Simulating Realistic User Mobility for Mobile Crowdsensing Using TACSim: A Performance Study

Simulating realistic user mobility for mobile crowdsensing using TACSim: a performance study

Matteo Rontini
Dept. of Computer Science and Engineering
University of Bologna
Bologna, Italy
matteo.rontini@studio.unibo.it

Christine Bassem
Dept. of Computer Science
Wellesley College
Wellesley, MA, USA
cbassem@wellesley.edu

Federico Montori
Dept. of Computer Science and Engineering
University of Bologna
Bologna, Italy
federico.montori2@unibo.it

***Abstract*—Mobile Crowdsensing (MCS) has recently taken up an important role in sensor data collection paradigms because of its reduced costs and flexibility. It allows crowdsourcers to recruit a number of mobile users to execute sensing tasks in an area without deploying physical sensors. However, MCS algorithms and policies are very different depending on the application and testing them in the real world is impractical, due to the difficulties in recruiting large crowds of volunteers. For this purpose, the research is mostly oriented to simulations, however, to date, there is no simulation platform that focuses enough on different aspects of MCS, often disregarding some in favor of others. In this paper we focus on TACSim, an extensible simulator and we propose an additional mobility module that fills the gap of accurate road network representation. Participants navigate a real road network offering an improved realism and yielding more accurate results. We also propose a caching system that helps in reducing the processing time of simulations and demonstrate its effectiveness through extensive benchmarks.**

***Index Terms*—Mobile Crowdsensing, simulation, smart city.**

I. Introduction

Mobile Crowdsensing (MCS) is a widespread data collection paradigm where a number of individual, called participants, are committed to collect sensor data over an area of interest using their mobile devices – most of the times their smartphone – about a phenomenon of common interest [1]. The data collection process is often referred to as *campaign*, often issued by a *crowdsourcer*, an entity which stakes a reward for participants on top of their performance. Each participant is then said to be performing a *task*. Tasks may be only one-off, such as taking a single picture of a subject, or continuous, such as detecting the noise pollution over time in different zones of the city. Regardless, MCS is nowadays a very heterogeneous paradigm, where campaigns may differ in their rewarding mechanism, their task assignment policy, their recruitment strategy and their network architecture [2].

One dominant aspect in MCS is the difficulty of testing new paradigms, being them data collection strategies or actual applications, reason being the need for a consistent user base willing to test a new proposal off-the-shelf. An actual recruitment of volunteers or paid users is a long and cumbersome process, which would dilate the time needed for setting up a complete scenario. In addition, there is no unified end-user application, therefore, scientists also need to develop a dedicate mobile application for the specific scenario, which may be different from time to time under different aspects – an example is push-based scenarios versus pull-based ones [3]. For this reason, MCS strategies are often validated through simulations, which must replicate the realism of a crowd traveling urban areas and performing MCS tasks under different conditions.

The current literature proposes a number of MCS simulators, however it is difficult to find an all-encompassing one, as most of them concentrate on a few aspects in detail, disregarding or simplifying the rest of them. For example, CrowdSenSim [4] in its original version was mostly devoted to analyzing the performance of resource utilization in opportunistic MCS, however it did not resolve events in their chronological orders, disregarding scenarios where event happening at one point in time could affect subsequent ones.

In this paper we take into account TACSim [5], a recent MCS simulator which we believe represents a good basis for future research because of its extensibility. TACSim features a powerful implementation of single participants, which can be extended by developers, and it is mostly oriented to one-off geolocated tasks, with different algorithms used for direct recruiting and rewarding. One thing missing from TACSim however, is the realistic representation of the urban scenario, as participants move among different tasks using their Euclidean distance. While actual navigation distance is an aspect that can be easily abstracted, we argue that many post-simulation analyses would benefit from embedding a realistic representation of a road network in the simulation software. For this reason, we propose an extension of TACSim, which we release open-source, that focuses on reproducing realistic road networks by leveraging the OpenStreetMap APIs. The road network is then translated into a graph which is the used for navigating the participants from task to task. Because navigating a graph is far more computationally expensive than using the euclidean distance, we also introduce two caching systems, namely the Accurate Cache and the Fast Cache, which consistently reduce the processing time for simulations. Results show that the use of these caching systems consistently improves the performances and that the topology of a road network may affect some aspects of the simulations beyond the mere scenario size, justifying the need for a realistic representation.

The paper is organized as follows: Section II presents related

works on the topic, Section III presents the legacy features of TACSim, while Section IV illustrates our additional module and how it blends into the original architecture. Section V presents our caching system, Section VI describes our experimental setup and Section VII its results. finally, Section VIII closes the paper.

II. Related Works

The most effective task allocation mechanism evaluations in mobile crowdsensing are performed via dedicated and goal-oriented simulators, such as those built for TaskMe [6] and PSAllocator [7], among many others. These simulators are effective for their single purposes, but cannot be easily re-used for other evaluation purposes.

On the other hand, existing general-purpose crowdsensing simulators have limitations when it comes to evaluating task allocation mechanisms. In [8], the open-source network simulator ns-3 is leveraged to build a simulator for crowdsensing purposes, but the complexity of ns itself as a general network simulator is limiting for the purposes of crowdsensing and the simulation of crowd behavior. In CrowdSenSim [4] and later CrowdSenSim 2.0 [9], a stateful and scalable simulation model was adopted to allow for the implementation of algorithms, in which chronological order of events matter. Although efficient, it lacks some core characteristics for effective evaluation of realistic crowdsensing scenarios, as it only focuses on opportunistic crowdsensing models with homogeneous participants roaming around a mobility field.

Finally, in TACSim [5], an agent-based simulation framework is designed to evaluate task allocation mechanisms specifically in participatory crowdsensing models. The design of TACSim allows for developer extensions to accommodate various task allocations, which we leverage in this work to incorporate diverse crowd behaviors in general crowdsensing platforms.

III. Architecture of TACSim

TACSim [5] is a crowdsensing simulation framework[1], that is dedicated to the evaluation of task allocation and route planning mechanisms within crowdsensing platforms. It is designed to provide an emulation of realistic sensing scenarios; running on urban road networks, and with heterogeneous tasks and rational participants with varying utilities.

In its current version, TACSim offers multiple features, some of which are,

1) Generating the road network from existing traces
TACSim accommodates importing data trace files, either for sensing tasks or participant trajectories. After a file of data traces is loaded, TACSim generates the corresponding road network graph, with the vertices corresponding to the points of interest in the loaded traces and their pair-wise distances calculated using the cosine-haversine formula. This importing option allows users to quickly set up a simulation scenario using pre-existing data traces, without having to set up the corresponding road network.

[1]Source code can be found at `https://github.com/cbassem-collab/TACSim`

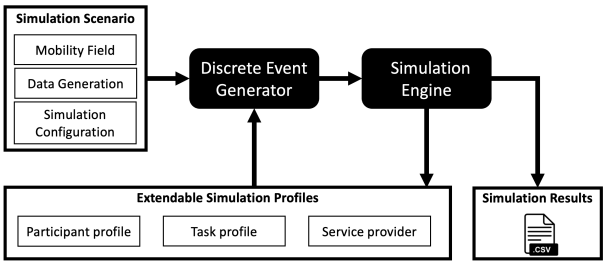


Fig. 1. The overall architecture of TACSim, with its backbone modules that represent the core of the simulator [5].

2) Realistic Crowd Behavior
TACSim models a diverse set of crowd participant behaviors within the simulation, while also providing the tools for developers to extend the simulator and add new participant profiles. Currently, the set of properties defined with the creation of a participant profile are their initial location, their time of availability within the simulation, their quality of sensing, and a model of their preferences [10]. These properties are then fed into the agent-based simulation engine, to generate mobility and task decisions within the simulation. The simulator has the functionality to model participant mobility in various forms based on the simulation needs. For instance, a participant's full trajectory can be pre-defined before the simulation starts based on pre-loaded trajectory traces, or they can be decided in real-time within the simulation based on the task allocation / routing algorithms implemented.

3) Heterogeneous Participants and Tasks
Leveraging the polymorphic characteristics of Java, unlimited number of participant utilities and task types can be defined within the TACSim framework. This feature also allows for the generation of heterogeneous participant profiles, in which task allocation, task completion and quality, and mobility choices can vary from one generated participant profile to the other, and heterogeneous task types, in which the complexity and reward of each generated task can vary. Thus, better emulating realistic crowdsensing scenarios.

IV. Dynamic Real-World Simulation

With respect to Section III, we observe that, despite the extensibility of the simulated entities (i.e., participants, tasks, and crowdsourcers), the representation of the mobility in the scenario remains quite simplified. In fact, in TACSim, participants travel the area by moving directly from one task to another. The simulator uses the cosine-haversine distance as a metric to estimate how long a participant will travel to get to the next task, however, this is hardly applicable to real urban environments. In real cities, the direct distance between two points is the shortest distance that a participant can travel, but in reality people would need to walk along roads and stick to the actual topology of the road network, which unavoidably stretches the path to be taken. Removing the details of the

road network, in some cases, can invalidate the outcome of simulations by giving substantially deviated results [11].

For these reasons, in this paper, we redesigned the mobility module of TACSim to accommodate the simulation of real cities. For this, we make use of the Overpass API[2], extensively used in literature to get detailed data about small spatial regions [12]. At startup, the system downloads an XML road network of a city through Overpass by asking the city name to the user. The road network is then converted into a graph with nodes and edges respectively listed in two separate files. In our case, nodes are crosses and edges are road segments, weighted accordingly to their length. Nodes are labeled with their GPS coordinates; this ensure uniqueness as two nodes cannot be perfectly overlapping. For simplicity, upon event generation, we deploy the number of tasks requested by the user over nodes. If a task must be located on a road segment not in correspondence of a cross, then a new node must be generated, splitting then the edge into two.

Participants are assigned to tasks in the same way as they were in legacy TACSim, however, the distance that they must travel between tasks must be calculated by navigating the graph instead of their cosine-haversine distance[3]. Our main contribution lies in a new module for TACSim: the **Graph Navigation** module, where developers can load their road graph data and implement their own graph navigation algorithm, as long as it takes in input a source node and a destination node. This interface fits most shortest path finding algorithms, such as Dijkstra's or A*, however, it gives the liberty to developers to provide their alternatives, as not in all scenarios participants wish to take the spatially optimal solution for their individual paths. This flexibility allows participants to favor specific metrics to choose where to go through. For instance, participants may wish to go through parks, or to avoid congested roads, if the originally loaded graph edge data captured information on traffic congestion. Pedestrian and cyclists in particular are mostly sensitive to other criteria that detach from the utter time convenience (e.g., cyclists may choose to only travel through road segments that have good floor quality to avoid bumps [13]).

V. Graph Navigation and Caching

In our implementation we exemplified, and repetitively tested, the Dijkstra's shortest path algorithm within the graph navigation module. The rationale behind this choice is that we wanted to provide a baseline by proposing the most famous and optimal known navigation algorithm, knowing that it has a considerably long running time when executed repeatedly within the same simulation. Dynamic programming algorithms with the same complexity as Dijkstra's, despite running in polynomial time, may display an unacceptably long running time on sufficiently large graphs if executed repeatedly for every single participant. Whatever the algorithm, this problem may hinder the usability of TACSim.

[2]`https://overpass-api.de/`

[3]Participant speed is considered uniform in this version of the work, but can be configured in future implementations.

Because we expect the algorithm to be executed many times within a single simulation, we designed two simple caching procedures that should alleviate the computation burden and avoid to recalculate the length of already known paths. The caching solutions are based on the assumption that, eventually, participants will need to take paths that other participants have taken in the past, because tasks are far less than the graph nodes, and graph navigation most of the times takes place from one task to another. The effectiveness of the caching system depends on many factors, one of which being the road topology: in many cities there are often key points through which pedestrians pass, making them hubs from which the calculation of paths would have likely taken place in the past.

Our caching system relies on a lookup table, which is filled every time the navigation algorithm ends by finding a path between a source and a destination. The path is inserted in the lookup table with the source-destination couple as a key. On top of that, in this section, we propose two different caching strategies: the *Fast Cache* and the *Accurate Cache*.

a) Fast Cache: Let us assume that a participant must travel from a source node S to a destination node D according to the elected graph navigation algorithm. At each step, as the algorithm lands on an intermediate node S', it verifies if the couple $\langle S', D\rangle$ is a valid key in the lookup table, that is, if the intermediate node is marked as a source of an already known path that leads to the same destination than that of the participant. In such a case, the algorithm stops and joins the path taken so far from S to S' with the known path from S' to D. Even though the path obtained this way leads from S to D, it is not guaranteed to be the one that best complies to the metrics of the graph navigation algorithm. In other words, if we consider the example of Dijkstra as the graph navigation algorithm, the use of the Fast Cache does not ensure that the path that we obtain from S to D is the shortest. To exemplify this concept in simple words, we may think of a triangle with vertices A, B, and C, such that $\overline{AB} < \overline{AC}$. If we assume that the path between two vertices is the straight segment connecting them, and we set BC as known, then a graph navigation that aims to find the shortest path between A and C will get to B first, setting the concatenation of AB and BC as the output. However, we know that AC is the shortest path in this case.

b) Accurate Cache: This cache works under the same premises as the previous one, with a source node S and a destination node D. When the algorithm encounters an intermediate node S', such that $\langle S', D\rangle$ is a valid key in the lookup table, it does not stop, differently from the previous case. Instead, it considers the path from S' to D to be known and does not visit further such branch, keeping in memory the metric of such a path. Then, it continues by visiting all other branches, stopping when it reaches D or the metric matches the one of the path obtained by joining the subpaths S to S' and S' to D. If we consider the example of Dijkstra as the graph navigation algorithm, the length of the path from S to D that passes through S' is known upon reaching S', therefore, the algorithm stops when all other reached paths

are above such a length (or D is reached first). The Accurate Cache is guaranteed to yield the same result as the plain graph navigation algorithm, although being considerably slower than the Fast Cache.

The usage of these caches provides a good trade-off between performance and optimality in graph navigation, and we deem both of them to be useful for our system. The reason behind this is that, generally, participants to MCS scenarios are pedestrians. These types of users may not always use a technological support to travel to one place to another, especially if they are within their own city. In such case they may as well decide to rely on their common sense and their habits, by referring to places they already know well, not always resulting in the optimal choice. Obviously, this depends on the use case and its criticality, thus we decided to include the implementation of both caches in TACSim, so that simulations can be launched favoring either time performance or precise calculation of paths.

VI. Experimental Setup

In order to validate the newly introduced graph-based distance calculator as well as the two cache systems, we implemented both components into TACSim, producing a new version. The new version of TACSim is, at the time of writing, available open-source as a fork of the original TACSim project[4].

We validated the new version of TACSim by running simulations over five differently sized Italian cities:

- Imola (nodes: 35,810 - edges: 39,252)
- Prato (nodes: 149,150 - edges: 157,059)
- Brindisi (nodes: 264,354 - edges: 283,314)
- Pisa (nodes: 511,841 - edges: 533,495)
- Bologna (nodes: 734,889 - edges: 773,360)

We varied the number of tasks and participants spawned in the cities: number of tasks varies among $\{20, 40, 60, 80, 100\}$ and number of participants varies among $\{40, 80, 120, 160, 200\}$. For each configuration, we ran simulations using no cache, the Accurate Cache and the Fast Cache by repeating each experiment 10 times. Each repetition uses the same seed across all configurations to ensure scientific rigor.

In a first batch of runs we configured the simulations so that we empty the cache for every run, thus considering each of them as standalone experiments. Additionally, we performed a second batch of runs where, for each configuration, we do not empty the caches across the 10 repetitions. This way, we wanted to test if similar runs repeated with the same inputs over time yield better results in terms of execution time. This corresponds to simulating similar scenarios occurring multiple times over workdays, which is closely related to simulate peoples' habits. We expect that, if we do not empty the cache across different runs, the processing time of a single simulation progressively decreases over time.

[4]https://github.com/rontinim/TACSim

According to the numbers above, we ran a total of 2,500 simulations, using a PC with an AMD Ryzen 5 PRO 3500U w/ Radeon Vega Mobile Gfx, 2100Mhz, 4 Core, 8 logical units, 8GB of RAM, hosing an AMD Radeon(Tm) Vega 8 Graphics and Microsoft Windows 10 PRO. Because the full graph needs to be kept in memory, the city of Bologna represents more or less the upper limit in terms of RAM size for the machine we used.

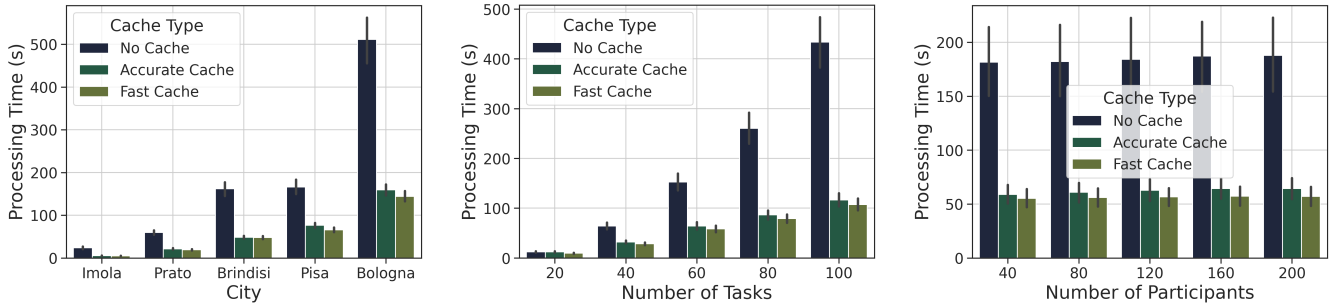
VII. Results

The results for each of the experiments are shown in Figures 2, 3, 4, and 5.

Figure 2 shows the results of the tests in terms of processing time when the cache is emptied at every repetition (first batch). Because the tests are repeated 10 times, each of the bar charts display a 95% confidence interval. The results displayed are in line to what we expected. As the size of the city grows, the processing time equally grows, with the Fast Cache yielding slightly better performance that the Accurate Cache. At a first look, the better performance of the Fast Cache does not seem to be enough to justify a worse precision in the identification of the shortest path. (we will quantify such difference later). Interestingly, we can see that Pisa does not impact the processing time much compared to Brindisi, even though the number of nodes and edges is almost the double. We argue that this depends mostly on the city topology: Brindisi is the only coastal city among the ones taken into account, moreover, it is built around a lagoon. This unavoidably forces participants to take long detours to reach their destination, affecting the duration of the shortest path search. Results also show that the number of tasks has a directly proportional impact on the time taken to run the simulation, which is expected, as this affects directly the number of participants to recruit in order to get them all completed. At the same time, we can also appreciate how the number of participants in the network does not affect the performance at all. This happens because, if the number of tasks stays the same, each participant must do less work in order to get all the tasks completed. Recall that a task is considered completed once one participant has completed it, with no need for the others to do the same.

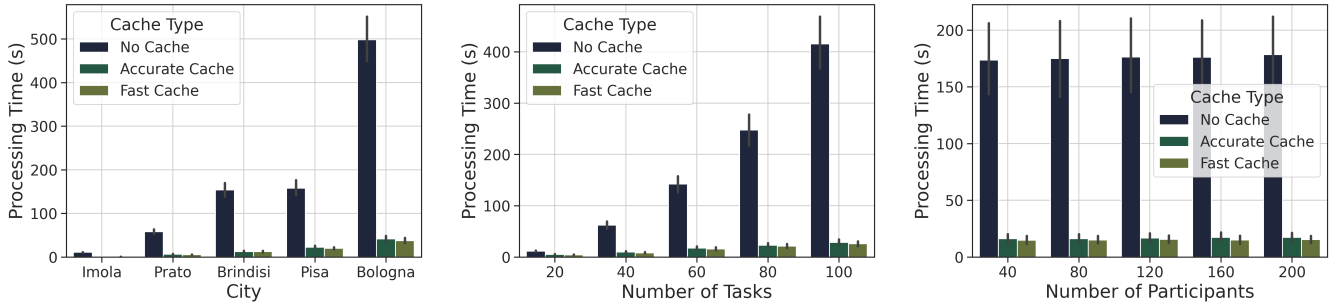
Figure 3 shows the same tests, executed when the cache is not emptied across repetitions within the same configuration. We can appreciate how, by keeping the cache, the processing time greatly reduces. It is worth recalling that different repetitions also have different seeds, which implies that the actual paths traveled by the participants are not always the same, still, the presence of a cache brings the processing time to less than an half. In this case, it is even more evident how the Fast Cache, in absolute, does not provide a tangible advantage.

A better understanding of how keeping the cache affects positively the processing time is given in Figure 4, where the processing time is shown on top of subsequent repetitions when the cache is kept. It is immediately visible how the processing time is reduced already from the second repetition, supporting the claim that, after running the simulation a couple



(a) Comparison over cities (b) Comparison over Tasks (c) Comparison over Participants

Fig. 2. Comparison of the simulation running time by using the three different caches by emptying the cache at every repetition.



(a) Comparison over cities (b) Comparison over Tasks (c) Comparison over Participants

Fig. 3. Comparison of the simulation running time by using the three different caches by keeping the cache at every repetition.

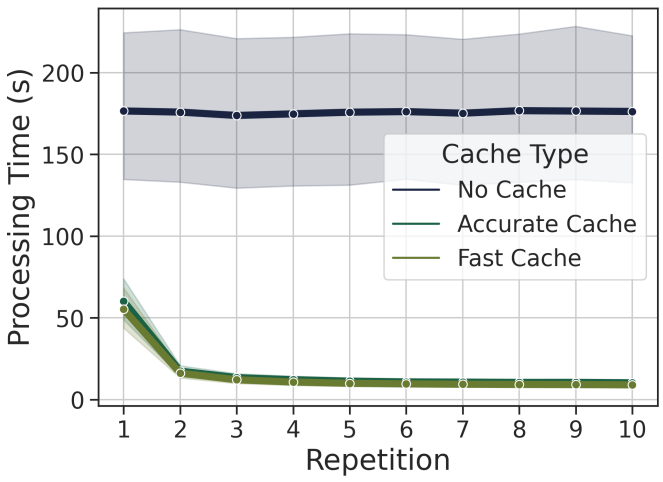


Fig. 4. Execution time of each iteration when keeping the cache

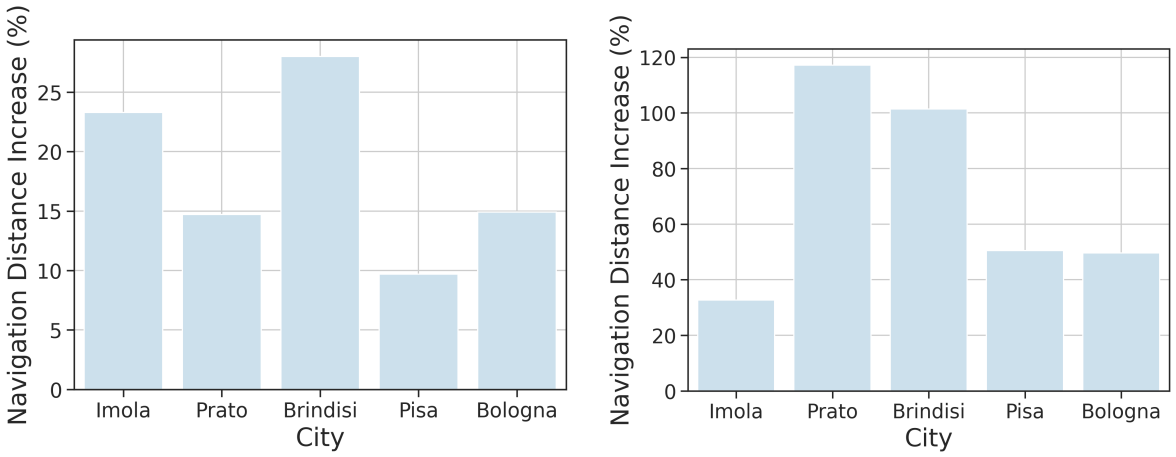
of times on a reasonably sized city, we already end up with a cache that would optimize subsequent runs.

Having proved that the Accurate Cache does not suffer much against the Fast Cache in terms of processing time, we also aim to show how much the results of the Fast Cache detach from the optimal. In other words, if we consider our routing algorithm as the Dijkstra's shortest path algorithm, will the algorithm using the Fast Cache output significantly longer paths? We analyzed this aspect in Figure 5, where we show, for each of the analyzed cities, how much, in percentage, the average path length increases when using the Fast Cache in comparison with the Accurate Cache. We evaluated this both in case the cache is emptied and when it is kept.

Results show that, for standalone simulations, the increase in the distance is between the 10% (for Pisa) and the 27% (for Brindisi). Here it is even more evident that these results highly depend on the city topology, i.e., a less centralized and connected topology would be less suitable for a Fast Cache, simply because basing part of a route on top of past routes is typically not the best choice. This becomes even more evident for the batch of simulations where the cache is not emptied: for two cities out of five, the Fast Cache yields a average path length that is more than twice the shortest. This basically suggests that the Fast Cache worsens its performance as the cache grows larger, therefore, it might be convenient to use only during cold starts, as it would diverge in the long run.

VIII. Conclusions

In this paper we developed a new mobility module for TACSim, efficiently simulating participants moving in urban scenarios according to a definite road network. Participants navigate such a network by actually navigating a graph, using the algorithm that promotes their condition of interest. In our example, we implemented the Dijkstra's shortest path algorithm and designed two types of cache, the Accurate Cache and the Fast Cache, that significantly reduce the processing time. Our experiments showed that both caches are both



(a) Emptying the cache every repetition

(b) Keeping the cache every repetition

Fig. 5. Increase in percentage of the average path length when using the Fast Cache in comparison to the Accurate Cache.

almost equally effective, demonstrating that the Fast Cache is typically less convenient as the time advantage it bring in is not worth the inaccuracy of the path, unless used in the cold start phase.

Future works concerning TACSim are directed towards a paradigm that can equally simulate push and pull-based scenarios, where users can directly recruited by the crowdsourcer for specific tasks, as well as opportunistically landing in the area of interest.

ACKNOWLEDGMENTS

This work has been funded by CN-MOST, Spoke 7.

REFERENCES

[1] A. Capponi, C. Fiandrino, B. Kantarci, L. Foschini, D. Kliazovich, and P. Bouvry, "A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities," *IEEE communications surveys & tutorials*, vol. 21, no. 3, pp. 2419–2465, 2019.

[2] F. Montori, P. P. Jayaraman, A. Yavari, A. Hassani, and D. Georgakopoulos, "The curse of sensing: Survey of techniques and challenges to cope with sparse and dense data in mobile crowd sensing for internet of things," *Pervasive and Mobile Computing*, vol. 49, pp. 111–125, 2018.

[3] T.-Y. Yu, X. Zhu, and M. Maheswaran, "Push vs pull participant recruitment system for personalized vehicular crowdsensing," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

[4] C. Fiandrino, A. Capponi, G. Cacciatore, D. Kliazovich, U. Sorger, P. Bouvry, B. Kantarci, F. Granelli, and S. Giordano, "Crowdsensim: a simulation platform for mobile crowdsensing in realistic urban environments," *Ieee access*, vol. 5, pp. 3490–3503, 2017.

[5] C. Bassem, "Tacsim: An extendable simulator for task allocation mechanisms in crowdsensing," in *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2023, pp. 74–81.

[6] Y. Liu, B. Guo, Y. Wang, W. Wu, Z. Yu, and D. Zhang, "Taskme: Multi-task allocation in mobile crowd sensing," in *Proceedings of the 2016 ACM international joint conference on pervasive and ubiquitous computing*, 2016, pp. 403–414.

[7] J. Wang, Y. Wang, D. Zhang, F. Wang, Y. He, and L. Ma, "Psallocator: Multi-task allocation for participatory sensing with sensing capability constraints," in *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 1139–1151.

[8] C. Tanas and J. Herrera-Joancomartí, "Crowdsensing simulation using ns-3," in *International Workshop on Citizen in Sensor Networks*. Springer, 2013, pp. 47–58.

[9] F. Montori, E. Cortesi, L. Bedogni, A. Capponi, C. Fiandrino, and L. Bononi, "Crowdsensim 2.0: A stateful simulation platform for mobile crowdsensing in smart cities," in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 289–296.

[10] C. Bassem, "Challenges of modeling participant behavior in crowdsensing evaluation," in *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 2024, pp. 1–6.

[11] F. Montori, M. Gramaglia, L. Bedogni, M. Fiore, F. Sheikh, L. Bononi, and A. Vesco, "Automotive communications in lte: A simulation-based performance study," in *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*. IEEE, 2017, pp. 1–6.

[12] R. M. Olbricht, "Data retrieval for small spatial regions in openstreetmap," *OpenStreetMap in GIScience: Experiences, Research, and Applications*, pp. 101–122, 2015.

[13] F. Montori, R. Pastore, L. Sciullo, L. Bononi, and L. Bedogni, "An mcs navigation system based on road surface quality for bicycle riders," in *2024 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2024, pp. 125–132.