Oriented Online Route Recommendation for Spatial Crowdsourcing Task Workers*

Yu Li, Man Lung Yiu, and Wenjian Xu

Department of Computing, Hong Kong Polytechnic University,Hong Kong
{csyuli, csmlyiu, cswxu}@comp.polyu.edu.hk

Abstract. Emerging spatial crowdsourcing platforms enable the workers (i.e., crowd) to complete spatial crowdsourcing tasks (like taking photos, conducting citizen journalism) that are associated with rewards and tagged with both time and location features. In this paper, we study the problem of online recommending an optimal route for a crowdsourcing worker, such that he can (i) reach his destination on time and (ii) receive the maximum reward from tasks along the route. We show that no optimal online algorithm exists in this problem. Therefore, we propose several heuristics, and powerful pruning rules to speed up our methods. Experimental results on real datasets show that our proposed heuristics are very efficient, and return routes that contain 82–91% of the optimal reward.

1 Introduction

Spatial crowdsourcing platforms[1] [2] publish crowdsourcing tasks that are associated with rewards and tagged with spatial / temporal attributes (e.g., location, release time and deadline). To complete a task, a worker must reach the task's location before its deadline. Popular tasks include taking photos, reporting activities / accidents, and verifying data on-site, etc.

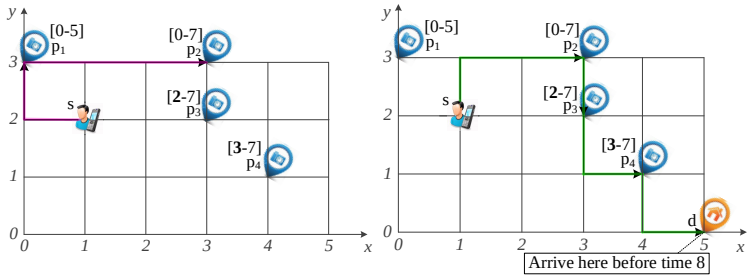
Regarding the matching between tasks and workers, existing approaches on spatial crowdsourcing can be divided into: (i) the *server-centric* mode [15,16], where the server assigns tasks to workers based on their reported locations / regions, or (ii) the *worker-centric* mode [3,7,10], where the server publishes its tasks and let workers to choose any task freely. In this paper, we adopt the worker-centric mode as it protects the location privacy of the worker [10] and enables the worker to choose tasks autonomously from different crowdsourcing platforms which he has registered in.

The closest work to ours is the *maximum task scheduling* (MTS) problem [10]. It returns a route that covers the maximum number of tasks (in a worker's specified region, e.g., his city). Since [10] considers the MTS problem at a snapshot, it would not update the worker's route when new tasks arrive. We illustrate it in Figure 1a. Assume that we use the Manhattan distance and each grid takes a time unit to travel. Each task p_i is tagged with its release time and deadline. Suppose that the worker starts from s at time 0. The MTS route is $s \rightarrow p_1 \rightarrow p_2$. The solution in [10] would not update the route when new tasks are released (e.g., p_3, p_4).

* The research is partly supported by grant GRF 152201/14E from Hong Kong RGC.

[1] www.clickworker.com/en/mobile-crowdsourcing

[2] features.en.softonic.com/mobile-crowdsourcing-does-it-work



(a) snapshot route by MTS [10] (b) online route by our method

Fig. 1. Route recommendation for the worker: each task p_i with [release time - deadline]

In this paper, we wish to support two extra requirements compared to [10]: ($\mathcal{R}1$) update the worker's route online with respect to newly released tasks and ($\mathcal{R}2$) align with the worker's trip, i.e., reaching a destination before expected time. It is important to support $\mathcal{R}1$ in order to assign a worker as many tasks as possible. New spatial crowdsourcing tasks are indeed being released continuously in real systems[3]. We also consider the requirement $\mathcal{R}2$ as the worker may have planned his own activities, e.g., reaching a specified destination by an expected time [17]. Such worker is willing to take crowdsourcing tasks along his trip provided that he can arrive at his destination on time.

To this end, we study the online route recommendation problem for spatial crowdsourcing workers, by taking requirements $\mathcal{R}1$ and $\mathcal{R}2$ into consideration. Figure 1b illustrates the route recommended by our method. Suppose that the worker starts from s at time 0 and plans to arrive at home $(5, 0)$ at time 8. At time 0, the worker is recommended to take the task p_2. When new tasks are released (e.g., p_3, p_4), the worker is recommended to take them. In summary, our recommended route is $s \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow d$, which covers 3 tasks and reaches the destination d on time.

To the best of our knowledge, this paper is the first on tackling the online route recommendation problem for spatial crowdsourcing workers with destination and arrival time constraints. We contribute the followings:

- We show that no algorithm can achieve a non-zero competitive ratio [2] in our online problem, meaning that the number of tasks found by any online algorithm may be arbitrarily small compared to the optimal offline solution.
- We propose two categories of heuristics (GetNextTask and Re-Route) that offer trade-offs between the response time and the number of tasks. GetNextTask greedily selects the next task to complete so it incurs a short response time. On the other hand, Re-Route produces a route with more tasks as it conducts a complete search to update the optimal route with respect to newly released tasks.
- We further propose pruning rules to reduce the response time of Re-Route.

Experiments on real datasets show that our methods take less than 1 second to update the route, and return routes that contain 82–91% of the optimal number of tasks.

The remainder of this paper is organized as follows. We formally define our problem in Section 2. Then, we illustrate our proposed heuristics in Section 3 and present

[3] www.clickworker.com/en/clickworkerjob
www.lionbridge.com

optimization techniques in Section 4. In Section 5, we test the performance of our proposed techniques on both real and synthetic datasets. Section 6 highlights the related work. Finally, we conclude our paper in Section 7.

2 Problem Statement

We first introduce some terminology and then define our problem formally.

Definition 1 (Task p). *We denote a task by $p_{sid,kid} = (loc, [t_p^-, t_p^+])$, where loc is the task's location, t_p^-, t_p^+ are the release time and deadline of the task, respectively. The subscripts sid and kid denote the task's server ID and task ID, respectively. A worker may complete p and collect the reward*[4] *if he can reach $p.loc$ before t_p^+.*

Definition 2 (Query q). *We denote a query q by $q = (s, d, [t_q^-, t_q^+])$. s and d are the worker's start and destination locations, respectively. t_q^- and t_q^+ are the start time from s and expected arrival time at d, respectively.*

Definition 3 (Travel Time τ). *We denote the travel time as $\tau(v, u) = \frac{dist(v,u)}{speed_q}$, where $dist(v, u)$ is the distance*[5] *between v and u, and $speed_q$ is the (constant) travel speed of the worker for q. $\tau(R)$ denotes the travel time along a route R (via vertices on R).*

With the above terminology, we are ready to define our problem formally below.

*Problem 1 (Oriented Online Route Recommendation (*OnlineRR*)).* Let a worker's query be $q = (s, d, [t_q^-, t_q^+])$. OnlineRR aims to find a route such that it covers *the maximum number of tasks* and the worker can arrive at d by t_q^+. It may update the route according to the worker's live location and the new tasks released by crowdsourcing servers.

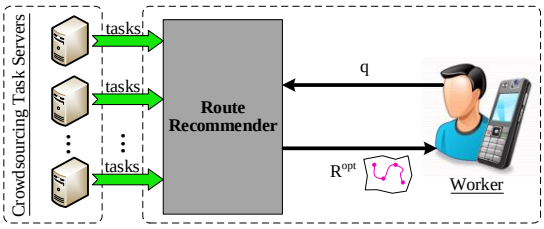


Fig. 2. System architecture

We adopt the system architecture as depicted in Figure 2. Spatial crowdsourcing servers publish new spatial crowdsourcing tasks. A worker may install our *route recommender* on his mobile device (smartphone). The route recommender is responsible for: (i) collecting task information from different servers continuously, (ii) recommending / updating a route based on the worker's current location and available tasks.

[4] The reward of a task can be collected by the same worker for only once. Similar to [10], we assume that each task has a unit reward and can be completed immediately.

[5] Our method can be applied to any distance function provided that it satisfies the triangle inequality, such as Euclidean distance, Manhattan distance, and road network distance.

3 Online Route Recommendation

First, we prove in Section 3.1 that no online algorithm can achieve a non-zero competitive ratio in OnlineRR. Then, we propose two categories of heuristic approaches for OnlineRR in Sections 3.2 and 3.3.

3.1 Competitive Analysis

We use the competitive ratio [2] to measure the performance of online algorithms. Since OnlineRR is a maximization problem, the competitive ratio $\mathcal{CR}$ is defined as:

$$\mathcal{CR} = \min_{e \in E} \frac{count(R_{alg}(e))}{count(R_{opt}(e))} \tag{1}$$

where E denotes the set of all problem instances, $R_{alg}(e)$ is the route recommended by an online algorithm alg for instance e, $R_{opt}(e)$ is the optimal route R_{opt} for instance e (cf. Definition 4), and $count(R_*(e))$ means the number of tasks on $R_*(e)$.

Definition 4 (Optimal route $R_{opt}(e)$ for OnlineRR). *Given a problem instance e, we denote its optimal route by $R_{opt}(e)$, which is obtained under assumption that the information of all tasks are known in advance (even before their release times).*

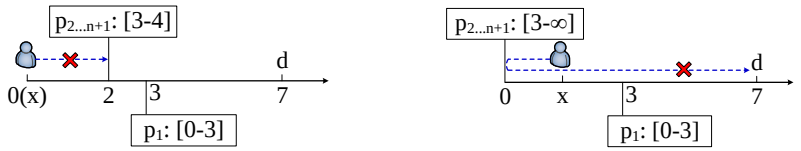
We show our competitive analysis below. It applies to any online algorithm, including both deterministic algorithms and randomized algorithms.

Theorem 1. *No online algorithm has a non-zero competitive ratio for* OnlineRR.

Proof. Since $\mathcal{CR} = \min_{e \in E} \frac{count(R_{alg}(e))}{count(R_{opt}(e))}$, it suffices to find a specific instance (i.e., the adversary) that makes $\mathcal{CR}$ as low as possible. Without loss of generality, in the following proof, we consider only locations on the positive half line $[0, +\infty)$. For the query, we set $t_q^- = 0$, $s = 0$, $t_q^+ = 10, d = 7$. Assume that $speed_q = 1$, that is $\tau_e(v, u) = |v - u|$. We simply denote a task p by $(p.loc, [t_p^-, t_p^+])$.

At time 0, the adversary releases a task $p_1 = (3, [0, 3])$. At time $m = 3$, the adversary will check the worker's current location (say x), and then decides to further release n tasks accordingly. There are two cases: (1) $x = 0$, or (2) $x > 0$. We show that the adversary can release those n tasks to make $\mathcal{CR}$ arbitrarily small.

Case 1: $x = 0$. In this case, the adversary will release tasks $p_{2 \le i \le n+1} = (2, [3, 4])$ (see Figure 3a). The worker cannot complete these tasks, since he cannot reach them



(a) Case 1: the worker cannot reach the location of $p_{2 \le i \le n+1}$ before their deadline (i.e., time 4)

(b) Case 2: the worker cannot proceed to $p_{2 \le i \le n+1}$ and arrive at d on time

Fig. 3. At time $m = 3$, adversaries release tasks $p_{2 \le i \le n+1}$ with [release time - deadline]

before their deadlines, and thus $count(R_{alg}) = 0$. But if all tasks are known in advance, the worker can wait at position 2 until all tasks are released and finish them on time $m = 3$. In this case, the competitive ratio is: $\mathcal{CR} = 0/n = 0$.

Case 2: $x > 0$**.** In this case, the adversary would release n tasks $p_{2 \le i \le n+1} = (0, [m, \infty])$ (see Figure 3b). As $m + x + d > m + d = 10 = t_q^+$, the worker cannot proceed to position 0 at time m; otherwise, he cannot reach d before t_q^+. So, the worker can finish at most the task p_1 only if he moves directly to m at time 0. However, if all tasks are known in advance, the worker could stay at 0 until time $m = 3$ to finish tasks $p_{2 \le i \le n+1}$, and thus $count(R_{opt}) = n$. Therefore, $\mathcal{CR} \le 1/n \to 0$ because n can be an arbitrary large value.

3.2 Greedy Task Approach

In this section, we present a greedy approach that incurs low response times.

The greedy approach works as follows. Initially, it calls GetNextTask (cf. Algorithm 1) to find the first task for the worker. Given the set of available[6] tasks P and the worker's location s_{now} at current time t_{now}, GetNextTask greedily selects the task with the highest score ψ_p. Upon reaching the chosen task, GetNextTask is involved to get the next task repeatedly until reaching d.

Algorithm 1 Get next best task

algorithm GetNextTask (Query $q = (s_{now}, d, [t_{now}, t_q^+])$, Set of available tasks P)
1: $Cand \leftarrow$ compute the set of feasible tasks from P ▷ apply Equation 2
2: **if** $Cand \neq \emptyset$ **then**
3: $p_{next} \leftarrow$ choose $p \in Cand$ with best score ψ_p ▷ ψ_p is a heuristic function
4: Return p_{next}
5: **else**
6: Apply policy $\mathcal{P}_{stay}$ or $\mathcal{P}_{go}$ until $Cand \neq \emptyset$ or $t_{now} + \tau(s_{now}, d) = t_q^+$

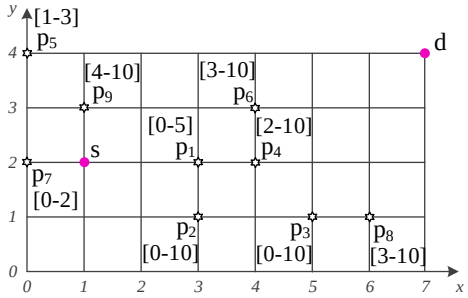
Due to the tasks' deadlines and the worker's expected arrivial time (cf. Definitions 1, 2), the worker may complete a task p if: (i) he can reach $p.loc$ before t_p^+, and (ii) he can reach d no later than t_q^+. Therefore, we call a task to be *feasible* if it satisfies:

$$\tau(s_{now}, p) + \tau(p, d) \le t_q^+ - t_{now} \qquad \text{and} \qquad t_{now} + \tau(s_{now}, p) \le t_p^+ \tag{2}$$

If there is no feasible task for q, the worker may stay or move based on a predefined policy (cf. Line 6 in Algorithm 1). In the policy $\mathcal{P}_{go}$, the worker simply moves towards the destination d. In the policy $\mathcal{P}_{stay}$, the worker waits at s_{now} until $t_{now} + \tau(s_{now}, d) = t_q^+$. When new feasible tasks are released, we resume the search and invoke GetNextTask to obtain the next task.

We illustrate several heuristics for computing the score ψ_p. Figure 4a shows the map of tasks which are labeled with release times and deadlines, and Figure 4b shows the

[6] Available tasks are tasks released before the current time t_{now}.



(a) map of tasks with [release time - deadline]

Heuristic	Route
G-NN	$\langle s, p_7, p_5, d\rangle$
G-ED	$\langle s, p_7, p_5, d\rangle$
G-MCS	$\langle s, p_1, p_4, p_6, d\rangle$
Re-Route	$\langle s, p_1, p_2, p_3, p_8, d\rangle$

(b) result routes (with $\mathcal{P}_{stay}$)

Route	# of tasks	Route	# of tasks	Route	# of tasks
$\langle s, p_1, d\rangle$	1	$\langle s, p_2, d\rangle$	1	$\langle s, p_3, d\rangle$	1
$\langle s, p_7, d\rangle$	1	$\langle s, p_1, p_2, d\rangle$	2	$\langle s, p_1, p_3, d\rangle$	2
$\langle s, p_2, p_3, d\rangle$	2	$\langle s, p_7, p_1, d\rangle$	2	$\langle s, p_1, p_2, p_3, d\rangle$	3
$\langle s, p_3, p_1, d\rangle$	not feasible	$\cdots$	not feasible	$\cdots$	not feasible

(c) all possible routes known at $t_{now} = 0$

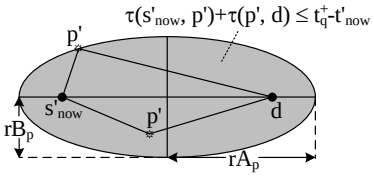
Fig. 4. Example of query $q = (s, d, [0, 10])$ in OnlineRR (using Manhattan Distance)

result route of each heuristic. In this example, we use the query $q = (s, d, [0, 10])$, the policy $\mathcal{P}_{stay}$, and the Manhattan distance.

Nearest Neighbor Heuristic (G-NN). It chooses the nearest feasible task to the worker's current location s_{now}, and thus setting $\psi_p = \tau(s_{now}, p)$. In Figure 4, G-NN produces the route $\langle s, p_7, p_5, d\rangle$.

Earliest Deadline Heuristic (G-ED). It chooses the task with the earliest deadline, and thus setting $\psi_p = t_p^+$. In Figure 4, G-ED recommends the route $\langle s, p_7, p_5, d\rangle$.

Maximum Candidate Space Heuristic (G-MCS). It chooses the task p that can maximize the search space of feasible tasks (Equation 2) in future. The search space in future is obtained under the assumption that p is just completed. The space shape differs for different distance metrics, but we can use a general approach Monte Carlo [20] to compare it. If a specific distance metric is used, then the exact candidate space size can be calculated. Take Euclidean distance for example, the space size is the area of the ellipse shown in Figure 5a, and thus we can calculate the score ψ_p using equations in Figure 5b for Euclidean distance metric.



(a) search space (in shade)

$$\psi_p = \pi \cdot rA_p \cdot rB_p$$
$$rA_p = (t_q^+ - t'_{now})/2$$
$$rB_p = \sqrt{rA_p^2 - (\tau(s'_{now}, d)/2)^2}$$
where $t'_{now} = t_{now} + \tau(s_{now}, p)$ and $s'_{now} = p.loc$

(b) search space size calculation

Fig. 5. Feasible candidates search space for Euclidean distance metric

We illustrate how G-MCS works in Figure 4. At time 0, the feasible tasks are p_1, p_2, p_3, p_7. Since p_1 has the highest score (ψ_{p_1}), p_1 is chosen to be visited. When the worker reaches p_1, a new task p_4 is released while p_7 expires, so the set of feasible tasks becomes $\{p_2, p_3, p_4\}$. Then p_4 is chosen as it has the highest score (ψ_{p_4}). Upon reaching p_4, the algorithm selects p_6 as it has the best score among $\{p_3, p_6, p_8\}$. After completing task p_6, there are no more feasible tasks. After waiting for two more time units, the worker moves toward d. In summary, G-MCS obtains the route $\langle s, p_1, p_4, p_6, d\rangle$.

3.3 Complete Search for Route Approach

In this section, we present a complete search approach that tends to find more tasks than the heuristics in Section 3.2.

Specifically, we formulate the following SnapshotRR problem, which takes the current query and the set of available tasks as input. Then, we solve SnapshotRR by enumerating all possible routes and obtain the one with the maximum number of tasks.

*Problem 2 (Snapshot Route Recommendation (*SnapshotRR*)).* Given a query $q = (s_{now}, d, [t_{now}, t_q^+])$ at the current snapshot t_{now}, SnapshotRR aims to find a route such that it covers *the maximum number of tasks* and the worker can arrive at d by t_q^+.

We illustrate this approach for the query $q = (s, d, [0, 10])$ in Figure 4. At time 0, we apply Equation 2 and obtain the set of feasible tasks: $P = \{p_1, p_2, p_3, p_7\}$. Figure 4c shows all possible routes (known at time 0). The optimal route at time 0 is $\langle s, p_1, p_2, p_3, d\rangle$.

We propose a simple optimization to solve SnapshotRR in Algorithm 2. At Line 3, we check whether there exists a new feasible task p (that was not available in the previous call of Algorithm 2). If such p exists, we must solve SnapshotRR again. Otherwise, the best route remains the same as in the previous call, so we need not solve SnapshotRR again.

Algorithm 2 Complete search the result route

```
    algorithm Re-Route (Query q = (s_now, d, [t_now, t_q^+]), Set of available tasks P)
1:  Let P_prev be the set of available tasks in the previous call
2:  if P ≠ ∅ then
3:      if ∃p ∈ P − P_prev such that p is feasible then          ▷ Equation 2
4:          R ← Solve SnapshotRR(q, P)                          ▷ conduct complete search
5:  else
6:      Apply policy P_stay or P_go      until P ≠ ∅ or t_now + τ(s_now, d) = t_q^+
```

We proceed to illustrate how Re-Route works in the example in Figure 4. At time 0, Re-Route computes the route $R_0 = \langle s, p_1, p_2, p_3, d\rangle$, and then the worker moves along R_0 to p_1. Upon reaching p_1, a new feasible task p_4 is found, so Re-Route re-calculates the route as $R_1 = \langle p_1, p_2, p_3, d\rangle$. When the worker reaches p_2, a new feasible task p_8 is found, so Re-Route updates the route to $R_2 = \langle p_2, p_3, p_8, d\rangle$. After reaching p_8, a new task p_9 is found but it is not feasible. Thus, Re-Route would not computes the route

again (cf. Line 3 in Algorithm 2). Eventually, the worker moves to d. In summary, the actual route traveled by the worker is: $\langle s, p_1, p_2, p_3, p_8, d\rangle$. It covers more tasks than other heuristics (cf. Figure 4b).

Since it is expensive to solve SnapshotRR by enumerating all possible routes, we will present optimizations to solve SnapshotRR efficiently in Section 4.

4 Optimization for SnapshotRR

We adapt the *bi-directional search* algorithm for the Orienteering Problem with Time Windows (OPTW) problem [19] to solve our problem. For brevity in discussion, we use $q = (s, d, [t_q^-, t_q^+])$ instead of $q = (s_{now}, d, [t_{now}, t_q^+])$. We will conduct bi-directional search for SnapshotRR in three steps:

Step 1: Search sub-routes in the forward direction (from s) and store them in $\overrightarrow{\mathbb{R}}$
Step 2: Search sub-routes in the backward direction (from d) and store them in $\overleftarrow{\mathbb{R}}$
Step 3: Join sub-routes between $\overrightarrow{\mathbb{R}}$ and $\overleftarrow{\mathbb{R}}$

According to Pruning Rule 1, the bi-directional search can reduce the search space. However, the method in [19] does not exploit spatial properties in our problem. In this section, we develop more effective pruning rules to accelerate bi-directional search on SnapshotRR.

Pruning Rule 1 (Half travel time bound property proved in [19]) *In the forward (or backward) route searching from vertex s (or d), only routes R with $\tau(R) \leq \tau_{max}/2$ are maintained and extended, where $\tau_{max} = t_q^+ - t_q^-$.*

4.1 Forward Search and Backward Search

In this section, we elaborate the forward search (Step 1) and discuss adaptations for the backward search (Step 2) at the end. In the following discussion, we use R instead of $\overrightarrow{R}$ to represent a sub-route found in forward search (which will be stored in $\overrightarrow{\mathbb{R}}$) for simplicity.

We first introduce the sub-route concept and its extension operation. Then, we propose a pruning rule and a search strategy to speedup the computation. In the following, we denote the set of vertices as $V = P \cup \{s, d\}$, where P is the set of available tasks.

Sub-route Extension.

We denote a path from s to $v \in V$ as a sub-route R_v, which contains four attributes $R_v = (\tau(R_v), B_{R_v}, C_{R_v}, v)$.

- $\tau(R_v)$ represents the travel time along R_v (i.e., from s to v).
- B_{R_v} stores a sequence of tasks visited before on the sub-route R_v. We denote the profit of R_v as $|B_{R_v}|$ because all tasks have the same reward.
- C_{R_v} is a set of candidate vertices (that are feasible for visiting in future), and its calculation is discussed in Equation 5.

During route search, for each vertex v, we store all sub-routes of the form R_v into a set $\mathbb{R}_v$. In addition, we only consider *feasible routes*. Recall that $\tau(R_v)$ represents the travel time (along R_v) from s to v. According to Equation 2, a sub-route R_v is said to be *feasible* if:

$$\tau(R_v) \leq t_v^+ - t_q^- \quad \text{and} \quad \tau(R_v) \leq t_q^+ - t_q^- \tag{3}$$

where t_v^+ is the deadline for vertex v when v is a task, or ∞ when $v \in \{s, d\}$.

For each vertex $u \in C_{R_v}$, we can extend R_v with an arc (v, u) to form a new sub-route R_u. The component of $R_u = (\tau(R_u), B_{R_u}, C_{R_u}, u)$ is calculated as follows:

$$B_{R_u} \leftarrow \langle B_{R_v},\ v \rangle \quad \text{and} \quad \tau(R_u) \leftarrow \tau(R_v) + \tau_e(v, u) \tag{4}$$

The set C_{R_u} contains each candidate vertex p that satisfies:

$$\begin{gathered} p \in C_{R_v}(\heartsuit) \quad \text{and} \quad p \notin B_{R_u}(\diamondsuit) \\ \tau(u,p) \leq t_p^+ - t_q^- - \tau(R_u) \quad \text{and} \quad \tau(u,p) \leq (t_q^+ - t_q^-)/2 - \tau(R_u)(\clubsuit, \spadesuit, \blacklozenge) \\ \tau(u,p) + \tau(p,d) \leq t_q^+ - t_q^- - \tau(R_u)(\clubsuit, \heartsuit) \end{gathered} \tag{5}$$

which involve the constraints in Equation 4 (♣), Equation 3 (♠), triangle inequality (♡), the constraint that each task can be visited only once (♢), the worker's arrival time t_q^+ (♥) and Pruning Rule 1 (♦).

We illustrate sub-route extension in Figure 6. Assume that $q = (s, d, [0, 10])$ and $P = \{p_1, p_2, \cdots, p_7\}$. We consider Manhattan distance in this example. First, we compute the candidate set of s. By Pruning Rule 1, we only consider tasks within $10/2 = 5$ units from s (i.e., tasks in the dotted diamond in Figure 6). Thus, tasks p_3, p_7 are not feasible. The tasks p_4 and p_5 are not feasible as they violate constraints on the task's deadline and the worker's arrival time, respectively. Thus, we obtain the candidate set of s as $C_s = \{p_1, p_2, p_6\}$, and compute the sub-route for s as $R_s = (0, \emptyset, C_s, s)$. Next, we append arcs (s, p_1), (s, p_2), (s, p_6) into R_s to generate three new sub-routes: $R_1 = (1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)$, $R_2 = (3, \langle p_2 \rangle, \{p_6\}, p_2)$, $R_6 = (5, \langle p_6 \rangle, \emptyset, p_6)$.

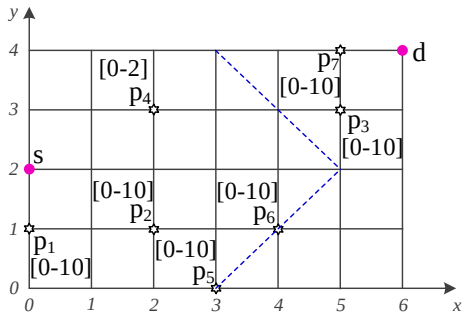


Fig. 6. Example query $q = (s, d, [0, 10])$ for SnapshotRR problem (using Manhattan distance)

Dominate Test Pruning.

We develop the following pruning rule to further reduce the search space.

Pruning Rule 2 (Dominating Pruning) *Let* $R_v = (\tau(R), B_{R_v}, C_{R_v}, v)$ *and* $R'_v = (\tau(R'_v), B_{R'_v}, C_{R'_v}, v)$ *be two feasible routes associated with* v*. We can prune* R'_v *if:*

$$\tau(R_v) \leq \tau(R'_v) \text{ and } |C_{R'_v} \cap B_{R_v}| \leq |B_{R_v}| - |B_{R'_v}|$$

Proof. Among all full routes with R'_v as the prefix, let $R'_{opt} = \langle s, B_{R'_v}, R'_{tail}, d\rangle$ be the maximum reward route. With the given condition $\tau(R_v) \leq \tau(R'_v)$, after traveling along R_v, we can still follow all tasks in R'_{tail} and arrive at d by t_q^+. There exists a route $R_{exist} = \langle s, B_{R_v}, R_{tail}, d\rangle$ where $R_{tail} = R'_{tail} - B_{R_v}$. R_{exist} ensures that the reward of each task is gained at most once as B_{R_v} and R_{tail} have no common tasks.

Since $R'_{tail} \subseteq C_{R'_v}$, we have $|R'_{tail}| = |R_{tail}| + |R'_{tail} \cap B_{R_v}| \leq |R_{tail}| + |C_{R'_v} \cap B_{R_v}|$.

By combining the above with the given condition $|C_{R'_v} \cap B_{R_v}| \leq |B_{R_v}| - |B_{R'_v}|$, we derive: $|B_{R_v}| + |R_{tail}| \geq |B_{R'_v}| + |C_{R'_v} \cap B_{R_v}| + |R_{tail}| \geq |B_{R'_v}| + |R'_{tail}|$. As the reward of R_{exist} (extended from R_v) is greater than or equal to that of R'_{opt} (extended from R'_v), we can prune the subroute R'_v.

Search Strategy.

Our strategy is to identify sub-routes with better reward values in order to utilize pruning rule 2. To do so, we introduce the concept of upper bound reward:

Definition 5 (Vertex upper bound reward $\$_v^+$). *Given a sub-route* $R_v = (\tau(R_v), B_{R_v}, C_{R_v}, v)$*, we define its upper bound reward as* $\$_{R_v}^+ = |B_{R_v}| + |C_{R_v}|$*. The upper bound reward of vertex* $v \in V$ *is defined as:* $\$_v^+ = \max\{\$_{R_v}^+ \mid R_v \in \overrightarrow{\mathbb{R}_v}\}$.

Initially, we begin the search from a sub-route at s. We iteratively extend sub-routes found so far and apply pruning rule 2 to discard unpromising sub-routes. During the search, we employ a heap H to process vertices in descending order of $\$_v^+$.

We illustrate this method on the example in Figure 6 and show the running steps in Table 1. Iteration 1 corresponds to the extension of the sub-route R_s at s, which we have discussed before. We obtain three new subroutes R_1, R_2, R_6, insert them in their corresponding route sets $\overrightarrow{\mathbb{R}_p}$, and also enheap p_1, p_2, p_6 into H. In each subsequent iteration, we deheap the vertex $v \in H$ with the largest $\$_v^+$, and extend its sub-routes R_v in the descending order of $|B_{R_v}|$.

In iteration 2, we generate a new sub-route $(3, \langle p_1, p_2\rangle, \{p_6\}, p_2)$ and apply Pruning Rule 2 to discard the previous subroute at p_2, i.e., $(3, \langle p_2\rangle, \{p_6\}, p_2)$. Similarly, the previous sub-routes for p_6: $(5, \langle p_6\rangle, \emptyset, p_6)$ and $(5, \langle p_1, p_6\rangle, \emptyset, p_6)$ are pruned in iterations 2 and 3, respectively.

Table 1. Forward space search

Iteration	Selected Vertex	Extended Route R	Modified $\mathbb{R}$	Heap H
1	s	$(0, \emptyset, \{p_1, p_2, p_6\}, s)$	$\overrightarrow{\mathbb{R}_{p_1}} = \{(1, \langle p_1\rangle, \{p_2, p_6\}, p_1)\}$ $\overrightarrow{\mathbb{R}_{p_2}} = \{(3, \langle p_2\rangle, \{p_6\}, p_2)\}$ $\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_6\rangle, \emptyset, p_6)\}$	$(p_1, 3)$ $(p_2, 2)$ $(p_6, 1)$
2	p_1	$(1, \langle p_1\rangle, \{p_2, p_6\}, p_1)$	$\overrightarrow{\mathbb{R}_{p_2}} = \{(3, \langle p_1, p_2\rangle, \{p_6\}, p_2)\}$ $\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_1, p_6\rangle, \emptyset, p_6)\}$	$(p_2, 3)$ $(p_6, 2)$
3	p_2	$(3, \langle p_1, p_2\rangle, \{p_6\}, p_2)$	$\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6)\}$	$(p_6, 3)$
4	p_6	$\emptyset$	$\emptyset$	$\emptyset$
$\overrightarrow{\mathbb{R}}$	$(5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6)$, $(3, \langle p_1, p_2\rangle, \{p_6\}, p_2)$, $(1, \langle p_1\rangle, \{p_2, p_6\}, p_1)$, $(0, \emptyset, \{p_1, p_2, p_6\}, s)$			

Algorithm 3 Forward search

function RouteSearchFW(Query $q = (s, d, [t_q^-, t_q^+])$, Vertex set $V = P \cup \{s, d\}$)
▷ Initialization
Create an empty set $\overrightarrow{\mathbb{R}_v}$ for each vertex $v \in V$ to store sub-routes associated with v
Calculate the candidate vertex set C_s of s ▷ Equation 5
$\overrightarrow{\mathbb{R}_s} \leftarrow \{(0, \emptyset, C_s, s)\}$
Create a max-heap $H \leftarrow \{(s, |C_s|)\}$ to store vertices whose routes will be extended
▷ Repeatedly generate feasible sub-routes
while $H \neq \emptyset$ **do**
  $(v, v.ub) \leftarrow$ Extract-Max(H) ▷ Searching strategy
  Sort routes $R \in \overrightarrow{\mathbb{R}_v}$ in the descending order of $|B_R|$ ▷ Searching strategy
  for all $R_v \in \overrightarrow{\mathbb{R}_v}$ **do**
    for all $u \in C_{R_v}$ **do**
      $R_u \leftarrow$ Extend(R_v, q, u) ▷ Equation 4, 5, Pruning Rule 1
      RemoveDominate($\overrightarrow{\mathbb{R}_u}, R_u$) ▷ Pruning Rule 2
      if $R_u \in \overrightarrow{\mathbb{R}_u}$ **then** ▷ R_u not pruned
        if $(u, u.ub) \notin H$ **then**
          Insert $(u, \$^+_{R_u})$ into H
        else
          $u.ub \leftarrow \max\{u.ub, \$^+_{R_u}\}$
Return $\overrightarrow{\mathbb{R}} \leftarrow$ all routes in each nonempty $\overrightarrow{\mathbb{R}_v}$

The forward search terminates when H becomes empty, i.e., no sub-routes can be extended. It returns the set $\overrightarrow{\mathbb{R}}$ of all surviving sub-routes.

Algorithm 3 illustrates the pseudo code of route search in forward direction. It is self-explanatory and summarizes what we have discussed above.

Backward Search. Route space search in backward direction is similar to that in forward direction. The pruning rules, searching strategies, and dominating testing discussed for forward search can be modified for backward search directly.

4.2 Route Join

In this section, we elaborate on how to join sub-routes obtained in the forward search and the backward search. Let $\overrightarrow{R_v} = (\tau(\overrightarrow{R_v}), B_{\overrightarrow{R_v}}, C_{\overrightarrow{R_v}}, v)$ and $\overleftarrow{R_u} = (\tau(\overleftarrow{R_u}), B_{\overleftarrow{R_u}}, C_{\overleftarrow{R_u}}, u)$ be two sub-routes in the forward and the backward directions, respectively. They are *feasible* to be joined if:

$$\tau(\overrightarrow{R_v}) + \tau(v, u) \leq u.t_p^+ \quad \text{and} \quad \tau(\overrightarrow{R_v}) + \tau(\overleftarrow{R_u}) + \tau(v, u) \leq t_q^+ - t_q^-$$
$$B_{\overrightarrow{R_v}} \cap B_{\overleftarrow{R_u}} = \emptyset \tag{6}$$

We denote the joined route as $R^{join} = \langle s, B_{\overrightarrow{R_v}}, rev(B_{\overleftarrow{R_u}}), d\rangle$, where $rev(B_{\overleftarrow{R_u}})$ refers to a list of vertices in $B_{\overleftarrow{R_u}}$ but in the reversed order. Its reward is: $|B_{\overrightarrow{R_v}}| + |B_{\overleftarrow{R_u}}|$.

We develop two optimization techniques to accelerate the join procedure. First, we apply pruning rule 3 to skip the *feasible* checking (cf. Equation 6) for pairs of sub-

Table 2. Route join

	sub-routes sorted in the descending order of $\lvert B_R \rvert$
$\overrightarrow{\mathbb{R}}$	$(5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6),\ (3, \langle p_1, p_2\rangle, \{p_6\}, p_2),\ (1, \langle p_1\rangle, \{p_2, p_6\}, p_1),\ (0, \emptyset, \{p_1, p_2, p_6\}, s)$
$\overleftarrow{\mathbb{R}}$	$(5, \langle p_7, p_3, p_6\rangle, \emptyset, p_6),\ (2, \langle p_7, p_3\rangle, \{p_6\}, p_3),\ (1, \langle p_7\rangle, \{p_3, p_6\}, p_7),\ (0, \emptyset, \{p_3, p_6, p_7\}, d)$

route join iterations

iteration	candidate join pairs		join result	$\$^{best}$
	$\overrightarrow{R}$	$\overleftarrow{R}$	R^{join}	
1	$(5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6)$	$(5, \langle p_7, p_3, p_6\rangle, \emptyset, p_6)$	not feasible (Equation 6)	0
2	$(5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6)$	$(2, \langle p_7, p_3\rangle, \{p_6\}, p_3)$	$R = \langle s, p_1, p_2, p_6, p_3, p_7, d\rangle$	5
$\cdots$	$\cdots$	$\cdots$	skipped (Pruning Rule 3)	5
optimal route for this snapshot			$R = \langle s, p_1, p_2, p_6, p_3, p_7, d\rangle$	

routes. Second, we sort sub-routes in the descending order of their $|B_R|$. This helps us find a tigher $\$^{best}$ earlier, and in turn boosts the power of Pruning Rule 3.

Pruning Rule 3 (Reward bound pruning) *Let $\$^{best}$ be the maximum reward on all joined routes found so far. If $|B_{\overrightarrow{R}}| + |B_{\overleftarrow{R}}| \leq \$^{best}$, then we need not join $\overrightarrow{R}$ and $\overleftarrow{R}$.*

Continuing with the example in Figure 6, we illustrate the join procedure in Table 2. First, we sort forward sub-routes $\overrightarrow{R} \in \overrightarrow{\mathbb{R}}$ and backward sub-routes $\overleftarrow{R} \in \overleftarrow{\mathbb{R}}$ in descending order of $|B_R|$. For each pair of $\overrightarrow{R}$ and $\overleftarrow{R}$, if it survives Pruning Rule 3, then we conduct feasible checking and then join the pair. After joining the forward sub-route $\overrightarrow{R} = (5, \langle p_1, p_2, p_6\rangle, \emptyset, p_6)$ with the backward sub-route $\overleftarrow{R} = (2, \langle p_7, p_3\rangle, \{p_6\}, p_3)$, we update $\$^{best}$ to 5. All remaining pairs are pruned according to Pruning Rule 3. The best route (known at this snapshot) is $\langle s, p_1, p_2, p_6, p_3, p_7, d\rangle$.

5 Experiment

This section studies the effectiveness and efficiency of our proposed methods on both real and synthetic datasets.

5.1 Experimental Setting

We first introduce the datasets used in experiments, and then describe the performance measures for algorithms.

Datasets.

Real datasets. Similar to [10], we obtain real check-in data in Foursquare[7] and convert them to crowdsourcing tasks in our problem. Specifically, we collect check-in data for New York city (NYC) and Los Angeles County (LA) in a month (September 2012). For each day in that month, we use all check-in items within a 90-minute duration. We take check-in items at the same location as a single task, set its release time and deadline to the earliest and the latest check-in time respectively[8]. We measure the travel time $\tau(v, u)$ as the Euclidean distance between two locations divided by the average speed.

[7] https://foursquare.com/

[8] For each location with only one check-in item (say, at time t), we choose its deadline randomly in $[t, t_q^+]$, where t_q^+ refers to the query's deadline.

Table 3. Experiment parameters

Parameter	Default	Range
total number of tasks	100	$20, 50, 100, 200, 500$
$t_q^+ - t_q^-$ [minutes]	90	$30, 60, 90, 120, 150$
Gaussian x	0.1	$0.05, 0.1, 0.25, 0.5$

We use a walking speed 6 km/h for NYC (whose map size 789 km^2 is small), and use a driving speed 60 km/h for LA (whose map size 10,570 km^2 is large).

Synthetic datasets. As NYC and LA have similar result trends (see Figure 7), we use the map domain of LA to generate synthetic datasets. For each synthetic task, we randomly choose its release time t_p^- randomly in $[t_q^-, t_q^+]$ and then choose its deadline t_p^+ in range $[t_p^-, t_q^+]$, as we consider queries of the form $q = (s, d, [t_q^-, t_q^+])$ in our experiments. We generate two types of datasets. In each uniform dataset (UNI), task locations are randomly chosen within the map domain. In each Gaussian dataset (GAU), task locations are generated based on four Gaussian bells, with the standard deviation of Gaussian bell as x times of the map domain length. The parameter values for the number of tasks and Gaussian standard deviation x are shown in Table 3.

Platform and Performance Measures. We implemented our methods (G-NN, G-MCS, G-ED, Re-Route) in C++, and conducted experiments on an Ubuntu 11.10 machine with a 3.4 GHz Intel Core i7-3770 processor and 16 GB RAM.

We use queries of the form $q = (s, d, [t_q^-, t_q^+])$, where $t_q^+ - t_q^- = 90$ minutes by default. We randomly choose s, d in the map domain such that $\tau(s, d) = 45$ minutes. The parameter values for $t_q^+ - t_q^-$ are given in Table 3.

In each experiment, we run a set Q of 50 queries and report (i) the quality ratio for Q, and (ii) the average response time per call of a method. Specifically, we define the *quality ratio* of a method as:

$$quality\ \ ratio\ \ = \frac{1}{|Q|} \cdot \sum_{q \in Q} \frac{count(R_{method}(q))}{count(R_{opt}(q))}$$

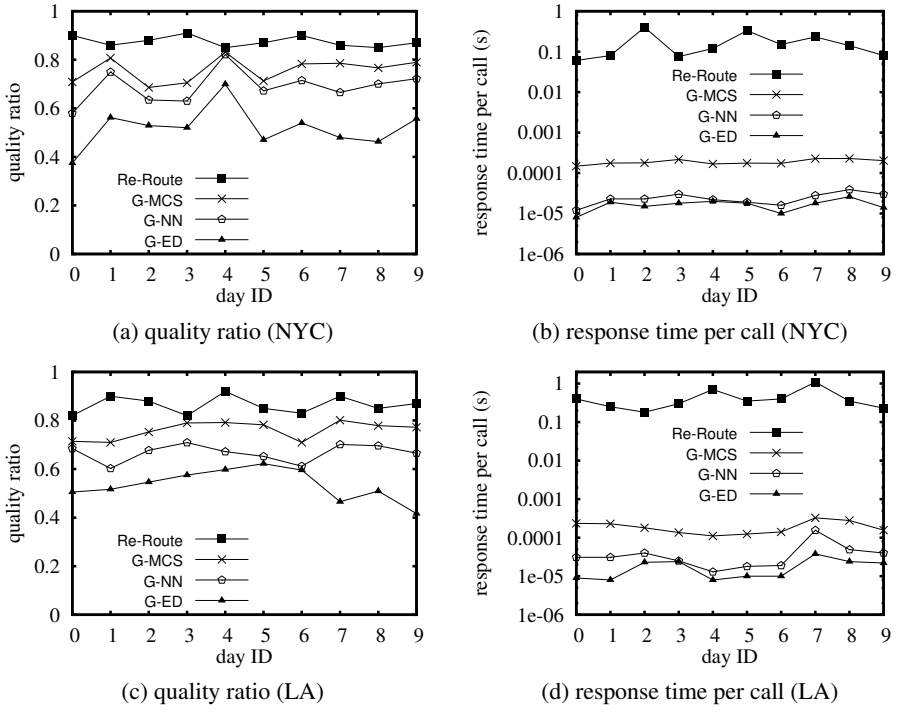
where q is a query in Q, $R_{method}(q)$ is the route for q found by our method, $R_{opt}(q)$ is the optimal route for q found by an offline method that knows all tasks in advance[9].

We have tested the effects of policies $\mathcal{P}_{stay}$ and $\mathcal{P}_{go}$ (cf. Section 3.2) on our methods. For the same method, the quality ratios between $\mathcal{P}_{stay}$ and $\mathcal{P}_{go}$ differ only by $0.01 - 0.02$. Thus, we take the default policy in our methods as $\mathcal{P}_{stay}$.

5.2 An Experiment on Real Datasets

We plot the performance of methods on real datasets (LA and NYC) on each day from Sep/21/2012 to Sep/30/2012 in Figure 7. Within the query period, LA and NYC contain 60 and 40 tasks on average, respectively. The optimal routes R_{opt} in LA and NYC

[9] As mentioned in Definition 4, $R_{opt}(q)$ is obtained with assumption that all tasks' information are known in advance at time t_q^-. With this assumption, OnlineRR becomes a special case of SnapshotRR where tasks can have release time larger than t_q^- and the approach for SnapshotRR can be used to find $R_{opt}(q)$ then.



(a) quality ratio (NYC) (b) response time per call (NYC)

(c) quality ratio (LA) (d) response time per call (LA)

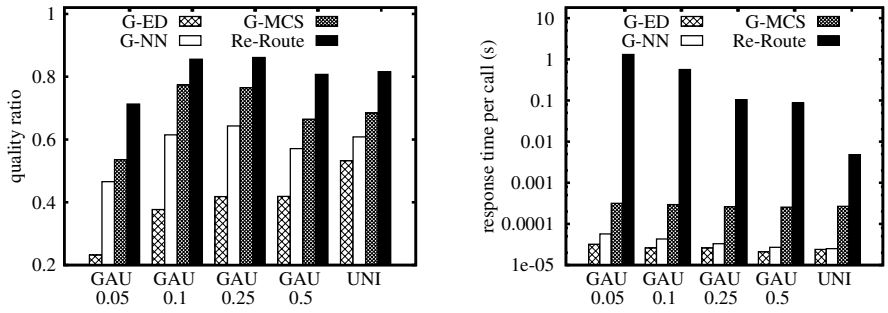
Fig. 7. Performance on real datasets

cover 10 and 5 tasks on average, respectively. Figures 7a,c show the quality ratio of the methods on NYC and LA, respectively. Re-Route outperforms other methods and achieves 0.82–0.91 quality. G-MCS is the second best and obtains 0.70–0.84 quality. Although Re-Route incurs higher response time, it takes less than 1 second per call, as depicted in Figure 7b,d. We consider such time acceptable for crowdsourcing workers. For example, for the LA dataset, Re-Route is called for 10 times (on average) during the query period (90 minutes). Observe that the time per call (1 second) is negligible compared to the average travel time between two tasks ($90/10 = 9$ minutes).

5.3 Scalability Experiments on Synthetic Datasets

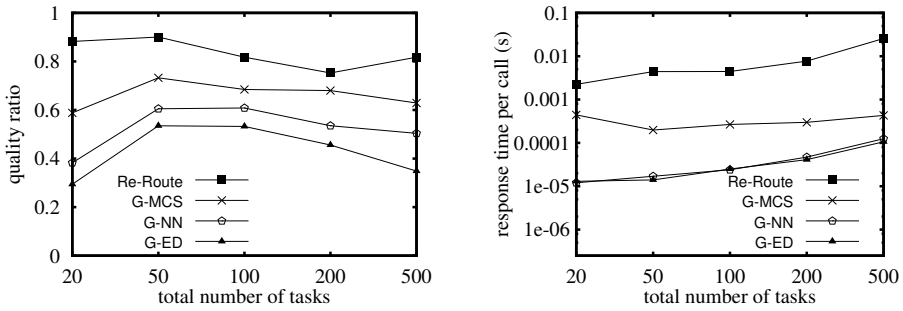
Effect of task distribution. Figure 8 depicts the performance of methods on GAU datasets with standard deviation x and on a UNI dataset. As illustrated in Table 4a, a more skewed dataset (i.e., with smaller x) leads to an optimal route with higher reward because tasks in the same cluster are close together. Since our methods can also find routes with higher reward on a more skewed dataset, the quality ratio does not vary much (See Figure 8a). Re-Route again outperforms other methods on the quality ratio. On the other hand, a more skewed dataset induces more feasible candidate tasks in Re-Route, and thus it incurs higher response time. Nevertheless, Re-Route takes at most around 1 second per call in Figure 8b, which is acceptable for crowdsourcing workers.

Since the trend on quality is consistent across different task distributions, we only use UNI datasets in the remaining experiments.



(a) quality ratio (b) response time per call

Fig. 8. Effect of task distribution



(a) quality ratio (UNI) (b) response time per call (UNI)

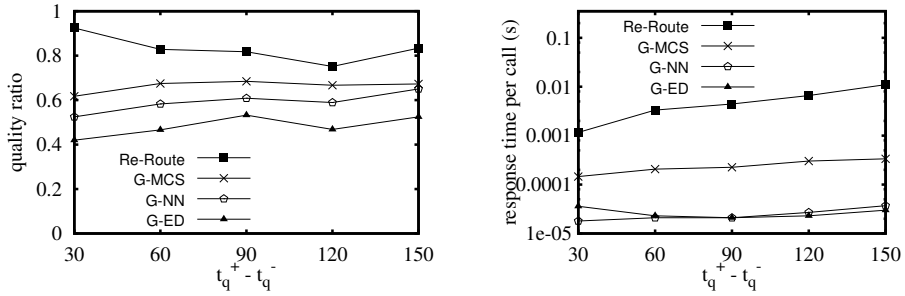
Fig. 9. Effect of the total number of tasks

Table 4. Reward on the optimal route

Task distribution	Gaussian	Uniform	
Parameter values	**(a)** standard deviation x	**(b)** total number of tasks	**(c)** query period $t_q^+ - t_q^-$
	$0.05,\ 0.1,\ 0.25,\ 0.5$	$20,\ 50,\ 100,\ 200,\ 500$	$30,\ 60,\ 90,\ 120,\ 150$
Reward of R_{opt}	$12.57, 9.39, 6.84, 4.72$	$1.7, 3.14, 5.26, 7.94, 13.2$	$1.62, 3.26, 5.26, 6.92, 8.92$

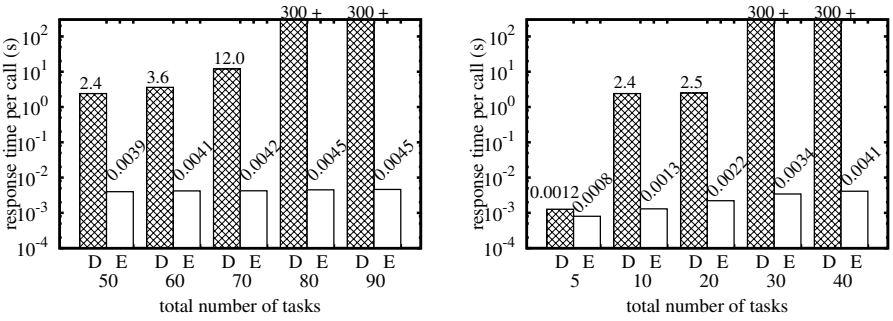
Effect of total number of tasks. When the total number of tasks increases, both the optimal route (cf. Table 4b) and our methods' routes would cover more tasks. Thus, the quality ratio is independent of the total number of tasks, as shown in Figure 9a. The response time of Re-Route increases slightly with the total number of tasks (see Figure 9b), but it is still within 0.1 seconds per call.

Effect of the query period $t_q^+ - t_q^-$. As the query period $t_q^+ - t_q^-$ widens, more tasks become feasible and thus the optimal route contains more tasks, as shown in Table 4c. We plot the performance of the methods with respect to $t_q^+ - t_q^-$ in Figure 10. The quality ratio is independent of $t_q^+ - t_q^-$ as our methods are also able to find routes with more tasks. The response time per call in Re-Route remains acceptable.



(a) quality ratio (UNI) (b) response time per call (UNI)

Fig. 10. Effect of the query period $t_q^+ - t_q^-$



(a) response time per call (UNI) (b) response time per call (GAU)

Fig. 11. Effect of pruning rules on Re-Route (E for ENABLE, D for DISABLE)

Effect of Pruning Rules on Re-Route**.** We proceed to test the effect of optimization techniques (cf. Section 4) on the response time per call of Re-Route. We consider two variations of Re-Route: (i) DISABLE applies only pruning rule 1 (in Ref. [19]), and (ii) ENABLE applies all three pruning rules in Section 4.

As DISABLE is very slow, we scale down the total number of tasks in this experiment, and terminate it if it takes more than 300 seconds per call. We show the response time per call of DISABLE and ENABLE on both UNI and GAU datasets in Figure 11. Observe that ENABLE runs much faster than DISABLE, implying that our pruning rules are able to shrink the search space significantly.

6 Related Work

Spatial crowdsourcing is an emerging topic in crowdsourcing research. Existing researches are divided into the *server-centric* mode [9, 15, 16, 18, 22] and the *worker-centric* mode [3, 7, 10]. We focus on the latter one as discussed in the introduction. However, [3, 7] do not consider the influence of the worker's travel time, which is critical in our OnlineRR problem. The closest work to ours is [10], which selects a route

with the maximum number of tasks for a worker. However, [10] does not discuss how to update a route with respect to online task arrivals. Also, it does not consider the worker's destination and deadline.

Our OnlineRR problem is related to the orienteering problem [13, 23]. The orienteering problem is a variant of the selective traveling salesman problem [11], where (i) not all requests need to be completed, and (ii) the cost is the sum of the total travel time and the penalty of rejected requests. The orienteering problem is well studied [13, 23], but only several works [5, 8, 12, 19] consider the Orienteering Problems with each request having a Time Window (OPTW). Those works focus on the offline scenario but not the online scenario. While there exist approximation algorithms for OPTW offline [5,8,12], OnlineRR is an online problem and does not permit any online algorithm to achieve a non-zero competitive ratio.

Righini et al. [19] propose an exact bi-directional search algorithm for OPTW, which can be adapted to solve our SnapshotRR problem. Unlike our solution, this algorithm does not exploit spatial properties to prune unpromising sub-routes. In Section 4, we have developed two pruning rules and a search strategy that are specific for SnapshotRR.

Other related route planning problems include the trip planning problem [17] and the optimal sequenced route problem [21]. They require finding the shortest route that passes through specific types of points-of-interests. On the other hand, our problem needs to maximize the number of tasks on a route subject to the tasks' deadlines and the worker's deadline.

OnlineRR problem is also related to online traveling salesman problem (OL-TSP) [4, 14]. Few works have studied OL-TSP with each request having a deadline [6, 24]. While OL-TSP aims to minimize the travel distance, our OnlineRR problem aims to maximize the number of tasks on a route. Moreover, the above works on OL-TSP do not consider the worker's destination and deadline. Finally, our problem is similar to an online job-scheduling problem whose tasks have dependent setup costs [1]. However, this problem does not exploit the spatial properties as in OnlineRR.

7 Conclusion

In this paper, we study the oriented online route recommendation (OnlineRR) problem for spatial crowdsourcing task workers. We prove that no online algorithm can achieve a non-zero competitive ratio for OnlineRR. Then we propose several heuristics for OnlineRR and optimizations to speedup the computation. According to our experimental findings, Re-Route produces routes with the highest quality (0.82–0.91) in acceptable response time per call (0.1–1 s), whereas G-MCS returns routes with the second highest quality (0.70–0.84) at real-time (below 1 ms). Workers preferring to save smartphone battery power may choose G-MCS as it has less computation cost. OnlineRR will be extended to consider the task diversity and task novelty in the future.

References

1. A. Allahverdi, C. T. Ng, T. C. E. Cheng, and M. Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, pages 985–

1032, 2008.
2. B. Allan and E.-Y. Ran. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
3. F. Alt, A. S. Shirazi, A. Schmidt, U. Kramer, and Z. Nawaz. Location-based crowdsourcing: extending crowdsourcing to the real world. In *NordiCHI*, pages 13–22, 2010.
4. G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, pages 560–581, 2001.
5. N. Bansal, A. Blum, S. Chawla, and A. Meyerson. Approximation algorithms for deadline-tsp and vehicle routing with time-windows. In *Symposium on Theory of Computing*, pages 166–174, 2004.
6. M. Blom, S. O. Krumke, W. E. D. Paepe, and L. Stougie. The online tsp against fair adversaries. *INFORMS Journal on Computing*, pages 138–148, 2001.
7. M. F. Bulut, Y. S. Yilmaz, and M. Demirbas. Crowdsourcing location-based queries. In *PERCOM Workshops*, pages 513–518, 2011.
8. C. Chekuri and N. Korula. Approximation algorithms for orienteering with time windows. *CoRR*, 2007.
9. Z. Chen, R. Fu, Z. Zhao, Z. Liu, L. Xia, L. Chen, P. Cheng, C. C. Cao, Y. Tong, and C. J. Zhang. gmission: A general spatial crowdsourcing platform. *PVLDB*, pages 1629–1632, 2014.
10. D. Deng, C. Shahabi, and U. Demiryurek. Maximizing the number of worker's self-selected tasks in spatial crowdsourcing. In *SIGSPATIAL*, pages 314–323, 2013.
11. H. A. Eiselt, M. Gendreau, and G. Laporte. Location of facilities on a network subject to a single-edge failure. *Networks*, pages 231–246, 1992.
12. G. N. Frederickson and B. Wittman. Approximation algorithms for the traveling repairman and speeding deliveryman problems. *Algorithmica*, pages 1198–1221, 2012.
13. D. Gavalas, C. Konstantopoulos, K. Mastakas, and G. E. Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *J. Heuristics*, pages 291–328, 2014.
14. P. Jaillet and M. R. Wagner. Online routing problems: Value of advanced information as improved competitive ratios. *Transportation Science*, pages 200–210, 2006.
15. L. Kazemi and C. Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *SIGSPATIAL*, pages 189–198, 2012.
16. L. Kazemi, C. Shahabi, and L. Chen. Geotrucrowd: trustworthy query answering with spatial crowdsourcing. In *SIGSPATIAL*, pages 304–313, 2013.
17. F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.
18. L. Pournajaf, L. Xiong, V. S. Sunderam, and S. Goryczka. Spatial task assignment for crowd sensing with cloaked locations. In *MDM*, pages 73–82, 2014.
19. G. Righini and M. Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Comput. Oper. Res.*, 36:1191–1203, 2009.
20. R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2011.
21. M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, pages 765–787, 2008.
22. H. To, G. Ghinita, and C. Shahabi. A framework for protecting worker location privacy in spatial crowdsourcing. *PVLDB*, pages 919–930, 2014.
23. P. Vansteenwegen, W. Souffriau, and D. V. Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, pages 1–10, 2011.
24. X. Wen, Y. Xu, and H. Zhang. Online traveling salesman problem with deadlines and service flexibility. *Journal of Combinatorial Optimization*, pages 1–18, 2013.