2015 International Conference on Distributed Computing in Sensor Systems

On Exploiting Logical Dependencies for Minimizing Additive Cost Metrics in Resource-Limited Crowdsensing

Shaohan Hu*, Shen Li*, Shuochao Yao*, Lu Su†, Ramesh Govindan‡, Reginald Hobbs§, Tarek F. Abdelzaher*
*University of Illinois at Urbana-Champaign, USA, †State University of New York at Buffalo, USA,
‡University of Southern California, USA, §US Army Research Laboratory
Email: {shu17, shenli3, syao9, zaher}@illinois.edu, lusu@buffalo.edu, ramesh@usc.edu, reginald.l.hobbs2.civ@mail.mil

***Abstract*—We develop data retrieval algorithms for crowdsensing applications that reduce the underlying network bandwidth consumption or any additive cost metric by exploiting logical dependencies among data items, while maintaining the level of service to the client applications. Crowdsensing applications refer to those where local measurements are performed by humans or devices in their possession for subsequent aggregation and sharing purposes. In this paper, we focus on resource-limited crowdsensing, such as disaster response and recovery scenarios. The key challenge in those scenarios is to cope with resource constraints. Unlike the traditional application design, where measurements are sent to a central aggregator, in resource limited scenarios, data will typically reside at the source until requested to prevent needless transmission. Many applications exhibit dependencies among data items. For example, parts of a city might tend to get flooded together because of a correlated low elevation, and some roads might become useless for evacuation if a bridge they lead to fails. Such dependencies can be encoded as logic expressions that obviate retrieval of some data items based on values of others. Our algorithm takes logical data dependencies into consideration such that application queries are answered at the central aggregation node, while network bandwidth usage is minimized. The algorithms consider multiple concurrent queries and accommodate retrieval latency constraints. Simulation results show that our algorithm outperforms several baselines by significant margins, maintaining the level of service perceived by applications in the presence of resource-constraints.**

***Keywords*—crowd sensing; logical dependency; resource limitation; cost optimization**

I. Introduction

This paper advances the state of the art in crowdsensing in resource-starved environments. Applications, such as disaster response, stability and support operations, or humanitarian assistance missions often do not have the infrastructure and bandwidth to collect large amounts of data about the current state on the ground due to severe resource constraints (e.g., due to depletion, destruction, or acts of war). More data might be collected by the sources than what the infrastructure would allow them to communicate. To enhance situation awareness, a key question becomes to reduce the amount of resources needed to answer queries about current state. We specifically consider queries issued at a central command center that serves as an aggregation point for collected data. Some of the pertinent data needed to answer a query might have already been collected. The question lies in collecting the remaining data at minimum cost.

The novelty of this work lies in exploiting data dependencies and mission specifics to minimize the cost of answering command center queries. In the scenarios mentioned above, applications don't make random requests. They typically have very specific information objectives derived from well-defined protocols and doctrine. A disaster recovery team might follow well-defined protocols for carrying out their rescue objectives. These protocols call for specific information objectives that allow first responders to systematically perform their tasks. For example, in recovering from an earthquake, a disease control team might want to find out where the worst afflicted neighborhoods are, what structures can be used as shelters that are closest from these afflicted neighborhoods, whether they are currently occupied or not, and how to get there to carry out disease prevention procedures. To find an occupied, accessible shelter, one might use the following boolean expression: $(shelterOccupied) \wedge (path1IsGood \vee path2isGood \vee path3isGood)$, where the shelter's occupancy can be estimated using pictures taken from cameras installed in the shelter, and the road conditions by retrieving and examining aerial images taken by a different team. Hence, given a specific protocol and terrain knowledge, application requests can be automatically converted into series of boolean expressions comprised of logically inter-dependent tests on data from multiple sources.

We therefore argue that crowdsensing systems can be designed to exploit *application knowledge and logic dependencies among data items* to potentially reduce the underlying network bandwidth consumption. The paper shows that when one knows application logic and data dependencies, it is possible to plan the retrieval of objects in such a way that the cost of answering queries is significantly reduced. The practice is close to what an optimizing compiler might do when evaluating complex logical expressions. Namely, if possible, it will evaluate first the variables that might short-cut much of the rest of the logic expression. For example, a predicate that evaluates to $true$, ORed with a large expression, is sufficient to yield a positive answer, obviating the need for evaluating the rest of the expression. The contribution of this paper lies in applying this insight to the domain of crowdsensing, thereby significantly reducing resource demand attributed to data retrieval.

The approach is further enhanced by exploiting additional information about the likely values of variables that have not yet been retrieved. In the above example, if the disease prevention team, by looking at the population distribution data, can tell that the particular shelter is unlikely to be occupied, then they don't need to bother asking for aerial images covering the candidate paths—they can focus their attention on the next shelter. A datastore is maintained that keeps historical data and

978-1-4799-8856-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DCOSS.2015.26

various application-specific information, which can be used for making educated guesses about cost of object retrieval and likelihood that specific tests on data would evaluate to $true$ or $false$. Evaluation shows that our approach is effective at reducing resource consumption while meeting application information needs.

The remainder of this paper is organized as follows. After discussing related work in Section II, we give overall problem description and system design in Section III, and then present our detailed analyses and algorithms in Section IV. Evaluation is presented in Section V. Finally Section VI concludes.

II. Related Work

Crowdsensing and socialsensing have become an important channel of data inputs, given rise by the fast growing popularity of various mobile and wearable devices and their ubiquitous connectivities. For example, BikeNet [1] is a sensor network for bikers to share data and map regions. CarTel [2] is a mobile sensing system for automobiles, where data can be collected, processed, and visualized. Coric et al. [3] design a crowdsensing system that helps to identify legal parking spaces. MaWi [4] is an indoor localization system with improved accuracy by relying on crowdsensing for spot survey. Hu et al. [5] design and implement the SmartRoad system that takes advantage of vehicular crowdsensing data to automatically detect and recognize traffic lights and stop signs. Given the richness of crowdsensing systems, to help cleanup and therefore better utilize crowdsensing data, various fact-finding techniques [6]–[8] have been proposed. The crowdsensing system design reported in this paper complements these prior studies by exploiting logic relations among data items when operating under resource limitations.

Resource-limited scenarios (e.g., disaster monitoring, alert, and response) have been studied in the community. For example, Breadcrumb [9] is an automatic and reliable sensor network for the firefighting situations. In the SensorFly [10] project, low cost mobile sensing devices are utilized to build an indoor emergency response system. PhotoNet [11] provides a post-disaster picture collection and delivery service for situation awareness purpose. Our crowdsensing system design can be used to operate on the above mentioned systems in improving communication efficiency under limited resources.

The goal of improving communication efficiency and data/information quality is shared with many existing studies. For example, time-series prediction techniques are used to reduce communication burden without compromising user-specified accuracy requirements in wireless sensor networks [12]. Regression models are used to estimate predictability and redundancy relationships among sensors for efficient sensor retrievals [13]. MediaScope [14] is a crowdsensing system with various algorithms designed to help with timely retrievals of remote media contents (e.g. photos on participants' phones) upon requests of multiple types (nearest-neighbor, spanners, etc). Gu et al. [15] design inference-based algorithms for data extrapolation for disaster response applications. Minerva [16] and Information Funnel [17] explore data prioritization techniques based on redundancy or similarity measures for information maximization. Data aggregation techniques are also studied to reduce network transmission and improve classification tasks' accuracies [18]–[20]. Gregorczyk et al. [21] discuss their experiences on in-network aggregation for crowdsensing deployments. Tham et al. [22] propose the Quality of Contributed Service as a new metric for crowdsensing systems. Different from the above work, our system design exploits logic relations among data items, and can perform cost optimization under deadline constraints.

In the theory community, the optimization of boolean predicate evaluation has been thoroughly studied. Greiner et al. [23] analyze and give theoretical results for the various subspaces of the general PAOTR (probabilistic and-or tree resolution) problem. Luby et al. [24] propose a set of tools for analyzing the probability an and-or tree evaluates to $true$. Casanova et al. [25] give various heuristic-based algorithms for the general NP-hard PAOTR problem and show performance comparisons. While heavily benefiting from the above theoretical studies, our work also extends beyond them as we apply to practical sensing scenarios where we also need to handle concurrent requests, deadline constraints, and existing partial retrieval sets when computing the optimal evaluation plans.

Most closely related to our work, Carlog [26] and ACE [27] also explore similar query request optimization techniques. In particular, Carlog explores latency optimization for vehicular sensing applications, and ACE aims at energy efficiency for continuous mobile sensing applications. However, ACE does not consider multi-request cases, and even though Carlog does, it limits requests to be of the conjunction form only. Furthermore, neither model the deadlines of query requests. Our data retrieval algorithm assumes no boolean expression form limitation, and can be used for optimizing cost under specified request deadline constraints.

III. System Overview

We consider crowdsensing applications (such as disaster response and recovery), where information at data sources is not immediately accessible to the collection point due to resource constraints. Retrieval of data from sources needs to be carried out very carefully such that the least amount of resource is used. Decisions based on received data need to be made at different time-scales depending on their criticality. Hence, in addition to resource bottlenecks, we have different application deadlines on information acquisition.

The goal is to fetch data from sources in a way that minimizes system resource consumption while meeting request deadlines. In general, a decision-maker might have several information needs, each translating into a query. Hence, multiple queries could be issued at the same time. We assume that each query is translated into a set of objects that must be retrieved. A retrieved object can be subjected to a test that evaluates a predicate. For example, an image of a building can be inspected to determine if the building is occupied. Given application logic and data dependencies, it is desired to determine which objects to retrieve in order to answer all queries with minimum resource consumption while meeting their deadlines.

In the rest of the paper, we describe cost as the bandwidth needed for retrieving a data item (i.e., the size in bytes of the retrieved data). However, our actual algorithm is not restricted to this cost definition—it can operate on any cost metric definition that is additive. For example, in a scenario where energy is the most important resource, we can easily define cost to be a node's energy consumption. We assume that the concurrent retrieval latency of multiple data object is the maximum single retrieval latency among all the single objects.

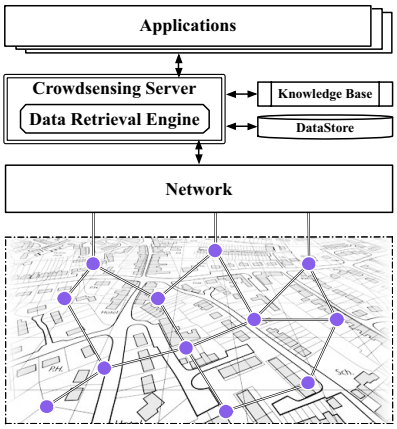


Fig. 1: System Design

Our resource-limited crowdsensing system design is depicted in Fig. 1. As seen, as application requests arrive, our crowdsensing engine uses the knowledge-base to convert the requests into boolean logic expressions. It then queries the datastore for the likelihood that different predicates in those expressions would evaluate to $true$ or $false$, to guide object retrieval accordingly for best odds of shortcutting logic expressions. It also fetches the estimated retrieval latencies between the collection point and the remote data sources. The data retrieval engine takes these estimates as input, computes a minimum cost retrieval plan, and performs the actual data retrieval. As data comes in, requests are updated, and the datastore is enriched with the latest data. Note that, in general, some data pertinent to a query might already be available. If so, part of the expression is immediately evaluated. The above applies to the remaining part, if any. Hence, without loss of generality, we focus on retrievals of missing data only.

IV. Analyses and Algorithms

In this section, we first talk about what analyses and building-block operations need to be done for each individual crowdsensing application request, and then discuss how the crowdsensing system carries out data retrievals when serving multiple requests concurrently under our targeted resource-constrained settings.

A. Single-Request Analyses

Given a single request q (please refer to Table I for all notations), we would like to compute the best (cost-minimizing) next source $n_E(q)$ for retrieving data from and the corresponding expected bandwidth cost $c_E(q)$ assuming sequential retrieval for all the rest of the involved data sources. We focus our discussion on bandwidth only, as the same methods can be used for the expected retrieval latency. Notice that this is generally *not* true because latencies for concurrent retrievals are not additive. But since all expected values that we discuss in this section are under the assumption of sequential retrievals, this *is* true.

Please note that we default to sequential retrieval because of our original objective of minimizing costs—more parallel retrievals lead to higher probability of unnecessarily retrieving data objects that could otherwise be avoided due to short-circuiting. Only when a query might miss its deadline if we stick to sequential retrieval do we schedule concurrent retrievals in order to decrease the total query resolution time.

Q	all requests	$n_E(q)$	the src leads to min $c_E(q)$
q	a request	$n_E(q,\mathcal{R})$	$n_E(q)$ given $\mathcal{R}$
$\mathcal{R}$	srcs to retrieve	$l_E(q)$	**E**[latency] of q
r_k	k^{th} src in $\mathcal{R}$	$l_E(q,\mathcal{R})$	$l_E(q)$ given $\mathcal{R}$
n_j	data source j	$c_E(q)$	**E**[cost] of q
s_j	true prob. of n_j	$c_E(q,\mathcal{R})$	$c_E(q)$ given $\mathcal{R}$
c_j	cost of n_j	$q_{\|\bullet}$	apply data of src $\bullet$ to q
$\mathcal{M}$	retrieved srcs	q_l	request of least laxity
p_i	test i	b_i	short-circuit prob. of p_i

TABLE I: Notation Table

We need to not only be able to compute the next best source and the expected cost when given just the request, but be able to do so when also facing a set $\mathcal{R}$ of (zero or more) sources that have already been selected for the next round of data retrieval. Formally, for a request q, let P be its set of tests, where each test p_{q_i} corresponds to a particular data source n_j, and thus has associated with it a retrieval bandwidth cost c_{n_j} and success (evaluating to $true$) probability s_{n_j}. Given a set $\mathcal{R}$ of data sources that are already selected for retrieval for the next round, we want to compute for request q the next best source $n_E(q,\mathcal{R})$ to retrieve data from and q's corresponding expected cost $c_E(q,\mathcal{R})$. The reason why we need to do this will be more clear, if not already so, when we talk about the actual data retrieval algorithm for multiple concurrent requests in Sec. IV-B.

If set $\mathcal{R}$ is empty, then the problem becomes the standard PAOTR problem, already known to be NP-hard. It is quite obvious that it is still so with an non-empty $\mathcal{R}$. Casanova et al. [25] give a greedy algorithm for computing q's best data retrieval plan without such a set $\mathcal{R}$ present. We denote it as $c_E(q,\varnothing) = c_E(q)$, and briefly introduce how this greedy algorithm works. For a test p_i with cost c_i and success probability s_i, we define the test's short-circuit probability to be the probability that performing it will cause its parent conjunction or disjunction to be resolved without needing to perform its sibling tests. Then p_i's short-circuit probability to cost ratio is $b_i = (1-s_i)/c_i$ in a conjunction, or $b_i = s_i/c_i$ in a disjunction. As each boolean expression can be converted into its disjunctive normal form (i.e., a disjunction of one or more conjunctions of one or more tests), for a conjunction we order its tests according to their (descending) b values. With this order $j_1, j_2, \ldots, j_m$, the expected cost of the conjunction can then be computed as

$$c_{conj} = c_{j_1} + s_{j_1}(c_{j_2} + s_{j_2}(c_{j_3} + \ldots + s_{j_{m-1}} c_{j_m})\ldots),$$

which, together with its derived success probability $\prod s_i$, yields the current conjunction's short-circuit probability to cost ratio $\frac{\prod s_i}{c_{conj}}$ for its parent disjunction, which can then be used to select the best conjunction to process first, the same way how the best test for a conjunction is selected.

After each test p_t is selected, the next best test p_{t+1} to be selected can depend on the evaluation result of p_t. So, both possibilities $p_{t+1}^{(T)}$ (upon $p_t == true$) and $p_{t+1}^{(F)}$ (upon $p_t == false$) are considered. The final retrieval plan computed is therefore a binary decision tree, where the evaluation result of each test performed dictates which source to retrieve data from and test to perform next. For ease of presentation, we use $n_E(q)$ and $c_E(q)$ to denote the algorithms for computing a request q's next best source to retrieve data from (i.e., the root node of the binary decision tree) and the corresponding

expected bandwidth cost. Please note that data sources' costs and the corresponding tests' success probabilities are the inputs to the mentioned algorithms. We, however, assume their static availabilities throughout our discussions, and thus omit them as being part of the input.

Note that due to the binary decision tree nature of the algorithm $n_E(q)$, it has exponential run time in request q's number of tests. This is not a problem in practice where a request contains say 5 to 10 tests. If, however, cases need to be handled where inputs are quite large (for example, a request consists of hundreds or thousands of tests in it), a simplified (and computationally more efficient) version of algorithm $n_E(q)$ can be used, which simply ranks all the conjunctions first and then proceeds with data retrievals according to the conjunction order and then the data source order within the same conjunction. This effectively reduces the computation complexity from exponential growth to $\mathcal{O}(|q| \lg |q|)$, and only slightly underperforms the original version in terms of ratio to optimal retrieval plan [25].

Algorithm 1 $c_E(q, \mathcal{R})$ - Request's Minimum Expected Bandwidth Cost Given A Predetermined Retrieval Set

Input: Request q, predetermined retrieval set $\mathcal{R}$
Output: The minimum expected bandwidth cost of q given $\mathcal{R}$

1: **if** q is already resolved **then return** 0
2: **else**
3: **if** $\mathcal{R} == \varnothing$ **then**
4: **return** $c_E(q)$
5: **else**
6: $r_l \leftarrow$ last element of $\mathcal{R}$
7: $\mathcal{R} \leftarrow \mathcal{R} \setminus \{r_l\}$
8: $q_{|r_l^{(T)}}, q_{|r_l^{(F)}} \leftarrow$ request q with $r_l = true, false$ values applied, respectively
9: **return** $s_{r_l} c_E(q_{|r_l^{(T)}}, \mathcal{R}) + (1 - s_{r_l}) c_E(q_{|r_l^{(F)}}, \mathcal{R})$
10: **end if**
11: **end if**

In our case, with the predetermined retrieval set $\mathcal{R}$ present, we can compute request q's expected cost in a recursive fashion. The base case would be when set $\mathcal{R}$ is empty, where the algorithm $c_E(q)$ mentioned above can compute q's cost directly. The recursive case goes as follows. For a set $\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$ with l elements, we remove from it its last element r_l, apply r_l's two possible values $true/false$ to the request and get two sub-requests $q_{|r_l^{(T)}}$ and $q_{|r_l^{(F)}}$. Then the expected cost of q given $\mathcal{R}$ can be computed recursively as follows,
$c_E(q, \mathcal{R}) = s_{r_l} c_E(q_{|r_l^{(T)}}, \mathcal{R} \backslash \{r_l\}) + (1 - s_{r_l}) c_E(q_{|r_l^{(F)}}, \mathcal{R} \backslash \{r_l\})$,
where s_{r_l} is the success probability of the test r_l. The pseudo code is depicted in Algorithm 1.

Again, due to the structure of the binary decision tree, the computational complexity of Algorithm 1 grows exponentially with the cardinality of the set $\mathcal{R}$. When fast decision makings on large inputs are needed, applications can opt to the static version of Algorithm 1 which only explores the most probable value of r_l, avoiding the exponential explosion. As the static algorithm travels a single path from root to leaf on the binary decision tree, the computational complexity is linear in $|\mathcal{R}|$. Therefore the total complexity is $\mathcal{O}(|\mathcal{R}| + |q| \lg |q|)$

Regarding how to compute, for request q, the best source $n_E(q, \mathcal{R})$ to retrieve data from next given the set $\mathcal{R}$, we let each member $r_l \in \mathcal{R}$ assume its most probable outcome determined by its success probability s_{r_l}, and apply this set of most probable values to q and get the resulting request q'. After that we can compute the best source as $n_E(q, \mathcal{R}) = n_E(q')$. The pseudo code is depicted in Algorithm 2. Similar to the notation $c_E(q, \mathcal{R})$ and $n_E(q, \mathcal{R})$, which denote the algorithms for computing request q's best data source and expected cost given predetermined retrieval set $\mathcal{R}$, we use $l_E(q, \mathcal{R})$ to refer to the algorithm for computing q's expected retrieval latency given a retrieval set $\mathcal{R}$.

Algorithm 2 $n_E(q, \mathcal{R})$ - Request's Best Next Source for Retrieval Data from Given A Predetermined Retrieval Set

Input: Request q, predetermined retrieval set $\mathcal{R}$
Output: q's best next data source for retrieval given $\mathcal{R}$

1: **if** q is already resolved **then return** $\varnothing$
2: **else**
3: **if** $\mathcal{R} == \varnothing$ **then**
4: **return** $n_E(q)$
5: **else**
6: $\mathcal{V}_{\mathcal{R}} \leftarrow$ the vector of most probable outcomes of tests corresponding to $\mathcal{R}$
7: $q_{|\mathcal{V}_{\mathcal{R}}} \leftarrow$ request q after applying all values from $\mathcal{V}_{\mathcal{R}}$
8: **return** $n_E(q_{|\mathcal{V}_{\mathcal{R}}})$
9: **end if**
10: **end if**

The computational complexity of Algorithm 2 is dominated by the sorting operation in $n_E(\cdot)$, resulting in $\mathcal{O}(|q| \lg |q|)$.

B. Data Retrievals for Multiple Concurrent Requests

Having established several building block algorithms, we are now ready to talk about the data retrieval algorithm for serving multiple concurrent requests. We first describe the various component stages, and then give the complete algorithm at the end.

At each round, the data retrieval engine needs to determine, from the set of all relevant data sources, a subset to retrieve data from. It proceeds by first selecting the one that leads to the minimum total expected cost. This is done by taking the Cartesian product of the set of all data sources and set of all requests, and for each source-request pair (n_i, q_j), computing the expected cost $c_E(q_j, \{n_i\})$. Then, the best source n_r is selected as the data source n that minimizes the sum of its own cost c_n plus the collective sum of the expected costs of all the requests, assuming the predetermined set of sources being the current retrieval set with n inserted, mathematically (and more clearly) as follows,

$$n_r = \arg\min_n \left(c_n + \sum_j c_E\left(q_j, \mathcal{R} \cup \{n\}\right) \right).$$

After n_r is selected, it is added to the retrieval set for this round $\mathcal{R} = \mathcal{R} \cup \{n_r\}$.

The data retrieval engine then checks the expected laxity (laxity = deadline − expected retrieval latency) of all requests. If the minimum expected laxity is negative, then it means the corresponding request q_l is expected to miss its deadline if we proceed with retrieval using the current $\mathcal{R}$ as is. Since all sources in $\mathcal{R}$ are to be retrieved in parallel, having more of q_l's relevant data sources in $\mathcal{R}$ for parallel retrieval might then shorten its expected retrieval latency and in turn make its laxity positive again. Under this intuition, we then pick q_l's best next

source $n_E(q, \mathcal{R})$ to add to $\mathcal{R}$ and carry out this laxity checks again with the updated $\mathcal{R}$. This process terminates when no request is expected to miss its deadline given the latest set $\mathcal{R}$. It might also happen that at some point some request q_l cannot be saved from having to miss its deadline, for example when the retrieval latency of set $\mathcal{R}$ already exceeds q_l's deadline, therefore adding more sources to $\mathcal{R}$ won't help in terms of saving q_l. Under such circumstances we have two options, either backtrack in $\mathcal{R}$ to try to save q_l, or accept that q_l might miss its deadline and disregard it for this current round of planning. In this paper we take the latter approach, and leave the former for our future efforts.

As we have taken care of bandwidth and deadlines at this point, it might seem like we are ready to carry out the retrieval of $\mathcal{R}$ for this round; our data retrieval engine, however, actually goes one step further here in preventing unnecessary request delays, as we explain next. We say two sources n_1 and n_2 are *related* if they co-appear in at least one request q. It then follows that the retrieval of n_1 will affect how n_2 is to be chosen in the future, because n_1's data upon retrieval will be applied to the request q, causing $c_E(q, \{n_2\}) \neq c_E(q_{|n_1}, \{n_2\})$. It is easy to see that this effect is transitive: If we build a graph of all the sources where each pair of sources share an edge iff they are *related*, then sources of the same connected component can affect each other in this fashion, either in the next round or after multiple rounds in the future. A pair of sources that are *disconnected*, however, will never affect each other. We then say two sets of sources are *disconnected* if no edge exist between them.

Algorithm 3 Multi-Request Serving

Input: The set $\mathcal{M}$ of retrieved data from previous round, the set $\mathcal{Q}$ of all unresolved requests, current time T
Output: The set $\mathcal{R}$ of sources to retrieve data from for this round

1: Use values in $\mathcal{M}$ to update all requests in $\mathcal{Q}$, remove all resolved requests from $\mathcal{Q}$
2: Initialize $\mathcal{R} \leftarrow \varnothing$
3: **repeat**
4: $n_{min} \leftarrow \arg\min_n \left(c_n + \sum_j c_E(q_j, \{n\})\right)$ the source that minimizes the total expected retrieval cost
5: $\mathcal{R} \leftarrow \mathcal{R} \cup \{n_{min}\}$
6: $\mathcal{L} \leftarrow \{q \in \mathcal{Q} | q.deadline - l_E(q, R) - T < 0\}$ the set of requests that are expected to miss their deadline after this round of retrieval
7: **repeat**
8: $q_{urgent} \leftarrow \arg\min_{q \in \mathcal{L}} (q.deadline - l_E(q, R) - T)$
9: $n_{q_{urgent}} \leftarrow n_E(q_{urgent}, \mathcal{R})$
10: **if** $n_{q_{urgent}} == NULL$ **then**
11: $\mathcal{L} \leftarrow \mathcal{L} \setminus \{q_{urgent}\}$
12: $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{q_{urgent}\}$
13: **else**
14: $\mathcal{R} \leftarrow \mathcal{R} \cup \{n_{q_{urgent}}\}$
15: **end if**
16: **until** $\mathcal{L} == \varnothing$
17: $\mathcal{Q} \leftarrow \{q \in \mathcal{Q} | q$ is disconnected from $\mathcal{R}\}$
18: **until** $\mathcal{Q} == \varnothing$
19: **return** $\mathcal{R}$

Now coming back to our multi-request data retrieval problem, if all sources in a request q_d are disconnected from the set $\mathcal{R}$, then there is no benefit for us not to start retrieving data from some source relevant to q_d at this current round, because we will not be saving total network bandwidth usage, and are just delaying the processing of this request, adding to the danger of it missing its deadline. Therefore, our data retrieval algorithm behaves as follows. At the beginning of each round (at which point $\mathcal{R} == \varnothing$), a set $\mathcal{Q}$ is initialized to contain all requests. As $\mathcal{R}$ is populated, elements of $\mathcal{Q}$ are removed from $\mathcal{Q}$ only if necessary s.t. $\mathcal{R}$ and $\mathcal{Q}$ remain disconnected. The previously mentioned cost-minimizing and deadline-miss-prevention operations are actually always carried out on the set $\mathcal{Q}$, until $\mathcal{Q}$ becomes empty, at which point, the data retrieval engine is actually ready to retrieve data from all sources in $\mathcal{R}$ in parallel. Upon receiving the data, the engine updates all the requests, and continues onto the next round of planning. The above detailed discussions are also collected and depicted in Algorithm 3.

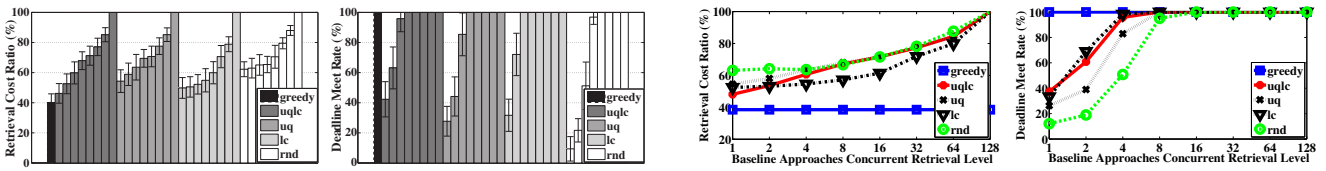
Assuming there are n sources and m requests. Then the outer loop repeats at most m times. In each iteration, the computation on line 4 costs $\mathcal{O}(n^2)$ time when the static version of algorithm $c_E(q, \mathcal{R})$ is in use, dominating the computational cost from line 4 to 6. The inner loop repeats n times in the worst case. Line 8 costs $\mathcal{O}(mn)$ time to enumerate all requests and compute their latencies, which is the most expensive operation in each inner iteration. Therefore, the computational complexity of Algorithm 3 is $\mathcal{O}(m(n^2+n(mn))) = \mathcal{O}(m^2n^2)$. In practice, this algorithm is highly parallelizable, more specifically the operations on Line 4, 6, and 8. Thus in case of extremely large inputs (e.g. hundreds of concurrent requests with thousands of relevant data sources), parallel computing techniques (e.g. GPGPU) can be utilized to boost performance. For more realistic input sizes the computing time is far smaller than the network transmission delay time for our targeted scenarios.

V. Evaluation

In this section we evaluate our proposed data retrieval algorithm (which we will mark as $greedy$ for ease of presentation) for resource-limited crowdsensing through simulations. First, we examine our algorithm's performance against several baseline methods under various settings. Then we specify a concrete application scenario and present our results and findings.

We compare our algorithm to the following four baseline methods in the evaluation.

- *Random* (*rnd*) — The elements of the retrieval set for each round are randomly selected from the set of all relevant data sources of the remaining requests. Here we experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time.
- *Lowest Cost Source First* (*lc*) — All requests' relevant data sources are put together and sorted according to their costs in ascending order. The retrieval set is then populated using the lowest cost sources. We also experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time.
- *Most Urgent Request First* (*uq*) — All requests are sorted according to their expected laxities in ascending order. The retrieval set is then populated by randomly selecting sources from the most urgent (least laxity) request. We also experiment with selecting $1, 2, 4, 8, \ldots, 128$ sources at a time. Note that if the most urgent request's relevant sources have been exhausted, we move to the next urgent request in line.



(a) Retrieval cost ratios (b) Deadline meet rates (c) Retrieval cost ratios (w/o std) (d) Deadline meet rates (w/o std)

Fig. 2: Retrieval cost ratios and deadline meet rates of our algorithm vs baseline methods

- *Most Urgent Request's Lowest Cost Source First* (*uqlc*) — The same as the *uq* approach, except that when selecting sources from the urgent request, the lowest cost source is selected first, as opposed to random selection.

A. Algorithm Behaviors

We examine our algorithm's behaviors under the following experiment settings. All requests are randomly generated. We experiment with different numbers of tests per request ($\{4, 8, 12, 16, 20\}$, indicating how many remote data sources are generally involved in a single crowdsensing application request), different numbers of conjunctions per request ($\{1, 3, 5, 7\}$, which could possibly correspond to the number of different options that a sensing application can take), and different numbers of concurrent requests that the crowdsensing system is serving ($\{5, 10, 15, 20, 25\}$, the number of sensing applications are being served all at the same time). All tests' success probabilities are uniformly randomly generated. Besides the uniform distribution on $(0, 100\%)$, we also experiment with ranges $(0, 10\%) \cup (90\%, 100\%)$ and $(40\%, 60\%)$, which represent situations where tests' success probabilities are generally less and more ambiguous in indicating how likely they are to evaluate to either $true$ or $false$ than the $(0, 100\%)$ case. Regarding sources' retrieval costs, we experiment with uniformly on $[1, 10]$, and 2^p where the power p is uniformly randomly generated on $[1, 10]$. Compared to the former, the latter represents a highly heterogeneous network where data collected by different sources can have orders of magnitude of differences in bandwidth costs. We set a source's retrieval latency to be proportional to its bandwidth cost plus a transmission setup latency, and for the actual request deadlines, we experiment with the following different settings: For a request, we take the sum of all its sources' individual retrieval latencies, and multiply it by 0.5, 1, 2, 4, or 8 to use as the request's deadline, representing different levels of "tightness" of the request deadlines.

We first take a look at how our algorithm's performance compares to the baseline methods in general. We set the total number of data sources in the field to be 100, whose bandwidth costs are uniformly distributed on $[1, 10]$ and success probabilities $(0, 100\%)$. We set the number of concurrent requests to be 20. Each request involves 8 individual tests grouped in 3 different conjunctions, and has its deadline set to be twice the sum of its relevant sources' individual retrieval latencies. Each set of experiments is repeated 20 times, for which we report the means and standard deviations of both the retrieval cost ratio, computed as

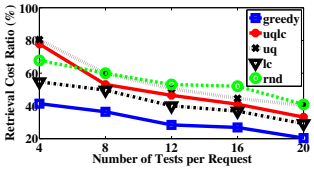
$$\frac{\text{total bandwidth cost of actually retrieved data}}{\text{total bandwidth cost of all sources involved in all requests}},$$

and the deadline meet rate—the percentage of requests whose resolutions are within their deadlines. Therefore, ideally we want low retrieval cost ratios and high deadline meet rates. Results are shown in Fig. 2, of which Fig. 2a shows the retrieval cost ratios and Fig. 2b the deadline meet rates. In each plot, the heights of the bars represent the mean values, and the error windows indicate the standard deviations. Results of our algorithm are captured by the leftmost bar, whereas each of the four baseline methods is represented by a cluster of adjacent bars with the same greyscale level, from left to right: $uqlc$, uq, lc, and rnd. Within the same baseline method cluster, the different bars stand for different concurrent retrieval levels: from left to right in the order 1, 2, 4, 8, ..., 128. For the sake of easier comparisons, the same results are also presented in a different manner without the standard deviation error windows, as illustrated in Fig. 2c and 2d.

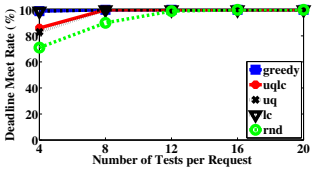
As can be seen, our data retrieval algorithm achieves a low retrieval cost ratio of about 40%, while maintaining a 99.8% average deadline meet rate. In comparison, among all the baseline strategies, the lowest retrieval cost ratio with 99+% deadline meet rate is achieved by lc with 4 concurrent retrievals at each round. Its retrieval cost ratio is about 52%, quite a bit higher than the 40% of our algorithm. For the rest of the baseline methods, in order to achieve 90+% deadline meet rate, the retrieval cost ratios have to be 60% or even greater.

After getting a general sense of how our algorithm and all the baseline strategies perform against each other, we would next like to investigate how various aspects affect the data retrieval algorithm's behaviors. Thus for each of the next sets of experiments, we tune one of the system and problem parameters while fixing all the rest to their default values as set in the previous experiment, and look at how our algorithm stacks up against the baseline methods. Note that for clarity of presentation, we pick, from each baseline strategies, a representative concurrent retrieval level. After examining results from all baseline strategies, we notice that with a concurrent retrieval level of 4, usually at least one of the four baseline methods can achieve 100% deadline meet rate without incurring additional retrieval costs. Therefore, we pick the concurrent retrieval level of 4 to be the representative of each of the baseline strategies. Each experiment is repeated 20 times. In order to better show the trend of changes reflected from the results, we omit showing standard deviations, and focus on the average results, similar to Fig. 2c and 2d.

First we look at how the number of tests in a request affects the data retrieval algorithm's performance. This in practice could indicate the general complexity of the requesting crowdsensing applications. For example, an application request from a user who just wants to find out if she could go get gas from either of the two gas stations near her home could be $gasStationAIsOpen \vee gasStationBIsOpen$, which is rather simple; another application request from a

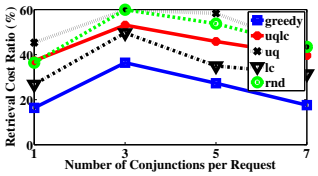


(a) Retrieval cost ratios

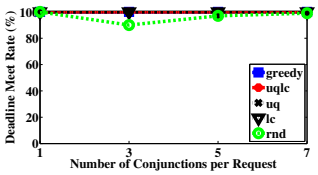


(b) Deadline meet rates

Fig. 3: Varying Number of Tests per Request

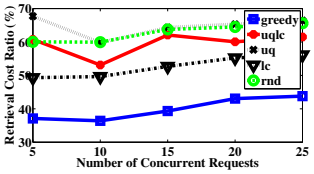


(a) Retrieval cost ratios

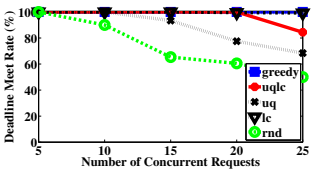


(b) Deadline meet rates

Fig. 4: Varying Number of Conjunctions per Request

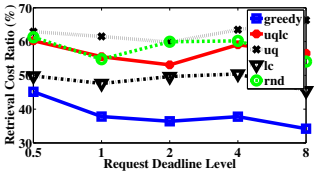


(a) Retrieval cost ratios

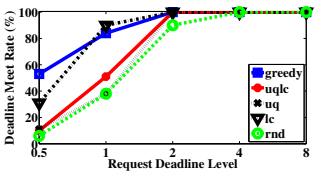


(b) Deadline meet rates

Fig. 5: Varying Number of Concurrent Requests



(a) Retrieval cost ratios



(b) Deadline meet rates

Fig. 6: Varying Request Deadlines

disaster response team that needs to find out how to reach their destination given three candidate routes could look like $(roadAIsGood \wedge roadBIsGood) \vee (roadAIsGood \wedge roadCIsGood \wedge roadDIsGood) \vee (roadEIsGood \wedge roadFIsGood)$, which is a bit more complex than the previous gas station example.

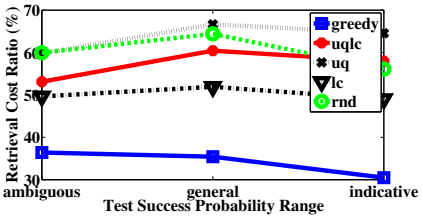
Results are shown in Fig. 3. As seen, our algorithm always achieves the lowest retrieval cost ratios with perfect/near-perfect deadline meet rate. We can observe that as the number of tests per request increases, the retrieval cost ratios of all approaches decrease. This is because more tests per request means generally more tests per conjunction (the number of conjunctions per request is fixed in this experiment). Having more tests included in a request increases the deadline of the request, but then there are more chances for tests within the same conjunction to be short-circuited by a sibling of theirs. Therefore, the total retrieval cost for resolving a request might not increase with the additional tests. So, we observe an downward trend of the retrieval cost ratio and an upward trend of deadline meet rate, as we increase the number of tests in requests. Among the baseline methods, the rnd (random) approach generally gives the worst results, which is expected. The lc (least cost sources first) approach leads to the second best retrieval cost ratio, which is understandable as minimizing cost is its sole objective. It is interesting to see that lc also leads to perfect deadline meet rate, considering the method itself does not concern requests' deadlines at all. One explanation we can think of is that since it always picks the lowest-cost sources, which generally also leads to small retrieval latencies, lc can therefore retrieve more data and fast, which helps it with request serving abilities.

Besides the number of tests, we also look at how the number of conjunctions within a request affects data retrieval algorithm's behavior. This in practice could correspond to the number of different options an application request can be satisfied. Taking the previous disaster-response-team-finding-routes example, the three conjunctions in the request correspond to 3 alternative routes that can be taken (of course different routes could possibly share common road segments among them). We experiment with 1, 3, 5, and 7 conjunctions per request, and show results in Fig. 4. Nothing interesting going on in Fig. 4b for deadline meet rates, we focus our attention to the retrieval cost ratios in Fig. 4a. Besides the obvious fact that our algorithm achieves the best retrieval cost ratios, we do see an interesting common trend among all approaches—the retrieval cost ratios generally decrease with a growing number of conjunctions per request except for when there is only one conjunction present in a request. This makes sense because the more conjunctions there are within a request (which has a fixed number of tests), the more the request itself looks like a disjunction, which means the more likely it is for a single test's evaluating to $true$ to short-circuit all other tests and completely resolve the request. When there is only one conjunction, however, the request itself is then just, well, a conjunction, which means a single test's evaluating to $false$ would completely resolve the request, and hence the low retrieval cost ratio.

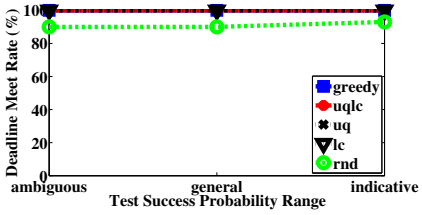
We next look at how the number of concurrent requests (which can be a measure of the load on the crowdsensing system) affects performance of various methods. Results are shown in Fig. 5. As seen, the increase in the number of concurrent requests has a pronounced effect on (increasing) the retrieval cost ratios and (decreasing) the deadline meet rates on all baseline methods, which is quite expected. We also observe that our algorithm still always achieves the lowest retrieval cost ratio, and, unlike many of the baseline methods, is able to maintain its perfect deadline meet rate in spite of the increase in the request quantities.

Next we look at how different request deadline settings affect the behaviors of our algorithm and the various baseline methods. Different deadlines obviously express different levels of urgency of the application requests. For example, a medical team wanting to find ways to get to a collapse site to save lives obviously would issue their route finding request with a tight deadline; a family looking for the nearest under-occupied shelter after losing their home to a hurricane probably doesn't need their request fulfilled with the same level of urgency.

We set a request's deadline to be a multiple of the sum of its sources' individual retrieval latencies. We experiment with multipliers 0.5, 1, 2, 4, and 8. Results are shown in Fig. 6. First of all, none of the baseline approaches concern requests' absolute deadline values, which is why in Fig. 6a we do not see a clear trend of change among baseline approaches. Looking at Fig. 6b, it's quite clear that the multiplier setting of 0.5 and

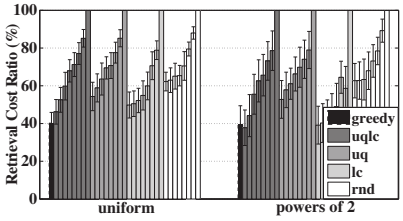


(a) Retrieval cost ratios

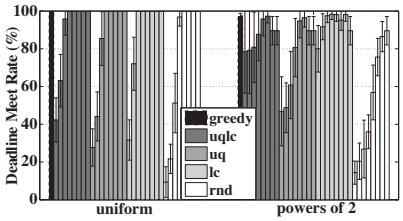


(b) Deadline meet rates

Fig. 7: Varying Test Success Probability Ranges

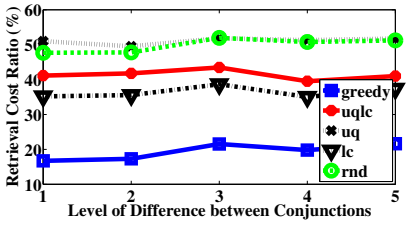


(a) Retrieval cost ratios

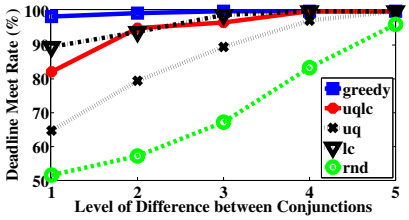


(b) Deadline meet rates

Fig. 8: Varying Levels of Sources' Retrieval Costs



(a) Retrieval cost ratios



(b) Deadline meet rates

Fig. 9: Varying Levels of Conjunction Differences

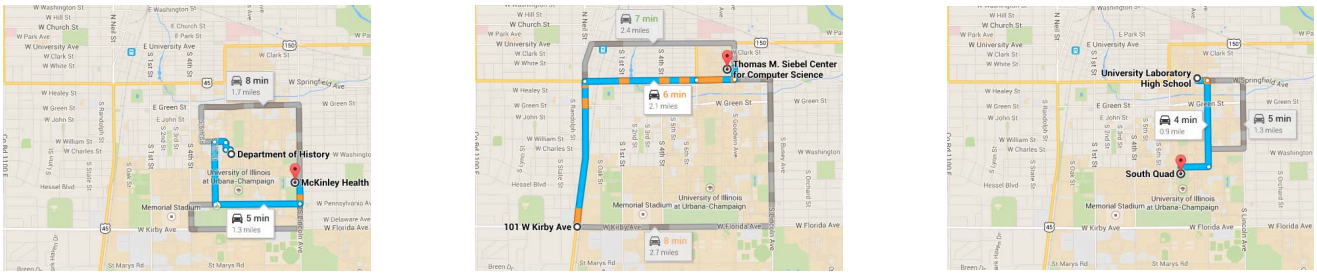
1 represent quite strict deadlines—even our algorithm results in significant deadline misses. Under a multiplier setting of 2, most approaches start to no longer struggle with avoiding deadline misses. As deadlines become more and more relaxed, we observe a trend of decrease in terms of the retrieval cost ratio of our algorithm. This is because the more relaxed requests' deadlines are, the less likely our algorithm is forced to include sources in each round for parallel data retrieval just to avoid deadline misses for some requests, which might otherwise be unnecessary and suboptimal.

As our algorithm exploits the logic relations among the tests within requests, it would be interesting to see how tests' success probabilities affect the performance of our data retrieval algorithm. These probabilities represent a measure of how good of a prior knowledge we already have regarding the crowdsensing environment. Intuitively, if the probabilities of all the tests evaluating to $true$ (or $false$) are around 50%, our algorithm will make more wrong guesses when it tries to retrieve data from sources in hoping they can help short-circuit other sources. On the other hand, if the probabilities are very close to 100% (or 0), indicating that a test is extremely likely to evaluate to $true$ (or $false$), then our algorithm will be correctly picking the right sources to retrieve data from for short-circuiting other sources more often. We experiment with uniformly generating success probabilities for all the tests within three different ranges, namely the quite ambiguous $(40\%, 60\%)$, the general $(0, 100\%)$, and the quite indicative (less ambiguous) $(0, 10\%) \cup (90\%, 100\%)$. Results are shown in Fig. 7. As baseline methods do not take into consideration these probabilities, we only look at our algorithm. As seen, our algorithm's retrieval cost ratio gradually improves as tests' success probabilities become less ambiguous, coinciding with our intuition.

One other interesting aspect to look at is the level of retrieval cost heterogeneity among different data sources. In other words, in a more homogeneous setting, sensors/sources tend to be of similar types with each other, generating data of closer natures and comparable retrieval bandwidth costs. In a highly heterogeneous setting, however, we can imagine sources being of drastically different natures. For example, for a network composed of carbon monoxide sensors, cameras, human reporters, and microphones, data generated by different sources could have retrieval costs differing from each other over multiple orders of magnitude. In order to capture this difference, we experiment with two different ways of setting sources' retrieval costs, one being uniformly on $[1, 10]$, and the other 2^p where p is uniformly randomly generated on $[1, 10]$. We include all concurrent retrieval levels for all baselines, and use the bar-with-error representation similar to Fig. 2. Results are shown in Fig. 8. As seen, when retrieval costs differ vastly from sources, all approaches' retrieval cost ratio standard deviations increase. From looking at Fig. 8a alone, it might seem like the single-source-per-round $uqlc$ scheme and lc scheme slightly outperform our algorithm in terms of retrieval cost ratio, but it can be seen from Fig. 8b that these two schemes lead to about 20% deadline miss rates, whereas for our algorithm only about 2.5% requests experience deadline misses. Even though our algorithm still gives the best performance, we do observe that the highly heterogeneous setting causes larger degree of deviations, compared to when sources are less different from each other. We are currently working on improving our algorithm to better handle crowdsensing scenarios where data sources' costs are substantially different from each other.

One last interesting aspect that we look at is the similarity between conjunctions, or how similar an application's different options are. Looking again at the previous route finding example, in an urban area where roads are highly interconnected, all alternative routes might share a lot of common road segments among each other; whereas for a more rural setting, the different possible routes might share few common road segments because roads are quite sparsely connected in the first place.

We set the number of conjunctions per query to 6 and the number of test per query to 6 as well. With 10 concurrent queries, we let, in each request, adjacent conjunctions differ by 1, 2, 3, 4, or 5 tests, and examine the data retrieval algorithm's performance using the different approaches. Results



(a) For the medical team (b) For the communication team (c) For the evacuation team

Fig. 10: Candidate Routes Map Illustrations (All candidate routes are shown, with the shortest routes highlighted.)

are shown in Fig. 9. As seen, our algorithm still outperforms all baseline methods, achieving perfect/near-perfect deadline meet rates, and reducing the network bandwidth cost by half compared to the second best competing method (lc). Looking at the trend of changes, we can see similarities to Fig. 3. As conjunctions share less common tests with each other, the number of different tests involved increases. We therefore see from Fig. 9b a general trend of rising deadline meet rates for all approaches, similar to what we have observed in Fig. 3b. However, rather than decreasing retrieval cost ratios as seen from Fig. 3a, we observe either roughly unchanged cost ratios from all baseline approaches, or an ever so slightly increase in cost ratios from our algorithm, as shown in Fig. 9a. From the route-finding perspective, as candidate routes become more different from each other, it becomes less likely for the bad condition of a single road segment to rule out multiple routes. As all baseline methods disregard the logic relations within a request, the benefit of having more tests roughly cancels out the damage of not being able to resolve multiple conjunctions with a single source. For our method, which does exploit the logic relations, the damage is likely slightly overshadowing the benefit, and hence the increasing retrieval cost ratios.

B. An Application Scenario

Now we adopt a concrete post-disaster application scenario and evaluate our algorithm's performance as compared to baseline methods. In particular, we assume the Urbana-Champaign, Illinois area has just been hit by an earthquake. The first responder team has made an initial damage report and deployed an emergency sensor network for further damage assessments and information passings. Various recovery teams have just arrived nearby and would like to go to different parts of the region to carry out their different recovery work (e.g., restore power-grid, search and rescue from collapsed building sites, etc.). We consider the following particular situations that three teams are facing:

1) The Department of History building suffered from severe damage due to its lack of maintenance, causing quite a few serious injuries. A medical team needs to rush send the injured personnels to McKinley Health Center.
2) The communication team has just arrived from the University Airport. They are currently at the southwest corner of the area, and would like to get to the Department of Computer Science Siebel Center to set up an emergency communication center.
3) The evacuation team has learnt that the University High School has not been fully evacuated. They plan to move the remaining kids to the South Quad open area.

One big problem is that some roads within the region are damaged, preventing recovery teams' vehicles to pass, thus they need to find out fast what paths to take to get to their respective destinations—if there is no viable paths for any particular team, they will have to call in nearby military helicopters for assistance. They can make guesses regarding the damage of roads using the reports generated by the first responder team, but to actually determine if a road is in a reasonable condition for vehicles to pass, a picture needs to be acquired from the camera sensor deployed along the road on the emergency network. The network is of low bandwidth and is shared by various teams and agencies, thus, it is desirable that the recovery teams use as little resource as possible when figuring out their paths.

Using Google map navigation service, each of the three teams has determined their candidate routes according to their respective source-destination pairs, as illustrated in Fig. 10. In particular, below are the exact routes.[1] For the medical team, from the Department of History to McKinley Health Center:

- **6th**→**Penn**→**Lincoln**, or
- *Chalmers*→*5th*→Green→**Lincoln**, or
- **6th**→**Penn**→*4th*→Florida→**Lincoln**,

For the communication team, from Kirby to Siebel Center:

- *Florida*→**Lincoln**→*Springfield*→***Goodwin***, or
- Neil→*Springfield*→***Goodwin***, or
- Neil→*University*→***Goodwin***,

For the evacuation team, from High School to South Quad:

- *Springfield*→**Lincoln**→*Nevada*→***Goodwin***→*Gregory*, or
- *Springfield*→***Goodwin***→*Gregory*.

Every route consists of several road segments, and each road is covered by a camera sensor connected to the emergency network. A request for each team is formed, where different conjunctions represent the different candidate routes, and each test within a single conjunction corresponds to a single road along the path being in OK condition for vehicles to pass. Due to the urgent nature of the post-disaster situation, we set the request deadline level to 1. The emergency network's topology is generated randomly, from which each camera sensor's individual retrieval latency is estimated and fed to the data retrieval engine. All retrievals are then handled by a network simulator running using the topology. Of all the baseline approaches, lc (lowest cost source first) achieves the best performance, so we compare our algorithm just to the lc approach. For the actual road conditions and exact camera

[1] Different font-faces indicate: non-italic—road is good; *italic—road is down*; **bold—retrieved by** $greedy$; underlined—retrieved by lc.

data retrieval details, please refer to Footnote 1. The retrieval cost ratio and deadline meet rate results are shown in Tab. II. The information of which approaches choose which roads' conditions to retrieve data for is also illustrated in Fig. 11. As seen, though no deadline misses are observed from either strategy, our data retrieval algorithm is able to finish all route-finding tasks using less than half of the number of road-side cameras that the competing lc approach uses, consuming only half the retrieval bandwidth cost.

	$greedy$	lc
Number of Road Conditions Retrieved	4	9
Retrieval Cost Ratio	28.95%	56.58%
Deadline Meet Rate	100%	100%

TABLE II: Multiple-Route-Finding Application Result

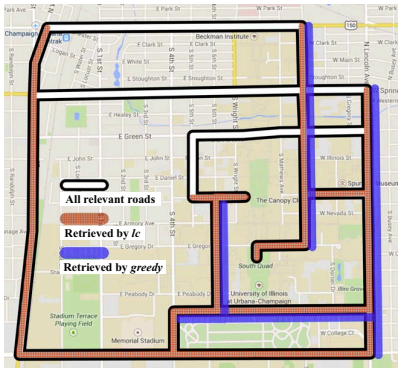


Fig. 11: Multiple-Route-Finding: All candidate routes of all teams are highlighted, different road segments whose conditions are retrieved by either approach ($greedy$ and lc) are marked accordingly.

VI. Conclusions

In this paper we develop data retrieval algorithms for crowd-sensing applications under resource constraints. Our algorithms reduce the underlying network bandwidth consumption by exploiting logical dependencies among data items, while offering a comparable level of service to application requests. The algorithms consider multiple concurrent queries and accommodate retrieval latency constraints. Simulation results show that our algorithms outperform several baselines by significant margins, while maintaining the level of service perceived by applications in the presence of resource constraints.

Acknowledgment

Research reported in this paper was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement W911NF-09-2-0053, DTRA grant HDTRA1-10-1-0120, and NSF grants NSF CNS 13-29886, NSF CNS 13-45266, and NSF CNS 13-20209. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell, "Bikenet: A mobile sensing system for cyclist experience mapping," *TOSN*, vol. 6, no. 1, p. 6, 2009.

[2] 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," in *SenSys*, 2006.

[3] V. Coric and M. Gruteser, "Crowdsensing maps of on-street parking spaces," in *IEEE DCOSS*, 2013.

[4] C. Zhang, J. Luo, and J. Wu, "A dual-sensor enabled indoor localization system with crowdsensing spot survey," in *IEEE DCOSS*, 2014.

[5] S. Hu, L. Su, H. Liu, H. Wang, and T. F. Abdelzaher, "Smartroad: a crowd-sourced traffic regulator detection and identification system," in *IPSN*, 2013.

[6] S. Wang, D. Wang, L. Su, L. Kaplan, and T. F. Abdelzaher, "Towards cyber-physical systems in social spaces: The data reliability challenge," in *RTSS*. IEEE, 2014, pp. 74–85.

[7] D. Wang, L. Kaplan, H. Le, and T. Abdelzaher, "On truth discovery in social sensing: A maximum likelihood estimation approach," in *IPSN*, 2012.

[8] S. Wang, L. Su, S. Li, S. Hu, T. Amin, H. Wang, S. Yao, L. Kaplan, and T. Abdelzaher, "Scalable social sensing of interdependent phenomena," in *IPSN*, 2015.

[9] H. Liu, J. Li, Z. Xie, S. Lin, K. Whitehouse, J. A. Stankovic, and D. Siu, "Automatic and robust breadcrumb system deployment for indoor firefighter applications," in *MobiSys*, 2010.

[10] A. Purohit, Z. Sun, F. Mokaya, and P. Zhang, "Sensorfly: Controlled-mobile sensing platform for indoor emergency response applications," in *IPSN*, 2011.

[11] M. Y. S. Uddin, H. Wang, F. Saremi, G.-J. Qi, T. Abdelzaher, and T. Huang, "Photonet: a similarity-aware picture delivery service for situation awareness," in *RTSS*, 2011.

[12] Y.-A. L. Borgne, S. Santini, and G. Bontempi, "Adaptive model selection for time series prediction in wireless sensor networks," *Signal Processing*, vol. 87, no. 12, pp. 3010 – 3020, 2007.

[13] C. C. Aggarwal, A. Bar-Noy, and S. Shamoun, "On sensor selection in linked information networks," in *DCOSS*, 2011.

[14] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan, "Mediascope: Selective on-demand media retrieval from mobile devices," in *IPSN*, 2013.

[15] S. Gu, C. Pan, H. Liu, S. Li, S. Hu, L. Su, S. Wang, D. Wang, T. Amin, R. Govindan *et al.*, "Data extrapolation in social sensing for disaster response," in *DCOSS*, 2014.

[16] S. Wang, S. Hu, S. Li, H. Liu, M. Y. S. Uddin, and T. Abdelzaher, "Minerva: Information-centric programming for social sensing," in *ICCCN*, 2013.

[17] S. Wang, T. Abdelzaher, S. Gajendran, A. Herga, S. Kulkarni, S. Li, H. Liu, C. Suresh, A. Sreenath, H. Wang *et al.*, "The information funnel: Exploiting named data for information-maximizing data collection," in *DCOSS*, 2014.

[18] L. Su, J. Gao, Y. Yang, T. F. Abdelzaher, B. Ding, and J. Han, "Hierarchical aggregate classification with limited supervision for data reduction in wireless sensor networks," in *SenSys*, 2011.

[19] L. Su, S. Hu, S. Li, F. Liang, J. Gao, T. F. Abdelzaher, and J. Han, "Quality of information based data selection and transmission in wireless sensor networks." in *RTSS*, 2012, pp. 327–338.

[20] L. Su, Q. Li, S. Hu, S. Wang, J. Gao, H. Liu, T. F. Abdelzaher, J. Han, X. Liu, Y. Gao *et al.*, "Generalized decision aggregation in distributed sensing systems," in *RTSS*, 2014.

[21] M. Gregorczyk, T. Pazurkiewicz, and K. Iwanicki, "On decentralized in-network aggregation in real-world scenarios with crowd mobility," in *DCOSS*, 2014.

[22] C. Tham and T. Luo, "Quality of contributed service and market equilibrium for participatory sensing," in *DCOSS*, 2013.

[23] R. Greiner, R. Hayward, M. Jankowska, and M. Molloy, "Finding optimal satisficing strategies for and-or trees," *Artificial Intelligence*, vol. 170, no. 1, pp. 19–58, 2006.

[24] M. G. Luby, M. Mitzenmacher, and M. A. Shokrollahi, "Analysis of random processes via and-or tree evaluation," in *SODA*, 1998.

[25] H. Casanova, L. Lim, Y. Robert, F. Vivien, and D. Zaidouni, "Cost-optimal execution of boolean query trees with shared streams," in *IPDPS*, 2014.

[26] Y. Jiang, H. Qiu, M. McCartney, W. G. J. Halfond, F. Bai, D. Grimm, and R. Govindan, "Carlog: A platform for flexible and efficient automotive sensing," in *SenSys*, 2014.

[27] S. Nath, "Ace: Exploiting correlation for energy-efficient and continuous context sensing," in *MobiSys*, 2012.