2021 IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS) | 978-1-6654-4935-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/MASS52906.2021.00061

Efficient Route Selection for Drone-based Delivery Under Time-varying Dynamics

Arindam Khanda*, Federico Corò*, Francesco Betti Sorbelli†, Cristina M. Pinotti†, and Sajal K. Das*
* Department of Computer Science, Missouri University of Science and Technology, USA
† Department of Computer Science and Mathematics, University of Perugia, Italy
Email: *{akkcm, federico.coro, sdas}@mst.edu, †{francesco.bettisorbelli, cristina.pinotti}@unipg.it

***Abstract*—The use of drones can be a valuable solution for the problem of delivering goods for many reasons. In fact, they can be efficiently employed in time-critical situations when there is a traffic jam on the roads, to serve customers in hard-to-reach places, or simply to expand the business. However, due to limited battery capacities and the fact that drones can serve a single customer at a time, a drone-based delivery system (DBDS) aims to minimize the drones' energy usage for completing a route from the depot to the customer and go back to the depot for new deliveries. In general, the shortest delivery route could not be the optimal choice since external factors like the wind (which varies with time) can affect energy consumption. Previous work has mainly considered simplified DBDSs assuming architectures with a single drone and with static costs on paths. Moreover, in these non-centralized architectures, the drones themselves compute the routes on the fly employing their onboard processing resources, making this choice costly. In this paper we develop a centralized system for computing energy-efficient time-varying routes for drones in a multi-depot multi-drone delivery system. Specifically, we propose a novel centralized parallel algorithm called Parallel Shortest Route Update (PSRU) that, over time, updates the drones' delivery routes avoiding the whole recomputation from scratch. A comprehensive evaluation proves that PSRU is up to 4.5x faster than the state-of-the-art algorithms.**

***Index Terms*—Drone, Dynamic graph, Parallel algorithm, GPU**

I. Introduction

Due to their versatility, Unmanned Aerial Vehicles (UAVs), or simply drones, can be efficiently used in a wide variety of applications including, but not limited to, surveillance service [1], localization [2], monitoring [3], precision agriculture [4], search and rescue [5]. Recently, this growing interest is particularly emphasized in the context of drone-based delivery systems (DBDSs) [6], [7]. With the usage of UAVs, a delivery system can be more effective and efficient due to various advantages of UAVs such as their capability to deliver in hard-to-reach places or to overcome possible traffic jam congestion on roads. Compared with the traditional truck-based delivery system, drones are faster as they can fly over small buildings and directly fly on straight lines shortening the traveled distance, and can easily traverse difficult terrain. On the other hand, there are several challenges to be addressed while relying on a DBDS, summarized as follows:

- *Energy constraints:* Drones are powered by limited capacity batteries whose energy consumption depends on the delivery route for performing a complete back and forth from the depot to the customers. Also, such energy requirement for a fixed route is not generally constant and changes due to various external dynamics like wind [8].
- *Payload constraints:* Drones have a maximum payload mass when carrying packages to customers. The consumed energy for delivering also depends on the actual payload. Moreover, due to technical constraints, a drone can deliver a single package at a time [9].
- *Limited communication range:* Drones have limited communication range which in turn impacts the maximum distance that they can go from the depot [10].
- *Limited computation resource:* For reducing the total mass, drones have limited onboard computation resources which heavily affects their capabilities in performing local high-demanding tasks [10].

In a DBDS, the delivery cost can be expressed in different metrics like distance to travel, time of flight, or energy consumed, which basically are sides of the same coin. However, in this paper, we take into consideration the energy consumption when computing efficient routes. In the real world, the drone's energy consumed not only depends on static parameters such as the drone's speed and mass but also depends on external dynamics like the current global wind [11]. Indeed, a tailwind can help a drone to fly through the air since it guarantees less energy to cut the air, whereas a headwind increases its energy usage [12]. Notice that, the wind characteristics like strength and direction can dynamically change over time. Therefore, given a drone's route, the actual energy usage can be different from the expected one since the delivery route comprises time-varying paths which in turn depend on various time-dependent variables. Moreover, the delivery drones are potentially small flying devices with limited computational resources. Thus assigning additional burdens (e.g., locally computing dynamic routes on the fly) can eventually speed up their energy consumption reducing the total flight autonomy.

The DBDS can be abstracted using a *graph* whose vertices are the locations (depots, customers), and the edges, which are also labeled with a cost (weight) in terms of energy, are the connections among locations. When dealing with dynamic costs on edges, the DBDS can be modeled using *temporal graphs* whose edges' weights change from time to time [6], while the set of vertices remains untouched. In order to analyze temporal graphs, traditional methods work with several snapshots of the temporal graph taken at different

978-1-6654-4935-9/21/$31.00 ©2021 IEEE
DOI 10.1109/MASS52906.2021.00061

time instances, and then invoke classical algorithms for static graphs (e.g., shortest path) on them. However, if at any subsequent snapshot it would be possible to exploit the previous graph properties (history) for computing routes, the required resources can be significantly reduced in terms of both space and time [13], [14]. In this paper, we devise a centralized DBDS that relies on a novel parallel algorithm called Parallel Shortest Route Update (PSRU) for efficiently computing time-varying delivery routes in a multi-depot multi-drone delivery model. Such computations are done by a centralized server and selectively sent to the proper delivery drones.

The major contributions of this paper are as follows.

- We devise a centralized DBDS which is in charge of efficiently computing the delivery routes for drones when external dynamics affect the drone's energy consumption. These routes are then transferred to the drones.
- We propose a novel parallel algorithm called PSRU to determine the shortest delivery route in temporal graphs.
- We implement PSRU using NVIDIA GPU architecture and prove its efficacy and efficiency by comparing it with state-of-the-art techniques.

The paper is organized as follows. Section II reviews related work and Section III describes the DBDS model. Section IV proposes our novel approach to efficient route selection. Section V designs a parallel algorithm for DBDS while Section VI evaluates the performance. Section VII offers conclusions.

II. Related Work

In this section we propose current techniques in a DBDS scenario. First, we discuss various dynamics that can affect the cost of flying, and then we argue about the existing algorithms for choosing the shortest routes for delivery.

A. Dynamics Affecting Drone-based Delivery

In [6], the Mission-Feasibility Problem is investigated for a DBDS in varying global wind conditions. One offline and two online algorithms are proposed for finding energy-efficient routes. The offline one, performed by the server, considers the initial graph snapshot when computing the Single Source Shortest Path (SSSP); while the online ones, locally computed by the drones, consider the dynamicity of the graph recomputing the delivery routes from scratch.

The cost of performing deliveries using drones in an energy-constrained scenario not only depends on the global wind but also on other parameters like the drone's speed, altitude, and payload that can affect the energy consumption [15], [16]. The problem of supplying multiple relief packages using a fleet of identical drones in a disaster scenario is considered in [17]. The proposed solution is not very restrictive about energy constraints and considers additional recharge stations on the route, allowing drones to recharge their batteries.

The mission planning problems for drones under weather uncertainty are studied in [8]. The mathematical formulation considers the demand of goods at the delivery point, collision avoidance, and customer satisfaction along with factors like wind conditions and ground speed of the drones. In [7], the authors deal with the problem of finding a suitable depot's location in a mixed landscape scenario (formed by two contiguous areas, each with a different metric), aimed at minimizing the overall drone's delivery distance from the depot and all possible customers in the area. For a more general approach, see [18].

B. Algorithms for Finding Delivery Paths

The success of the DBDS depends on the considered system model and the algorithms for calculating the delivery routes. In an energy-constrained scenario, the delivery cost is considered to be the energy required to deliver a package from the depot to the customer and, go back. Therefore, this routing problem can be easily converted into multiple instances of SSSP in which a drone serves a single customer at a time, and the weights of the routes are the costs in terms of energy.

A high-performance graph library, Gunrock [19], provides a data-centric abstraction on a set of vertices or edges providing a three-step architecture (advance, filter, and compute) to compute SSSP on GPUs. However, their algorithm only focuses on static graphs. In [20], a GPU implementation of the Bellman-Ford shortest path algorithm is proposed. This algorithm exploits dynamic parallelism, a feature of modern Kepler GPU. A detailed study on the performance of various algorithms on graphs including SSSP on temporal graphs, different multi-core architectures, and GPU accelerators, is proposed in [21]. A dynamic incremental and decremental SSSP algorithm is implemented using JavaScript in [22]. However, the results in this paper show that the algorithm performs well only if the number of changed edges is less than 10%. A parallel algorithm template for updating SSSP in large-scale dynamic networks is proposed in [23]. This paper deals with generic undirected graphs and empirically shows good scalability. The authors claim that the template is computing-architecture independent, and they also provide two implementations, one for shared memory, and another for NVIDIA GPU architecture. The proposed algorithm first finds the affected subgraph due to change in the network and then updates the shortest distance of the affected vertices. In this paper, we adopt a similar approach for updating the shortest delivery route for our DBDS in time-varying dynamics.

In a recent work in [24] the authors implement the Bellman-Ford algorithm using parallel hypergraph algorithms, while others [25] provide two implementations of Δ–stepping algorithm on static graphs in shared multi-core memory architecture. A shared-memory-based amorphous data-parallelism programming model, Galois [26], provides an implementation of Dijkstra's algorithm. It supports priority scheduling and processes on active elements comprised of a subset of vertices. In a distributed platform, a software named Havoqgt [27] can compute SSSP. Both Galois and Havoqgt do not have any support for dynamic networks. Srinivasan et al. [28] propose the first shared-memory algorithm for updating SSSP on dynamic networks, and implemented it using OpenMP. A Spark-based implementation to update SSSP on dynamic networks is reported in [29].

III. System Model

In this paper, we consider a DBDS that includes multiple depots each capable of simultaneously serving multiple customers with the help of multiple drones. The objective is to efficiently determine the minimum drone's energy cost routes in order to increase the total number of successful deliveries. Specifically, we aim to find a suitable route for drones that start from the depot, go up to the customers, and then return to the depot. The cost of such routes also depends on external dynamics, and drones need to recompute their routes accordingly. However, when pursuing this goal, we consider drones with minimal computation power. For this goal, we develop a centralized system for all drones associated with many depots to dynamically compute (and quickly recompute) minimum cost delivery routes, where the drone's energy consumption on the edges is time-dependent.

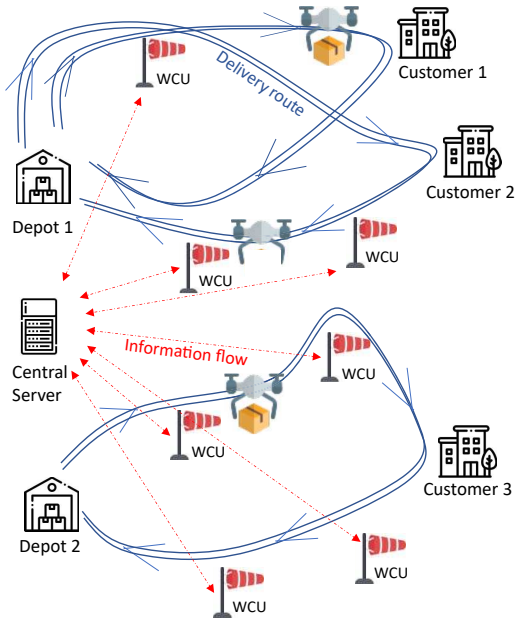


Fig. 1. A DBDS scenario where multiple drones are delivering items from multiple depots. WCUs are measuring the time varying dynamics and sending data to a central server, which is analyzing the data and controlling every delivery by sending instructions to drones.

As said before, the DBDS can be influenced by external factors like the global wind, which in turn affects the actual drone's energy consumption. When the energy consumption changes, the drone's routes also vary accordingly, and so the system should be efficient in recalculating these routes. An illustration of our proposed DBDS is depicted in Figure 1. In this multi-depot multi-drone system, the Wind Control Units (WCUs) are capable to measure wind characteristics (i.e., strength and direction) to be periodically sent to the centralized server for further analysis. With the help of these data, the server performs some analysis in order to find the updated drones' routes which are then sent to them when they are currently performing a delivery task. So, instead of relying on the onboard drone's computation, this model delegates the computational tasks to the central server. Also, the last-mile deliveries associated with many depots can be planned, monitored, and efficiently executed from a centralized vertex.

Assumptions: In our proposed model, the following assumptions are taken into account: *(i)* All delivery orders and customer locations are known in advance; *(ii)* All possible paths between a depot and a customer are known; *(iii)* All drones are identical; *(iv)* A drone is assumed to serve a single customer at a time as it can carry a single item due to the payload constraints; *(v)* A drone returns back to its starting depot after the delivery is performed; *(vi)* All drones are fully charged when they start for their deliveries and can recharge or swap a new battery only after returning to the depot.

A. Dynamic Graph Model

We can consider the entire DBDS as a *dynamic graph* whose topology remains the same while the weight of the edges changes over time. Let $G = (V, E; t, \omega_t)$ be a graph where $V = V^D \cup V^W \cup V^C$ is the set of *vertices* and V^D, V^W, and V^C denote the set of depots, WCUs, and customers, respectively; E is the set of directed *edges* among pair of vertices; $t \geq 0$ is the discrete current *time instance*; and $\omega_t : E \to \mathbb{R}^+$ is a *temporal weight function* that associates edges $(u, v) \in E$ with a drone's cost in terms of energy required $\omega_t(u, v)$ for flying from vertex u to vertex v at time t. Notice that G is not a complete graph as there can be no flying path available between two vertices due to obstructions or flying restrictions. The traveling cost is time-dependent as it depends on several time-varying dynamics including wind characteristics, flying restrictions, and drone-flying parameters, as already discussed in Section II-A.

The energy required for a trip from a depot vertex $u \in V^D$ to a customer vertex $v \in V^C$ is equal to the sum of all edges' weights on the path from u to v, plus the costs from v to u for going back to the depot. As the total mass (drone itself plus its payload) affects the energy usage, under the same flight conditions, and the hypothesis of no wind, the drone requires more energy to fly from the depot to the customer (with payload) than from the customer to the depot (without payload). Therefore, let G^F be the *forward graph* whose edges' weights denote the cost of flying when a payload is present, and similarly, let G^R be the *return graph* whose edges' weights denote the cost of flying without a payload. G^F is used to find a route from the depot to the customers, and similarly, G^R is considered to find the return route from the customers to the depot. We assume that the total number of depots is fixed, and each of them has a fixed number of drones. So, it is possible to generate delivery graphs G^F and G^R where only the customers' vertices and the associated edges change depending on the customers' location. Let $w_i \in V^D$ be the vertex that represents the i^{th} depot in the area with $1 \leq i \leq k$, where k is the total number of depots. Since we assume that two depots are geographically far enough, each depot can be considered independent from the others and, therefore, also

the customers to be served. Therefore, for each depot i we define a subgraph $G_i \subseteq G$ such that $G_1 \cup \ldots \cup G_k = G$ and $G_i \cap G_j = \varnothing$, for $i \neq j$. Moreover, for each $G_i \subseteq G$ there are m_i drones for performing deliveries from w_i. Let $\Pi_i = \{\pi_i^1, \ldots, \pi_i^{m_i}\}$ be the set of actual drones that fly in G_i.

IV. Delivery Graph based Route Selection

In this section, we propose the PreProcessing (PP) algorithm (sketched in Algorithm 1) that creates the initial single source delivery graphs G^F and G^R so that the SSSP update algorithm can be efficiently applied in the following.

A. Initial Delivery Graph Preparation

Since by design our algorithm makes changes to the edges of already visited vertices, possible conflicts (on routes) between different drones can arise. To avoid overlapping routes, we replicate the graph G_i for m_i times, and considering these as separate graphs $G_i^1, \ldots, G_i^{m_i}$, where each of them can be used by a single drone to serve a single customer at a time. Analogously, for each j^{th} copy of G_i, i.e., G_i^j, we also consider its duplicated depot vertex w_i^j, with $1 \leq j \leq m_i$. Moreover, after this duplication, a graph G_i^j can be only used by drone π_i^j in our algorithm. As each of these graphs can serve only a single customer at a time, the algorithm adds a dummy customer vertex γ_i^j in each of these graphs. Finally, G^F is prepared by considering a dummy source vertex s, and connecting it to all the depot vertices w_i^j with an edge with weight zero. Similarly, G^R is prepared by considering another dummy source vertex s^R, and connecting it to all the dummy customer vertices γ_i^j with edge weight zero. Hence, G^F and G^R become single-sourced (s and s^R, respectively), but they are not connected yet as the actual edges related to customer vertices are not known at this stage.

B. Customer Dependent Delivery Graph

The number of actual customers and their locations (at vertices) vary depending on the *delivery order*. The preprocessing algorithm generates the actual delivery graphs by modifying G^F and G^R while considering the delivery order. For each $G_i \subseteq G$ there are z_i customers to be served by the drones. Let $C_i = \{c_i^1, \ldots, c_i^{z_i}\}$ be the set of actual customer vertices to be served from depot w_i such that the customer c_i^x is served before the customer c_i^y, with $x < y$. At any time instance, a depot w_i has the capacity to serve m_i customers, since m_i is the number of available drones in G_i. Therefore, the algorithm first selects m_i customer vertices from C_i and replaces the dummy customer vertices in G^F and G^R by the actual customers. All the associated edges of the customer vertices are also added in both G^F and G^R. Now, G^F and G^R become single source connected graphs. After this first delivery graph generation, the rest of the customers can be later accommodated in the graph when some drone becomes available after the first round of deliveries (e.g., a customer c_i^j with $j > m_i$). The edge weights of the initial delivery graphs are assigned depending on the considered system model and the value of associated time-varying factors at the initial time.

Algorithm 1: PreProcessing

```
Initialize G^F and G^R with a single vertex s and s^R, resp.
for i ∈ 1, ..., k do
    for j ∈ 1, ..., m_i do
        Duplicate G_i creating G_i^j whose depot is w_i^j
        Add γ_i^j in G_i^j
        G^F ← G^F ∪ G_i^j
        Add edge (s, w_i^j) in G^F with 0 weight
        G^R ← G^R ∪ G_i^j
        Add edge (s^R, γ_i^j) in G^R with 0 weight
for i ∈ 1, ..., k do
    for j ∈ 1, ..., z_i do
        Wait for a drone and assign it to customer c_i^j
        In both G^F and G^R, γ_i^j ← c_i^j
        Add associated edges of c_i^j in both G^F and G^R
SSSP(G^F, s)
```

C. Initial Shortest Delivery Route Computation

In the delivery graph G^F the drones start their deliveries from the depot, and the objective is to find all the shortest routes from such depot to the customers at the initial time. We can use any state-of-the-art parallel SSSP algorithm for computing the initial shortest routes from a source s to all customers. Since the weights of all the edges from s to any other depot vertex are zero, the shortest route from s to c_i^j always goes through w_i^j, and the path is actually the shortest path from w_i^j to c_i^j.

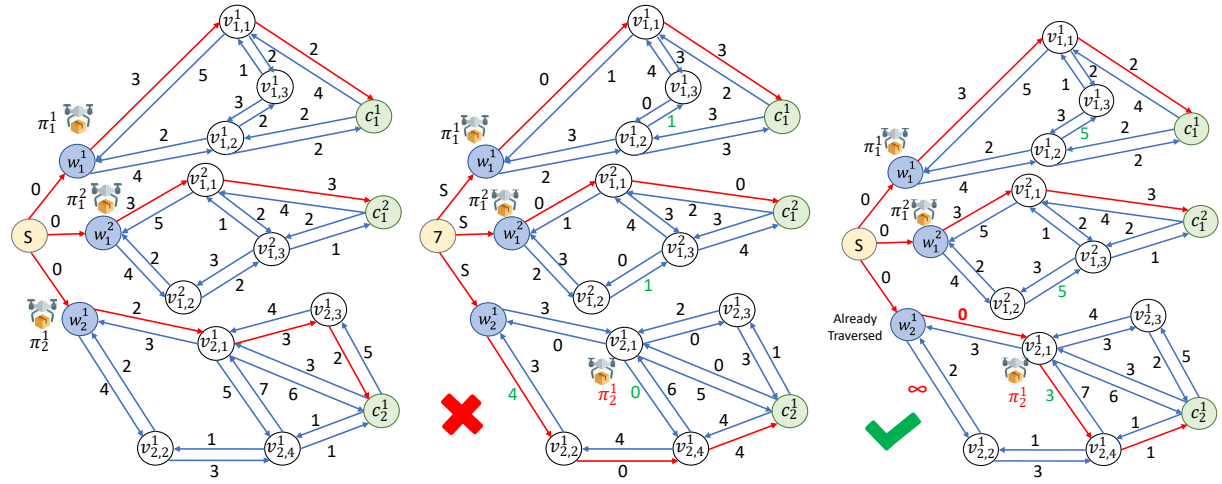
Example: In Figure 2a, we show how the DBDS introduced in Figure 1, can be modeled as a connected single-source delivery graph G^F. In that graph, the yellow vertex s is the dummy source vertex, the blue vertices are the depots, and the green vertices are the customers. It can be observed that Depot 1 has two drones, and hence there are two copies related to this depot, while Depot 2 has only a single drone, and hence there is no replication. Only the customer vertices and their related edges are different in these subgraphs. Interestingly, the initial shortest delivery routes are shown with red color.

V. The Novel PSRU Algorithm

In this section, we propose a novel algorithm called Parallel Shortest Route Update (PSRU) for updating SSSP in parallel for DBDS under time-varying dynamics.

We avoid recomputing the delivery routes from scratch, and we also store the last SSSP information in a tree structure named *SSSP tree* similar to the approach proposed by Khanda et al. in [23]. Here, the SSSP tree is rooted at the source vertex that maintains i) the shortest distance of every vertex from the source, and ii) a parent-child relationship among vertices.

Given the graph $G = (V, E; t, \omega_t)$, let T_t be the SSSP tree of G at time t (e.g., Figure 3b shows an SSSP tree of the graph shown in Figure 3a). Let ΔE_t be the set of *changed edges* from time steps $t-1$ to t due to time-varying dynamics reported by the WCUs. ΔE_t consists of both the set of inserted edges I_t and deleted edges D_t which have been changed from



(a) Graph model related to DBDS. Dummy vertex s is added to make the graph single source.

(b) A wrong approach.

(c) Right approach.

Fig. 2. Effect of drone's location on shortest delivery route: the drone π_2^1 moves from w_2^1 to $v_{2,1}^1$ at time $t-1$ (b)–(c), and $D_t = \{(v_{1,2}^1, v_{1,3}^1, 2), (v_{1,2}^2, v_{1,3}^2, 2), (w_2^1, v_{2,2}^1, 4), (v_{2,1}^1, v_{2,4}^1, 5)\}$, $I_t = \{(v_{1,2}^1, v_{1,3}^1, 5), (v_{1,2}^2, v_{1,3}^2, 5), (w_2^1, v_{2,2}^1, 1), (v_{2,1}^1, v_{2,4}^1, 3)\}$.

time steps $t-1$ to t, such that $\Delta E_t = I_t \cup D_t$. Generally, in dynamic graphs edges between vertices can be added, changed (weight), and deleted. For simplicity, in PSRU, we only focus on edge addition and deletion since a change of weight can be done by combining a deletion first and addition then. The *objective* is to efficiently find the shortest delivery routes by computing the SSSP tree T_t at time t based on the structure and edge weights of the previous SSSP tree T_{t-1} and ΔE_t.

Data Structures: We build the SSSP tree by storing the parent of each vertex in a vector $\mathcal{P}$, and the shortest distance of each vertex from the source in a vector $\mathcal{D}$. Moreover, for additional information about vertices, we use three more vectors of length $|V|$, specifically, τ which captures whether a vertex is already traversed by a drone or not, α_D which captures whether a vertex has been affected due to deletion, and α_C which checks whether a vertex has been affected by any changed edge.

In the following, we discuss the details of the PSRU algorithm which can be used for both G^F and G^R to efficiently find the updated shortest routes. To keep the simplicity of the notations, we use only G in PSRU.

A. Proposed Parallel Shortest Route Update Algorithm

The PSRU algorithm (sketched in Algorithm 2) consists of three steps that are detailed in the following.

Step 1 – Preparation of effective changed edges: After the preprocessing phase is performed, the changed edges only consider the time-varying dynamics sensed and reported by the WCUs. However, also the current drone's location can significantly impact the shortest delivery routes. Hence, if an SSSP is computed without considering the drone's location, erroneous results can be provided. For instance, an erroneous SSSP update (just by considering ΔE) is shown in Figure 2b. Here, the updated SSSP does not go through the vertex $v_{2,1}^1$ which is the current location of the drone π_2^1, and therefore the shortest delivery route for such a drone cannot be provided. Hence, we have to add a few constraints in order to ensure that the updated SSSP also considers the drone's current location. Notice that we do not use each drone's location as source vertex for computing SSSP because PSRU uses the same single-source delivery graph structure for all the deliveries to compute the shortest route in parallel.

This first step prepares the *effective set of changed edges* $\Delta\bar{E} = \bar{D}_t \cup \bar{I}_t$. If a drone moves from u to v at time $t-1$, or reaches v at time t, then the algorithm changes the weight of the edge (u, v) to 0, and sets the weight of the other outgoing edges (u, n) to $+\infty$ for each $n \in Adj(u), n \neq v$, in order to enforce the updated SSSP to visit v. Particularly, the edge (u, v) is added in $\bar{I}_t$, and all the edges (u, n) are added in $\bar{D}_t$ for deletion (Lines 5–7). Figure 2c illustrates how these restrictions can help to find the correct updated shortest delivery route. It can be observed that, due to the enforced restrictions, a changed edge $(u, v) \in \Delta E$ can impact the SSSP tree only when no endpoint of this edge is already traversed. Therefore, the algorithm finds out the effective edges from ΔE which can impact the SSSP adding them in $\Delta\bar{E}$ (Lines 8–11).

Step 2 – Identification of affected subgraphs: In this step, the affected edges are parallelly processed evaluating first the edges in $\bar{D}_t$ (also for identifying the affected vertices for deletion), and second the edges in $\bar{I}_t$.

Processing of $\bar{D}_t$: For each directed edge $(u, v) \in \bar{D}_t$, the algorithm first checks if this edge was part of T_{t-1}, since a non-SSSP tree edge deletion does not affect the shortest distance of any vertex, and hence this kind of edges require no further processing. Therefore, if the edge (u, v) belongs to T_{t-1}, only the shortest distance of v can be affected since u is the parent of v, and (u, v) is a directed edge. The algorithm

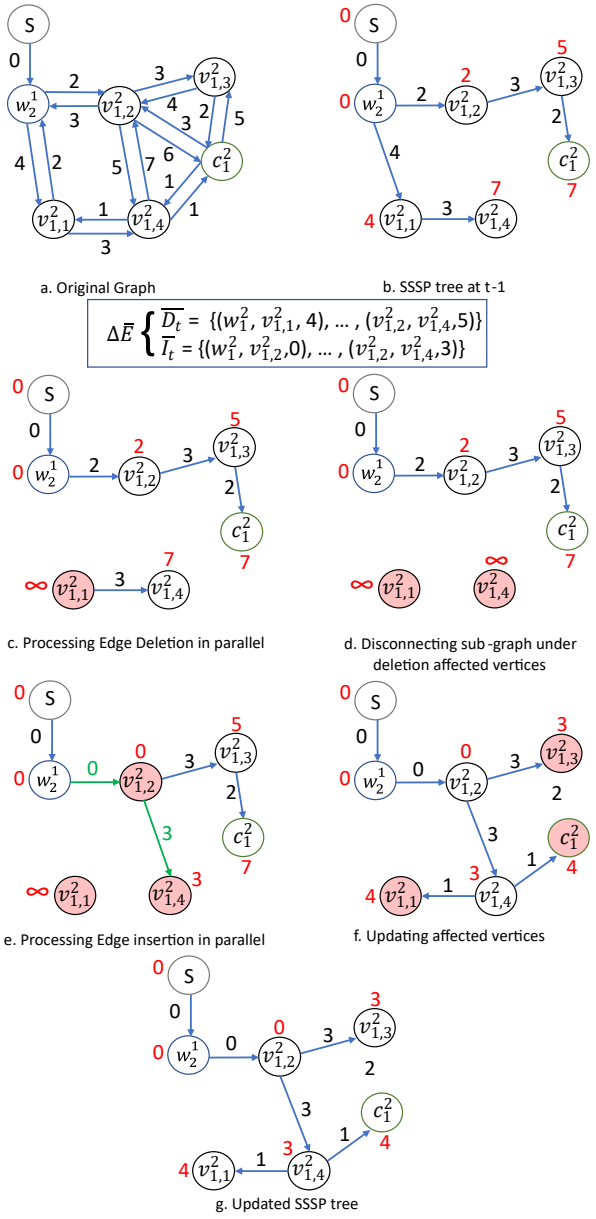


Fig. 3. An example of updating SSSP tree.

Algorithm 2: Parallel Shortest Route Update

1 **Step 1** `effectiveCE`$(G, T_{t-1}, \Delta E_t, \tau)$**:**
2 **foreach** *drone* π_i *parallelly* **do**
3 **if** π_i *travels from* u *to* v *at* $t-1$ **then**
4 $\tau[u] \leftarrow true$
5 Add (u, v) in $\bar{I}_t$ with weight 0 for insertion
6 **for** *each neighbor* n *of* u *and* $n \neq v$ **do**
7 Add (u, n) in $\bar{D}_t$ with weight $\omega_{t-1}(u, n)$ for edge deletion
8 **foreach** *edge* $(u, v) \in I_t$ *parallelly* **do**
9 **if** $\tau[u] \neq true \wedge \tau[v] \neq true$ **then** Add (u, y) in $\bar{I}_t$
10 **foreach** *edge* $(u, v) \in D_t$ *parallelly* **do**
11 **if** $\tau[u] \neq true \wedge \tau[v] \neq true$ **then** Add (u,v) in $\bar{D}_t$
12 $\Delta\bar{E}_t \leftarrow \bar{D}_t \cup \bar{I}_t$
13 **return** $\Delta\bar{E}_t$
14 **Step 2** `findAffected`$(G, T_{t-1}, \Delta\bar{E}_t, \mathcal{P}, \mathcal{D})$**:**
15 **foreach** *edge* $(u, v) \in \bar{D}_t$ *parallelly* **do**
16 **if** $(u, v) \in T_{t-1}$ **then**
17 $\mathcal{D}[v] \leftarrow \infty, \mathcal{P}[v] \leftarrow \varnothing, \alpha_D[v] = \alpha_C[v] \leftarrow true$
18 Change weight of (u, v) to ∞ in G
19 **while** α_D *has true values* **do**
20 **foreach** *vertex* $v \in V$ *s.t.* $\alpha_D[v] = true$ *parallelly* **do**
21 $\alpha_D[v] \leftarrow false$
22 **forall** *vertex* c, *where* c *is child of* v *in* T_{t-1} **do**
23 $\mathcal{D}[c] \leftarrow \infty, \mathcal{P}[c] \leftarrow \varnothing, \alpha_D[c] = \alpha_C[c] \leftarrow true$
24 **foreach** *edge* $(u, v) \in \bar{I}_t$ *parallelly* **do**
25 **if** $\mathcal{D}[v] > \mathcal{D}[u] + \omega_t(u, v)$ **then**
26 $\mathcal{D}[v] \leftarrow \mathcal{D}[u] + \omega_t(u, v), \mathcal{P}[v] \leftarrow u, \alpha_C[v] \leftarrow true$
27 Change weight of (u, v) to $\omega_t(u, v)$ in G
28 **return** α_C
29 **Step 3** `updateAffected`$(G, T_{t-1}, \mathcal{D}, \mathcal{P}, \alpha_C)$**:**
30 **while** α_C *has true values* **do**
31 **foreach** *vertex* $v \in V$ *s.t.* $\alpha_C[v] = true$ *parallelly* **do**
32 $\alpha_C[v] \leftarrow false$
33 **for** *edge* (v, n), *with* $n \in V, (v, n) \in E$ **do**
34 **if** $\mathcal{D}[n] > \mathcal{D}[v] + \omega_t(v, n)$ **then**
35 $\mathcal{D}[n] \leftarrow \mathcal{D}[v] + \omega_t(v, n)$
36 $\mathcal{P}[n] \leftarrow v$; $\alpha_C[n] \leftarrow true$
37 **for** *edge* (n, v), *with* $n \in V, (n, v) \in E$ **do**
38 **if** $\mathcal{D}[v] > \mathcal{D}[n] + \omega_t(n, v)$ **then**
39 $\mathcal{D}[v] \leftarrow \mathcal{D}[n] + \omega_t(n, v)$
40 $\mathcal{P}[v] \leftarrow n$; $\alpha_C[v] \leftarrow true$

then sets $\mathcal{D}[v] \leftarrow \infty$ and $\mathcal{P}[v] \leftarrow \varnothing$, making the vertex v, now disconnected from the parent tree. To indicate that v is affected by deletion, $\alpha_D[v]$ and $\alpha_C[v]$ are changed to $true$ (Line 17), and the weight of the edge (u, v) is updated to ∞ in the updated delivery graph (Line 27). Since v is disconnected, also the subtree rooted at v will be disconnected. Therefore, the algorithm updates the distance of the vertices descendant from v to ∞ and sets α_C to $true$ for each of them (Line 23).

Processing of $\bar{I}_t$: The insertion of an edge $(u, v) \in \bar{I}_t$ can affect the SSSP tree only when $\mathcal{D}[v] > \mathcal{D}[u] + \omega_t(u, v)$. If such an edge (u, v) satisfies this condition, $\mathcal{D}[v]$ is updated to $\mathcal{D}[u] + \omega_t(u, v)$, $\mathcal{P}[v]$ is updated to u, and $\alpha_C[v]$ is set to $true$ to indicate that the vertex v is affected (Line 26). For each edge $(u, v) \in \bar{I}_t$, the weight is changed to $\omega_t(u, v)$ in G. Therefore, at the end of Step 2, G becomes $(E \cup \bar{I}_t) \setminus \bar{D}_t$.

Step 3 – Update on affected subgraph: The last step performs a visit of all the affected vertices for both deletion and insertion marked by α_C and updates their shortest distances as well. For each marked vertex v and its outgoing edges (v, n), if the distance $\mathcal{D}[n]$ is shortened by visiting v, then the distance of n is updated to $\mathcal{D}[v] + \omega_t(v, n)$. Also $\mathcal{P}[n]$ is set to v. On the other hand, i.e., if $\mathcal{D}[v] > \mathcal{D}[n] + \omega_t(n, v)$,

then $\mathcal{D}[v]$ is updated, and $\mathcal{P}[v]$ is updated to n as well. The algorithm then iteratively proceeds tending at decreasing the distance of the vertices. Hence, the process always converges and occurs when there is no vertex left whose distance can be updated. Eventually, the algorithm returns an updated SSSP tree T_t after Step 3.

The PSRU adapts the SSSP update algorithm (proposed in [23]) for individual drones. Therefore, the correctness of the SSSP update algorithm on the graph related to each drone (starting at the drone's current location) is preserved. For our DBDS, multiple SSSP starting at the drone's current location are connected using directed edges with weight zero coming from dummy source vertex. Therefore correctness is preserved cumulatively for the whole system.

Example: Figure 3 illustrates an example of SSSP update relatively to the subgraph identified by the vertices s, w_2^1, $v_{2,1}^1, \ldots, v_{2,4}^1$, and c_2^1 in Figure 2a (which came from Depot 2 of Figure 1). The effective set of change edges $\Delta\bar{E}$ is prepared using D_t and I_t mentioned in Figure 2 and by considering that the drone π_2^1 has recently moved from vertex w_2^1 to $v_{2,1}^1$. For simplicity, in this illustration, we only show a part of the whole delivery graph. The $\mathcal{D}$ values are shown in red, while the affected vertices at each step are shown in light red. Figures 3(c–g) show the steps of the PSRU algorithm.

B. Complexity and Speedup Analysis

This section analyzes the computational complexity and speedup of our parallel algorithm.

Theorem 1. *For p number of processing units and d diameter of the graph, the time complexity of PSRU is* $\mathbb{T}_p = \mathcal{O}\left(|\Delta E|/p\right) + \mathcal{O}\left(d\chi\delta_{avg}/p + d\right)$.

Proof. For Step 1, each changed edge can be processed in parallel requiring $\mathcal{O}(|\Delta E|/p)$ time. For Step 2, each changed edge in $\bar{D}_t$ and $\bar{I}_t$ is processed in parallel taking $\mathcal{O}(|\Delta\bar{E}|/p) \approx \mathcal{O}(|\Delta E|/p)$ time. For disconnecting the descendants of the deletion affected vertices, in each iteration, the algorithm visits the neighbors of each affected vertex, and only if a neighbor is the child of an affected vertex, the neighbor is disconnected and marked as affected. Therefore, the work at this stage is proportional to the degree of affected vertices. If χ_D vertices are affected by processing edge deletion and δ_{avg} is the average degree of vertices, then the time required for each iteration is $\mathcal{O}(\chi_D\delta_{avg}/p)$ and the maximum number of iterations required is the diameter of the graph d. Hence, the time complexity for Step 2 is $\mathcal{O}(d\chi_D\delta_{avg}/p + d)$, where the last term d comes from the fact that at least constant time has to be paid for each iteration.

Similar to the last part of Step 2, in Step 3 the algorithm visits the neighbors of the affected vertices in each iteration and marks a new set of vertices as affected after updating their distance. Therefore, similarly to Step 2 time complexity, the total time required for Step 3 will be $\mathcal{O}(d\chi\delta_{avg}/p + d)$ where χ is considered as the number of affected vertices (sum of both deletion affected and insertion affected) at each iteration in Step 3. Therefore, overall time complexity of the algorithm is $\mathbb{T}_p = \mathcal{O}\left(|\Delta E|/p\right) + \mathcal{O}\left(d\chi_D\delta_{avg}/p + d\right) + \mathcal{O}\left(d\chi\delta_{avg}/p + d\right) \approx \mathcal{O}\left(|\Delta E|/p\right) + \mathcal{O}\left(d\chi\delta_{avg}/p + d\right)$. □

Space Complexity: The preprocessing phase duplicates the graph G_i, related to a depot w_i, into m_i copies requiring $\mathcal{O}(m_i(|V_i| + |E_i|))$ space, where V_i and E_i are the sets of vertices and edges in G_i, respectively. However, our main algorithm works on SSSP trees processing edges only when these actually belong to an SSSP tree. Therefore, instead of relying on copies of graphs, it is enough to duplicate for m_i times only the initial SSSP tree associated with G_i and store the adjacency list of G_i only once. Therefore, the space requirement related to a single depot is $\mathcal{O}((m_i + 1)|V_i| + |E_i|) = \mathcal{O}(m_i|V_i| + |E_i|)$. Overall space requirement will consider the graphs related to all k depots and it will be sum over i of $\mathcal{O}(m_i|V_i| + |E_i|)$.

For instance, let T_i^j be an SSSP tree associated with G_i and drone π_i^j. Notice that, the adjacency list of G_i becomes shared with all SSSP trees T_i^j, where $j = 1, \ldots, m_i$. Hence, we cannot directly change the weight of an edge to 0 if traversed by a drone in G_i. To work around this issue, we need to follow the below steps:

(i) An already traversed edge should be updated with weight 0 directly in the related SSSP tree, without updating it in G_i. For example, if a drone π_i^j moves from u to v in G_i, $\omega(u, v) = 0$ in T_i^j while no change is required in G_i.

(ii) A new Boolean array τ_i^j is required to track if a vertex in G_i is traversed by π_i^j. Here $\tau = \cup_{i=1}^{k} \cup_{j=1}^{m_i} \tau_i^j$.

(iii) Insertion/deletion or usage of an edge (x, y) in the algorithm should be avoided for a specific SSSP tree T_i^j if any endpoint of this edge is already traversed by π_i^j. For instance, deletion/insertion of edge $(u, n) \in \Delta E$ should be avoided in T_i^j if $\tau_i^j[u]$ is true. However, this edge addition/deletion should be updated in G_i.

Speedup: The speedup is defined as the ratio between the time $\mathbb{T}_{A,1}$ of a sequential algorithm A and the time $\mathbb{T}_{A,p}$ of the same algorithm performed in parallel with p processors. Therefore, in our case speedup can be written as:

$$\mathcal{S}_p = \frac{\mathcal{O}(|\Delta E|) + \mathcal{O}(d\chi\delta_{avg} + d)}{\mathcal{O}(\frac{|\Delta E|}{p}) + \mathcal{O}(\frac{d\chi\delta_{avg}}{p} + d)} \tag{1}$$

This equation indicates that the speedup becomes optimal if $\frac{|\Delta E| + (d\chi\delta_{avg})}{p}$ dominates the diameter d, which means number of changed edges $|\Delta E| + d\chi\delta_{avg}$ is at least dp.

We note that the computational complexity and the speedup of our algorithm depend on the diameter d of the graph. In this regard, since our approach can be applied to any delivery network, we may consider random graphs and give a few examples of bounds. In the case of Barabàsi and Albert process in which vertices are sequentially added to the graph with a probability proportional to its degree, it was proven that $d = \frac{\log n}{\log\log n}$ [30]. In the case of sparse random graph, in which vertices are added to the graph with a given fixed probability ρ, then $d = \frac{\log n}{\log(n\rho)}$ [31].

VI. Performance Evaluation

We implemented our PSRU algorithm using NVIDIA CUDA C++ language and evaluated its performance on an NVIDIA Tesla V100 GPU (80 streaming multiprocessors) with 32GB memory whose host processor is a Dual 32 core AMD EPYC 7452. In this massively parallel GPU architecture, we use 1024 Compute Unified Device Architecture (CUDA) threads per thread-block. In our implementation, we assign each changed edge to each thread, and for iterative operations on affected vertices, we assign each of them to a thread.

To evaluate the performance of PSRU in terms of *execution time* and *scalabilty*, we consider several networks (datasets). Networks are reported in Table I showing also their number of vertices and edges. We decided to use these datasets to consider scenarios when a large delivery company uses our algorithm, on a centralized server, to calculate all the possible deliveries in the area. In this scenario, it is reasonable to consider millions of changes to the weights of the edges due to the meteorological change. Therefore, is important to have algorithms to be able to calculate online, and therefore quickly, the possible paths of all drones.

To prove the efficacy of PSRU, we compare it against, to the best of our knowledge, the fastest parallel algorithm in the literature. This compared GPU-based algorithm is an implementation provided by the Gunrock [19] library. Differently from us, Gunrock computes from scratch the graph updates after every change.

TABLE I
USED NETWORKS FOR COMPARING THE ALGORITHMS.

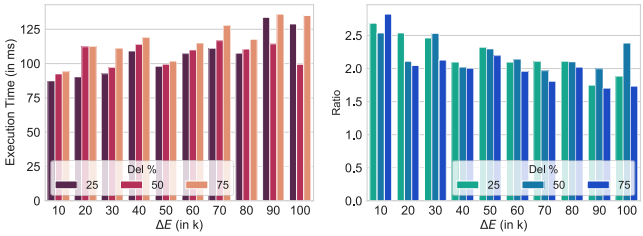
Network Name	Alias	Ref	Num. of Vertices	Num. of Edges
roadNet-PA	DS3	[32]	1,087,562	1,541,514
graph500-scale23-ef16	DS4	[32]	4,606,315	258,503,995
RMAT24_G	DS5	[33]	16,777,215	134,511,383

A. Execution Time Comparison

In Figure 4, we start evaluating the performance of PSRU on a realistic dataset (DT3, see Table I) considered as a delivery graph. To inject dynamicity in the graph, we randomly insert and delete edges as a batch of edge updates. In this time-varying graph, we find out the shortest paths from the source vertex 0 and record the execution time for performing the updates for each batch of changes.

In order to understand the goodness of our technique, we consider different *mixes* of edge deletions and insertions. If a batch of edge updates has a total of changed edges ΔE, we vary the *percentage of edge deletions* (25%, 50%, and 75%) of the total ΔE. Specifically, we have three possible combinations, i.e., more insertions (25%), more deletions (75%), and perfectly balanced (50%). In the evaluation, we vary the total number of changed edges from 10,000 to 100,000 and use the three aforementioned edge deletions percentages. The number of iterations in Step 2 and Step 3 of PSRU depends on the location of the change in the graph, and the total execution time depends on the total number of affected vertices. As we generate the changed edges randomly, the execution time in our evaluation is dependent on the graph topology, and the batch of changed edges. Finally, each scenario is run five times and the average is taken.

In Figure 4a, we report the execution time of PSRU applied on DS3. In general, the time required for updating all the edges is approximately 100ms on average. In particular, one can observe the small increase of time when the number of changed edges increases. In Figure 4b, we compare the performance among PSRU and Gunrock. Specifically, we report the *ratio* of the execution time taken by Gunrock divided by PSRU. We highlight that our implementation outperforms that of Gunrock since this ratio is always larger than 1.0 (the average ratio is 2), which means that PSRU is faster than Gunrock in terms of execution time.

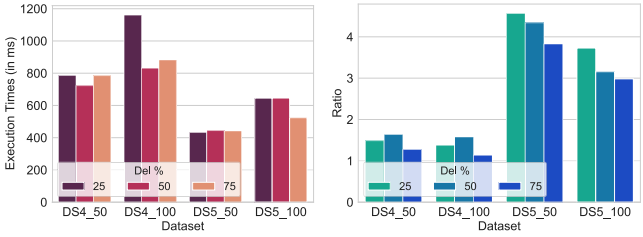


(a) Execution time. (b) Execution time ratio.

Fig. 4. Performance analysis on DS3. (a) execution time of PSRU on different batches, and (b) execution time ratio among PSRU and Gunrock.

B. Scalability Comparison

To evaluate the *scalability* of PSRU, we use two large scale graphs as input (DS4 and DS5, see Table I). Here, we vary the size of the batches considering 50 and 100 million change edges. As before, we balance the percentage of deletions and insertions using the same percentages.



(a) Execution time (b) Execution time ratio

Fig. 5. Execution time and performance comparison for large scale graphs when $\Delta E = 50, 100$ millions.

Figure 5 shows the performance on large-scale instances. Figure 5a highlights the execution time of PSRU showing that the maximum reported execution time is about 1.2s even when the input is very large. Interestingly, even though the DS5 graph has more vertices than DS4, the execution time of DS4 is larger than the DS5 since the DS4 graph is denser than the DS5 graph in terms of edges. Finally, we compare the execution time ratio among PSRU and Gunrock. Overall, PSRU outperforms Gunrock (up to 4.5×) on large instances.

VII. Conclusion

In this paper, we devise a centralized DBDS for a multi-drone multi-depot architecture, and present PSRU, a parallel shortest delivery route update algorithm for efficiently computing the drones' delivery routes under time-varying dynamics, such as wind. We implement our proposed algorithm on GPUs and comprehensively test its performance on large-scale inputs for evaluating the execution time and scalability. The performance of PSRU outperforms the state-of-the-art implementation provided by Gunrock. In future work, we would like to change the system model allowing drones to start and finish missions to different depots. It would be also interesting to investigate the presence of rechargeable stations inside the delivery area in order to prolong the lifetime of drones, especially in case of unexpected battery drainage.

Acknowledgments

This work was partially supported by NSF grants CNS-1818942, OAC-1725755, OAC-2104078, and SCC-1952045; and also partially supported by "HALY-ID" project funded by the European Union's Horizon 2020 under grant agreement ICT-AGRI-FOOD no. 862665, no. 862671, and from MIPAAF.

References

[1] I. Bisio, C. Garibotto, F. Lavagetto *et al.*, "Blind detection: Advanced techniques for wifi-based drone surveillance," *IEEE Trans. on Vehicular Technology*, vol. 68, no. 1, pp. 938–946, 2018.

[2] F. Betti Sorbelli, C. M. Pinotti, S. Silvestri, and S. K. Das, "Measurement errors in range-based localization algorithms for *UAVs*: Analysis and experimentation," *IEEE Trans. on Mobile Computing*, to appear, 2021.

[3] A. Khochare, Y. Simmhan, F. B. Sorbelli, and S. K. Das, "Heuristic algorithms for co-scheduling of edge analytics and routes for *UAV* fleet missions," in *IEEE INFOCOM*, 2021.

[4] D. Murugan, A. Garg, and D. Singh, "Development of an adaptive approach for precision agriculture monitoring with drone and satellite data," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 10, no. 12, pp. 5322–5328, 2017.

[5] T. Calamoneri and F. Corò, "A realistic model for rescue operations after an earthquake," in *16th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, 2020, pp. 123–126.

[6] F. B. Sorbelli, F. Corò, S. K. Das, and C. M. Pinotti, "Energy-constrained delivery of goods with drones under varying wind conditions," *IEEE Transactions on Intelligent Transportation Systems*, to appear, 2021.

[7] L. Bartoli, F. B. Sorbelli, F. Corò, C. M. Pinotti, and A. Shende, "Exact and approximate drone warehouse for a mixed landscape delivery system," in *IEEE International Conference on Smart Computing (SMARTCOMP)*, 2019, pp. 266–273.

[8] A. Thibbotuwawa, G. Bocewicz, P. Nielsen, and B. Z., "Planning deliveries with *UAV* routing under weather forecast and energy consumption constraints," *IFAC-PapersOnLine*, vol. 52, no. 13, pp. 820–825, 2019.

[9] S. Sawadsitang, D. Niyato, P. S. Tan, P. Wang, and S. Nutanong, "Multi-objective optimization for drone delivery," in *IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, 2019, pp. 1–5.

[10] W. Shi, H. Zhou, J. Li, W. Xu, N. Zhang, and X. Shen, "Drone assisted vehicular networks: Architecture, challenges and opportunities," *IEEE Network*, vol. 32, no. 3, pp. 130–137, 2018.

[11] M.-h. Hwang, H.-R. Cha, and S. Y. Jung, "Practical endurance estimation for minimizing energy consumption of multirotor unmanned aerial vehicles," *Energies*, vol. 11, no. 9, p. 2221, 2018.

[12] T. Nguyen and T. Au, "Extending range of delivery drones by exploratory learning of energy models." in *AAMAS*, 2017, pp. 1658–1660.

[13] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick, "A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks," in *IEEE 25th International Conference on High Performance Computing*, 2018, pp. 245–254.

[14] S. Srinivasan, S. Pollard, S. K. Das, B. Norris, and S. Bhowmick, "A shared-memory algorithm for updating tree-based properties of large dynamic networks," *IEEE Transactions on Big Data*, to appear, 2021.

[15] C. Di Franco and G. Buttazzo, "Energy-aware coverage path planning of uavs," in *IEEE Int Conf on Autonomous Robot Systems and Competitions*, 2015, pp. 111–117.

[16] K. Dorling, J. Heinrichs, G. G. Messier, and S. Magierowski, "Vehicle routing problems for drone delivery," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 1, pp. 70–85, 2016.

[17] B. Rabta, C. Wankmüller, and G. Reiner, "A drone fleet model for last-mile distribution in disaster relief operations," *International Journal of Disaster Risk Reduction*, vol. 28, pp. 107–112, 2018.

[18] F. Betti Sorbelli, F. Corò, C. M. Pinotti, and A. Shende, "Automated picking system employing a drone," in *15th IEEE Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2019, pp. 633–640.

[19] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[20] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler gpu architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, 2015.

[21] A. Rehman, M. Ahmad, and O. Khan, "Exploring accelerator and parallel graph algorithmic choices for temporal graphs," in *11th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2020, pp. 1–10.

[22] A. Ingole and R. Nasre, "Dynamic shortest paths using javascript on *GPUs*," in *IEEE 22nd Int Conf on High-Performance Computing (HiPC)*, 2015, pp. 1–5.

[23] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[24] J. Shun, "Practical parallel hypergraph algorithms," in *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 232–249.

[25] E. Duriakova, D. Ajwani, and N. Hurley, "Engineering a parallel δ-stepping algorithm," in *IEEE Int. Conf. on Big Data*, 2019, pp. 609–616.

[26] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *24th ACM Symposium on Operating Systems Principles*, 2013, pp. 456–471.

[27] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 549–559.

[28] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick, "A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks," in *25th IEEE Int. Conf. on High Performance Computing (HiPC)*, 2018, pp. 245–254.

[29] S. Riazi, S. Srinivasan, S. K. Das, S. Bhowmick, and B. Norris, "Single-source shortest path tree for big dynamic graphs," in *IEEE International Conference on Big Data*, 2018, pp. 4054–4062.

[30] B. Bollobás and O. Riordan, "The diameter of a scale-free random graph," *Comb.*, vol. 24, no. 1, pp. 5–34, 2004.

[31] F. Chung and L. Lu, "The diameter of sparse random graphs," *Adv. Appl. Math.*, vol. 26, no. 4, pp. 257–279, 2001.

[32] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[33] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765 –1775, October 2011.