Developing a Deterministic Patrolling Strategy for Security Agents

Nicola Basilico, Nicola Gatti, and Francesco Amigoni
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
Email: {basilico, ngatti, amigoni}@elet.polimi.it

Abstract—Developing autonomous systems that patrol environments for detecting intruders is a topic of increasing relevance in security applications. An important aspect of these systems is the patrolling strategy; namely, the determination of where to move in order to conveniently detect intrusions. While a large part of patrolling strategies proposed so far adopt some kind of random movements, deterministic strategies can be useful in some situations of interest. In this paper, we propose an approach to find a deterministic strategy that allows the patrolling agent to always detect an intruder that attempts to enter an environment. The problem is formulated as the determination of a cyclic path that visits, under temporal constraints, all the vertexes of a graph representing the environment. We propose a solving algorithm, study its properties, and experimentally evaluate it.

Index Terms—Robotic patrolling with adversaries, traveling salesman problem, constraint satisfaction problem

I. INTRODUCTION

Protecting sites against intrusions is a topic of increasing social importance, and autonomous mobile patrolling systems have been developed in the last years [1], [2]. Under the assumption that their sensors are limited and cannot perceive all the environment at a time, these systems usually adopt some kind of randomized movements in order to maximize the probability of detecting an intruder and to make their routes unpredictable for an observing intruder [3]. However, there are situations in which a deterministic strategy can be found that makes entering the environment inconvenient for an intruder.

In this paper, we propose an approach for determining a deterministic strategy for a mobile patrolling agent. We consider an environment modeled as a directed graph, where vertexes are locations that can be intruded and arcs connect locations that can be reached directly by the patroller. Each vertex is characterized by a value, called relative deadline, that denotes the amount of time required by an intruder to enter the corresponding location or, equivalently, the maximum permitted time interval between two successive visits of the patrolling agent to that vertex. Arcs are characterized by weights that represent the amount of time the patroller spends to traverse them. We assume that time is discrete and that, when the intruder decides to attack a vertex (location), it stays in that vertex for a time equal to the corresponding relative deadline. The patroller captures the intruder when it visits a vertex where the intruder is staying from a time shorter than the relative deadline. In doing this, we assume that the patroller can sense the presence of the intruder only in its current vertex. Hence, the patroller can secure a vertex by periodically visiting it within its relative deadline, while an intruder can break in if the vertex is left unvisited for sufficient time.

Starting from this last consideration, we formulate the problem of finding a deterministic strategy for a patrolling agent in the considered setting as the problem of cyclically visiting vertexes of the graph under temporal constraints. Specifically, the problem we study is the determination of a finite sequence of vertexes, starting and ending with the same vertex and including all the vertexes, such that no relative deadline is violated when the sequence is indefinitely repeated. Note that, in the setting considered in the paper, if such a deterministic strategy exists, the patroller following it will prevent any intrusion, by construction. Indeed, if no relative deadline is violated, the intruder cannot enter any vertex without being captured by the patroller. We formulate the problem of cyclically visiting graph vertexes with relative deadlines as a constraint satisfaction problem (Section III), we provide a sound and complete solving algorithm and discuss some of its properties (Section IV), and we experimentally evaluate our approach (Section V).

II. PATROLLING STRATEGIES FOR A SECURITY AGENT

In this work, we consider the following patrolling context. There is a patrolling agent that has to prevent intrusions in some locations, which could be access points along a perimeter [4] or areas of interest [5]. The patrolling agent can be a mobile robot [1], [2]. This kind of scenarios has been studied by several works in literature, mainly devoted to find good strategies for the patrolling agent. A patrolling strategy determines the movements of the patrolling agent in the environment. Some patrolling strategies do not consider any explicit model of the adversary [1], [2], [3], while some others do [4], [5], [6], [7], [8]. As we have shown in [9], patrolling strategies that consider models of adversaries can give the patrolling agent a larger expected utility than strategies that do not. Usually, these patrolling strategies are non-deterministic, employing random movements to make them unpredictable for an observing intruder [4], [5], [6], [7], [9], [10]. However, sometimes patrolling agents can employ deterministic strategies to “force” the intruders to never attempt to enter.
Deterministic patrolling strategies can be adopted when the scenario presents some particular characteristics (for example, related to the speed of the patroller and time needed by a possible intruder to attack to a location) and when the intruder’s knowledge of the patroller is very limited (as shown in [11]). Deterministic patrolling strategies can guarantee to visit a given vertex within an exact time bound, while non-deterministic patrolling strategies can leave a vertex unvisited for long time. Moreover, a deterministic strategy that makes inconvenient for the intruder to enter (namely, that guarantees that the intruder will be always captured if it attempts to enter a location) is better than any non-deterministic strategy, which can only provide a probability for capturing the intruder. Hence, deterministic strategies, when they can be found, can be preferred to non-deterministic strategies, which, on the other hand, can be always found. Notwithstanding their important role in some cases, development of deterministic strategies for patrolling agents has not received much attention, except for some results in very special ring-like environments [11]. In this paper, we aim at contributing to fill this gap by proposing an approach to determine a deterministic strategy for a patrolling agent operating in an arbitrary environment.

Before presenting our contributions, we note that similar strategic problems have been addressed in the pursuit-evasion field (e.g., [12], [13]). However, some assumptions, including the fact that the evader’s goal is only to avoid capture and not enter a location, make the pursuit-evasion strategies not directly applicable to our patrolling scenario.

III. PROBLEM STATEMENT AND RELATED WORKS

A. Problem Statement

The problem we deal with is described by a weighted directed graph with relative deadlines, represented as a tuple $G = (V, A, w, d)$. $V = \{v_1, \ldots, v_n\}$ is a set of vertexes (locations) and $A$ is a set of arcs connecting pairs of different locations. We assume that the graph is connected, i.e., for every pair of vertexes $(v_i, v_j)$ there exists at least one sequence of arcs that allows one to reach $v_j$ starting from $v_i$. Function $w : A \rightarrow \mathbb{R}$ assigns each arc $(v_i, v_j)$ a weight $w(v_i, v_j)$, denoting the time (or temporal cost) needed for the patroller to move from $v_i$ to $v_j$. Function $d : V \rightarrow \mathbb{R}$ assigns each vertex $v_i$ a relative deadline $d(v_i)$, denoting the amount of time required by an intruder to enter $v_i$ or, equivalently, the largest permitted temporal interval between two successive patroller’s visits of $v_i$. We report in Fig. 1 a (self-explanatory) example of such a graph.

We denote a finite sequence of visits to vertexes of $G$ as a function $S : \{1, 2, \ldots, s\} \rightarrow V$, where $S(j)$ is the $j$-th element of the sequence. The length of the sequence is $s$. The temporal length of a sequence of visits is the sum of the weights of covered arcs, i.e., $\sum_{j=1}^{s-1} w(S(j), S(j+1))$. The time interval between two visits of two vertexes is calculated accordingly as the sum of the weights of the arcs covered between the two visits. Note that temporal costs are associated only to movements of the patroller between vertexes while the time for visiting a vertex is assumed negligible. (This does not limit the generality of the model, since weights can be set to account for vertexes’ visit times.)

A solution of our problem is a sequence $S$ such that the following properties are satisfied: (1) $S$ is cyclic, i.e., the first vertex coincides with the last one, namely, $S(1) = S(s)$; (2) every vertex in $V$ is visited at least once, i.e., there are no uncovered vertexes; and (3) when indefinitely repeating the cycle, for any $v_i \in V$, the time interval between two successive visits of $v_i$ is never larger than $d(v_i)$. It is clear that, when the agent follows repeatedly such a sequence $S$ as its patrolling strategy, no intrusion can occur.

Let us call $O_i(j)$ the position in $S$ of the $j$-th occurrence of $v_i$ and by $o_i$ the total number of $v_i$’s occurrences in $S$. For instance, consider Fig. 1: given $S = \{01, 02, 05, 02, 01\}$, $O_{02}(1) = 2$ and $O_{02}(2) = 4$, while $O_{02} = 2$ and $O_{03} = 0$. Note that, given a sequence $S$, quantities $O_i(j)$ and $o_i$ can be easily calculated. A matrix $T(n \times n)$ describes the direct connection between vertexes in the graph: $T(v_i, v_j) = 1$ if arc $(v_i, v_j) \in A$ and $T(v_i, v_j) = 0$ otherwise. Given such definitions we can formally state the problem in a mathematical programming fashion. We aim at finding a sequence $S$ of $s$ visits such that the following constraints are satisfied:

\[
S(1) = S(s) \quad \forall i \in \{1, 2, \ldots, n\} \tag{1}
\]
\[
o_i \geq 1 \quad \forall j \in \{2, 3, \ldots, s\} \tag{2}
\]
\[
O_i(k + 1) - 1 \sum_{j=O_i(k)}^{O_i(k+1)-1} w(S(j), S(j+1)) \leq d(v_i) \quad \forall k \in \{1, 2, \ldots, o_i - 1\} \tag{3}
\]
\[
O_i(1) \sum_{j=1}^{s-1} w(S(j), S(j+1)) + \sum_{j=O_i(1)}^{O_i(s)} w(S(j), S(j+1)) \leq d(v_i) \quad \forall i \in \{1, 2, \ldots, n\} \tag{4}
\]

Constraint (1) states that $S$ is a cycle, i.e., the first and last
vertexes of \( S \) coincide; constraints (2) state that every vertex is visited at least once in \( S \); constraints (3) state that for every pair of consecutively visited vertexes, say \( S(j-1) \) and \( S(j) \), arc \((S(j-1), S(j))\) is in \( A \), i.e., vertex \( S(j) \) can be directly reached from vertex \( S(j-1) \); constraints (4) state that, for every vertex \( v_i \), the temporal interval between two successive visits of \( v_i \) in \( S \) is not larger than \( d(v_i) \); similarly, constraints (5) state that for every vertex \( v_i \) the temporal interval between the last and first visits of \( v_i \) is not larger than \( d(v_i) \), i.e., the deadline of \( v_i \) is satisfied along the cycle closure. Hence, our goal is to find a sequence of vertexes \( S(1), S(2), \ldots, S(s) \) such that the above constraints are satisfied. Note that also the length \( s \) of the sequence must be found as part of the solution.

### B. Related Works

The problem of visiting vertexes of a graph under temporal constraints has been vastly studied in related literature. However, to the best of our knowledge, no previous work can be directly applied to our specific problem. For example, in the deadline-TSP (Travel Salesman Problem) [14], vertexes have deadlines after the first visit. Rewards are collected when a vertex is visited before the deadline. The goal is to find a tour that maximizes the reward. A more general variant is the vehicle routing problem with time windows [15] where deadlines are replaced with time windows in which visits of vertexes must occur. Cyclical visits are addressed in the period routing problem [16], where vehicle routes are constructed to run for a finite period of time in which every vertex has to be visited according to a given frequency. Frequencies can be given also as lower bounds, considering the real frequencies of visits as decision variables of the problem [17]. In the cyclic inventory routing problem [18] vertexes represent customers with a given demand rate and storage capacity. The objective is to find a tour such that a distributor can repeatedly restock customers under some constraints on visiting frequencies.

The above problems differ from ours in the following two issues. The first one is that our problem is defined according to relative deadlines (calculated wrt two consecutive visits to the same vertex) and the absolute deadlines (calculated wrt the beginning of the sequence) depend on the solution itself, as expressed by constraints (4)-(5). The extension of the above works, mostly considering absolute deadlines, to our problem introduces highly non-linear constraints and does not seem straightforward. The second issue is that in our problem we aim at finding a solution, and not the optimal solution according to some metric. So, we are solving a feasibility problem and not an optimization problem.

### IV. THE SOLVING ALGORITHM AND ITS PROPERTIES

#### A. The Algorithm

The problem described in the previous section can be formulated as a Constraint Satisfaction Problem (CSP) [19]. Each \( S(j) \) is considered as a variable with domain \( F_j \subseteq V \). The constraints over the values of the variables are (1)-(5). A solution is an assignment of values to all variables such that all the constraints are satisfied. This problem presents some peculiarities wrt works on CSP-based scheduling. Although a large literature in AI studies scheduling problems, cyclic scheduling presents only some partial results (e.g., [20]). Differently from the problems studied in the literature, in our problem, the number \( s \) of variables \( S(j) \) to which values must be assigned is not known in advance, but it is part of the solution to be found. Indeed, we have not any prior information on the length \( s \) of \( S \) (even if, as we shall show in the following, an upper bound over its temporal length can be derived). This is because a vertex can appear more times in \( S \). As a consequence, we will need a criterion to stop the search preserving the completeness of the solving algorithm.

The algorithm we propose for finding a solution basically searches the space of possible sequences with backtracking. To improve efficiency, forward checking [19] is used in the attempt to reduce the branching of the search tree. We report our algorithm in Algorithms 1, 2, and 3.

Algorithm 1 simply assigns \( S(1) \) a vertex \( v_i \). Note that if a solution exists, it can be found independently of the first vertex appearing in \( S \). Indeed, the solution \( S \) being a cycle comprising all vertexes, every vertex can be chosen as the initial one. Hence, the choice of \( v_i \) in Algorithm 1 does not affect the possibility of finding a solution.

Algorithm 2 assigns \( S(j) \) a vertex from domain \( F_j \subseteq V \), which contains available values for \( S(j) \) that are returned by the forward checking algorithm (Algorithm 3). If \( F_j \) is empty or no vertex in \( F_j \) can be successfully assigned to \( S(j) \), then Algorithm 2 returns failure and a backtracking is performed.

Algorithm 3 restricts \( F_j \) to the vertexes \( v_i \) directly reachable from the last assigned vertex \( S(j-1) \) such that their visits do not violate constraints (4)-(5). Note that checking constraints (4)-(5) requires knowledge about the weights (temporal costs) related to the arcs between the vertexes that will be assigned subsequently, i.e., between the variables \( S(k) \) with \( k > j \). For example, consider the graph of Fig. 1 and suppose that the partial solution currently constructed by the algorithm is \( S(1) = v_1 \) (namely, \( j = 2 \)). In this situation, we cannot check the validity of constraints (4)-(5) since we have no information about times to cover the arcs between the vertexes that will complete the solution. In order to cope with this, we estimate the unknown temporal costs by employing an admissible heuristic (i.e., a non-strict underestimate) based on the minimum cost between two vertexes. The heuristic being admissible, no feasible solution is excluded. We denote the heuristic value by \( \overline{w} \), e.g., \( \overline{w}(v_i, S(1)) \) denotes the weight of the minimum path between \( v_i \) and \( S(1) \). We assume \( \overline{w}(v_i, v_i) = 0 \) for any \( v_i \).

Given a partial solution \( S \) from 1 to \( j - 1 \), the forward checking algorithm considers all the vertexes directly reachable from \( S(j-1) \) and keeps those that do not violate the relaxed constraints (4)-(5) computed with heuristic values. It considers a vertex \( v_i \) directly reachable from \( S(j-1) \) and assumes that \( S(j) = v_i \). Step 5 of Algorithm 3 checks relaxed constraints (5) with respect to \( v_i \), assuming that the weight along the cycle closure from \( S(j) = v_i \) to
$S(1)$ is minimum. In the above example, with $S(1) = 01$, the vertexes directly reachable from $S(1)$ are 02 and 05. The algorithm considers $S(2) = 02$. By Step 5, we have $w(S(1), 02) + \overrightarrow{m}(02, S(1)) = 4 \leq d(02) = 18$ and then Step 5 is satisfied. It can be easily observed that such condition holds also at the next iteration of the cycle, when $S(2) = 05$. Step 8 of Algorithm 3 checks relaxed constraints (5) with respect to all the vertexes $v_k \neq v_i$, assuming that both the weight to reach $v_k$ from $S(j) = v_i$ and the weight along the cycle closure from $v_k$ to $S(1)$ are minimum. Consider again the above example. It can be easily observed that when $S(2) = 02$ such conditions hold for all $v_k$. Instead, at the next iteration of the cycle, when $S(2) = 05$ and $v_k = 03$ we have $w(S(1), 05) + \overrightarrow{m}(05, 03) + \overrightarrow{m}(03, S(1)) = 16 > d(03) = 14$. The relaxed constraint is violated and vertex 05 will be not inserted in $F_j$. Similarly, Step 6 checks relaxed constraints (4) with respect to $v_i$, and Step 9 checks relaxed constraints (4) with respect to any $v_k$ assuming that the weight to reach $v_k$ from $S(j) = v_i$ is minimum. In the above example, the relaxed constraints are satisfied only when $v_i = 02$ and therefore $F_j = \{02\}$. Finally, we notice that Steps 5 and 8 are checked only when $o_i = 0$ and $o_k = 0$, respectively, since it can be easily proved that when $o_i > 0$ and $o_k > 0$ these conditions always hold.

**Algorithm 1: FIND_SOLUTION(V, T, w, d)**

1. select a vertex $v_i$ in $V$
2. assign $S(1) \leftarrow v_i$
3. call RECURSIVE_CALL(V, T, w, d, S, 2)

**Algorithm 2: RECURSIVE_CALL(V, T, w, d, S, j)**

1. if $S(1) = S(j - 1)$ and condition (2) holds then
   2. if conditions (5) hold then
      3. return $S$
   4. else
      5. return FAILURE
8. else
7. assign $F_j \leftarrow$ FORWARD_CHECKING(V, T, w, d, S, j)
8. for all the $v_i$ in $F_j$ do
9. assign $S(j) \leftarrow v_i$
10. assign $S' \leftarrow$ RECURSIVE_CALL(V, T, w, d, S, j + 1)
11. if $S'$ is not FAILURE then
12. return $S'$
13. return FAILURE

**Algorithm 3: FORWARD_CHECKING(V, T, w, d, S, j)**

1. assign $F_j \leftarrow \emptyset$
2. assign $s \leftarrow j - 1$
3. for all members $v_i$ in $V$ such that $T(S(s), v_i) = 1$ do
4.   if conditions
5.     $(o_i = 0 \land \sum_{l=1}^{j-1} w(S(l), S(l + 1)) + w(S(s), v_i) + \overrightarrow{m}(v_i, S(1)) \leq d(v_i))$ or
6.     $(o_i > 0 \land \sum_{l=0}^{s} w(S(l), S(l + 1)) + w(S(s), v_i) \leq d(v_i))$
7.     for all $v_k \neq v_i$,
8.     $(o_k = 0 \land \sum_{l=1}^{s} w(S(l), S(l + 1)) + w(S(s), v_i) + \overrightarrow{m}(v_i, v_k) + \overrightarrow{m}(v_k, S(1)) \leq d(v_k))$
9.     $(o_k > 0 \land \sum_{l=0}^{s} w(S(l), S(l + 1)) + w(S(s), v_i) + \overrightarrow{m}(v_i, v_k) \leq d(v_k))$
10. hold then
11.     add $v_i$ to $F_j$
12. return $F_j$

B. An Example

We apply our algorithm to the example of Fig. 1. We use the lexicographic order in Step 1 of Algorithm 1 (to choose the first visited vertex of the sequence) and in Step 7 of Algorithm 2 (to choose the elements of $F_j$ as part of the current candidate solution). We report part of the execution trace (Fig. 2 depicts the complete search tree):

1) the algorithm assigns $S(1) = 01$;
2) the domain $F_2$ (depicted between curly brackets beside vertex 01) is produced as follows (recall the discussion of the previous section):
   - vertex 02 is added to $F_2$, since all the conditions in Algorithm 3 with $v_i = 02$ are satisfied;
   - vertex 05 is not added to $F_2$, since the condition in Step 8 of Algorithm 3 with $v_k = 03$ is not satisfied, formally, $w(01, 05) + \overrightarrow{m}(05, 03) + \overrightarrow{m}(03, 01) > d(03)$;
   - no other vertex is added to $F_2$, since no other vertex is directly reachable from 01;
3) the algorithm assigns $S(2) = 02$;
4) the domain $F_3$ is produced similarly as above, yielding to $F_3 = \{03\}$;
5) the algorithm assigns $S(3) = 03$ and continues.

Some issues are worth noting. First, in the 9th node of the search tree, a sequence $S$ with $S(1) = S(s)$ and including all the vertexes was found. However, this sequence did not satisfy constraints (5). Second, in the 5th node, no possible successor is allowed by the forward checking, and therefore the algorithm backtracks. Third, if the search was not stopped and backtracked at the the 9th node (in Step 5 of Algorithm 2), the algorithm would never terminate. Indeed, the subtrees that would follow this vertex would be the infinite repetition of part of the already built tree and, in particular, of the solution in bold of Fig. 2.
Some Properties of the Algorithm

In this section, we present some properties of the proposed algorithm.

**Theorem 4.1:** The above algorithm is sound and complete.

**Proof.** We initially prove the soundness of the algorithm. We need to prove that all the solutions it produces satisfy constraints (1)-(5). Constraints (1), (2), and (5) are satisfied by Algorithm 2. If at least one of them does not hold, no solution is produced. The satisfaction of constraints (3) is assured by Algorithm 3 in Step 3, while the satisfaction of constraints (4) is assured by Algorithm 3 in Step 9. The above algorithm is sound and complete.

The second point is the stopping criterion in Algorithm 2: when all the vertices occur in S (at least once) and the first and the last vertex in S are equal, no further successor is considered and the search is stopped. If S satisfies all the constraints, then S is a solution, otherwise backtracking is performed. We show that, if a solution can be found without stopping the search at this point, then a solution can be found also by stopping the search and backtracking (the vice versa does not hold). This issue is of paramount importance since it assures that the algorithm terminates (as we remarked in the example of the previous section, without this stopping criterion the search could not terminate). Consider a S such that S(1) = S(s) and including all the vertexes in V. The search subtree following S(s) and produced by the proposed algorithm is (non-strictly) contained in the search tree following from S(1). This is because the constraints considered by the forward checking from S(s) on are (non-strictly) harder than the corresponding ones from S(1). The increased hardness is due to the activation of constraints (4) that are needed given that at least one occurrence of each vertex is in S. Thus, if a solution can be found by searching from S(s), then a shorter solution can be found by stopping the search at S(s) and backtracking. This concludes the proof of completeness.

We give an upper bound to the temporal length of S.

**Theorem 4.2:** If the problem defined in Section III-A admits a solution, then there exists at least a solution S with temporal length no longer than max_{i} \{d(v_{i})\}.

**Proof.** In order to prove the theorem it is sufficient to prove that, if a problem is solvable, then there exists a solution S' wherein there is at least a vertex that only appears once, excluding S'(s). Indeed, if this statement holds then the maximum temporal length of S' is d(v_{i}) where v_{i} is the vertex that appears only one time in S. It easily follows that in the worst case the maximum temporal length of S' is max_{i} \{d(v_{i})\}. Figure 3 shows a situation in which the temporal length of the unique solution is exactly max_{i} \{d(v_{i})\} = 4.

In order to prove completeness we need to show that the algorithm produces a solution whenever at least one exists. In the algorithm there are only two points in which a candidate solution is discarded.

The first one is the forward checking in Algorithm 3. Indeed, it iteratively applies constraints (4)-(5) to a partial sequence S exploiting a heuristic over the future weights (i.e., the time spent to visit the successive vertexes). The employed heuristic being admissible, no feasible solution can be discarded.

The second point is the stopping criterion in Algorithm 2: when all the vertices occur in S (at least once) and the first and the last vertex in S are equal, no further successor is considered and the search is stopped. If S satisfies all the constraints, then S is a solution, otherwise backtracking is performed. We show that, if a solution can be found without stopping the search at this point, then a solution can be found also by stopping the search and backtracking (the vice versa does not hold). This issue is of paramount importance since it assures that the algorithm terminates (as we remarked in the example of the previous section, without this stopping criterion the search could not terminate). Consider a S such that S(1) = S(s) and including all the vertexes in V. The search subtree following S(s) and produced by the proposed algorithm is (non-strictly) contained in the search tree following from S(1). This is because the constraints considered by the forward checking from S(s) on are (non-strictly) harder than the corresponding ones from S(1). The increased hardness is due to the activation of constraints (4) that are needed given that at least one occurrence of each vertex is in S. Thus, if a solution can be found by searching from S(s), then a shorter solution can be found by stopping the search at S(s) and backtracking. This concludes the proof of completeness.

We give an upper bound to the temporal length of S. Theorem 4.2: If the problem defined in Section III-A admits a solution, then there exists at least a solution S with temporal length no longer than max_{i} \{d(v_{i})\}.

**Proof.** In order to prove the theorem it is sufficient to prove that, if a problem is solvable, then there exists a solution S' wherein there is at least a vertex that only appears once, excluding S'(s). Indeed, if this statement holds then the maximum temporal length of S' is d(v_{i}) where v_{i} is the vertex that appears only one time in S. It easily follows that in the worst case the maximum temporal length of S' is max_{i} \{d(v_{i})\}. Figure 3 shows a situation in which the temporal length of the unique solution is exactly max_{i} \{d(v_{i})\} = 4.

In order to prove completeness we need to show that the algorithm produces a solution whenever at least one exists. In the algorithm there are only two points in which a candidate solution is discarded.

The first one is the forward checking in Algorithm 3. Indeed, it iteratively applies constraints (4)-(5) to a partial sequence S exploiting a heuristic over the future weights (i.e., the time spent to visit the successive vertexes). The employed heuristic being admissible, no feasible solution can be discarded.

The second point is the stopping criterion in Algorithm 2: when all the vertices occur in S (at least once) and the first and the last vertex in S are equal, no further successor is considered and the search is stopped. If S satisfies all the constraints, then S is a solution, otherwise backtracking is performed. We show that, if a solution can be found without stopping the search at this point, then a solution can be found also by stopping the search and backtracking (the vice versa does not hold). This issue is of paramount importance since it assures that the algorithm terminates (as we remarked in the example of the previous section, without this stopping criterion the search could not terminate). Consider a S such that S(1) = S(s) and including all the vertexes in V. The search subtree following S(s) and produced by the proposed algorithm is (non-strictly) contained in the search tree following from S(1). This is because the constraints considered by the forward checking from S(s) on are (non-strictly) harder than the corresponding ones from S(1). The increased hardness is due to the activation of constraints (4) that are needed given that at least one occurrence of each vertex is in S. Thus, if a solution can be found by searching from S(s), then a shorter solution can be found by stopping the search at S(s) and backtracking. This concludes the proof of completeness.

We give an upper bound to the temporal length of S.

**Theorem 4.2:** If the problem defined in Section III-A admits a solution, then there exists at least a solution S with temporal length no longer than max_{i} \{d(v_{i})\}.

**Proof.** In order to prove the theorem it is sufficient to prove that, if a problem is solvable, then there exists a solution S' wherein there is at least a vertex that only appears once, excluding S'(s). Indeed, if this statement holds then the maximum temporal length of S' is d(v_{i}) where v_{i} is the vertex that appears only one time in S. It easily follows that in the worst case the maximum temporal length of S' is max_{i} \{d(v_{i})\}. Figure 3 shows a situation in which the temporal length of the unique solution is exactly max_{i} \{d(v_{i})\} = 4.

In order to prove completeness we need to show that the algorithm produces a solution whenever at least one exists. In the algorithm there are only two points in which a candidate solution is discarded.

The first one is the forward checking in Algorithm 3. Indeed, it iteratively applies constraints (4)-(5) to a partial sequence S exploiting a heuristic over the future weights (i.e., the time spent to visit the successive vertexes). The employed heuristic being admissible, no feasible solution can be discarded.

The second point is the stopping criterion in Algorithm 2: when all the vertices occur in S (at least once) and the first and the last vertex in S are equal, no further successor is considered and the search is stopped. If S satisfies all the constraints, then S is a solution, otherwise backtracking is performed. We show that, if a solution can be found without stopping the search at this point, then a solution can be found also by stopping the search and backtracking (the vice versa does not hold). This issue is of paramount importance since it assures that the algorithm terminates (as we remarked in the example of the previous section, without this stopping criterion the search could not terminate). Consider a S such that S(1) = S(s) and including all the vertexes in V. The search subtree following S(s) and produced by the proposed algorithm is (non-strictly) contained in the search tree following from S(1). This is because the constraints considered by the forward checking from S(s) on are (non-strictly) harder than the corresponding ones from S(1). The increased hardness is due to the activation of constraints (4) that are needed given that at least one occurrence of each vertex is in S. Thus, if a solution can be found by searching from S(s), then a shorter solution can be found by stopping the search at S(s) and backtracking. This concludes the proof of completeness.

We give an upper bound to the temporal length of S.

**Theorem 4.2:** If the problem defined in Section III-A admits a solution, then there exists at least a solution S with temporal length no longer than max_{i} \{d(v_{i})\}.

**Proof.** In order to prove the theorem it is sufficient to prove that, if a problem is solvable, then there exists a solution S' wherein there is at least a vertex that only appears once, excluding S'(s). Indeed, if this statement holds then the maximum temporal length of S' is d(v_{i}) where v_{i} is the vertex that appears only one time in S. It easily follows that in the worst case the maximum temporal length of S' is max_{i} \{d(v_{i})\}. Figure 3 shows a situation in which the temporal length of the unique solution is exactly max_{i} \{d(v_{i})\} = 4.
the vertexes appear in the subsequence \( S(1) - S(k) \). We show that, if the problem is solvable, then it is not necessary that vertex \( S(k) \) appears again after \( k \). A visit of \( v = S(k) \) after \( k \) would be observed if either it is necessary to pass through \( v \) to reach \( S(1) \) or it is necessary to re-visit \( v \), due to its relative deadline, before \( S(1) \). However, since all the vertexes but \( S(k) \) are visited before \( k \), all the vertexes but \( S(k) \) can be visited without necessarily visiting \( S(k) \). Furthermore, the deadline of \( S(1) \) is by hypothesis harder than \( S(k) \)’s one and then the occurrence of \( v = S(k) \) after \( k \) is not necessary. Therefore, vertex \( S(k) \) occurs only one time.

The above theorem provides an upper bound on the temporal length of \( S \). Note that there are cases for which this bound is exact; namely, there are instances of the problem for which the temporally shortest solution has length exactly \( \max_i \{d(v_i)\} \) (as in Fig. 3). In other cases, the bound is not exact (as in Fig. 3 with \( d(01) = 5 \)). In any case, the upper bound can be exploited to limit the depth of the search tree preserving the algorithm’s completeness. Indeed, if \( \sum_{i=1}^{n-1} w(S(l), S(l + 1)) + \tau(S(s), S(1)) > \max_i \{d(v_i)\} \), then the search can be safely stopped and backtracked.

Finally, we show some results on the algorithm’s computational complexity. We note that, if we were looking for the minimum cost (i.e., shortest) solution \( S \) for our setting, this optimization problem could be easily reduced to a travel salesman problem showing that it is NP-hard. However, as discussed in Section III, in this paper we are considering the problem of find a solution \( S \) for our setting. This problem has not a straightforward reduction to well-known NP-hard problems and its computational complexity characterization is still an open issue. Here we present an initial result: In the special case of linear settings, the computational complexity of our approach is linear. We initially consider the problem independently of our algorithm.

**Proposition 4.3:** If a problem defined on a linear graph (see, e.g., Fig. 3) admits a solution, then the linear sequence of the vertexes is a solution.

**Proof sketch.** Consider a setting like that in Fig. 3, with any functions \( w \) and \( d \). Suppose \( S(1) = 01 \) and consequently \( S(2) = 02 \). If there is no feasible \( S \) with \( S(3) = 03 \), then the problem is not feasible. Indeed, if \( S(3) = 03 \) does not satisfy the constraints over the deadline of \( 01 \) along the cycle closure, then there is not any \( k \) such that \( S(k) = 03 \) satisfies such constraints. The same argument can be applied to any other vertex and to any linear graph.

The above proposition suggests a simple method to check the feasibility of a problem defined on a linear graph. If the linear sequence is not feasible, then the problem is unfeasible, otherwise it constitutes a solution. The length of such solution rises linearly in the number of vertexes. Notice that the problem could admit more solutions, and the length of some of them can be larger than that of the linear solution. For example, in Fig. 3, with \( d(03) \) arbitrarily large, a solution could be \( S = (01, 02, 01, 02, \ldots, 03, 02, 01) \).

Let us turn to study our algorithm in linear settings. We consider at first the case in which the problem is unfeasible. If the root node of the search tree is one of the two extremes of the linear graph, then there is no successor. Indeed, the conditions considered in the forward checking correspond exactly to the feasibility of the linear sequence. Instead, if the problem is feasible, the size of the search tree is at most \( O(2^{\max_i \{d(v_i)\}}) \). Anyway, it can be easily observed that if at each node of the search tree the successors are ordered from the minimum to the maximum \( \alpha_i \), the algorithm produces a search tree whose size is linear in the number of vertexes.

**D. Improving Efficiency and Ordering Criteria**

In this section, we show how to reduce the number of constraints to be checked in the forward checking and propose some criteria to order the vertexes in domains \( F_j \).

Consider the conditions in Steps 5 and 8 of Algorithm 3. Except for the first execution of Algorithm 3 (i.e., when \( j = 2 \)), the satisfaction of the condition at Step 5 for a given \( j \) is granted if the condition in Step 8 for \( j - 1 \) is satisfied. Therefore, we can safely limit the algorithm to check the conditions at Step 5 exclusively when \( j = 2 \). The same considerations hold also for the conditions in Steps 6 and 9. Therefore, we can safely limit the algorithm to check the conditions at Step 6 exclusively when \( j = 2 \).

We can exploit different ordering criteria in choosing the next vertex to expand in Step 8 of Algorithm 2, including: lexicographic \( (h_1) \), random with uniform probability distribution \( (h_r) \), maximum and minimum number of incident arcs \( (h_{max \ a} \) and \( h_{min \ a} \), and less visited \( (h_{min \ v}) \). In the next section we experimentally evaluate the impact of these criteria in searching for a solution.

**V. Experimental Results**

In this section, we experimentally evaluate the performance of our algorithm in producing a deterministic strategy for the patrolling agent or in returning a failure. We developed a random generator of graphs with parameters \( n \) (number of vertexes) and \( m \) (number of arcs) and working as follows. Given two values \( n \) and \( m \), firstly a connected graph with \( n \) vertexes is randomly produced, then \( m - n \) arcs are added. All the arcs weights are set equal to 1 (it can be easily shown that this is the worst case for computational complexity). Values \( d(v_k) \) are uniformly drawn from the interval \( [\min_{ij} \{w(v_i, v_j) + w(v_j, v_i)\}, 2n^2 \max_{ij} \{w(v_i, v_j)\}] \), where \( w(v_i, v_j) \) is the length of the shortest path between vertexes \( v_i \) and \( v_j \). The lower bound of the interval comes from the consideration that settings with \( d(v_k) < \min_{ij} \{w(v_i, v_j) + w(v_j, v_i)\} \) are infeasible and our algorithm immediately detects that infeasibility. The upper bound is justified by considering that if a problem is feasible then it always admits a solution shorter than \( 2n^2 \max_{ij} \{w(v_i, v_j)\} \). Graphs differ from each other in the topology and in the relative deadlines of the vertexes. This program and that implementing our algorithms have been coded in C and executed on a UNIX computer with 2.4 GHz CPU and 4 GB RAM. We developed our programs from scratch because in our problem the number
of variables to be assigned is not known in advance and off-the-shelf constraint solvers can hardly deal with this kind of problems.

We evaluate, for the different ordering criteria of Section IV-D the percentage of terminations of the algorithm and, in the case of termination (either with a solution or with a failure), the computational time. Since, as discussed in the Section III our approach is aimed at finding a solution and not the optimal solution, we do not present any way to measure the quality of a solution (e.g., its cost). Evaluating solutions on the basis of their quality is a promising line for future research. The results are the following.

- **All the ordering criteria but** $h_{\text{min}}$: all these criteria produced similar average results both in terms of termination percentage and in terms of computational times. When an execution took more than 10 minutes we considered it unterminated. We initially disabled the stopping criterion based on Theorem 4.2: the termination percentage has been very low even with small instances, for example \( \sim 22\% \) with 5 vertexes and 10-20 arcs. In this case, the average computational times have been: 153 s with $h_t$, 135 s with $h_r$, and 140 s with $h_{\text{max \ a}}$ and $h_{\text{min \ a}}$. With the stopping criterion enabled, the algorithm terminated for all the instances with 5 vertexes. With a larger number of vertexes, the stopping criterion only slightly improved the termination percentage. Overall, these ordering criteria do not provide acceptable results.

- **$h_{\text{min \ v}}$**: in this case, the stopping criterion based on Theorem 4.2 is always enabled. Table I shows the performance of the $h_{\text{min \ v}}$ ordering criterion with ties lexicographically broken. For each number $n$ of vertexes, we extracted 5 evenly-separated values for $m$ (number of arcs) from the interval \([n, (n-1)n]\), then for each pair of values $n$ and $m$, we called our random generator of graphs for creating 100 graphs (instances of our problem). The table shows the average values for these 500 instances for each $n$. In particular, we show the percentage of termination of our algorithm within a 10 minutes deadline, the average computational time (only for instances that terminated within the deadline), the standard deviation of computational time, the maximum and the minimum computational times. Table II shows the performance of the $h_{\text{min \ v}}$ ordering criterion with ties randomly broken.

The results are good, since the percentage of termination is very high (always over 90%) and the average computational time is of the order of few seconds, even with several vertexes and arcs. Note that few hundreds of vertexes is the size expected for graphs modeling real-world patrolling environments. Moreover, if a deterministic strategy can be found, it needs to be calculated only once (unless the environment in which the patrolling agent operates changes). We have verified that, unsurprisingly, given a number $n$ of vertexes, the larger the number $m$ of arcs, the more complex finding the solution (results are not shown here). The behavior of the proposed algorithm resembles that of many constraint programming algorithms, whose termination time is usually either very short (when a solution is found) or the algorithms do not terminate within the deadline. The random generation of graphs explains the data relative to the maximum computational time: some cases are harder than the average and require a lot of time to be solved (in practice, they both reduce the percentage of termination and increase the computational time). These hard cases, which represent outliers of the population of graphs, are characterized by complicated topologies or oddly-distributed relative deadlines. Comparing the two tables, a random heuristics to break ties seems more effective than the lexicographic order.

<table>
<thead>
<tr>
<th>$n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>98.25</td>
<td>97.25</td>
<td>98</td>
</tr>
<tr>
<td>dev [s]</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>0.04</td>
<td>0.14</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td>max [s]</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>125</td>
<td>396</td>
<td>213</td>
<td>115</td>
</tr>
<tr>
<td>min [s]</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
</tr>
</tbody>
</table>

**TABLE I PERFORMANCE OF THE $h_{\text{min \ v}}$ ORDERING CRITERION WITH TIES LEXICOGRAPHICALLY BROKEN.**

<table>
<thead>
<tr>
<th>$n$</th>
<th>100</th>
<th>250</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>99.8</td>
<td>96.4</td>
<td>90.2</td>
</tr>
<tr>
<td>dev [s]</td>
<td>0.16</td>
<td>0.87</td>
<td>6.3</td>
</tr>
<tr>
<td>max [s]</td>
<td>78.26</td>
<td>316.9</td>
<td>413.94</td>
</tr>
<tr>
<td>min [s]</td>
<td>$&lt; 0.01$</td>
<td>0.01</td>
<td>0.07</td>
</tr>
</tbody>
</table>

**TABLE II PERFORMANCE OF THE $h_{\text{min \ v}}$ ORDERING CRITERION WITH TIES RANDOMLY BROKEN.**

VI. CONCLUSIONS AND FUTURE WORKS

In this paper, we have proposed a method to find a deterministic patrolling strategy for a security agent. The problem is translated to that of finding a cyclic path for visiting the vertexes of a graph under relative deadlines, which constrain the time interval between two successive visits of a vertex. Our solution reduces the problem to a CSP, which is then solved with a tree search with backtracking. Use of forward checking and ordering criteria makes the approach viable for large graphs. When the method proposed in this paper succeeds to produce a deterministic patrolling strategy, a security agent can adopt it, ensuring that any intruder attempting to enter will be captured. In this case, there is no need to adopt randomized patrolling strategies.

Beyond issues discussed throughout the paper, some aspects of our approach can be improved. For example, a more sophisticated stopping criterion for tree search can be
introduced, in order to detect unfeasible sequences earlier, and more powerful heuristics can be devised, in order to further reduce the domains during forward checking. Moreover, a complete characterization of the computational complexity of the problem of finding a cyclic path under relative deadlines is still missing. In the future we aim at extending our approach to multirobot settings, using Distributed CSP techniques. Finally, we shall study the situations where the patroller has augmented sensing capabilities, being able to observe multiple vertexes from its current one.

VII. Acknowledgements

The authors are glad to thank Alessandro Busi for his contribution in implementing and evaluating the algorithm.

REFERENCES


