be pruned. The integration of Lemma 1 to our dynamic programming is as follows. In each iteration, instead of considering each set with the same size, we just focus on the valid sets. In particular, in the subset generation phase, we generate candidate sets with size x from the valid sets with size $x - 1$. If two valid sets with size $x - 1$ share the first $x - 2$ tasks, we join them and form a candidate set. If none of the candidate sets is generated, we can terminate the loop. Otherwise, for each candidate set Q, we use the same equation in Algorithm 1(lines 7–8) to compute table $\texttt{opt}$ and $\texttt{pre}$. In this process, for any of the task $j \in Q$, if it can be finished from any second-to-last task i by following the optimum subsequence to achieve $\texttt{opt}(Q - \{j\}, i)$ (which means $\delta_{ij} = 1$), then Q is a valid set because at least one valid task sequence exists for Q. If all of the candidate sets with size l are invalid, the loop terminates. We omit the pseudo-code here.

One drawback of the optimization strategy is that it incurs an extra cost for the generation of the valid sets. Specifically, to generate x-element candidate set, pairs of valid $(x-1)$-element set are merged

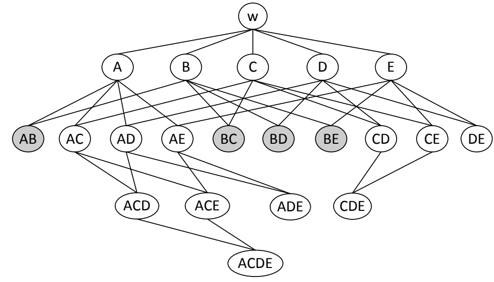


Figure 3: **Pruned search space for optimized dynamic programming**

to determine whether they have at least $x - 2$ tasks in common. In the best case scenario, every merging step produces a feasible candidate set. In the worst case, the algorithm must merge every pair of valid $(x - 1)$-element set. Thus, the overall cost of generating candidate sets is (Q_{x-1} is the $(x - 1)$-element set):

$$\sum_{x=1}^{n}(x-2)|Q_x| < \text{Cost of Generating} < \sum_{x=1}^{n}(x-2)|Q_{x-1}|^2$$

Therefore, when most of the sets are valid, the optimization strategy may not be effective because the cost of set generation surpasses its benefits.

Example: Figure 3 illustrates the corresponding sets examined after using the pruning technique and the invalid sets are shaded. Initially, all sets with one task such as $\{A\}, \{B\}$ are valid. Subsequently, it checks the candidate sets with 2 tasks and finds that $\{A, B\}$, $\{B, C\}$, $\{B, D\}$ and $\{B, E\}$ are invalid, and hence be discarded in the next iteration. Next, it generates the 3-element sets using only the remaining six 2-element sets. This is because Lemma 1 ensures that all the supersets of the invalid 2-element sets must also be invalid. Consequently, four valid 3-element sets are generated and only two of them can be used to form 4-element subset. Finally, $\{A, C, D, E\}$ is generated from $\{A, C, D\}$ and $\{A, C, E\}$, then the program terminates. With the pruning using Lemma 1, only $5 + 10 + 4 + 1 = 20$ task sets are examined instead of 31, which reduces the search space by 35%.

3.2 Branch-and-Bound Algorithm

In this section, we propose a branch-and-bound algorithm to compute the exact solution of MTS. Assume that the search space of a branch-and-bound algorithm is represented as a tree, then the general idea is to conduct a depth-first search on this search tree but with a carefully designed pruning strategy. In particular, for each node of the search tree we maintain a candidate task set which filters out the unpromising branches, and thus reduce the search space. Moreover, the size of the candidate task set can be used to derive an upper bound for ordering and pruning. Finally, we can use this candidate set to compute a lower bound as another ordering and pruning metric. Figure 4 illustrates the high-level idea of our branch-and-bound algorithm: starting from the worker (i.e., the root of the search tree), at each level we branch the tasks in the candidate task set according to their ordering metrics (e.g., upper bound). The process is repeated recursively until we find a feasible solution. When backtracking to the upper level, we prune the branches whose upper bound is lower than the length of the current best solution or lower bounds of other branches. Before presenting the details of our branch-and-bound algorithm, we first discuss the merits of using candidate task set.

3.2.1 Candidate Task Set

For each node in the search tree, the candidate task set ($cand$) keeps the promising tasks to be expanded for the next level. For example, in Figure 4, for node C at Level 1, originally there are four

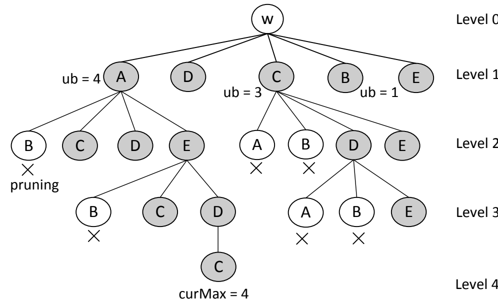


Figure 4: An overview of the proposed branch-and-bound algorithm for Figure 1

branches, i.e., $\{A, B, D, E\}$, but only D and E are promising. This is because the arriving time of A and B by following C is greater than their deadlines. Therefore, if we know the candidate task set of one node, we can focus on this smaller subset of tasks instead of blindly choosing any of the remaining branches. To efficiently calculate one node's candidate task set, we present an important property in the following:

PROPOSITION 2. *A node's candidate task set in the search tree is the subset of its parent's candidate task set.*

PROOF. This is trivial to show using the triangle inequality. ■

For example, in Figure 4, the candidate task set of node C at Level 1 contains $\{D, E\}$, its children node (C, D)'s candidate task set only contains E at Level 3, and excludes tasks A and B because we already know that task A (or B) cannot be completed after C, and thus after (C, D).

Therefore, to compute the candidate task set of a node, we go through each task in its parent's candidate task set and keep the task as promising only if it is available from the current node. Algorithm 2 outlines the computation of the candidate task set of R_s when we are branching from node R to a task s for the next level (we denote the children node of R, whose next branches is s as R_s), given R's candidate set $\texttt{cand_}R$ and $s \in \texttt{cand_}R$. The algorithm first finds R's current task and the task's arriving time (line 1). Initially, the candidate task set of R_s (i.e., $\texttt{cand_}R_s$) is empty (line 2). Subsequently, we examine each of the remaining tasks s' in $\texttt{cand_}R$ and its possibility to be finished after R_s (lines 3–5). If the arriving time of s' is less than its deadline, we add s' to $\texttt{cand_}R_s$.

3.2.2 Using the Candidate Task Set for Pruning

The candidate task set not only directs our search space into a much smaller and more promising subset, but also helps to derive an upper bound (ub) for that node. The upper bound represents the maximum number of tasks that can be finished by following the corresponding branch. Formally, node R's upper bound ub_R can be computed using the following equation:

$$ub_R = |R| + |\texttt{cand_}R| \tag{2}$$

where $\texttt{cand_}R$ denotes the candidate task set of node R.

Intuitively, the upper bound is the number of already finished tasks by arriving at R plus the maximum possible number of tasks that can be completed after R. Let us consider the same example in Figure 4, for node C at Level 1, its candidate task set is $\{D, E\}$, thus its upper bound is $1 + 2 = 3$. Similarly, for node B at Level 1, its upper bound is 1 since its candidate task set is empty. With this upper bound, one branch can be pruned if its bound is less than the length of the current best solution found so far. Specifically, we have:

LEMMA 2. *Assume that the length of the best known solution found so far (local best) is $curMax$ and the current branch is R, then R can be pruned if and only if $ub_R \leq curMax$.*

Algorithm 2 Calculate_Cand(R, $\texttt{cand_}R$, s)

Input: R is the current task sequence, $\texttt{cand_}R$ is the current candidate set for expansion, $s \in \texttt{cand}_R$ is the next task to be expanded.
Output: A candidate task set $\texttt{cand_}R_s$ for node R_s.

1: get curTask and its arriving time curTime of R
2: $\texttt{cand_}R_s \leftarrow \emptyset$
3: **for** each task $s' \in \texttt{cand_}R \setminus s$ **do**
4:     **if** curTime $+c(s, s') \leq d(s')$ **then**
5:         $\texttt{cand_}R_s \leftarrow \texttt{cand_}R_s \cup s'$
6: **return** $\texttt{cand_}R_s$

PROOF. Obvious from the definition. ■

For example, in Figure 4, by following $A \to E \to C \to D$, we obtain the current best valid task sequence with length 4. Then, if we are back to the node C at Level 1, since its upper bound is 3 which is less than $curMax$, we can safely skip this branch.

3.2.3 Branching Strategy

Clearly, the larger the value of $curMax$, the stronger the pruning power of Lemma 2. For example, in Figure 4, suppose that node B at Level 1 is visited first, which produces a local best $curMax$=1; then Lemma 2 is not efficient in terms of pruning since none of the other branches' upper bounds are greater than one. On the contrary, assume that initially we choose node A at Level 1, we can find a local best solution with 3 tasks finished and thus $curMax$=4. The other branches such as B, C, D, E can be pruned accordingly. Consequently, in the search tree, if nodes which leads to a larger local best $curMax$ can be accessed in order, then more nodes can be pruned using Lemma 2. Unfortunately, it is impossible to know the local best $curMax$ in advance. Intuitively, a node with higher upper bound is more likely to result in a better solution and larger values of $curMax$. Towards this end, at each level of the search tree, we visit the nodes in the decreasing order of their upper bounds.

Even though branching from a node with larger upper bound has a higher likelihood of obtaining a larger value of $curMax$, there is no guarantee. As a result, we further propose the other ordering metric: the lower bound of a node (lb), which denotes the minimum number of tasks that can be finished by following this node. Intuitively, the upper bound is the most optimistic choices possible, while the lower bound results in the most pessimistic ordering. In addition, the lower bound can also be used for pruning. Specifically, if the upper bound of one node is less than the lower bound of any other nodes, it can be discarded safely. In order to calculate the lower bound for the node R, we first use any approximation algorithm introduced in Section 4 that computes the number of tasks that can be completed in its candidate task set. Next, we compute the lower bound as the number of tasks finished at R, plus the least number of tasks that can be finished in $\texttt{cand_}R$.

3.2.4 Algorithm and Complexity

We explain our branch-and-bound algorithm using a depth-first search. We use the upper bound as the ordering metric, and use the lower bound only for pruning. The details are outlined in Algorithm 3. Initially, we search from the root node, where the task sequence R is empty, $\texttt{cand_}R$=S (i.e., all the tasks are considered as candidates), and $curMax$ is 0, and then recursively call Algorithm 4. For each node at the next level, the algorithm first calculates its corresponding candidate task set and its upper and lower bound (lines 1–3). All the tasks in the current candidate set $\texttt{cand_}R$ are sorted in the decreasing order of their upper bounds, and the branches with its upper bound less than the lower bound of other branches are discarded in $\texttt{cand_}R$ (line 4). For each candidate task s, if the upper bound is larger than $curMax$, we continue

Algorithm 3 MST_Branch_Bound(S, w)

Input: A set of tasks S and a worker w
Output: Optimum solution R^*

1: MST_Branch_Bound_Search($\emptyset$, S, 0)

Algorithm 4 MST_Branch_Bound_Search(R, `cand_R`, $curMax$)

Input: R is the current task sequence, $\texttt{cand}_R$ is a set of candidate tasks, curMax is the length of current best solution.
Output: Optimum solution R^*

```
1: for each task s ∈ cand_R do
2:    cand_R_s ← Calculate_Cand(R, cand_R, s)
3:    calculate ub_R_s and lb_R_s
4: sort cand_R in the descending order of ub, prune branches
   based on ub and lb
5: for each task s ∈ cand_R do
6:    if ub_R_s > curMax then
7:       MST_Branch_Bound_Search(R∪s, cand_R_s, curMax)
8:    else
9:       if |R| > curMax then
10:         curMax ← |R|
11:         R* ← R
```

our search in the next level using its candidate task set `cand_`R_s (lines 6–7). Otherwise, we examine whether we need to update the current best solution (lines 9–11). In the following we discuss and compare the space and time complexity of our dynamic programming and branch-and-bound algorithm.

Space complexity: The branch-and-bound algorithm is more efficient than dynamic programming algorithm in terms of space requirements. The reason is that the recursive depth is bounded by the number of tasks n, and in each call of Algorithm 4, we only need to store R and `cand_`R with maximum size n. Therefore, the space complexity for the branch and bound algorithm is $O(n^2)$, which is much smaller than the exponential space requirement of our dynamic programming approach.

Time complexity The time complexity of the branch-and-bound algorithm is proportional to the size of the search tree. Generally speaking, with a good branching and pruning strategy, the branch-and-bound algorithm can discard significant number of unnecessary nodes and achieve much better efficiency than dynamic programming. Unfortunately, there is no tight bound as the pruning power depends on the distribution of the location of tasks and their deadlines. It is possible that the branch-and-bound algorithm searches the entire tree without eliminating any branch. Thus, the worst case time complexity of the branch-and-bound algorithm is still $O(n!)$.

Example: Figure 5 illustrates the corresponding search space of our branch-and-bound algorithm for solving the problem of Figure 1. For each task A, B, C, D and E, at the first level, it first computes the candidate task sets and their upper bounds. The node B is pruned since its upper bound is 1, which is lower than A's lower bound. Subsequently A is searched first since its upper bound is 4. At the second level, we only check tasks $C, D, E \in$ `cand_(A)`. Similarly, after calculating their candidate task sets and upper bounds, we consider branch (A, E) because its bound is largest among the three branches. We continue our search until we reach a candidate solution (A, E, C, D) and update the value of $curMax$ to 4. Subsequently, we observe that all of the remaining branches can be pruned based on Lemma 2.

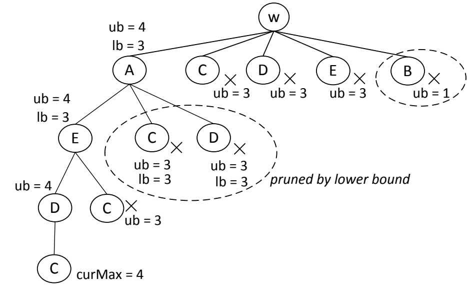


Figure 5: An illustration of the branch-and-bound algorithm

4. APPROXIMATION ALGORITHMS

Considering the real-world application scenario and the resource limitations of mobile platform in spatial crowdsourcing, we would like to achieve faster response time as well as less memory consumption. However, the time complexity of both dynamic programming and branch-and-bound algorithms grow exponentially as the number of tasks grows. Therefore, in this section, we present three approximation algorithms based on three different heuristics and a class of progressive algorithms to enable real-world applications.

Algorithm 5 Least_Expiration_Heuristic(S, w)

Input: A set of tasks S and worker w.
Output: Optimum solution R^*.

```
1: R* ← ∅, curTime ← 0, last ← w
2: sort S in the ascending order of tasks' expiration time
3: for each task s ∈ S do
4:    if curTime +c(s, last) ≤ d_s then
5:       R* ← R* ∪ s
6:       curTime ← curTime +c(s, last)
7:       last ← s
8: return R*
```

4.1 Least Expiration Time Heuristic (LEH)

The main idea of LEH is to form a task sequence by greedily choosing the task with the least expiration time. We explain LEH in Algorithm 5. Initially a worker w starts from time unit 0 with an empty task sequence R^* (line 1). The tasks in S are sorted in the increasing order of their expiration time (line 2). Subsequently, in each iteration, the current task s with the least expiration time is examined. If s can be completed, we add it to the current task sequence R^* (line 5), and update the last task in R^* as well as its arrival time (lines 6–7); otherwise, we continue to examine the next task in S. Consider the example in Figure 1, with LEH, the worker w chooses task B to start with since it is the most "urgent" with expiration time 8. Then he will consider A, E, C and D respectively based on their expiration times. Unfortunately, none of these tasks can be completed afterwards and thus only B is returned to the worker.

4.2 Nearest Neighbor Heuristic (NNH)

NNH exploits the spatial proximity between the tasks by iteratively choosing the nearest available task to the last task added in the task sequence. For example, in Figure 1, worker w initially chooses task A which is nearest to him. Next task B is considered since B is the closest task to A. However, B cannot be reached on time, hence A's second nearest task E is checked. We find E is available and add it to the task sequence. Subsequently task D is examined and added to the task sequence. In the end we find task C cannot be completed after D, thus task sequence (A, E, D) is returned. We omit the pseudo-code here.

4.3 Most Promising Heuristic (MPH)

The third approximation algorithm is MPH, which is based on

the branch-and-bound algorithm presented in Section 3.2. Like branch-and-bound, MPH iteratively chooses the most promising branches (i.e., nodes with the highest upper bound at the same level); however, instead of exploiting the entire search tree, MPH terminates when the first candidate task sequence is found. For instance, consider the search tree of the branch and bound algorithm in Figure 5, by following the most promising branches, it retrieves the first candidate task sequence (A, E, C, D). With MPH, the search stops here and task sequence (A, E, C, D) is returned. We omit the pseudo-code here.

4.4 Progressive Algorithms

In real-world applications, sometimes it is sufficient to report a small number of spatial tasks to a worker quickly and then continue to solve the remaining problem off-line, i.e., the complete solution is computed in the background while the worker is performing the initial tasks. In response to it, we present a class of progressive algorithms. Given a worker and a set of tasks, the idea is to use any of the approximation algorithms (e.g., NNH) to identify a small number of initial tasks quickly. Subsequently we use the exact branch and bound algorithm to find the optimum task sequence for the remaining tasks.

The advantage of the progressive algorithms is obvious: quicker response time as compared with the exact algorithms, and higher accuracy as compared with the approximation algorithms. However, there are several drawbacks. One drawback is that a worker's potential tasks may be preempted by other workers when the worker is on the way to process the initial tasks. In addition, workers may prefer to see the entire task sequence before starting to work.

5. EXPERIMENTS

5.1 Experimental Setup

Datasets. We conducted our experiments with both synthetic (SYN) and real (REAL) data. For synthetic data generation, we used two distributions: uniform (SYN-UNIFORM) and skewed (SYN-SKEWED). In order to generate SYN-SKEWED data set, 99% of the tasks were generated into four Gaussian clusters (with $\delta = 0.05$ and randomly chosen center) and the other 1% of the tasks were uniformly distributed. Given a worker and his region, we varied the average number of spatial tasks inside his spatial region, denoted by tasks per worker (T/W), from 10 to 40. In addition, given a worker and a set of tasks,the expiration time of the tasks was generated as follows: starting from the worker's location, we greedily chose the next nearest task to form a task sequence and got the total travel cost t, which was used as the upper bound for expiration time generation. Subsequently we defined a range $[d^l, d^u]$, where $0 < d^l < d^u < 1$, and the expiration time was generated from the uniform distribution $[d^l \cdot t, d^u \cdot t]$. With SYN, we used 5 pairs of values for d^l and d^u, which were $[0.2, 0.3]$, $[0.3, 0.4]$, $[0.4, 0.5]$, $[0.5, 0.6]$ and $[0.6, 0.7]$. Basically range $[d^l, d^u]$ determines the percentage of the tasks that can be completed.

The real data was obtained from Gowalla [4], a location-based social network, where users are able to check in at different locations in their vicinity. The check-ins include the location and the time that the users enter the locations. For our experiments, we used the check-in data over a period of one month (i.e., August, 2010). We defined the tasks as the locations of restaurants, in the area of Los Angeles, CA. For each day during that month, we found all the check-ins within a time range (e.g., three hours) from different uses and removed the duplicate tasks for the same user. The remaining check-ins were used as spatial tasks. For each check-in, we used its location and time as the location and expiration time of the task. Intuitively checking in a spot is equivalent to finishing a spatial task at that location. The travel cost was calculated by the Euclidian distance divided by the average travel speed (i.e., 30 miles/hour). The user with the earliest check-in time in one time range was treated as the spatial worker.

Algorithms. We compared the performance for both exact and approximation algorithms. Specifically, we consider three exact solutions namely dynamic algorithm (DA), dynamic algorithm with optimization strategy (DA_OPT) and branch-and-bound algorithm (B&B), and three approximation algorithms, namely nearest neighbor heuristic (NNH), least expiration time heuristic (LEH) and most promising heuristic (MPH).

Besides, we studied the performance of the progressive algorithm. We consider NNH-1 (NNH-2, NNH-3) which uses nearest neighbor heuristic to return one (two, three) task(s) to the worker initially, and then uses the exact branch-and-bound algorithm for the remaining tasks.

Configuration and Measures. We evaluated the scalability of the algorithms by varying both the number of tasks per worker (T/W) and range $[d^l, d^u]$ for tasks' expiration time generation. For each of the experiments, we ran 50 cases and reported the average of the results. The CPU cost (in milliseconds) was reported [3]. In addition, for the approximate algorithms, we also reported their accuracies (i.e., approximation ratio).

All experiments were run on an Intel Core i5-2400 CPU @ 3.10G HZ with 8 GB RAM.

5.2 Experiments on Synthetic Data Sets

5.2.1 Effect of Number of Tasks Per Worker(T/W)

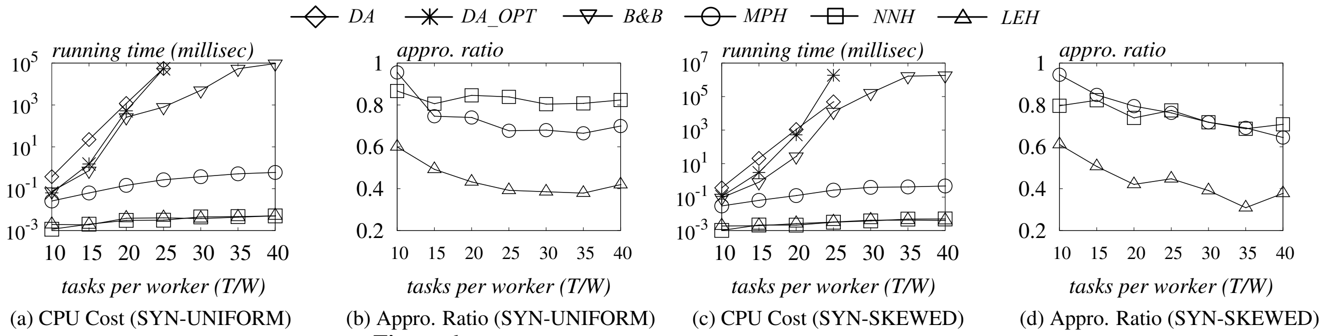
In the first set of experiments, we evaluated our approaches by varying the number of tasks per worker (T/W) with range [0.3, 0.4] for the expiration time generation.

Efficiency of Different Algorithms. Figure 6(a) shows the runtime (i.e., response time) of all of the algorithms on SYN-UNIFORM. As expected, the running time of all three exact algorithms is much slower than those of approximation algorithms. In addition, we observe the runtime of the exact algorithms increases exponentially as T/W increases. Among them, DA is the slowest since it enumerates the entire subsets of tasks. DA_OPT is faster than DA because it avoids examining the invalid subsets. When T/W is larger than 25, both DA and DA_OPT are very time-consuming within more than hours response time, thus we do not report their results here. B&B performs better than both DA and DA_OPT. It demonstrates the usefulness of the pruning and ordering strategies of B&B. However, note that in real-world applications the mobile platforms have limited resources, which is much inferior than our experimental platform. Moreover, in general, waiting for the answer more than 300 ms makes users feel non-interactive. As our experiment setting, more than 100 ms response time could make it non-interactive environment in practical scenario. Therefore, B&B cannot scale either and it turns out that the exact algorithms are only applicable when T/W is relatively small (i.e., less than 20).

The runtime of the approximation algorithms increases almost linearly with the increase of T/W. For NNH (LEH), it keeps searching the next available tasks with nearest distance (least expiration time) until no tasks are left, hence it is very efficient. For MPH, it runs relatively slower than NNH and LEH because before choosing the next task, it needs to calculate the candidate task set, then choose the next branches based on their upper bounds.

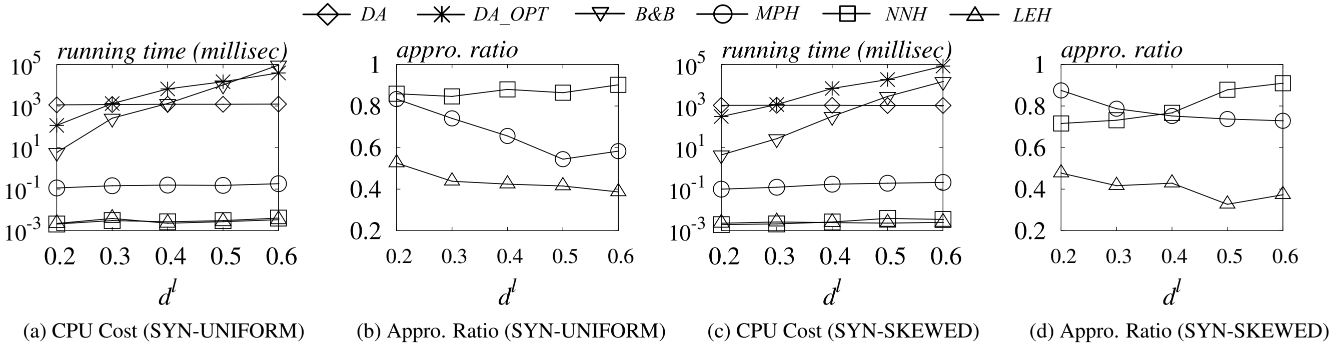
Accuracy of Approximation Algorithms. Figure 6(b) shows the accuracy of the three approximation algorithms on SYN-UNIFORM. The accuracy of different approximation techniques varies significantly. NNH achieves the best accuracy and LEH performs worst.

[3]For progressive algorithms we did not report the response time since it was not the concern.



(a) CPU Cost (SYN-UNIFORM) (b) Appro. Ratio (SYN-UNIFORM) (c) CPU Cost (SYN-SKEWED) (d) Appro. Ratio (SYN-SKEWED)

Figure 6: Effect of T/W on synthetic data with range [0.3, 0.4]



(a) CPU Cost (SYN-UNIFORM) (b) Appro. Ratio (SYN-UNIFORM) (c) CPU Cost (SYN-SKEWED) (d) Appro. Ratio (SYN-SKEWED)

Figure 7: Effect of range $[d^l, d^u]$ on synthetic data ($d^u = d^l + 0.1$) with T/W =20

Table 1: Avg. No. of completed tasks by the exact and progressive algorithms of varying T/W on SYN-UNIFORM

Alg. \ T/W	10	15	20	25	30	35	40
Exact Alg.	4.5	6.7	10.4	13	15.3	18.2	19.3
NNH-1	4.5	6.6	10.3	13	15.2	17.6	18.6
NNH-2	3.6	6.2	10.3	12.9	14.9	17.2	18.4
NNH-3	3.5	5.7	10.3	12.3	14.9	17	17.8

Table 2: Avg. No. of completed tasks by the exact and progressive algorithms of Varying T/W on SYN-SKEWED

Alg. \ T/W	10	15	20	25	30	35	40
Exact Alg.	5.4	7.9	10.7	14.3	18.4	19.85	22.5
NNH-1	5.4	7.4	10.7	14.3	18.3	18.5	21.75
NNH-2	4.9	7.3	10.5	14.3	17.7	17.2	21.2
NNH-3	4.1	7	9.3	14	16.8	16.4	21

The reason is that LEH does not consider the spatial proximity of the tasks, the worker may miss many tasks located far from the most "urgent" task he chooses. We also list the number of completed tasks by the exact algorithm on SYN-UNIFORM in Table 1, from which we can observe the difference between the exact and approximation algorithms in terms of number of completed tasks. Taking T/W = 25 as an example, the exact algorithms can complete 13 tasks on average. Thus, the worker is able to complete almost 3 more tasks (i.e., 13×20%, the best ratio is around 80% in Figure 6(b)) than the approximation algorithm. Therefore, the difference between exact and approximation algorithms might still be significant when T/W increases.

Accuracy of Progressive Algorithms. Table 1 shows the number of completed tasks by the progressive algorithms on SYN-UNIFORM. NNH-1, NNH-2 and NNH-3 all achieve near-optimum task number, hence they are superior than the approximation algorithms in terms of accuracy. In addition, NNH-1 performs better than NNH-2 and NNH-3. This is because the more tasks fetched at the beginning, the more deviation from the optimum task sequence.

Experiments on SYN-SKEWED. The set of experiment studied the efficiency and accuracy of our algorithms on SYN-SKEWED when T/W varies. Figure 6(c) shows the runtime and Figure 6(d) depicts the accuracy. Table 2 depicts the number of completed tasks by the exact and progressive algorithms. The results are qualitatively similar with that of on SYN-UNIFORM. From Tables 1 and 2, we observe that the average number of tasks that can be completed in the skewed distribution is greater than that of the uniform distribution. This is because in skewed data, many tasks resides in one cluster which increases the possibilities to be completed before their expiration. With more tasks completed in SYN-SKEWED, it also affects the running time of the exact algorithms. For example, DA_OPT performs much worse than on SYN-UNIFORM. The reason is that the valid sets in DA_OPT increases significantly with more tasks completed, which makes the cost of valid set generation surpasses its benefit. With respect to accuracy, MPH achieves better performance than on SYN-UNIFORM. This is because in uniform distribution, each task is equally promising, while in skewed distribution, naturally some tasks are more promising than the others, which makes MPH more effective. In addition, both MPH and NNH perform worse as T/W grows. Finally, the progressive algorithms achieve near optimal results as on SYN-UNIFORM and NNH-1 is better than NNH-2 and NNH-3. In the remaining section, we do not report the results of progressive algorithms as they are similar due to the stability of progressive algorithms.

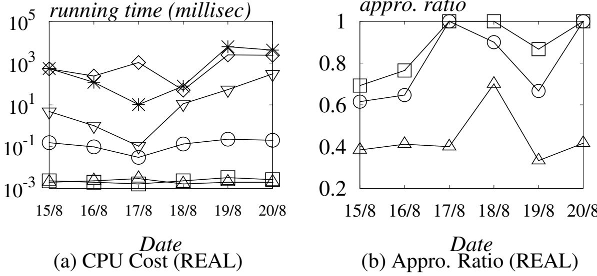
5.2.2 Effect of Range $[d^l, d^u]$

In this set of experiments, we evaluated the scalability and accuracy of our proposed approaches by varying range $[d^l, d^u]$ for the tasks' deadline generation. The number of tasks per worker (T/W) was fixed at 20.

Efficiency of Different Algorithms. With this set of experiment, we studied the efficiency as range $[d^l, d^u]$ varies. Figure 7(a) illustrates the runtime of our approaches on SYN-UNIFORM. Clearly, this range has significant influence on the runtime of DA_OPT and B&B, whereas has little influence on DA. When d^l grows, the dead-

Table 3: **Avg. No. of completed tasks by the exact algorithms of varying d^l ($d^u = d^l + 0.1$, W/T = 20)**

Data set \ d^l	0.2	0.3	0.4	0.5	0.6
SYN-UNIFORM	7.8	10.4	12.5	14.7	16.3
SYN-SKEWED	8.8	10.8	13.3	15.6	17.7



(a) CPU Cost (REAL) (b) Appro. Ratio (REAL)

Figure 8: **Real Data**

line of tasks increases, and hence the number of tasks that can be completed also increases. Table 3 shows this trend. For DA_OPT more tasks that can be completed results in a significant growth of the number of valid sets, which makes the additional cost incurred for valid sets generation surpasses its benefit. Therefore, the performance of DA_OPT degrades significantly as d^l grows. As shown in Figure 7(a), the runtime of B&B increases almost exponentially. The reasons is twofold. First it takes longer time for B&B to find a good candidate solution with more tasks that can be completed as $curMax$. In addition, this increase leads to more candidate branches that needs to be explored. However, the running time of DA stays almost the same as range increases. This is because DA always enumerates all the subsets of tasks independent of the range. As expected, the approximation algorithms run much faster than the exact algorithms.

Accuracy of Approximation Algorithms. Figure 7(b) depicts the efficiency of the three approximation algorithms when varying d^l on SYN-UNIFORM. As in the the previous set of experiments, LEH performs worst and NNH performs better than MPH. It is interesting to observe that on SYN-SKEWED, NNH performs better as d^l increases, whereas MPH performs much worse. One reason is that with MPH the increased deadline results in all the remaining tasks with similar higher upper bound, which makes them become equally promising. It renders MPH to randomly choose the tasks, thus affects the effectiveness. However, for NNH, larger number of completed tasks generally means more available choices for the worker and more accurate results.

Experiments on SYN-SKEWED. Figure 7(c) shows the runtime and Figure 7(d) depicts the accuracy. The results of SYN-SKEWED are similar to those of SYN-UNIFORM except MPH. We notice MPH achieves higher accuracy than on SYN-UNIFORM, as similar to Figure 6(b) and Figure 6(d). Another observation is that MPH performs better than NNH when d^l is small but worse than NNH when d^l becomes larger (They perform almost equally when d^l =0.4). The reason is twofold. First, when d^l is small, MPH performs better due to the effectiveness of its upper bound on SYN-SKEWED. Besides, as d^l increases, MPH performs significantly worse but NNH achieves better accuracy.

5.3 Experiments on Real Data Set

Figure 8 shows our experiment results on real data set for a period from Aug/15/2010 to Aug/20/2010. For this real data set, the average number of tasks per worker is around 20 and the range of expiration time is around $[0.2, 0.4]$, which conforms to our setting with synthetic data. Figure 8(a) depicts the efficiency and Figure 8(b) illustrates the accuracy. As expected, B&B performs better than DA and DA_OPT, which makes it more suitable for real-world applications. In terms of accuracy, NNH performs best.

6. RELATED WORK

In this section, we first review the related studies on crowdsourcing and spatial crowdsourcing. We then discuss the related work in the area of job scheduling and route planning queries.

Crowdsourcing has attracted much interest from both the industrial and research community. A recent survey can be found in [9]. With the increasing popularity, a set of crowdsourcing market platforms such as Amazon Mechanical Turk and CrowdFlower [5] have emerged, which enable human workers to perform tasks on the Internet. Crowdsourcing applications have been adopted in a wide range of applications such as image search [24], natural language annotations [22] and information retrieval [11]. Moreover, crowdsourcing has also been incorporated into database design and relational query processing [10, 12, 18].

Despite all the studies on crowdsourcing, not as much work has been done in spatial crowdsourcing [7, 13, 15]. Kazemi and Shahabi [15] defined a maximum task assignment problem under SAT mode. In [7], a crowdsourcing system with the WST mode was proposed, which integrates location into the processing of distributing tasks. Besides, participatory sensing [8,13,14,19], a particular type of WST based spatial crowdsourcing has been studied to involve human workers to perform sensor-dependent tasks. Among these work, both Cartel [13] and Nericell [19] have used GPS-enabled phones mounted on vehicles to collect information about traffic, the WiFi access points on the route and road condition. Frameworks under WST mode [8] and SAT mode [14] have been proposed in participatory sensing. However, none of these existing studies considers the influence of the worker's travel cost on task's completion, which is critical in spatial crowdsourcing. Furthermore, existing works on participatory sensing cannot be generalized to any type of spatial crowdsourcing.

MTS can be formulated as an instance of a job-scheduling problem, which incorporates a job setup cost that is sequence dependent, called DCS (dual criteria scheduling with setup cost). With DCS, the objective is to find a schedule that maximizes both the number of completed jobs and the total completion time. With MTS, we can consider the travel time between two tasks as the setup cost of the latter task in DCS. In [16], Lee et. al proposed a genetic programming (GA) approach to solve DCS. First, the GA approach is an overkill for our problem setting because the number of tasks per worker is small. Second, with MTS we need to provide a schedule for the worker in millisecond, which is different than solving DCS as a one time optimization problem. Finally, we can exploit the spatial property of MTS, i.e., the fact that the costs are actual travel times depending on the location of tasks, to solve this problem more efficiently.

MTS is also related to the route planning queries in the area of spatial databases. Sharifzadeh et al. [21] addressed Optimal Sequenced Route (OSR) Query. Given a source location, a number of points with different types and a particular order imposed to visit these types, OSR aims to find a route of minimum length passed through the locations as the specified sequence. Besides, Li et al. [17] studied Trip Planning Queries (TPQ) and Terrovitis et al. [23] addressed Constrained Shortest Path problem (CSP). Given a source and a destination, TPQ asks for a route with minimum length which passes through a subset of these location types (not the strict sequence order), while CSP seeks to find a shortest path which passes through exactly k intermediate points (no constraint on location types). The main difference between these studies and

our problem is that with MTS we want to maximize the number of tasks that can be completed, whereas, these route planning queries aim to minimize the total travel cost.

7. CONCLUSION

In the context of spatial crowdsourcing, we introduced a novel problem, termed Maximum Task Scheduling (MTS), to maximize the number of spatial tasks performed by a worker. We proved that MTS is NP-hard and for which we proposed several exact, approximate and progressive algorithms. Extensive experiments on both real-world and synthetic datasets offered us insights on how to apply these algorithms in practical scenarios.

There are a number of promising directions for future work. First, we intend to develop algorithms to optimize dual criteria, namely, the number of tasks that can be completed and the total travel cost of the worker. Moreover, we plan to consider other properties of spatial tasks, for instance, the processing time and priority of the spatial tasks. Finally, we would like to extend MTS to the SAT mode while addressing the privacy issues.

Acknowledgments. This research has been funded in part by NSF grants IIS-1115153 and IIS-1320149, a contract with Los Angeles Metropolitan Transportation Authority (LA Metro), the USC Integrated Media Systems Center (IMSC), HP Labs and unrestricted cash gifts from Google, Northrop Grumman, Microsoft and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the National Science Foundation or LA Metro.

8. REFERENCES

[1] http://picasa.google.com.
[2] http://www.waze.com.
[3] http://www.openstreetmap.org/.
[4] http://snap.stanford.edu/data/loc-gowalla.html.
[5] http://crowdflower.com/.
[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. VLDB '94, pages 487–499, San Francisco, CA, USA, 1994.
[7] F. Alt, A. S. Shirazi, A. Schmidt, U. Kramer, and Z. Nawaz. Location-based crowdsourcing: extending crowdsourcing to the real world. NordiCHI '10, pages 13–22, NY, USA, 2010.
[8] M. Bulut, Y. Yilmaz, and M. Demirbas. Crowdsourcing location-based queries. In *PERCOM Workshops*, pages 513–518, 2011.
[9] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.
[10] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. SIGMOD '11, pages 61–72, NY, USA, 2011.
[11] C. Grady and M. Lease. Crowdsourcing document relevance assessment with mechanical turk. NAACL HLT '10, pages 172–179, PA, USA, 2010.
[12] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. SIGMOD '12, pages 385–396, NY, USA, 2012.
[13] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden. Cartel: a distributed mobile sensor computing system. SenSys '06, pages 125–138, NY, USA, 2006.
[14] L. Kazemi and C. Shahabi. A privacy-aware framework for participatory sensing. *SIGKDD Explor. '11*, 13(1):43–51.
[15] L. Kazemi and C. Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *SIGSPATIAL '12*, pages 189–198, NY, USA, 2012.
[16] S. M. Lee and A. A. Asllani. Job scheduling with dual criteria and sequence-dependent setups: mathematical versus genetic programming. *Omega*, 32(2):145 – 153, 2004.
[17] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. SSTD'05, pages 273–290, Berlin, Heidelberg, 2005.
[18] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.
[19] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. SenSys '08, pages 323–336, NY, USA, 2008.
[20] J. M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):pp. 102–109, 1968.
[21] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *The VLDB Journal*, 17(4):765–787, July 2008.
[22] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. EMNLP '08, pages 254–263, PA, USA, 2008.
[23] M. Terrovitis, S. Bakiras, D. Papadias, and K. Mouratidis. Constrained shortest path computation. In *SSTD'05*, volume 3633, pages 181–199. 2005.
[24] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones. MobiSys '10, pages 77–90, NY, USA, 2010.

APPENDIX

A. PROOF OF THEOREM 1

Proof (Sketch): We show that Ham-Path $\leq_p$ sTSP. The Ham-Path problem $< G >$ decides whether there exists a simple path in G which visits each vertex exactly once. Let $G(V, E)$ be an instance of Ham-Path with n vertices, we construct an instance of sTSP as follows: first, we form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function c by

$$c(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E \\ 2 & \text{if } (i,j) \notin E \end{cases}$$

Next we add a new vertex u' in G', and for each $i \in V$, add (u', i) with edge cost 1. Now we have a complete graph $G' = (V', E'')$ where $V' = V \cup \{u'\}$ and $E'' = E' \cup \{(u', i) : i \in V\}$. We choose $x = u'$ and $k = n + 1$, it is easy to see $k \geq 2 \cdot c(x, i)$ for $i \in V'$ and $i \neq x$.

We now show that graph G has a hamiltonian path if and only if graph G' has a tour starting from u' with cost of at most $n + 1$. Suppose that graph G has a hamiltonian path $p = (u, \cdots, v)$ which has $n - 1$ edges. Each edge in p belongs to E and thus p has cost $n - 1$. If we connect u' to path p by adding edges (u', u) and (v, u') in G', we can form a tour h' with cost $n + 1$ since $c(u', u) = 1$ and $c(v, u') = 1$. Conversely, suppose that graph G' has a tour h' start from u' with cost less than $n + 1$. Since edge cost in E' is either 1 or 2, the cost of tour h' which contains $n + 1$ edges must be exactly $n + 1$. Therefore, by excluding u' in h' we can form a hamiltonian path that contains only edges in E. It completes our proof. ■

B. PROOF OF THEOREM 2

Proof (Sketch): First we show that the decision problem of MTS is in NP. Given any task sequence R, it is easy to decide the number of finished tasks in R by calculating the arrive time of each task. This process can be done in polynomial time.

Next we show that the proof can be established by a reduction from the sTSP problem. Consider an instance of sTSP$< G, c, x, k >$ problem, let $G(V, E)$ be a complete graph with $n + 1$ vertices, given starting vertex $x = 0$ and cost k, where $k \geq 2 \cdot c(0, i)$ for $i \in V$ and $i \neq 0$, now we construct an instance of MTS$< w, S, z >$ as follows. Let vertex 0 represents the worker w, vertices $1, 2, \cdots, n$ represent n tasks. The travel cost from i to j defined as $c(i, j)$ is the same as sTSP. The deadline of each task i is $d(i) = k - c(0, i)$, which guarantees that the worker can arrive at each task before its deadline initially since $d(i) \geq c(0, i)$. Finally we choose $z = n$.

We now show that sTSP $< G, c, x, k >$ has a tour of cost at most k if and only if MTS $< w, S, z >$ has a valid task sequence R, st. $|R| = n$, which means all the tasks can be completed on time. Suppose we have a tour in G starting from vertex 0 with cost less than k, which is $(0, v_1, v_2, \cdots, v_n)$ and its last edge $(v_n, 0)$, we know $c(0, v_1) + \sum_{j=1}^{n-1} c(v_j, v_{j+1}) + c(v_n, 0) \leq k$. Next we prove all tasks can be finished on time by induction. First consider the simple case: for the last task v_n, its arrive time $a(v_n) = c(0, v_1) + \sum_{j=1}^{n-1} c(v_j, v_{j+1}) \leq k - c(v_n, 0) = d(v_n)$, which means v_n can be finished on time. Then suppose $v_m (m < n)$ can be finished, we prove v_{m-1} can also be finished. Since v_m can be finished, we know $a(v_m) = a(v_{m-1}) + c(v_{m-1}, v_m) \leq k - c(v_m, 0)$, from the triangle inequality, $c(v_{m-1}, v_m) + c(v_m, v_0) \geq c(v_{m-1}, v_0)$. Combining these, we can get $a(v_{m-1}) \leq k - c(v_{m-1}, v_0) = d(v_{m-1})$, which means v_{m-1} can be finished on time. Therefore if we have a tour of cost k, we also have a valid task sequence $R = (v_1, v_2, \ldots, v_n)$ that contains n tasks.

Conversely, if we have a valid task sequence R with n tasks, suppose i is the last task in R, we know the cost at i is at most $k - c(0, i)$. Thus, We can get a tour for sTSP starting from 0, following the vertices according to the orders in R and returning back to 0. It is easy to see the tour's cost is less than k. This completes our proof. ■