IMPLEMENTATION AND EVALUATION OF DYNAMIC VEHICLE ROUTING ALGORITHMS

ZHIHAO RUAN [RUANZH@SEAS.UPENN.EDU];
LEJUN JIANG [LEJUNJ@SEAS.UPENN.EDU]

ABSTRACT. We have implemented and experimented with six different policies (the random assignment policy, the first-come-first-serve policy, the m-Stochastic Queue Median policy, the Unbiased Travelling Salesman policy, the m-vehicle Divide and Conquer policy, the No-Communication policy) for the dynamic vehicle routing problem. We compared their performance in terms of System Wait Time and analyzed stability under both light-load and heavy-load settings. Under heavy load and for a long time horizon, the Unbiased Travelling Salesman Problem (UTSP) Policy turns out to be the best choice due to its stability. If the time horizon is short, or the tasks are light-load, the No Communication (NC) Policy would be the most efficient in terms of System Wait Time.

1. INTRODUCTION

The vehicle routing problem (VRP) is a set of important problems in terms of multi-agent planning and control. It focuses on finding the lowest-cost routing policy to arrange a fleet of *vehicles* to finish a set of *tasks* scattering within some convex subspace $\mathcal{Q}$ of $\mathbb{R}^n$. For example, consider the scenario where a fleet of robots deployed in an Amazon warehouse are asked to provide service to a set of outstanding demands emerging at different locations. The optimal planning policy for such scenario would be the solution to the VRP problem, which involves algorithms from combinatorial mathematics.

While there have been significant results in finding the exact and approximate solutions to the normal VRP problems [1, 2, 3], the dynamic vehicle routing problem (DVRP), as a variant of VRP, has gained particular interest over the years. The tasks in a typical DVRP problem keep emerging *during* the mission execution rather than being *static* as in the case of a normal VRP problem. As a consequence, solutions to the DVRP problem aims at stabilize the system rather than reaching the lowest travel costs. [4] has provided an overview to some of the classical solutions to the DVRP problem.

While [4] has provided sufficient theoretical analyses to each DVRP policy, it does not offer any experimental results. Hence, we propose to implement and visualize some of the policies in a simple Python-based simulator[1], hoping that we could realize those theoretical guarantees in practice.

[1]Our code can be accessed at `https://github.com/lb-robotics/python-vehicle-routing`.

2. Related Work

All of our implemented DVRP policies directly come from the paper [4]. However, it is interesting to notice that some of the policies are built on mathematical assumptions that are not easy to achieve in code. The m-SQM policy relies on the result of finding the m-median of a convex set, where there only exist some complex optimization algorithms [5] that fail to generalize or fit well into the DVRP setting; The UTSP policy builds upon the exact solution to the general Traveling Salesman Problem (TSP), whereas in most practical cases only approximate solutions are available [1, 6]; The m-DC policy requires an implementation of a distributed spatial partitioning algorithm [7], which involves the computation of a Laguerre-Voronoi diagram (aka. power diagram) [8, 9]. We proposed strategies around those issues and realized them as discussed in the next section.

3. Approach

Our implementation utilizes the multi-agent simulation framework provided by MEAM 624 homework assignments written in Python. The simulation framework has the following classes:

- A `Node` class representing a single vehicle. Each vehicle is able to `send` (send messages to connected neighbors), `transition` (receive and process messages), `systemdynamics` (update vehicle state and velocities), and `updategoal` (update the goal of the vehicle).
- A `Graph` class consisting all the nodes (vehicles). It is responsible for setting up a Matplotlib animation for simulation as well as plotting all necessary data.
- A `Task` class reponsible for generating tasks according to some pre-defined rate λ_p and distributing them to vehicles according to different policies.

Using this simulation framework, we implemented the naive random assignment policy, the first-come-first-serve (FCFS) policy, the m-SQM policy, the UTSP policy, the m-DC policy, and the No-Communication (NC) policy [4].

3.1. The Random Assignment Policy and the First-Come-First-Serve (FCFS) Policy.

The random assignment policy can be described as follows:

- For each generated task, assign it to a random vehicle. For each vehicle, execute the task in first-come-first-serve order.

The FCFS policy can be described as follows:

- For each generated task, assign it to the closest vehicle. For each vehicle, execute the task in first-come-first-serve order.

Both of the two policies are implemented exactly according to their definitions. They are only used as a test to the simulation framework and it is verified that the number of outstanding tasks accumulate over time without bound.

3.2. The m-Stochastic Queue Median (m-SQM) Policy.

The m-SQM policy can be described as follows:

- Find the m-median of the entire convex space $\mathcal{Q}$ and initialize the positions of the vehicles to be the m-median points. For each generated task, assign it to the nearest m-median point. For each vehicle, execute the task in first-come-first-serve order.

This policy is proven to be stable in light-load scenarios. The overall implementation is straightforward, with the focus being solving the m-median points of an arbitrary convex space $\mathcal{Q}$ under an arbitrary spatial distribution. In order to make the algorithm generalizable, we propose the k-median algorithm, which combines the Monte-Carlo method with an adapted version of k-means:

(1) Sample N points from the distribution within the subspace $\mathcal{Q}$;
(2) Execute the k-median algorithm:
   (a) Initialize m centroids within $\mathcal{Q}$;
   (b) Associate each sampled point with the nearest centroid;
   (c) Update each centroid's position to be the geometric median of all associated points using [10];
   (d) Repeat from (2b) until convergence.

From a Monte-Carlo point of view, the process of finding the geometric median of a cluster minimizes the sum of distances to each point in the cluster, which converges to the definition of m-median when the number of samples is large. In practice, sampling 10,000 points from the distribution has already achieved a descent result of the m-median points.

Figure 1 shows the result of a 10-median of a rectangle under uniform distribution from the k-median algorithm on 10,000 sampled points.

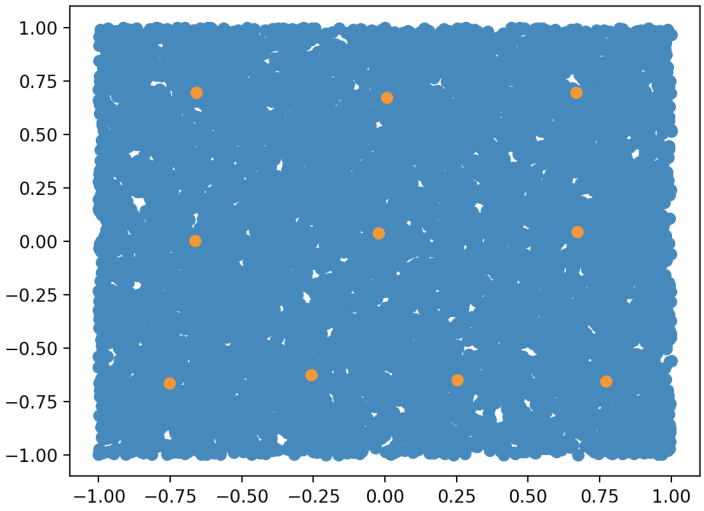

FIGURE 1. The resulting 10-median under uniform distribution from 10,000 sampled points.

3.3. **The Unbiased Travelling Salesman Problem (UTSP) Policy.** The UTSP policy can be described as follows:

- Subdivide $\mathcal{Q}$ into r wedges centering at an arbitrary point $p \in \mathcal{Q}$ such that each wedge share an equal fraction share of the entire distribution. Form sets of tasks in each wedge of size n/r (n is a design parameter). As sets are formed, deposit them in a queue and assign to the first available vehicle in first-come-first-serve order. For each vehicle, form a TSP loop on each set of tasks and execute them in the TSP order.

This policy is proven to be stable under heavy-load scenarios. For actual implementation, we used an open source Python-based TSP solver to compute TSP loops, which essentially applies the greedy algorithm [11]. The sets of tasks are stored in a `tasksetqueue` defined in the `Task` class, and each task set is assigned to a vehicle all at once.

For wedge partitioning of $\mathcal{Q}$ under spatial distribution φ, we design the following pipeline:

(1) Choose a fixed central point $p \in \mathcal{Q}$;
(2) Sample N points from the distribution within $\mathcal{Q}$ and transform them into polar coordinates centering at p;
(3) Starting from $\theta_0 = -\pi$, gradually increase θ until the resulting wedge between angle $[\theta_0, \theta]$ has exactly N/r points. Denote $\theta_1 = \theta$. Add the wedge between angle $[\theta_0, \theta_1]$ into the partition;
(4) Repeat from (3) until the entire $[-\pi, \pi]$ is sweeped.

Figure 2 shows the result of partitioning a rectangle under uniform distribution into 10 wedges centering at $(0.7, 0.7)$ by sampling 10,000 points. With the spatial distribution being uniform, the partition is close to equal in area.

3.4. **The m-Divide and Conquer (m-DC) Policy.** The m-DC policy provides a routing policy without any requirement of central storage or computation. It essentially consists of two steps:

(1) A distributed partitioning algorithm that divides the entire space $\mathcal{Q}$ into m partitions that is simultaneously equitable to φ and $\varphi^{1/2}$;
(2) A routing algorithm that batch-serve the queued tasks using a TSP loop.
(3) For each vehicle, if there is no outstanding tasks in the queue, move to the location that minimizes the sum of distance to all the past serviced tasks.

For the actual implementation of Step (1) (which is essentially a gradient descent algorithm on the weights of a power diagram [7]), we made use of the analytical solution mentioned in the paper:

$$\dot{w}_i(t) = -\frac{\partial H_{\mathcal{V}}}{\partial w_i}, \quad \frac{\partial H_{\mathcal{V}}}{\partial w_i} = \sum_{j \in N_i} \frac{1}{2\gamma_{ij}} \left(\frac{1}{\lambda_{V_j}^2} - \frac{1}{\lambda_{V_i}^2} \right) \int_{\Delta_{ij}} \lambda(x)\, dx \tag{1}$$

where γ_{ij} is the Euclidean distance from the generator of the power cell i to the generator of the power cell j, λ_{V_j} is the integral of the distribution φ over the power cell j, and Δ_{ij} is the line boundary between the power cell i and j. Due to the complexity of implementing a fast line integral algorithm on arbitrary 2-D function for the integral term of Eq. (1), we simplified our code by

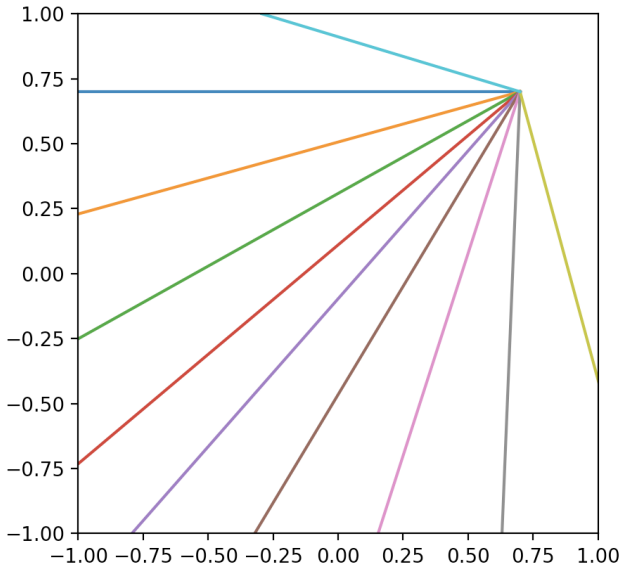

FIGURE 2. Dividing a rectangle under uniform distribution into 10 wedges centering at $(0.7, 0.7)$ by sampling 10,000 points.

limiting to a set of known distribution types (i.e., uniform distribution and normal distribution), and hard-code the computed analytical solution of the line integral according to the provided type of distribution. So far we have tested successfully on the simplified equation of the weight gradient under uniform distribution [7]:

$$\frac{\partial H_{\mathcal{V}}}{\partial w_i} = \frac{|\mathcal{Q}|}{2\lambda_{\mathcal{Q}}} \sum_{j \in N_i} \frac{\delta_{ij}}{\gamma_{ij}} \left(\frac{1}{|V_j|^2} - \frac{1}{|V_i|^2} \right) \tag{2}$$

where δ_{ij} is the length of the boundary between power cell i and power cell j. To generate the corresponding power diagram given a set of weights w_i and generators p_i and calculate its area $|V_i|$ and $|V_j|$, we incorporated an open-source Python-based power diagram generator tool pyVoro [12].

Figure 3 shows the power diagram-based distributed partitioning result of a rectangle under uniform distribution with 5 vehicles.

For the actual implementation of Step (2) and (3), the TSP loop computation is still based on the TSP solver [11]. [10] is re-used to compute the point that minimizes the sum of distances to all the past serviced tasks (aka. the geometric median of the set of all past serviced tasks).

3.5. **The No-Communication (NC) Policy.** The NC policy can be described as follows:

- Let D be the set of outstanding tasks. For each vehicle, if $D = \varnothing$, move to the location minimizing the sum of distances to the past serviced tasks; if $D \neq \varnothing$, move to the nearest task.

The actual implementation of this policy requires the set up of a global queue for all vehicles to access the tasks. Since each task could be taken by any vehicle at any time which could lead to

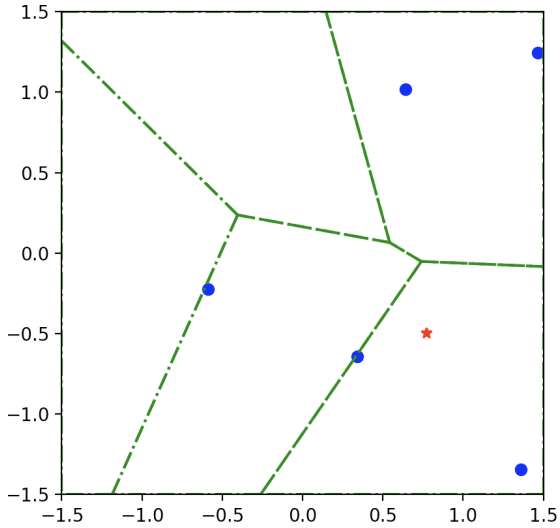

FIGURE 3. Distributed equitable partition of a rectangle under uniform distribution with 5 vehicles. The blue points are the vehicle (generator) locations.

serious race conditions, a lock is critical to the implementation. We designed a global task queue with reader-writer lock based on [13] implemented in the `GlobalTaskQueue` class, which supports atomic operations of adding new tasks and popping finished tasks. The other parts of the NC policy follows the basic FCFS policy implementation. [10] is reused to find the location minimizing the sum of distances to the past serviced tasks.

4. EXPERIMENTAL RESULTS

In this section, we have performed experiments on all six algorithms described above to testcase their performance on light and heavy loads. For light loads, we selected a load factor of 0.1, with $\lambda = 0.5, n = 5$ and service time s distributed normally (mean 1 s and std 0.33 s). For light loads, we selected a load factor of 0.9, with $\lambda = 4.5, n = 5$ and service time s distributed normally (mean 1 s and std 0.33 s). We ran the experiments for 120 s, and in order to test the steady-state behavior (stability) of the algorithms, we ran another experiment for heavy load for 600 s. Shown below are the results of our experiment:

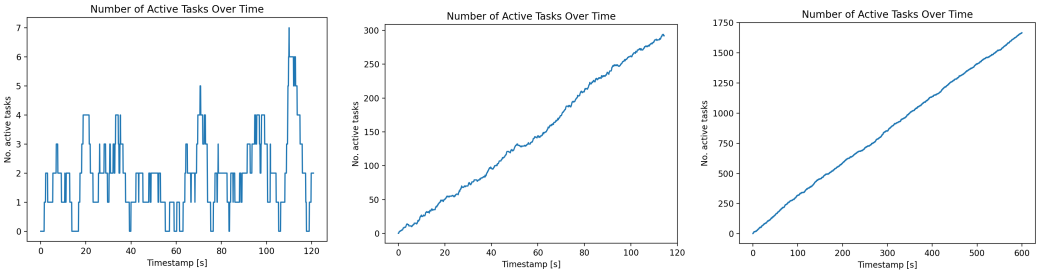

FIGURE 4. Number of active tasks for the Random policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

As expected, the random policy is stable under light load but unstable under heavy load.

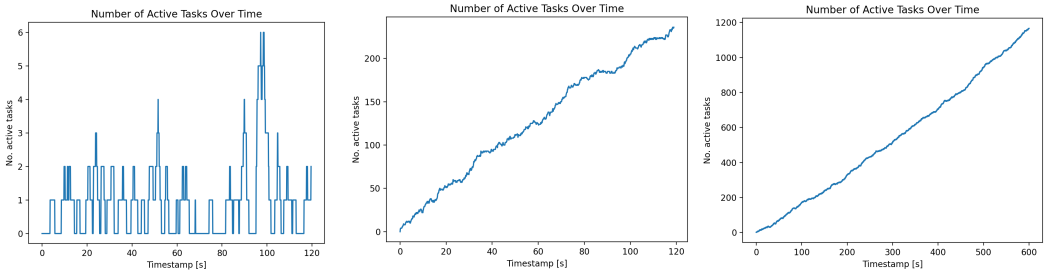

FIGURE 5. Number of active tasks for the First Come Fist Serve (FCFS) policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

Though better than the random policy, the FCFS policy is also stable under light load but unstable under heavy load.

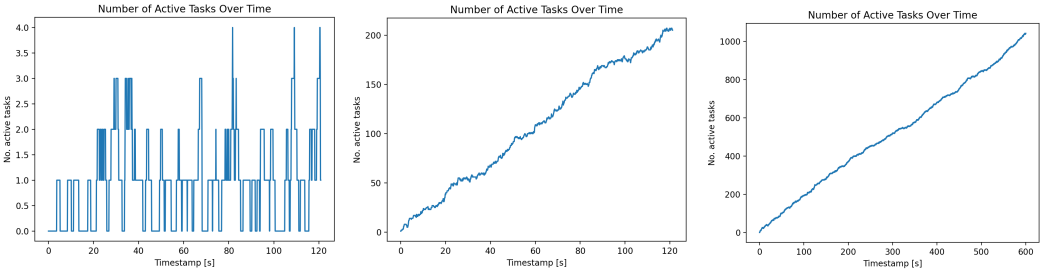

FIGURE 6. Number of active tasks for the m-SQM policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

Similarly, the m-SQM policy is also stable under light load but unstable under heavy load.

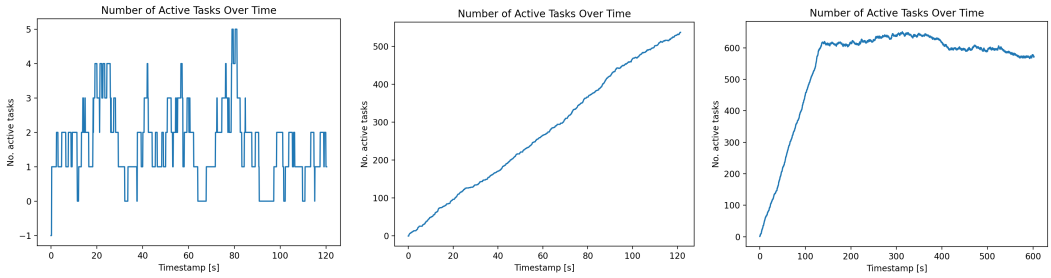

FIGURE 7. Number of active tasks for the UTSP policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

In contrast, although the UTSP policy accumulates a huge amount of active tasks in the beginning, it is stable under both light and heavy loads.

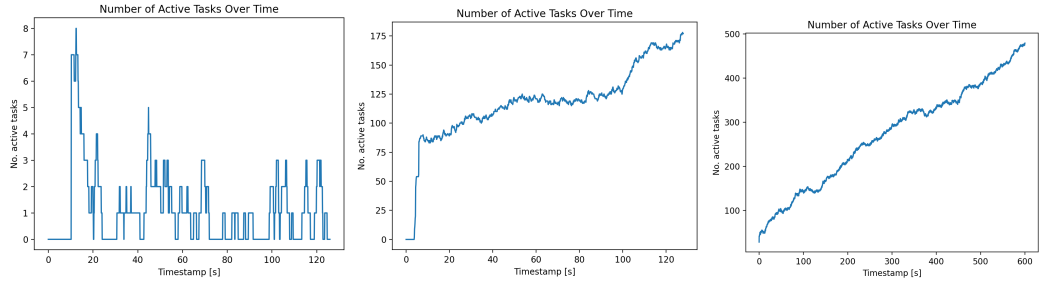

FIGURE 8. Number of active tasks for the Divide and Conquer policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

As expected, the Divide and Conquer policy is also stable under light load but unstable under heavy load.

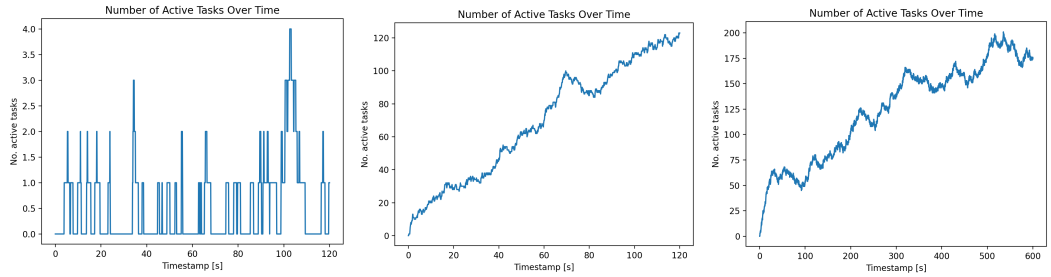

FIGURE 9. Number of active tasks for the No Communication policy at a) low load fatcor ($\rho = 0.1$) over 120 s, b) high load factor ($\rho = 0.9$) over 120 s, c) high load factor ($\rho = 0.9$) over 600 s.

The No Communication policy performs really well under light load. Meanwhile, it also gives good performance under heavy load even though it is not stable.

Below is a table for quantitative comparisons between the algorithms:

	System Wait Time at $\rho = 0.1$ (s)	System Wait Time at $\rho = 0.9$ (s)
Random	3.40	34.38
FCFS	1.83	27.56
m-SQM	1.69	24.76
UTSP	3.05	58.90
Divide and Conquer	2.23	24.40
No Commnication	**1.27**	**14.25**

TABLE 1. Quantitative comparison of System Wait Time between different policies at low Load factor ($\rho = 0.1$) and high Load Factor ($\rho = 0.9$). Each experiment was run for a time horizon of about 120 seconds.

It can be observed that the No Communication policy in this case has the best behavior in System Wait Time in both the light-load and heavy-load settings. It is a surprise since the No Communication policy has its limit of lacking information and is supposed to work worse than the other policies with communications (and the communication was perfect in our experiments). However, it also makes sense in a way that the No Communication policy is very greedy and ensures the system to serve the closest (easiest serving) request as soon as possible. It does not consider the order in which the requests come in and thus could do better on metrics like System Wait Time, as compared to other policies that often serves the requests in a first-come-first-serve manner.

5. CONCLUSION AND FUTURE WORK

In conclusion, we have implemented, experimented and compared six different policies for the dynamic vehicle routing problem. We ran the policies for 120 seconds under both light-load and heavy-load settings, with same task generators and the same environment.

Compared to the naive policies of Random and FCFS, we found the m-SQM policy manage to reduce System Wait Time in both light-load and heavy-load settings. The Divide and Conquer policy behaved not as well under light load since it used TSP as the underlying serving mechanism (batch-serve), but it showed better performance under heavy load. Meanwhile, the UTSP policy took the batch-serve idea further, which resulted in its longest System Wait Time in both load settings. However, according to Fig. 7, it is the only algorithm that is stable under heavy load, and therefore it will be a good choice if we need to carry out heavy load tasks for a long time horizon. If the time horizon is short, or the tasks are light-load, we discovered that the No Communication policy would be the most efficient in terms of System Wait Time.

We have a few ideas for future works:

- Investigate further the underlying factor resulting in the outstanding behavior of the No Communication policy, and combine it with the policies with communication to achieve better performance.
- Move the simulation framework to ROS to allow more realistic experiments, especially in terms of communication.
- Introduce penalty on overdue tasks, this will result in a more realistic metric for measuring different policies' performance.

REFERENCES

[1] L. Perron and V. Furnon, "Or-tools," Google. [Online]. Available: https://developers.google.com/optimization/

[2] A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck, "A generic exact solver for vehicle routing and related problems," *Mathematical Programming*, vol. 183, no. 1, pp. 483–523, 2020. [Online]. Available: https://doi.org/10.1007/s10107-020-01523-z

[3] K. Ledvina, H. Qin, D. Simchi-Levi, and Y. Wei, "A new approach for vehicle routing with stochastic demand: Combining route assignment with process flexibility," *Available at SSRN 3656374*, 2020.

[4] F. Bullo, E. Frazzoli, M. Pavone, K. Savla, and S. L. Smith, "Dynamic vehicle routing for robotic systems," *Proceedings of the IEEE*, vol. 99, no. 9, pp. 1482–1504, 2011.

[5] P. K. Agarwal and M. Sharir, "Efficient algorithms for geometric optimization," *ACM Comput. Surv.*, vol. 30, no. 4, p. 412–458, dec 1998. [Online]. Available: https://doi.org/10.1145/299917.299918

[6] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. Cook, "Concorde." [Online]. Available: https://www.math.uwaterloo.ca/tsp/concorde.html

[7] M. Pavone, A. Arsie, E. Frazzoli, and F. Bullo, "Distributed algorithms for environment partitioning in mobile robotic networks," *IEEE Transactions on Automatic Control*, vol. 56, no. 8, pp. 1834–1848, 2011.

[8] The CGAL Project, *CGAL User and Reference Manual*, 5.4 ed. CGAL Editorial Board, 2022. [Online]. Available: https://doc.cgal.org/5.4/Manual/packages.html

[9] C. H. Rycroft, "Voro++." [Online]. Available: http://math.lbl.gov/voro++/

[10] Y. Vardi and C.-H. Zhang, "The multivariate l1-median and associated data depth," *Proceedings of the National Academy of Sciences*, vol. 97, no. 4, pp. 1423–1426, 2000.

[11] D. Shintyakov, "tsp-solver2." [Online]. Available: https://pypi.org/project/tsp-solver2/

[12] J. Jordan, "pyvoro." [Online]. Available: https://github.com/joe-jordan/pyvoro

[13] Éric Larivière, "readerwriterlock." [Online]. Available: https://pypi.org/project/readerwriterlock/