A Dijkstra-Inspired Graph Algorithm for Fully Autonomous Tasking in Industrial Applications

Mohamed Lotfi, *Member, IEEE*, Gerardo J. Osório, Mohammad S. Javadi, *Senior Member, IEEE,* Abdelrahman Ashraf, Mustafa Zahran, Georges Samih, and João P. S. Catalão, *Senior Member, IEEE*

***Abstract*—An original graph-based model and algorithm for optimal industrial task scheduling is proposed in this paper. The innovative algorithm designed, dubbed "Dijkstra Optimal Tasking" (DOT), is suitable for fully distributed task scheduling of autonomous industrial agents for optimal resource allocation, including energy use. The algorithm was designed starting from graph theory fundamentals, from the ground up, to guarantee a generic nature, making it applicable on a plethora of tasking problems and not case-specific. For any industrial setting in which mobile agents are responsible for accomplishing tasks across a site, the objective is to determine the optimal task schedule for each agent, which maximizes the speed of task achievement while minimizing the movement, thereby minimizing energy consumption cost. The DOT algorithm is presented in detail in this manuscript, starting from the conceptualization to the mathematical formulation based on graph theory, having a thorough computational implementation and a detailed algorithm benchmarking analysis. The choice of Dijkstra as opposed to other shortest path methods (namely, A* Search and Bellman-Ford) in the proposed graph-based model and algorithm was investigated and justified. An example of a real-world application based on a refinery site is modeled and simulated and the proposed algorithm's effectiveness and computational efficiency is duly evaluated. A dynamic obstacle course was incorporated to effectively demonstrate the proposed algorithm's applicability to real-world applications.**

***Index Terms*—Graph theory, algorithms, task scheduling, energy management, Dijkstra, industrial applications.**

I. INTRODUCTION

In an exceedingly dynamic and digital world, the old saying of "time is money" presents itself in all modern problems. Energy systems witnessed momentous change during the past few decades [1], which in turn affected all sectors that are heavily energy-dependent, including industrial and transport sectors [2].

The work of Mohamed Lotfi was supported by the MIT Portugal Program (in Sustainable Energy Systems) by Portuguese and EU funds through FCT, under Grant PD/BD/142810/2018. The work of J.P.S. Catalão was supported in part by FEDER funds through COMPETE 2020 and in part by the Portuguese funds through FCT under POCI-010145-FEDER029803 (02/SAICT/2017).

M. Lotfi and J.P.S. Catalão are with the Faculty of Engineering of the University of Porto (FEUP) and the Institute for Systems and Computer Engineering, Technology and Science INESC TEC, 4200-465 Porto, Portugal (e-mails: mohd.f.lotfi@gmail.com, catalao@fe.up.pt).

G.J. Osório is with Portucalense University Infante D. Henrique (UPT), 4200-072 Porto, Portugal. (e-mail: gerardo@upt.pt).

M. Javadi is with the Institute for Systems and Computer Engineering, Technology and Science INESC TEC, 4200-465 Porto, Portugal. (e-mail: msjavadi@gmail.com).

A. Ashraf, M. Zahran, and G. Samih are with the German University in Cairo (GUC), New Cairo, Egypt. (emails: mubbyashraf@gmail.com, mustafazahran1996@gmail.com, georgessami7@gmail.com).

Despite the advancement of technologies leading to overall abundance or resources, increased socio-technical complexities associated with the availability of resources make the optimal management thereof of paramount importance [3].

In addition to the added complexity of the energy supply infrastructure, process automation levels are at an all-time high, making it necessary to develop and deploy new algorithms for optimized task management in order to guarantee cost-efficiency and reliability of these automated processes [4], [5].

A. Literature Review

A rundown of recent scientific studies is performed and subsequently presented to establish the state-of-the-art of scientific literature addressing optimal task scheduling and management in modern automated systems. As previously mentioned, two of the most affected sectors are the industrial and transport sectors [2], with maximizing cost-efficiency already of pivotal importance for the two.

For the transport sector, there has been a lot of recent focus on developing optimal management algorithms for consumer-owned electric vehicle (EV) fleets [6], with an emphasis on cost-optimal energy management in the presence of hybrid technologies [7] and considering smart homes [8] and other modern solutions for optimal utilization of distributed energy resources (DERs) [9].

This is especially important with dynamic electricity pricing schemes adopted through demand response (DR) implementation [1], [2]. Recent research on this matter was not only confined to consumer-owned EVs, with a lot of research also investigating smart public transport systems with increased proliferation of electric buses (EBs) and smart charging infrastructures [10]. The priority is ensuring cost-efficiency of the systems [11] through optimal scheduling [12].

In industrial applications, optimal management of time and resources is even more critical due to the profit-centered character of industry, throughout the wide-ranging spectrum of industrial specializations. In the context of smart factories, multi-agents systems are proposed as a model for coordination between autonomous systems working on performing preset tasks in factories with high levels of automation and a smart communication infrastructure [13].

The adoption of intelligent algorithms for optimal task scheduling in industry has been shown to result in significant cost savings, whether performed by automated mobile agents or human labor. Such saving are crucial for industries and economic growth [14].

B. Novel Contributions

Algorithms for optimal task-handling are being investigated for a wide array of applications, ranging from coordinating autonomous self-driving EVs [15], to cooperative robotics [16], industrial site inspection [17], and the management of modern warehouses [18]. As such, the development of these algorithms for optimal cost-efficiency of task handling, regardless of the type of task, becomes imperative for modern industries.

With increased intertwining of modern systems and cross-industry designs it becomes even more important to design algorithms for generic systems [19], which are not application-specific, and could be employed regardless of the target sector, be it smart factories, modern warehouses, EV fleet management, etc.

In this study, a novel algorithm for optimal task scheduling, dubbed "Dijkstra Optimal Tasking" (DOT), is proposed, implemented and validated. This algorithm was initially conceptualized by the authors of this work in a preliminary phase in [20]. The proposed algorithm is generic in nature, meaning that it can be adapted to different problems in which a limited number of mobile agents are required to perform several tasks, with minimal energy. The novel contributions of this work can thus be summarized as:

- Model a generic tasking problem using graph theory to guarantee applicability to a wide range of modern problems (particularly in the industrial and transportation sectors).
- Design and implement an original graph-based model and algorithm for task scheduling by a limited number of autonomous mobile agents.
- Perform several benchmarking analyses to determine the computational complexity of the proposed algorithm, optimal setting of tunable parameters, and assessing the performance of the proposed algorithm incorporating different shortest path methods (Dijkstra, A* Search and Bellman-Ford).
- Ensure the computational efficiency of the algorithm to enable application in real-time.
- Demonstrate the proposed algorithm considering a case study based on a real-world industrial site, including the effect of dynamic (moving) obstacles.

C. Paper Organization

This paper is organized as follows: In Section I, the motivation behind this work, literature review, and novel contributions are presented. In Section II, the proposed algorithm is documented in detail, including: the conceptual model, mathematical formulation, algorithm design, and computational implementation.

In Section III, a thorough benchmarking analysis is performed to determine optimal settings of tunable parameters and determine the time complexity. A case study is used to demonstrate a real-world application in Section IV. Finally, in Section V, conclusions of this paper are summarized and a discussion of anticipated applications of the proposed algorithm in different sectors is examined, deriving recommendations for future work.

II. Algorithm Design

A. Conceptual Model

A generic task scheduling problem in a modern industrial setting is illustrated in Fig. 1. The main elements thereof can be defined and listed as follows:

Map: is a confined space where all the tasks and mobile agents are located. All events and scheduling are performed within this local environment.

Mobile Agent: is an agent which can move around the map and perform tasks. The agent is electric, meaning it consumes electric energy on a local battery as a cost of movement. This agent could be autonomous or human-operated (e.g. Segway, electric pallet jack, or golf cart).

Charging Station: is where the mobile agent is stationed to recharge onboard batteries. Although commonly found on the edges of most maps, they can be located anywhere across the traversable map.

Tasks: must be reached by a mobile agent in order to be accomplished. This is generic, i.e., in inspection problems the task is merely for the agent to be there every period of time. In handling problems, the agent must stay until task completion.

Obstacles: are non-traversable areas. The mobile agent must plan a path around them to reach tasks or charging stations.

B. Mathematical Formulation: Graph Theory Model

These are the defining elements of any generic industrial task scheduling problem, and an algorithm aiming to provide a non-case-specific solution must be capable of incorporating them in a versatile and flexible manner.

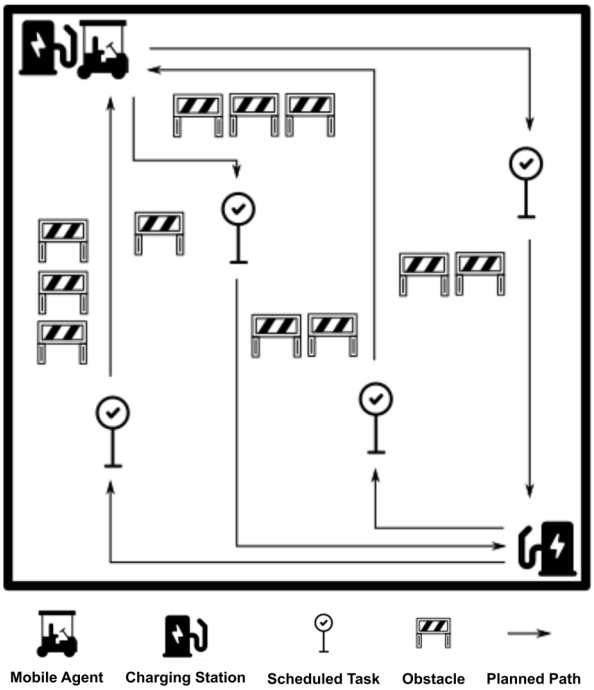

Fig. 1. An illustration of a generic tasking problem: A mobile agent needs to perform specified tasks located at different locations in a confined map, in the presence of non-traversable obstacles. The movement is associated with energy consumption and recharging is performed at set locations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIA.2021.3091418, IEEE Transactions on Industry Applications

Graph theory provides the tools to mathematically model such a problem and is therefore chosen to construct the basis of the proposed algorithm. The first step is to divide the map into a mesh of equidistant and isomorphic "cells". The size of each cell should be based on the smallest element in the map. By doing so, the problem can be defined as a graph G whose elements are mathematically defined subsequently.

$$G = (\mathbf{V}, \mathbf{E}) \tag{1}$$

1) *Graph Vertices*

The graph defined in (1) consists of a set of vertices **V** (which correspond to the "cells"), and a set of edges **E**.

$$i \in \mathbf{V} \ \forall \ i \in \mathbb{Z} \ \cap [0, |\mathbf{V}| - 1] \tag{2}$$

As defined in (2), **V** is a set of vertices *i*: *{0, 1, … , |V|-1}*. At this stage it is established that a zero-based numbering convention (initial element assigned index 0) is used throughout this paper, due to its compatibility with graph theory modeling and algorithm design. The vertices are numbered sequentially, row-by-row, as shown in Fig. 2.

The equidistant and isomorphic division of the map results in a "lattice graph" (illustrated in Fig. 2), a unique type of graph whose properties can be exploited. First, the size of the graph is decided by the number of rows *m* and number of columns *n*. The total number of vertices is easily expressed as in (3).

$$|\mathbf{V}| = m \cdot n \tag{3}$$

Afterwards, each vertex in the lattice graph can be uniquely mapped to a row and column value *R(i)* and *C(i)* in a set of rows **R** and columns **C**, respectively. Being a single-values unique mapping between the sets, no two different vertices *(i,j)* can have both the same row and column values as shown in (4), and the sets have the same size as shown in (5).

The mapping functions of the row and column values for each vertex *i* is done using (6) and (7), respectively. In (6) the row number of node *i* is obtained by applying the *modulo* operator of *i* to *n* (remainder of division), while the column is calculated using integer (truncated) division in (7). This is a simple demonstration of the advantage of using zero-based numbering to obtain simple and computationally efficient operations within the graph.

$$\big(C(i) = C(j)\big) \ \wedge \ \big(R(i) = R(j)\big) \ \leftrightarrow \ i = j \tag{4}$$

$$|\mathbf{C}| = |\mathbf{R}| = |\mathbf{V}| \tag{5}$$

$$R(i) = (i \bmod n) \tag{6}$$

$$C(i) = [(m \cdot i) \ / \ |\mathbf{V}|] \tag{7}$$

The *x* and *y* coordinates of each vertex on the original (physical) map can be retrieved using (8) and (9), where d_x and d_y correspond to the horizontal and vertical spacing between cells, respectively.

$$x(i) = C(i) \cdot d_x \tag{8}$$

$$y(i) = R(i) \cdot d_y \tag{9}$$

For equidistant and isomorphic spacing, this is simplified by setting $d_x = d_y = \Delta v$. Furthermore, the relationship between row and column values of adjacent vertices the graph is defined using (10)-(13).

$$R(i) = R(i-1) \leftrightarrow row(i) > 0 \tag{10}$$

$$C(i) = 1 + C(i-1) \leftrightarrow R(i) > 0 \tag{11}$$

$$C(i) = C(i-n) \leftrightarrow C(i) > 0 \tag{12}$$

$$R(i) = 1 + R(i-n) \leftrightarrow C(i) > 0 \tag{13}$$

Finally, vertices are identified as boundary vertices (set **B**) or interior domain (set **D**) nodes according to (14) and (15).

$$\begin{aligned} i \in \mathbf{B} \leftrightarrow \ i \in \mathbf{V} \wedge \Big(& (R(i) \cdot C(i) = 0) \\ & \vee \big((R(i) - m + 1) \\ & \cdot (C(i) - n + 1) = 0\big)\Big) \end{aligned} \tag{14}$$

$$i \in \mathbf{D} \leftrightarrow \ i \in \mathbf{V} \wedge \ i \notin \mathbf{B} \ \Rightarrow \mathbf{D} = \mathbf{V} - \mathbf{B} \tag{15}$$

2) *Graph Edges*

The other main element of the graph is the set of edges, **E**. An edge is a set of two vertices *{i,j}* that are connected in the graph. The set of all possible edges **E** can be defined using the condition in (16).

While loops (an edge connecting a node to itself, or subsequently a path which starts and ends at the same node) are mathematically possible in a generic graph, they do not exist in this mode, as it would correspond to indefinite circling in a closed "loop" within the map. Thus, the condition $i \neq j$ is imposed in (16). Moreover, the modeled graph is an undirected one, hence the condition in (17).

$$\mathbf{E} \subseteq \{\, \{i,j\} \mid (i,j) \in \mathbf{V} \wedge i \neq j \,\} \tag{16}$$

$$\{i,j\} = \{j,i\} \ \forall \ (i,j) \in \mathbf{V} \tag{17}$$

Based on this property, all edges of the lattice graph can be constructed by defining an edge between all interior domain vertices and their adjacent neighbors. This is mathematically expressed using the interjection in (18).

$$\begin{aligned} \exists \ \{\mathrm{i},\mathrm{j}\} \in \mathbf{E} \leftrightarrow (i \in \mathbf{D}) \\ \wedge \big((|i-j| = 1) \\ \vee \ (|i-j| = n)\big) \end{aligned} \tag{18}$$

The total number of edges in the graph can be obtained using the expression in (19).

$$\begin{aligned} |\mathbf{E}| &= (m-1) \cdot n + (n-1) \cdot m \\ &= 2 \cdot m \cdot n - m - n \end{aligned} \tag{19}$$

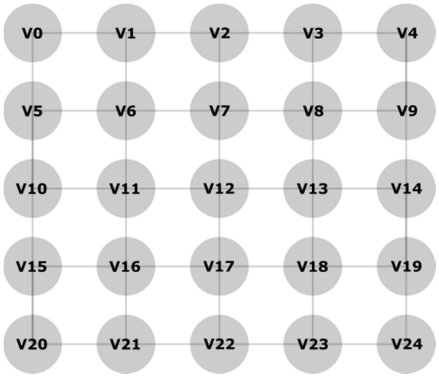

Fig. 2. An illustration of five-by-five map modeled as a lattice graph. Cells are assigned as vertices in the graph, and connections between adjacent vertices correspond to edges of the graph.

3) *Shortest Paths*

The constructed lattice graph is a connected graph, meaning that any two set of vertices *{i,j}* can connected using a number of edges. The simplest connection is a walk, in which a sequence of edges joins two vertices. A walk can either be finite or infinite, in which the edges contained in the sequence need not to be unique.

A path is defined as a walk in which all the elements are unique, i.e., every vertex in the path is only visited once. Some path *P(i,j)* connecting *i* and *j* is therefore defined in (20), with *K* being the number of elements in the path.

$$\mathrm{P(i,j)} = (P_0, P_1, \dots P_{K-1}) \mid \{P_0, P_1, \dots P_{K-1}\} \in \mathbf{V} \wedge (P_0, P_{K-1}) = (i,j) \tag{20}$$

For every set of vertices *i* and *j*, there exists a finite number of paths between them, where $\mathbf{\Pi}_{i,j}$ is a set containing all possible paths *P(i,j)*. In a weighted graph every edge *E* is associated with a weight value *ω(E)* such that $\mathbf{E} \mapsto \boldsymbol{\omega}$, with the latter being the set of edge weights. Accordingly, the weighted length *l(P)* of a path can be calculated as shown in (21).

$$l(P) = \sum_{k=0}^{K-1} \omega(\{P_k, P_{k+1}\}) \tag{21}$$

This function can be used to map $\mathbf{\Pi}_{i,j} \mapsto \mathbf{L}_{i,j}$ (set of corresponding path weighted lengths). The graph distance between two vertices is defined in (22) as the weighted length of the shortest path between them:

$$d(i,j) = \min\{\mathbf{L}_{i,j}\} \tag{22}$$

Finding the shortest path between any two vertices is a fundamental problem in graph theory for which solution algorithms have been well established. Shortest path first (SPF) algorithms are vital algorithms for graph analysis.

Dijkstra's algorithm is [21] one of the most popular and well-established fundamental SPF algorithms in graph theory. The algorithm is a highly computationally efficient algorithm finding the shortest path between two nodes in a graph as expressed in (23).

$$SPF : (i,j) \rightarrow P(i,j) \ni l\big(P(i,j)\big) = d(i,j) \tag{23}$$

It is duly noted that the proposed model accommodates the use of any shortest path method, and not necessarily Dijkstra's SPF. The choice of Dijkstra as opposed to other alternatives is discussed and analyzed in detail in Section III.C.

With the mathematical formulation being specified, the designed algorithm can now be expressed in terms of the graph elements and defined relations.

C. Designed Algorithm and Computational Implementation

Recalling the original motive, the objective of the proposed algorithm is to be generic in nature, easily adaptable to different problems with elements defined in II.A. To do this, the designed algorithm was implemented in an object-oriented programming environment. The pseudocode is shown in Algorithm I, followed by a detailed description of the implementation.

ALGORITHM I. PSEUDOCODE OF THE DESIGNED DOT ALGORITHM.

```
Input Map, TaskList, MobileAgents
while isRunning do
  tau += 1
    for each A in MobileAgents
      if A.isCharging then
          A.chargeStep()
          if A.fullyCharged then
              A.isCharging := false
              tau += 1
          end if
      else if A.atStation then
        A.path ← DijkstraSPF(A.orSta,A.deSta)
        A.atStation := false
      else
        A.loc[t] ← A.move(path,loc[t-1])
        if map[A.loc[t]].isStation then
          A.atStation := true
          tau += tau
        else
          Map.vertexHeat[loc[t]] += inc
        end if
        for each N in Map.vertexHeats
          N-=Map.CDF()
        end for
      end if
    end for
    Map.refresh()
    TaskList.refresh()
    MobileAgents.refresh()
end while
```

1) *Map (Class):* The industrial site map model is implemented as a class. A map object contains all information about the graph (edges and vertices) and the class methods to update them.

2) *Agent (Class):* Each agent is modeled as a class. The class contains information about the agent, e.g. its current location, battery state-of-charge (SoC), current path and the class methods to update all the aforementioned values.

3) *Checkpoints (Class Property of Map, Agent):* Checkpoints are locations in the map where the agents stop between map traversals. Checkpoints don't necessarily have to also contain a charging stations, whilst all charging stations are checkpoints, since the agent can stop at a station and not charge depending on its current path or schedule.

4) *Timer (Global Variable):* The time *t* is constantly incremented as a global counter in the implemented program with any update in the map. Given a lattice graph, the movement time from one node to the other can be used as the unit of time if all agents are the same model (i.e., same speed), which is common in most industrial facilities. MAXTIME can be used as a termination criterion for the program.

$$t \in \mathbf{t} = (0,1, \dots \mathbf{MAXTIME}) \tag{24}$$

5) *Traversal Timer (Class Property of Map, Agent):* A second time variable is incorporated to increase the versatility of the algorithm. Since the algorithm relies on the agents traversing between checkpoints, the flow of time can be alternatively tracked as a counter of the number of traversals the agent makes. This traversal time τ is the one used to keep track of the vertex properties and update the heat values in the map.

A good feature of this implementation, which provides the versatility, is that τ can always be set simply as $\tau = t$ in the code, switching back to real time in the dependent functions according to the type of tasking problem at hand. Like MAXTIME, TMAX can also be used as a termination criterion.

$$\tau \in \boldsymbol{\tau} = (0,1, \dots, \mathbf{TMAX}) \tag{25}$$

A mapping $\tau(t)$ can give the current increment time at any *t*.

6) *Heat Values (Class Property of Map):* Each vertex has a time-varying "heat" value H(i,t) assigned to it in the Map class. The movement cost through an edge is defined as the mean of the heat values of the two connected vertices as in (26).

$$\omega(\{i,j\},t) = \frac{1}{2}\big(H(i,t) + H(j,t)\big) \; \forall \, \{i,j\} \in \mathbf{E}, t \in \mathbf{t} \tag{26}$$

This heat property is the main premise of the designed algorithm and is used to establish all other relations to model a given problem and calculate the task scheduling. Heat values of the vertices are continuously updated to "guide" each agent through the shortest path in the graph such that all tasks are achieved while traversing between its checkpoints (Fig. 3).

7) *Obstacles* are directly incorporated by setting a very large number as heat value in that vertex. In this manner, obstacles can be modeled in a very computationally efficient way (as opposed to the use of exceptions or conditional statements), since the movement to/from that vertex is never chosen over any other alternative. This number is imposed as the largest float value of the machine where the algorithm is running, which for most modern processors is 1.7976^{308}.

8) *Tasks,* on the contrary, are by setting a low value, depending on the type and/or urgency of each task. This acts as an "attractor" for the mobile agent since the Dijkstra SPF algorithm will be attracted to pass through that vertex when constructing the path instead of other alternatives.

Once an agent has reached a vertex with a task, the heat value of this vertex is incremented. In this way, the movement cost to/from this location is increased, removing the "attractor" as the task is accomplished. A "cooldown" effect is applied by decrementing heat values of all vertices with each increment of *t*, such that for an idle map with no activity, heats are eventually reset to their initial values.

To sum up the flow of the algorithm:

- The industrial site or facility is modeled as a lattice graph.
- Each vertex has a heat property that is updated with every increment of time t based on the mobile agents' movement through the map, the flow of time, and nature of the tasks to be performed (as visualized in Fig. 3). The heat value update is done by incrementation and cooldown.
- Tasks and obstacles are modeled by setting the heat value accordingly to guide the agents.
- The heat values set the edge weights for the graph.
- The pathing of each mobile agent between its checkpoints is determined using Dijkstra's SPF (or an alternative shortest path method) every increment of traversal time τ.

The algorithm was implemented using Python 3.6.7. All subsequent tests were run on a standard laptop computer with an Intel Core i7-8550U CPU @ 1.80 GHz, 16.0 GB RAM, and Windows 10 64-bit operating system.

III. Parameter Tuning and Benchmark Testing

In this section, tuning parameters of the algorithm are identified, and a benchmark analysis is performed to test the proposed algorithm on a benchmark case, assess the appropriate values for the parameters, and analyze the computational performance and time complexity of the algorithm.

Finally, the choice of the most adequate shortest path method (Dijkstra is justified by discussing other commonly employed shortest path methods is graph theory (A* Search and Bellman-Ford) and performing a comparative analysis between feasible candidates.

A. Identifying Tunable Parameters

From the algorithm description it can be seen that there are two main parameters which can be used to tune the algorithm:

1) *Increment Value:* The first tuning parameter is the incremental heat value of a vertex once an agent completes a task there. This is defined as INC in (27). Note that this function is only invoked once an agent reaches a vertex marked with a task.

$$H(i,t) = H(i,t) + \text{INC}, \qquad \text{if task completed} \tag{27}$$

2) *Cooldown Function:* This second tuning parameter is how the vertex heat values of the whole graph are updated every time increment *t*. This is defined as a function CDF in (28).

$$H(i,t) = CDF(i, H(i,t), t) \tag{28}$$

In this study, four different types of functions are considered. A Fixed Cooldown (FCD) decrements the H(i,t) by a constant value CD every time increment, while a Zero Cooldown sets $CD = 0$ as in (29) and (30).

In (31) and (32), exponential functions are used instead for a scaled cooldown (SCD), making the cooldown value increase exponentially with every traversal time. In SCD1 and SCD2, the initial decrement values are 1 and 0, respectively. T_{obj} scales the function, increasing the exponential growth as $T_{obj} \to 0$, as shown in (33).

$$CDF_{FCD}(i, H(i,t), t) = H(i,t) - \text{CD} \tag{29}$$

$$CDF_{ZCD}(i, H(i,t), t) = H(i,t) - \text{CD}, \qquad \text{CD} = 0 \tag{30}$$

$$CDF_{scd1}(i, H(i,t), t) = H(i, \tau(t) - 1) - e^{\text{CDS}\cdot\tau(t)} + 1 \tag{31}$$

$$CDF_{scd2}(i, H(i,t), t) = H(i, \tau(t) - 1) - e^{\text{CDS}\cdot\tau(t)} \tag{32}$$

$$\text{SCD} = |V| \cdot \log\left(\frac{\text{T}_{obj}}{\text{TMAX}}\right) \tag{33}$$

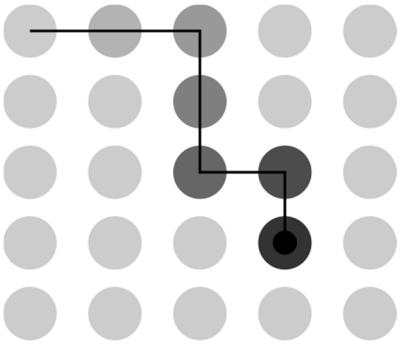

Fig. 3. Vertex properties being updated as the mobile agent moves and follows a path through the map.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIA.2021.3091418, IEEE Transactions on Industry Applications

B. Benchmark Analysis and Limit Testing

The objective of benchmark analysis is limit-testing the proposed algorithm by conducting a parametric study to assess the performance and stability of the solution A generic case study is used based on the facility inspection problem [17].

In this problem, a mobile agent travels between the charging stations, at the top-left and bottom-right corners of the map. The mobile agent must inspect the site, making sure all areas are frequently visited and no areas are ignored. The inspection problem makes is an ideal benchmark case study for limit testing, since it is an extreme case of the task scheduling problem: every vertex of the map is itself a task since the goal is to patrol the full map continuously.

The ideal solution in this case is for the agent to be pathed across the map to avoid any areas of the map being neglected on the long term (i.e., avoid some areas being visiting more than others as much as possible). Three benchmark studies and limit tests were performed.

1. Benchmark Test 1: Time Complexity Analysis

When proposing a new computational algorithm, one of the most important features to establish is its time complexity. The time complexity of the Dijkstra SPF (computing only one traversal of the map) is $\Theta(|V|\cdot\log|V|)$. Using analytical analysis of the implemented code, the time complexity of the proposed algorithm was determined to be $\Theta(|V|\cdot(\log|V|)^3)$.

The benchmark problem is run for a grid size of 10x10, 25x25, 50x50, 75x75, and 100x100. The ZCD function was used (chosen for simplicity, since the choice is irrelevant and doesn't affect the time complexity results since all the CDF functions are $\Theta(1)$). The termination criterion was set as TMAX=100 (100 traversals). For each map size the code is run 10 times, and the average run time is recorded.

The results are plotted in Fig. 4 in comparison to other common time complexities of graph algorithms, expressed in big-Θ notation. From the results, it is indeed confirmed to be $\Theta(|V|\cdot(\log|V|)^3)$.

The small offset for larger values is attributed to approaching physical limits of memory allocation on a laptop PC. Thus, the proposed algorithm is deemed computationally efficient, being marginally slower than Dijkstra's SPF for a single shortest path solution, yet faster than any $\Theta(N^2)$ algorithm, i.e., $\Theta(|V|\cdot\log|V|) < \Theta(|V|\cdot(\log|V|)^3) < \Theta(N^2)$.

2. Benchmark Test 2: INC and CDF Selection

The second benchmark test aims to test the effect of varying the value of INC and the choice of the CDF. To perform a full parametric analysis which considers the grid size as well, the testing is performed and comparatively evaluated on small (10x10), medium (25x25), and large (50x50) maps.

Four each map size, ten different INC values are tested, varying from $0.5\cdot|V|$ to $5\cdot|V|$ with increments of $0.5\cdot|V|$. For each value of INC, the problem is run using each of the four proposed CDF functions (i.e., a total of 3 maps x 10 INC values x 4 CDF runs). For each run, the termination criterion was set as TMAX=100 (100 traversals). As a performance metric the number of times each vertex was visited/inspected by the agent is counted, recalling that the anticipated solution is to have no uninspected parts of the grid.

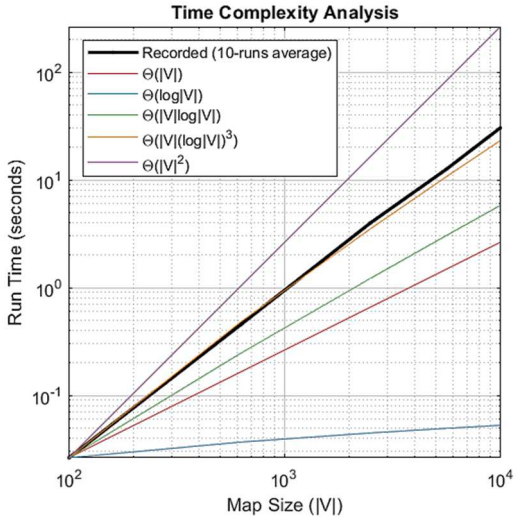

Fig. 4. Results for the first benchmark test: time complexity analysis showing the recorded run time vs. other time complexities in big-Θ notation. The designed algorithm is confirmed to be $\Theta(|V|\cdot(\log|V|)^3)$

In Fig. 5, the number of uninspected nodes at the end of each run (at $\tau = TMAX = 100$) is plotted for all cases. Another performance metric is associated with the frequencies of vertex inspections. The ideal solution is for the number of visits for the maps vertices to be as close to the median value as possible (i.e., no parts are neglected compared to others).

To analyze this, a box plot with summary statistics for each run of the medium map is shown in Fig. 6. In this sense what is desired is to have: 1) no zero values; and 2) minimum inter-quartile range. From the results in Fig.5 and Fig. 6, the following points can be made by observing both performance metrics:

- ZCD is the only CDF that provides a stable operation, being independent of the grid size.
- If a FCD is to be chosen, its value should be set as a function of INC to guarantee improved performance.
- In comparison, SCD1 and SCD2 do not perform as well and are less stable. SCD1 shows more stability than SCD2, but better tuning of the function is necessary.

The results of this third benchmark test strengthen the points made previously. ZCD and FCD both provide stable performance, with the number of traversals required converging to a finite value as the grid size is increased. However, FCD is dependent on the INC value, thus being proportional to the grid size for a stable operation. SCD1 and SCD2 are shown to have stability problems in their current form for large maps.

It can be argued that the SCD function may provide better performance depending on the type of tasking problem involved. While this may be true, the objective of this test is to determine the choice of parameters that guarantee a reliable and stable operation for any type of tasking problem, thus establishing a "benchmark" for the designed algorithm. Nevertheless, the implementation makes it flexible for users to freely tune these parameters to best fit the specific problem.

Therefore, from the benchmark analysis it is possible to show that: 1) the proposed algorithm is $\Theta(|V|\cdot(\log|V|)^3)$ and 2) ZCD is recommended as the "default" option for the CDF, being the most reliable and least dependent on other parameters.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIA.2021.3091418, IEEE Transactions on Industry Applications

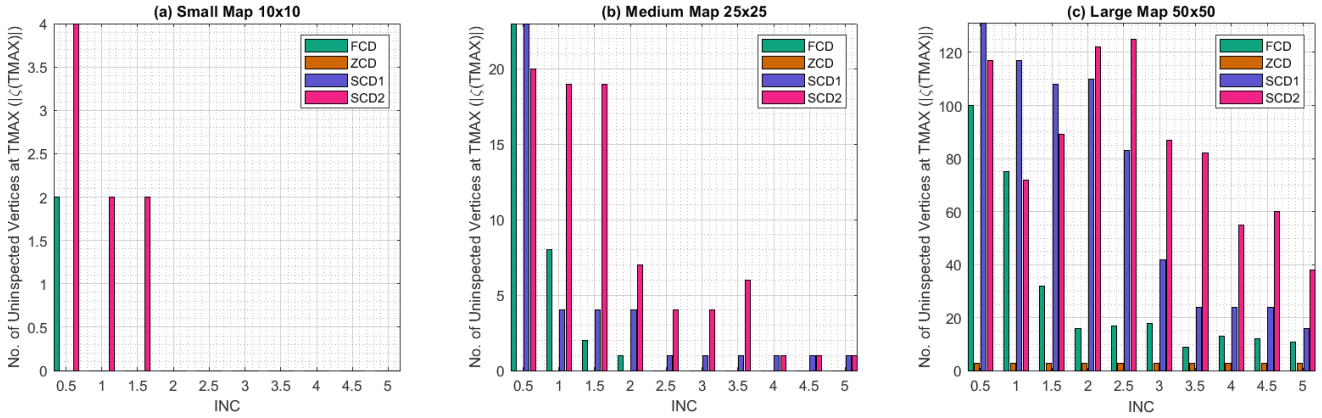

Fig. 5. Results for the second benchmark test: number of uninspected vertices at τ=TMAX=100. Results are shown for the total of number of runs corresponding to: 3 maps x 10 INC values x 4 CDF choices.

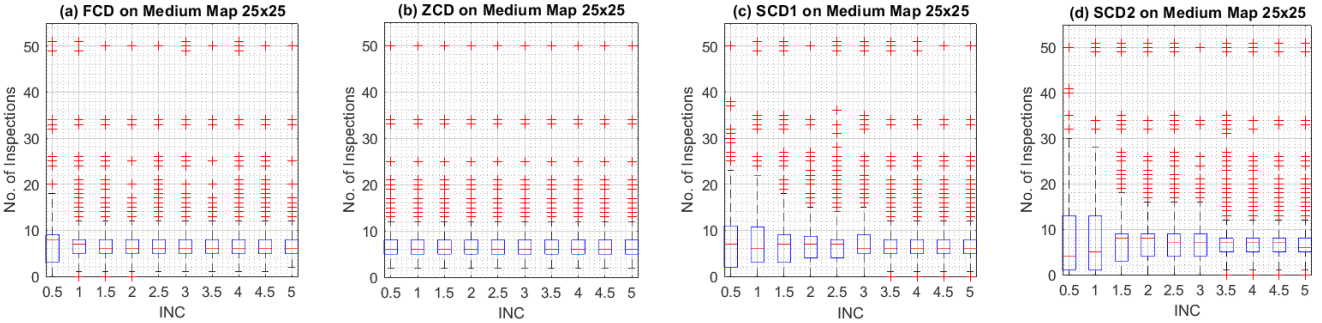

Fig. 6. Results for the second benchmark test: box plot to show summary statistics of the number of vertex inspections at τ=TMAX=100. A box plot for each of the 10 INC values x 4 CDF choices is plotted for the medium 25x25 map. The blue boxes correspond to the 25th to 75th percentile range. The red line is the median value, and the whiskers show the maximum and minimum values. Outliers (>1.5 times inter-quartile range) are shows as red crosses.

3. Benchmark Test 3: Stability and Termination Criteria

By using TMAX as the termination criterion in the previous study, it was observed that the number of required traversals to fully span the map is dependent both on the tunable parameters and the grid size. It is very critical to verify that the number of traversals required to fully span the map does not diverge with the grid size, i.e., it is critical to establish the stability of the algorithm and the CDF functions and INC values.

Therefore, another limit test is performed by letting the simulation run for a very large number of traversals (TMAX = 500) and recording the number of traversals required to inspect the full grid once all nodes have been inspected at least once. The results are listed in Table I and plotted in Fig. 7.

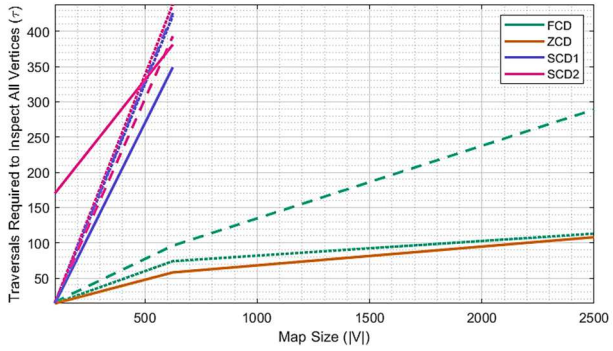

Fig. 7. Results for the third benchmark test: number of traversals required to inspect the full map relative to the map size, for each CDF selection. Dotted, dashed, and solid lines correspond to INC=1|V|, 3|V|, and 5|V|, respectively.

TABLE I RESULTS FOR THE THIRD BENCHMARK TEST: NUMBER OF TRAVERSALS REQUIRED TO INSPECT THE FULL MAP.

CDF	Map Size (|V|)	INC		
		1|V|	3|V|	5|V|
FCD	100	36	16	14
	625	>500	96	74
	2500	>500	289	113
ZCD	100	14	14	14
	625	58	58	58
	2500	108	108	108
SCD1	100	170	16	15
	625	381	393	438
	2500	>500	>500	>500
SCD2	100	14	14	14
	625	349	426	422
	2500	>500	>500	>500

C. Choice of the Shortest Path Method

In the designed and implemented algorithm, for each traversal of a mobile agent through the modeled map, Dijkstra's SPF method is used to determine the path taken of the mobile agent. Since the shortest path is guided by the node values set iteratively according to the designed algorithm (as the map dynamically changes), the obtained path would maximize the tasks being achieved while minimizing the movement cost (and hence, electricity consumption).

Indeed, numerous other shortest path methods exist in graph theory applications, with common well-known alternatives to Dijkstra's SPF being A* Search and Bellman-Ford [22].

All the aforementioned methods achieve the same objective: find the shortest path between two nodes in a weighted graph.

The designed graph-based model and algorithm are versatile such that any shortest path method can be used, and the same results would be achieved, since the shortest path for a given state of the map (node values and corresponding edge weights) would be the same regardless of the method used to find it.

In this case, the choice of the most suitable shortest path method to incorporate in the proposed algorithm depends on the computational burden. To justify the choice of Dijkstra as opposed to other alternatives, a discussion thereof and a comparative analysis is performed in this section.

By reperforming the benchmark analysis considering all three candidates (Dijkstra, A* Search, and Bellman-Ford), Dijkstra was shown to guarantee the best performance in terms of computational complexity (and thereby scalability) for the proposed algorithm. In Fig. 8, A* Search and Bellman-Ford are seen to have a similar performance, being significantly slower than Dijkstra, especially for larger maps.

Another critical point to note is that with the proposed model and algorithm, Bellman-Ford is unable to converge to a solution when all the edge weights are equal (e.g., in the first iteration), and an alternative method must be employed whenever this occurs.

This issue does not occur neither with Dijkstra nor with A* Search, which both robustly find the shortest path in all iterations for all map conditions. Therefore, the use of Bellman-Ford is not recommended, and the two feasible candidates are Dijkstra and A* Search.

Both methods provide the same results for the designed model and algorithm, with Dijkstra being superior in terms of computational time, especially for larger maps (i.e., better scalability). Therefore, Dijkstra's method is shown to guarantee a reliable performance while providing the fastest computational time (which is critical as the proposed algorithm is intended for real-time application).

Nevertheless, the implementation of the proposed model and algorithm makes it flexible for users to use any shortest path method at their convenience. A comparison between the choice of Dijkstra and A* Search in the proposed model and algorithm is revisited in the next section considering a real-world application.

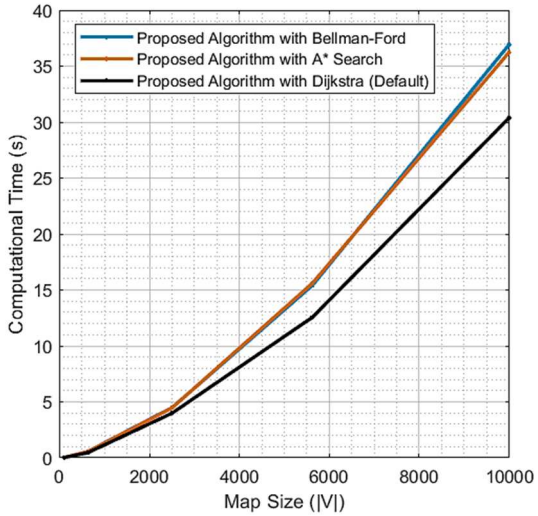

Fig. 8. Comparing the performance of the proposed algorithm while using Dijkstra, A* Search, and Bellman-Ford for different map sizes.

IV. Real-World Application

In this section, a real-world case study is used to demonstrate the applicability of the proposed algorithm to real-life problems. An oil refinery located at coordinates (53.090, 14.254) is considered. Due to their nature, oil refineries require constant safety inspection, particularly with the hazardous nature involving the oil tanks and pipelines on the site. These refineries span very large areas, and so automating the safety inspection process is highly desired.

A. Validation Case Study with Stationary Obstacles

In this case study, one autonomous mobile agent is allocated to perform the security inspection and patrolling the refinery, as shown in Fig. 9 (left). The SMP S5.2 series security robot 2020 model [22] is considered as a commercially available option for an autonomous mobile agent, with its specifications listed in Table II. Thus, the objective is to test the performance of the proposed DOT algorithm in effectively scheduling its fully autonomous operation in the sites' safety inspection. In this first case study, only stationary obstacles are considered.

The physical limitations of the agent's motion must be considered to determine the correct discretization of the lattice graph. The dimension of each grid element Δv must be larger than both the minimum width of the patrol path (S3) and the minimum turning radius (S4). Meanwhile, the grid elements must also be smaller than the minimum object recognition range of the onboard cameras and detection systems (S5). This is expressed in (34).

$$\max(S3, S4) \leq \Delta v \leq S5 \tag{34}$$

With the real site area being 350x350 (m^2), a spacing $\Delta v = 7$m would satisfy (38), thus resulting in a 50x50 grid as shown in Fig. 8 (right). The graph can then be constructed as formulated in Section II. As mentioned, obstacles (in this case being the tanks) are modeled by setting the vertex heat values to 1.7976^{308}. Each time step would correspond to the average traveling time between two vertices at the agent's average autonomous traveling speed (S2), as shown in (35).

$$\Delta t = \Delta v \cdot S2 = 0.035\ h \tag{35}$$

The agent's onboard battery SoC is updated according with each timestep to (36). The charging and discharging values (per timestep) are calculated according to (37) and (38). The minimum allowed SoC is 0.1. Accordingly, the maximum range of a fully charged agent is be obtained in (40).

$$SoC(t) = \begin{cases} SoC(t-1) - SoC_{discharge}, & if\ moving \\ SoC(t-1) + SoC_{charge}, & if\ charging \end{cases} \tag{36}$$

$$SoC_{discharge} = 100 \cdot \frac{\Delta v}{S1}\% = 0.029\% \tag{37}$$

$$SoC_{charge} = 100 \cdot \frac{\Delta t \cdot S9}{S8}\% = 0.7\% \tag{38}$$

$$max\ range = (1 - SoC_{min}) \cdot S1 = 19.2\ km \tag{39}$$

The algorithm is run for this problem with ZCD and $INC = 1$. In order to simulate the real-life case, the termination criteria is set according to the maximum range at full charge, by setting $MAXTIME = 3085$. In this sense, the case study aims to assess the effectiveness of the algorithm in scheduling the agent's inspection paths through the map, making the best use of one full battery charge.

TABLE II SPECIFICATIONS OF THE MOBILE AGENT USED FOR THE CASE STUDY (SMP S5.2 SERIES 2020 MODEL) [22].

S1 - Cruising Range	24 km
S2 - Autonomous Traveling Speed (average)	5 km/h
S3 - Width of patrol route path (minimum)	0.9 m
S4 - Turning radius (minimum)	5 m
S5 - Object Recognition Range (minimum)	50 m
S6 - Operating Time (average)	12 h
S7 - Charging Time (average)	5 h
S8 - On-Board Battery Capacity	3 kWh
S9 - Charger Power	600 W

It has been mentioned that the proposed algorithm is the first of its kind in generically model the industrial task scheduling problem for autonomous agents. While this is a novel contribution, it does add a difficulty since there were no similar existing algorithms to use for comparison at the time of this work, especially in terms of the problem modeling.

Obtaining a deterministic solution is not possible, as it would require a full graph search to be performed being $\Theta(|V|!)$, corresponding to more than $1.6^{7411}\cdot$path determinations, which is infeasible even on high-performance computers.

However, there are multiple graph theory algorithms for path spanning and sampling that can be modified for this purpose. Accordingly, the directed random walk (DRW) algorithm [23], [24] was used with two variations: normal (DRW1), and brute-force (DRW2). Those algorithms are detailed in Appendix 1.

In addition, the case study is used to reaffirm the choice of Dijkstra's SPF as opposed to the other feasible alternative (A* Search) in the proposed model and algorithm. While it is anticipated that both Dijkstra and A* Search would yield the same results and the main advantage of choosing Dijkstra would be in the computational efficiency, this is revalidated by comparing the results of the proposed algorithms using both shortest path methods.

The objective now is to evaluate a) how effectively is the site being inspected by the assigned agent and b) how efficiently is this being done by limiting the operation to one full battery charge. Four evaluation criteria are used:

- Percentage of Site Area Inspected [%].
- Mean number of vertex visits.
- Mean Area per Charge Consumed [m^2/kWh]
- Ratio of Algorithm Running Time to Real Operating Time.

Snapshots of the resulting paths through the site by the DOT algorithm are visualized in Fig. 10, and the performance metrics are compared with those of DRW1 and DRW2 in Fig. 11. The latter are also listed in Table III. The proposed algorithm outperforms the others in all performance metrics.

It is noted that the algorithm run time is calculated per map traversal as a normalized figure. As anticipated, the choice of Dijkstra or A* has no effect on the performance metrics except the computational time, where the choice of Dijkstra outperforms A* Search (by 12.5%).

Note that the DRW algorithms have a random element and the results shown are for optimized runs (best cases). Therefore, the actual real-life performance of DRW is worse than shown here, as opposed to the deterministic solution of the proposed DOT.

The results shown are for an operation limited to one full battery charge to simulate a real-life restriction. Removing this constraint (with recharging or a substitute agent) results in even better performance by DOT compared to DRW. Finally, it is worth noting that the ratio of scheduled real time to the algorithm running time is ~1000, confirming that DOT is deployable for real time scheduling of autonomous agents.

It can be seen in Fig. 10 how the obstacles were provided directly in the map data input file without the need for any conditional statement modifications to the code. Applying a heat value of 1.7976^{308} guaranteed that the obstacle vertices are never selected in computed paths. This provides great versatility, since new obstacles can be introduced or moved in real-time, a feature which is not possible by other graph methods that construct random paths such as DRW. To demonstrate this, a second case study with dynamic (mobile) obstacles is performed.

B. Case Study with Stationary and Mobile Obstacles

In this case study, two dynamic (mobile) obstacles are introduced into the map. In the real-world setting, this would correspond to construction work along the pipelines in the oil refinery, which would in untraversable by the mobile agent during its inspection patrols. This is illustrated in Fig. 12, with dynamic obstacles 1 and 2 set on a path that is eastbound and westbound, respectively. The speed of the obstacles is set to 3m/h, corresponding to a realistic relocation of maintenance workers along the pipelines.

Snapshots of the resulting paths through the site by the DOT algorithm is visualized in Fig. 13, and the performance metrics are shown in Table IV.

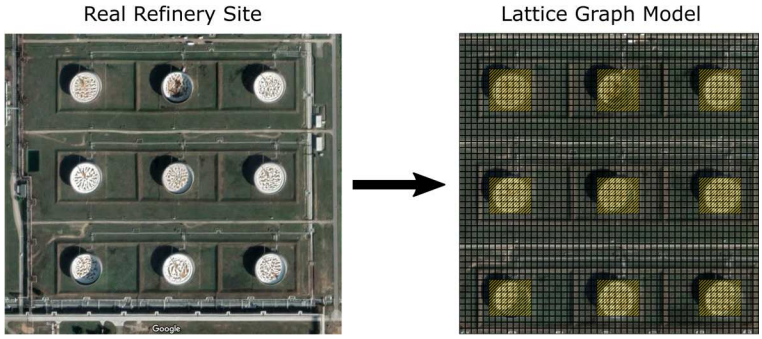

Fig. 9. Satellite image of the oil refinery located at coordinates (53.090, 14.254) used as for the case study (left), and modeling as a lattice graph (right) with the obstacles/non-traversable vertices highlighted in yellow. The real-life area of the site is 350x350m^2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIA.2021.3091418, IEEE Transactions on Industry Applications

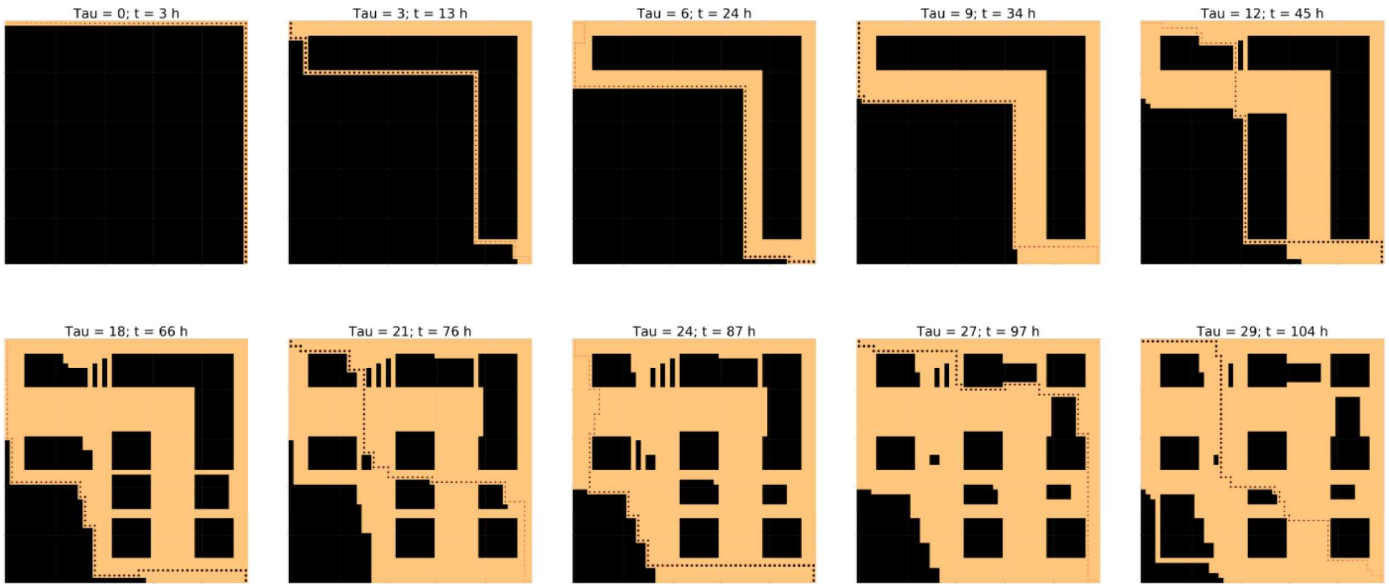

Fig. 10. Snapshots visualizing the resulting paths through the site by the DOT algorithm at $\tau = (0,3,6,9,12,18,21,24,27,28)$ on a single onboard battery charge. Dark and light colored nodes correspond to inspected vs. uninspected vertices, respectively.

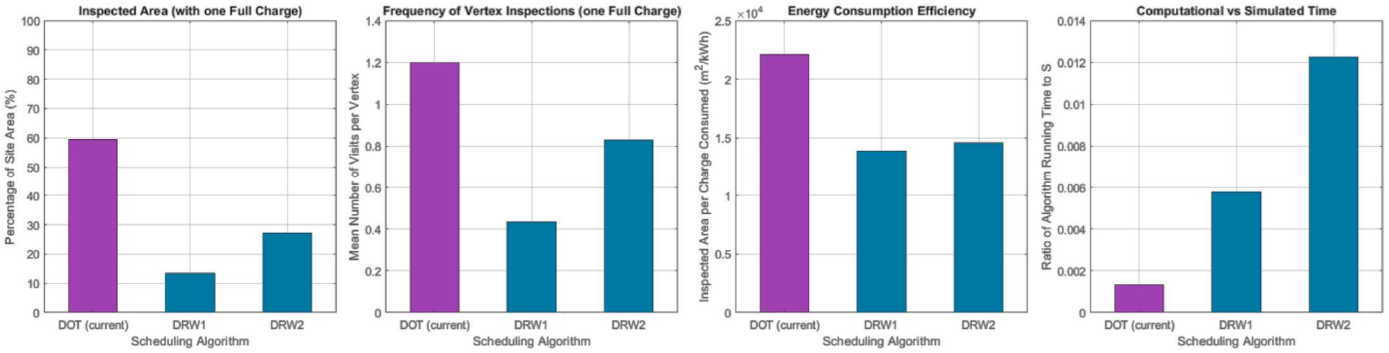

Fig. 11. Comparison between the proposed DOT algorithm vs. DRW1 and DRW2 in terms of the performance metrics: a) Percentage of Site Area Inspected [%], b) Mean number of vertex visits, c) Mean Area per Charge Consumed [m^2/kWh], and d) Ratio of Algorithm Running Time to Real Operating Time.

TABLE III PERFORMANCE METRICS FOR DOT WITH STATIONARY OBSTACLES ONLY.

Performance Metric	Method Used for Task Scheduling			
	Proposed Graph Model and Algorithm		DRW1	DRW2
	With Dijkstra SPF	With A* Search		
Algorithm Running Time (s)	**3.99**	**4.5**	**36.6**	**17.2**
Mean number of vertex visits	1.20	1.20	0.43	0.83
Total area inspected (m^2)	57526	57526	13083	26313
Percentage site area inspected (%)	60%	60%	14%	27%
Area per Charge Consumed (m^2/kWh)	22103	22103	13874	14569

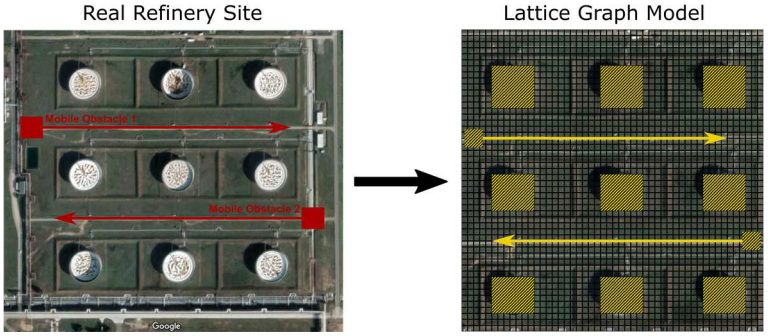

Fig. 12. Satellite image of the oil refinery located at coordinates (53.090, 14.254), including mobile obstacles used as for the second case study (left), and modeling as a lattice graph (right) with the obstacles/non-traversable vertices highlighted in yellow (stationary and mobile). The real-life area of the site is 350x350m^2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIA.2021.3091418, IEEE Transactions on Industry Applications

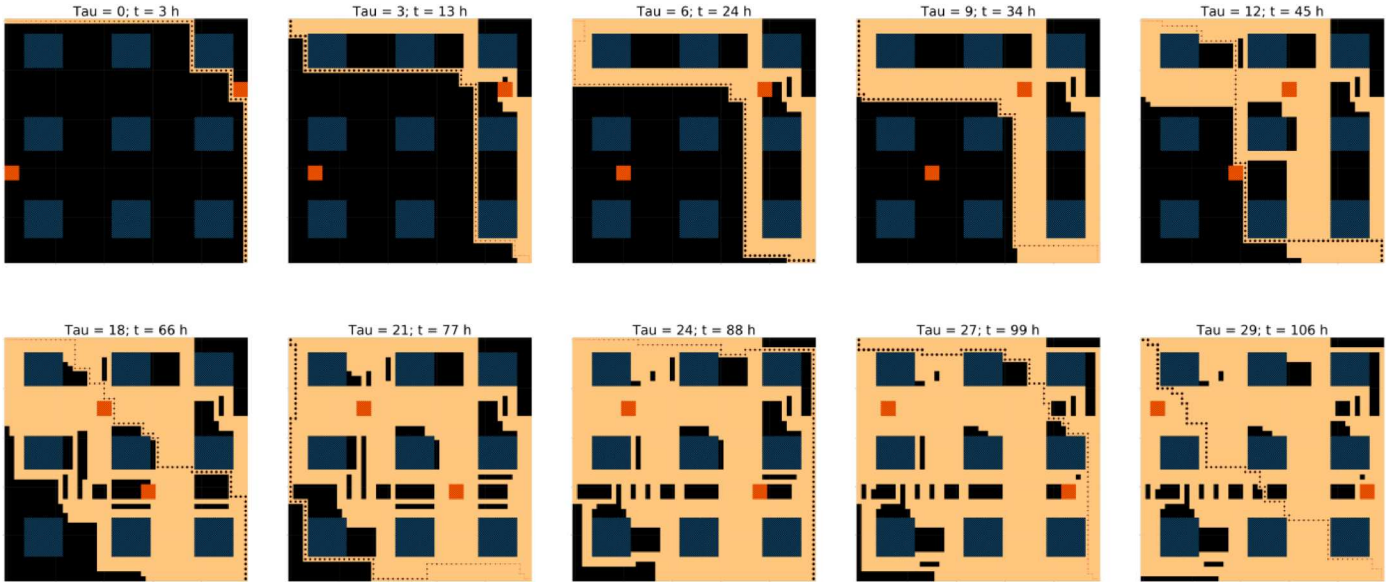

Fig. 13. Snapshots visualizing the resulting paths through the site by the DOT algorithm at $\tau = (0,3,6,9,12,18,21,24,27,28)$ on a single onboard battery charge. Dark- and light-colored nodes correspond to inspected vs. uninspected vertices, respectively. Blue and red squares correspond to stationary and mobile obstacles, respectively. The dotted path corresponds to the agent's current path at the given time.

The presence of dynamic obstacles slightly increases the computational burden (due to the necessity of updating the node heat values every time the obstacle moves). Moreover, the presence of the moving obstacles seems to (very slightly) facilitate the inspection problem, since it forces the mobile agent to cover a wider area to avoid the additional obstacles present.

It is verified that the resulting paths for the agent never intersect with neither the stationary nor the dynamic (moving) obstacles, while successfully maximizing the inspected area for an operation limited to one full battery charge to simulate a real-life restriction.

To reaffirm the statements made in Section III.C regarding the choice of Dijkstra as opposed to other shortest path, the case study with dynamic obstacles is re-simulated using the proposed method, incorporating A* Search instead of Dijkstra.

As anticipated and previously stated (also as the results of the first case study showed), the choice of the shortest path method does not affect the results. Dijkstra is demonstrated again to guarantee the best computational efficiency and algorithm stability.

The objective of this paper was to clearly describe the proposed model and algorithm present the mathematical formulation. A thorough limit testing was performed to recommend the set of parameter settings (i.e., INC and CDF) and the choice of the shortest path method (i.e., Dijkstra), that guarantees reliable and stable execution of the algorithm, in addition to minimal computational burden.

However, the algorithm was designed such that the building blocks can be easily changed by the users (e.g., choice of the shortest path method), without influencing the results). This is in fact a compelling advantage of the proposed graph model and algorithm, being that the obtained solution is independent on the choice of the shortest path function.

In this way, the designed algorithm is versatile and can be easily adapted or modified by users for different industrial tasking problems cases while guaranteeing a reliable and robust performance for real world applications.

TABLE IV PERFORMANCE METRICS FOR DOT (PROPOSED ALGORITHM WITH DIJKSTRA) VS. PROPOSED ALGORITHM CONSIDERING DYNAMIC OBSTACLES.

Performance Metric	Proposed Graph Model and Algorithm	
	With Dijkstra (DOT)	With A*Search
Algorithm Running Time (s)	**4.25**	**4.85**
Mean number of vertex visits	1.21	1.21
Percentage site area inspected (%)	61%	61%
Area per Charge Consumed (m^2/kWh)	22235	22235

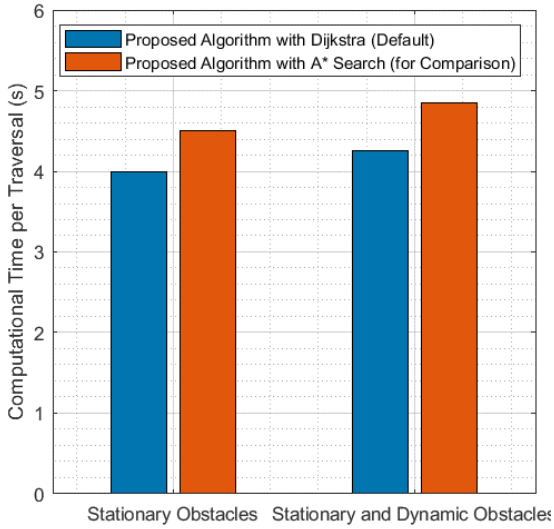

Fig. 14. Comparing the computational performance of the proposed algorithm while using Dijkstra vs. A* Search in the presence of stationary obstacles only (left) and stationary and dynamic obstacles (rights).

V. Conclusions

An innovative graph-based model and algorithm for optimal task scheduling was proposed, implemented and tested. The designed DOT algorithm was designed based on graph theory to guarantee a generic nature, making it applicable on a plethora of tasking problems and not being case-specific. For any industrial setting where mobile agents are responsible for accomplishing tasks across a site, an optimal task schedule for each agent is obtained to maximize the speed of the task achievement with high energy consumption efficiency. The algorithm's versatility in modeling different problems and high computational efficiency make it perfectly suitable for a fully distributed task scheduling of autonomous agents. A real-world case study has demonstrated the effectiveness of the proposed algorithm for an industrial site inspection problem, including the presence of dynamic (moving obstacles). In future work, the algorithm can be applied to other problems in smart industries with dynamic environments where energy consumption efficiency is required.

Acknowledgements

The Authors sincerely thank the Editor and Reviewers from the IEEE-IAS Industrial Automation and Control Committee for their detailed revision and comments which have greatly helped improve the quality of the manuscript.

Appendix 1: Directed Random Walk Algorithm

In a random walk, the next vertex *j* in a path is chosen at random from the neighbors of a vertex *i*. In this study a variation of this is used for comparison with DOT, a directed random walk (DRW) [24], [25]. In a DRW the next vertex is chosen randomly, but the probability of a vertex being chosen is inversely proportional to its distance from the destination. This is illustrated in Fig. A1, where the current vertex *i* has four neighbors 1, 2, 3, and 4, with the distances (Cartesian) to the destination being d1, d2, d3, and d4, respectively. The next vertex in the path is selected using a roulette wheel approach. The aim is to have a random selection while assigning a higher priority to vertices closer to the destination. Therefore, the selection probability is proportional to d', which is the inverse (1/d) of the Cartesian distance. A random variable X is generated such that *X=U(0,sum(d1',d2'd3',d4')*, based on an uniform distribution. As illustrated in Fig A2, the probability of each of the neighbors being selected is proportional to its inverse distance from the destination. With the random element performed, the results reported in this study are based on the 25th percentile (best case) of 1000 runs.

The DRW guarantees to provide a finite path; however, the random element can result in excessively long ones. Therefore, two variations of the DRW were used in this work: normal (DRW1) and an improved brute force one (DRW2). At the beginning of every traversal while stationed at a checkpoint, a path is calculated. Once a path requires the agent to drop below SoC minimum, the pathing is halted. With DRW2, ten trials are attempted at finding a shorter path until the solution is halted, in which case the agent must recharge before proceeding, since no shorter path can be found that can be traversed with the remaining SoC.

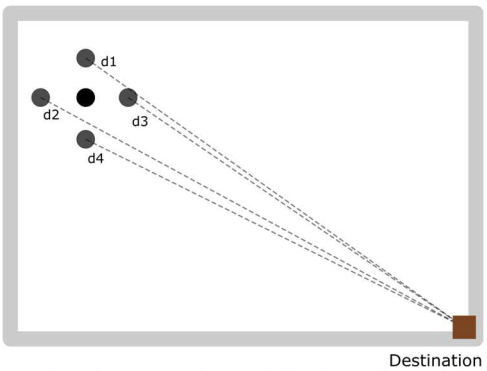

Fig. A1 Illustration of current node *i* , neighboring nodes, and their Cartesian distance to the destination.

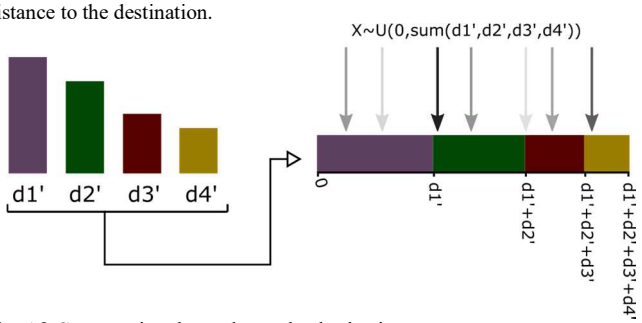

Fig. A2 Constructing the roulette wheel selection.

VI. References

[1] M. Lotfi, C. Monteiro, M. Shafie-Khah, and J. P. S. Catalao, "Evolution of Demand Response: A Historical Analysis of Legislation and Research Trends," in *2018 20th International Middle East Power Systems Conference, MEPCON 2018 - Proceedings*, 2018, pp. 968–973.

[2] J. A. P. Lopes *et al.*, "The future of power systems: Challenges, trends, and upcoming paradigms," *WIREs Energy Environ.*, Dec. 2019.

[3] R. K. Jain, J. Qin, and R. Rajagopal, "Data-driven planning of distributed energy resources amidst socio-technical complexities," *Nat. Energy*, vol. 2, no. 8, pp. 1–11, Aug. 2017.

[4] M. Chowdhury and M. Maier, "Local and nonlocal human-to-robot task allocation in fiber-wireless multi-robot networks," *IEEE Syst. J.*, vol. 12, no. 3, pp. 2250–2260, Sep. 2018.

[5] V. Ortenzi *et al.*, "Robotic manipulation and the role of the task in the metric of success," *Nat. Mach. Intell.*, vol. 1, no. 8, pp. 340–346, Aug. 2019.

[6] A. Ovalle, A. Hably, S. Bacha, G. Ramos, and J. M. Hossain, "Escort Evolutionary Game Dynamics Approach for Integral Load Management of Electric Vehicle Fleets," *IEEE Trans. Ind. Electron.*, vol. 64, no. 2, pp. 1358–1369, Feb. 2017.

[7] X. Hu, C. Zou, X. Tang, T. Liu, and L. Hu, "Cost-optimal energy management of hybrid electric vehicles using fuel cell/battery health-aware predictive control," *IEEE Trans. Power Electron.*, vol. 35, no. 1, pp. 1–1, 2019.

[8] X. Hou, J. Wang, T. Huang, T. Wang, and P. Wang, "Smart Home Energy Management Optimization Method Considering Energy Storage and Electric Vehicle," *IEEE Access*, vol. 7, pp. 144010–144020, 2019.

[9] H. M. D. Espassandim, M. Lotfi, G. J. Osorio, M. Shafie-Khah, O. M. Shehata, and J. P. S. Catalao, "Optimal operation of electric vehicle parking lots with rooftop photovoltaics," in *2019 IEEE International Conference on Vehicular Electronics and Safety, ICVES 2019*, 2019.

[10] M. Bhaskar Naik, P. Kumar, and S. Majhi, "Smart public transportation network expansion and its interaction with the grid," *Int. J. Electr. Power Energy Syst.*, vol. 105, no. December 2017, pp. 365–380, 2019.

[11] M. Rogge, E. Van Der Hurk, A. Larsen, and D. U. Sauer, "Electric bus fleet size and mix problem with optimization of charging infrastructure," *Appl. Energy*, vol. 211, pp. 282–295, 2018.

[12] E. Yao, T. Liu, T. Lu, and Y. Yang, "Optimization of electric vehicle scheduling with multiple vehicle types in public transport," *Sustain. Cities Soc.*, vol. 52, no. August 2019, p. 101862, 2020.

[13] S. Wang, J. Wan, D. Zhang, D. Li, and C. Zhang, "Towards smart factory for industry 4.0: A self-organized multi-agent system with big data based feedback and coordination," *Comput. Networks*, vol. 101, pp. 158–168, Jun. 2016.

[14] X. Gong, Y. Liu, N. Lohse, T. De Pessemier, L. Martens, and W. Joseph, "Energy- and Labor-Aware Production Scheduling for Industrial Demand Response Using Adaptive Multiobjective Memetic Algorithm," *IEEE Trans. Ind. Informatics*, vol. 15, no. 2, pp. 942–953, Feb. 2019.

[15] B. Beirigo, F. Schulte, and R. Negenborn, "Dual-Mode Vehicle Routing in Mixed Autonomous and Non-Autonomous Zone Networks," in *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, 2018, vol. 2018-November, pp. 1325–1330.

[16] A. Khamis, A. Hussein, and A. Elmogy, "Multi-robot task allocation: A review of the state-of-the-art," *Stud. Comput. Intell.*, vol. 604, pp. 31–51, 2015.

[17] C. Liu, *Multi-Robot Task Allocation for Inspection Problems with Cooperative Tasks Using Hybrid Genetic Algorithms*. .

[18] C. Sarkar, H. S. Paul, and A. Pal, "A Scalable Multi-Robot Task Allocation Algorithm," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2018, pp. 5022–5027.

[19] M. Hermann, T. Pentek, and B. Otto, "Design principles for industrie 4.0 scenarios," *Proc. Annu. Hawaii Int. Conf. Syst. Sci.*, vol. 2016-March, pp. 3928–3937, 2016.

[20] M. Lotfi, A. Ashraf, M. Zahran, G. Samih, M. Javadi, G.J. Osório, J.P.S. Catalão, "A Dijkstra-Inspired Algorithm for Optimized Real-Time Tasking with Minimal Energy Consumption," in *Proceedings - 2020 IEEE International Conference on Environment and Electrical Engineering and 2020 IEEE Industrial and Commercial Power Systems Europe, EEEIC / I and CPS Europe 2020*, 2020.

[21] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[22] D.K. Smith, "Shortest Paths," *Networks and Graphs*, pp. 27–45, 2003.

[23] SMP Robotics, "S5.2 series 2020 models datasheet," 2020. [Online]. Available: https://smprobotics.com/wp-content/uploads/2019/09/security_robot_s5.2_is_prompt_2020.pdf.

[24] S. Y. Huang, X. W. Zou, and Z. Z. Jin, "Directed random walks in continuous space," *Phys. Rev. E - Stat. Physics, Plasmas, Fluids, Relat. Interdiscip. Top.*, vol. 65, no. 5, p. 4, May 2002.

[25] B. Ribeiro, P. Wang, F. Murai, and D. Towsley, "Sampling directed graphs with random walks," in *Proceedings - IEEE INFOCOM*, 2012, pp. 1692–1700.