Swarm of Agents for Guarding an Art Gallery: A Computational Study

Mahdi Moeini
BISOR, University of Kaiserslautern,
Erwin-Schrödinger-Str.,
D-67653 Kaiserslautern, Germany.
Email: mahdi.moeini@wiwi.uni-kl.de

Daniel Schermer
BISOR, University of Kaiserslautern,
Erwin-Schrödinger-Str.,
D-67653 Kaiserslautern, Germany.
Email: daniel.schermer@wiwi.uni-kl.de

Oliver Wendt
BISOR, University of Kaiserslautern,
Erwin-Schrödinger-Str.,
D-67653 Kaiserslautern, Germany.
Email: wendt@wiwi.uni-kl.de

***Abstract*—The Art Gallery Problem (AGP) is one of the classic problems in Computational Geometry. For a given art gallery, represented by a polygon, the AGP seeks for the minimum number of guards that are necessary for overseeing the entire polygon. Many variants of this problems have already been studied. In this paper, we are interested in examining and visualizing two algorithms for the distributed version of the AGP, where guards are autonomous and have limited communication abilities. For this purpose, we present a self-contained simulator that is able to read or generate non convex polygons and to simulate the movements of robotic guards inside the polygonal environment by using the navigation algorithms. In particular, we study two algorithms: Random Search (RS) and Depth-First Search (DFS). We compare RS and DFS, in terms of computation time, by testing them on benchmark instances as well as randomly generated polygons. According to our experiments, each algorithm has a better performance on specific types of polygons.**

I. Introduction

In Computational Geometry, one of the well-known and interesting problems is the Art Gallery Problem (AGP) [4], [18], [19] that was proposed by Victor Klee, in 1973. In the AGP, the objective consists in finding the minimum number of guards that we need for having the complete view on a polygonal environment. It has been shown that, even for restricted cases, the AGP is NP-hard [11], [12], [13]. One of the interesting results is known as *Chvátal's art gallery theorem*. According to this theorem, for a given polygon (without hole) with n vertices, $\lfloor \frac{n}{3} \rfloor$ guards are sufficient (and sometimes necessary) to oversee the whole polygon. Similar result concerns the orthogonal polygons for which the bound is $\lfloor \frac{n}{4} \rfloor$. In general, these bounds are not tight; hence, many studies have been conducted to design algorithms for finding the exact number of necessary guards [4], [11], [12], [18].

The Art Gallery Problem has several applications in different domains, for example robotics and telecommunications. Deploying autonomous robots inside an unknown area is one of the interesting application topics in which AGP has been used [4], [5], [6], [14]. More precisely, for a given polygonal environment, the objective consists in distributing autonomous agents (such as robots) with limited communication abilities in a way to oversee the entire polygon [5], [6], [14]. In particular, due to recent technical advances that allow creation of small, robust, and inexpensive robotic units, the AGP has attracted the attention of researchers in some engineering fields such as sensor networks and robotics [1], [10], [15], [17]. However, these researchers are interested in a variant of AGP that is different with the classic one. Indeed, the focus is on deployment of autonomous robots for solving (autonomously) the AGP, under the condition that the agents have no global knowledge on the polygon, prior to deployment [5], [6], [7], [8], [9], [16]. Consequently, each robot can only get information by sensing the environment and by communicating with other robots. Due to this fact, the solution will not be the same as in the classic AGP: This approach will not focus on finding the minimum number of the guards that are necessary for guarding the polygon. Therefore, the main objective consists in covering the whole polygon without having global knowledge about the polygonal area. We use the term *distributed AGP* (or, when there is no ambiguity, *AGP*) in order to specify placing the autonomous agents on (suitable) vertices of a given polygon with the objective of observing the whole polygon.

In this paper, we introduce an "AGP simulator" in order to solve the (distributed) AGP. To the best of our knowledge, this is the first simulator that is self-contained, does not depend on external libraries or software packages (such as CGAL and Matlab), and focuses on solving the (distributed) AGP [3], [5], [6], [7], [11]. In particular, this AGP simulator is free, portable, and easy-to-use. Two deployment algorithms (for distributing and placing a swarm of autonomous agents) are incorporated inside the simulator that permit to solve the AGP, visualize the algorithms, and export the results. The implemented algorithms are Random Search (RS) and Depth-First Search (DFS) navigation procedures [5], [6], [7]. Furthermore, we carried out some numerical experiments by using benchmark instances as well as randomly generated polygons. According to the numerical results, each of the algorithms has a better performance in solving AGP on specific types of polygons.

We organize this paper as follows: Section II is devoted to the basic definitions and notation. In Section III, we present the necessary materials that we need for modelling the (autonomous) robotic agents and studying the (distributed) AGP. Section IV explains the algorithms RS and DFS for solving the (distributed) AGP. The AGP-simulator is presented in Section V. Section VI consists of the computational experiments, the

numerical results, and comparison of RS versus DFS. In the last section, we discuss about some concluding remarks and future research avenues.

II. Preliminaries, Basic Definitions, and Notation

We suppose that an art gallery, in form of a simple and non convex polygon Q (without hole), is given. The set of the vertices of Q is $Ve(Q) = \{v_1, \ldots, v_n\}$, where n is the number of vertices, $v_i = (x_i, y_i)$, and x_i, y_i are real numbers. Let $E(Q) = \{e_1, \ldots, e_n\}$ refer to an ordered set of edges of Q. We denote the *exterior* of Q, its *interior*, and the boundary of Q, respectively, by $ext(Q)$, $int(Q)$, and $\partial(Q)$. A line $[v_i, v_j] \in Q$ connecting two non-consecutive vertices $v_i, v_j \in Ve(Q)$, where $]v_i, v_j[\in int(Q)$, is called a *diagonal* of Q. The notation $P(Q)$ is used to denote the set of points that are inside the polygon (i.e., $P(Q) = int(Q) \cup \partial(Q)$). Suppose that a given polygon Q is partitioned into m smaller polygons Q_i $(i = 1, \ldots, m)$, with disjoint interiors. Any diagonal of Q, that is shared by two adjacent partitions, is called a *gap* [5], [6], [7]. A *triangulation* $T(Q)$ of a polygon Q is a set of $n-2$ triangles that partition Q [18], [20].

A point $q \in P(Q)$ is said to be *visible* from another point $p \in P(Q)$, if $[p, q] \in int(Q) \cup \partial(Q)$ [11], [12]. For a given point $p \in Q$, the *visibility polygon* $S(p) \subset Q$ of p consists of all points of Q that are visible from p. In a similar way, the *vertex-limited* visibility polygon $S_{ver}(p) \subset Q$ is a polygon that is defined by the subset of vertices in $Ve(Q)$, that are *visible* from p. A *star-shaped* polygon Q_{star} contains at least one point from which any point of Q_{star} is visible. Such a point, e.g., k is called a *kernel point* and $S(k) = Q_{star}$. We denote the set of kernel points by $K(Q_{star})$ and call it the *kernel* of Q_{star} [20].

In this paper, we consider non-convex polygons (without hole) and we accept only vertex guards, i.e., the vertices of Q are the only positions where a guard (agent) can be placed; however, the guards can move freely inside Q. A subset of vertices $G(Q) \subset Ve(Q)$ is said to be a *(vertex) guard set* if its members *guard* or *oversee* the whole polygon, i.e., any point $p \in P(Q)$ is visible from at least one vertex $g \in G(Q)$.

III. Distributed Art Gallery Problem

For a given polygon Q, the (distributed) Art Galley Problem (AGP) wants to place the guards (robotic agents) on some vertices of Q such that the set of placed guards oversees the entire polygon. In this case, the number of agents may not be equal to the minimum number of guards, as it is in the classic AGP; however, due to the fact that the agents (guards) have no knowledge about the polygon (prior to deployment) and they have limited communication abilities, guarding the whole polygon is the most important objective of the problem. More precisely, the objective consists in designing a multi-agent system in order to deploy a set A of (autonomous) agents (i.e., guards) from a given vertex of Q and place them on suitable vertices for guarding the whole polygonal area. In this context, the agents can only use the information that they get by sensing the environment and by communicating with other agents (that are located within a predefined limited distance). Based on the gathered information, the agents decide in which direction and up to which point they can move. Based on the approach of Ganguli et al. [5], [6], [7], [8], the procedure consists of partitioning the given polygon, constructing a tree, exploring the tree, and placing the guards on suitable vertices.

Let A be the set of agents that we want to deploy in Q by starting from a given vertex $s \in Ve(Q)$. We suppose that each agent $a_i \in A$, has a unique identifier (ID) i, where $i \in \{1, \ldots, |A|\}$. Furthermore, each agent is equipped with an omni-directional line-of-sight sensor with an unlimited range [5], [6], [7]. With this equipment, the agents can sense their star-shaped vertex-limited visibility polygon from their current position in $P(Q)$. Each agent a_i has also a limited amount of memory M_i that is updated based on the agent's movements and its communications with other agents [5], [6], [7]. Defining R as the maximum broadcast range, contents of the memory of any agent a_i as well as its ID can be broadcast to any other agent $a_j \in A$, where $j \neq i$, if a_j is within the range and line-of-sight of a_i. We denote such a broadcast by `Broadcast(`i, M_i`)`. Furthermore, each agent can:

- send a predefined amount of repeated `Broadcast(`i, M_i`)`, each $\delta > 0$ seconds;
- while broadcasting, `Listen` for incoming data, received from other agents `Broadcast(`j, M_j`)`;
- `Process` the received information and continue to `Listen` during this period;
- `Move` to a desired point, based on the decision made during `Process`.

Let us suppose that the polygon Q is partitioned into m star-shaped polygons with kernel points $k_l \in K(Q) = \{k_1, \ldots, k_m\}$. Based on this partition, we have a directed tree $T(Q)$ with $K(Q)$ as its nodes and k_1 as the root. Using this set of information, the memory of each agent is composed of a quadruple of points in Q, labelled as $(p_{parent}, p_{last}, g_{left}, g_{right})$:

- p_{parent} refers to the parent kernel point;
- p_{last} refers to the last way-point i.e., the point before moving to the current position;
- (g_{left}, g_{right}) is the gap that is shared by the current partition and its parent partition.

The initial location of the agents is used for initializing these parameters. During each broadcast, each agent sends these values to all agents that are within its range and line-of-sight.

Whenever an agent moves from a kernel point k_i to a child kernel point k_j, the agent performs a `Move-to-Child` operation (see Algorithm 2) through the gap, described by two vertices $[v_n, v_m]$, (where $v_n, v_m \in Ve(Q)$). The memory M_a of the agent a is updated as follows:

- $p_{parent} = k_i$,
- $p_{last} = k_j$,
- $g_{left} = v_n$,
- $g_{right} = v_m$.

Whenever we want to perform a `Move-to-Parent` operation (see Algorithm 3) an agent moves from a kernel point k_j to a

parent kernel point k_i and the memory M_a of the agent a is updated as follows:

- $p_{last} = w$, where w is the mid-gap point of the diagonal that is shared by partitions Q_j and Q_i,
- p_{parent}, g_{left}, and g_{right} are updated during the next `Process` action. Every agent receives the content of each agent's memory from all other agents within the range and line-of-sight during the `Listen` action. We update the values of p_{parent}, g_{left}, and g_{right} by listening to the incoming message that is received from the agent with the highest ID.

IV. Algorithms for Deploying Agents in an Art Gallery

In order to introduce the navigation algorithms that we use for exploring and guarding a polygonal environment, we present preliminary algorithms that are necessary for using the navigation algorithms.

A. Vertex-Induced Partition and Tree Algorithm

Suppose that a non-convex polygon Q is given, the *vertex-induced partition and tree algorithm* starts from a given vertex $s \in Ve(Q)$ and:

- partitions Q into a set $\mathcal{S}^*(Q)$ of star-shaped polygons $\{Q_1, \ldots, Q_m\}$;
- finds a list of kernel vertices $K^*(Q) = \{k_1, \ldots, k_m\}$ for each respective star-shaped polygon;
- constructs a rooted tree $T^*(Q)$ with $K^*(Q)$ as the set of nodes and k_1 as the root.

The details of the vertex-induced partition and tree algorithm can be found in [6]; however, for the sake of completeness and clarity, we provide a brief description of the procedure in Algorithm 1 that outlines different steps of the procedure. Figure 1 highlights different steps of the algorithm.

For a polygon Q with n vertices, $T^*(Q)$ has, at most, $\lfloor n/2 \rfloor$ vertices. If we put a guard on each vertex of $T^*(Q)$, the whole polygon will be overseen. Consequently, the cardinality of the guard set $G(Q)$ is $\leq \lfloor n/2 \rfloor$ (see also [6]).

B. Node-to-Node Navigation Algorithms

Using the vertex-induced partition and tree algorithm, for a given polygon Q, we can create a directed rooted tree $T^*(Q)$ with the kernel points $\{k_1, \ldots, k_m\}$ as its vertices, where $s = k_1 \in K^*(Q)$ is the root of the tree. The tree $T^*(Q)$ can be used for deploying a swarm of (robotic) agents in order to guard Q. For this purpose, we consider two algorithms for navigating through the vertex-induced tree. In these algorithms, we use the mid-point of the gaps and the vertices of the polygon for the movements. Through the navigation, we use two basic operations as follows [5], [6], [7]:

- **Move-to-Child:** refers to moving to a child $k_j \in K^*(Q)$ of a node $k_i \in K^*(Q)$ (see Algorithm 2).
- **Move-to-Parent:** refers to moving from a node $k_j \in K^*(Q)$ to its parent node $k_i \in K^*(Q)$ (see Algorithm 3).

Figure 1 (c) shows the paths that are created by movements using **Move-to-Parent** and **Move-to-Child** operations.

Algorithm 1 Vertex-Induced Partition and Tree Algorithm

`Inputs`: Vertex s and Polygon Q;
`Outputs`: Vertex-Induced Partition and Tree;
Build a list of vertices that are visible from $k_1 = s$.
Let P_1 be the polygon determined by the vertices that are visible from $k_1 = s$ (P_1 includes k_1).
Create a FIFO-list, named `openGaps`. Identify the edges of P_1 that are diagonals of Q, call each of them a gap of P_1, and add them to `openGaps`.
while `openGaps` is not empty **do**
    Popup `openGaps` to get the *current gap*.
    Find a new point $k_j \in Ve(Q)$, outside the star-shaped polygon of the current gap, such that k_j is able to see the current gap entirely.
    Add the current gap to the list `closedGaps` and remove it from the list `openGaps`.
    Build a list of the vertices that are visible from k_j and do not cut the current gap; Let P_j be the polygon determined by these vertices (by definition, $k_j \in K(P_j)$).
    Identify the edges of P_j that are diagonals of Q and call each of them a gap of P_j. If they are not in `closedGaps`, add them to the list of `openGaps`.
end while

Algorithm 2 *Move-to-Child* Algorithm

`Input`: Position k_i;
`Output`: Position k_j, $j > i$;
compute the mid-point of the gap shared by the current partition Q_i and the child partition Q_j.
go to the mid-point of the gap;
compute the nearest vertex from which the entire gap is visible and which is not located inside the parent partition;
go to that vertex.

C. Depth-First Search Deployment Procedure

Ganguli et al. [5], [6], [7] introduced the Depth-First Search (DFS) navigation algorithm by which the vertex-induced tree is explored via a Depth-First Search (DFS) approach. Indeed, the procedure consists in exploring $T^*(Q)$ as deep as possible along each branch before returning (if necessary) to a parent node. Once the full covering of the polygon is achieved, the algorithm stops. Algorithm 4 presents the different steps of DFS navigation algorithm.

For a given polygon Q with n vertices, if we deploy sufficient number of agents (e.g., $\lfloor \frac{n}{2} \rfloor$ agents), Algorithm DFS is able to

Algorithm 3 *Move-to-Parent* Algorithm

`Inputs`: Position k_j, gap shared with parent partition;
`Output`: Position k_i, $i < j$;
compute the mid-point of the gap shared by the current partition Q_j and the parent partition Q_i;
go to the mid-point of the gap;
go to p_{parent}, representing the parent node.

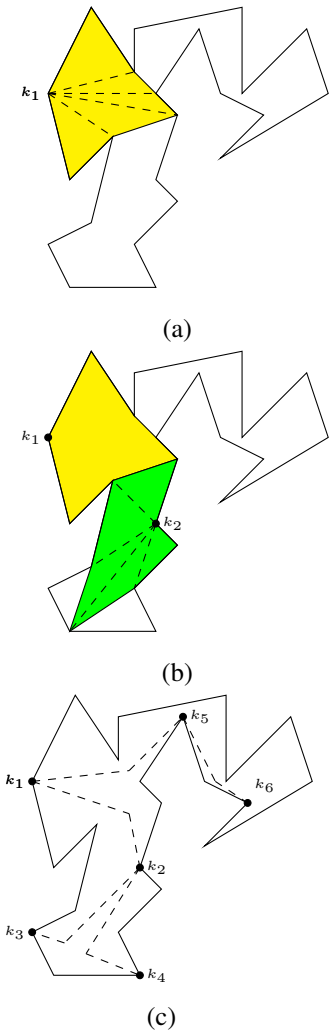

Fig. 1. Computation of the vertex-induced partition and tree on a polygon with $n = 23$ vertices.

explore the vertex-induced tree and guard the whole polygon in a finite time (for more details, see [5], [6]).

D. Random Deployment

Another way of exploring the vertex-induced tree consists in using a random (however, systematic) navigation approach. More precisely, the Random Search (RS) algorithm is, essentially, the same as Algorithm 4; except, in place of visiting all children of a node one-after-one, and doing a backtracking (if necessary), we choose randomly either a child and execute *Move-to-Child* operation or we do a *Move-to-Parent* [5], [6]. At a given node, we draw a random number such that all children of the current node as well as its parent have a same chance (probability) to be selected. This method of selection is rational because, if there are many children to visit, then there will be higher probability to move in the direction of children, rather moving back towards the parent node.

The RS can be stopped as soon as the polygon is entirely covered; however, theoretically, there is no guarantee that the RS algorithm covers the whole polygon in finite time.

Algorithm 4 Depth-First Search (DFS) Algorithm for AGP

All agents are initially located at the root vertex s of the vertex-induced tree.
During each *Process* operation, the agents execute the following steps:
Find maximum ID received during the *Listen* operation;
if maximum received ID is less than its own ID **then** do not move.
else
    if the current kernel point has no children **then**
        Move-to-Parent via the gap $\{g_{left}, g_{right}\}$ of the current position.
    else Order the children in a consistent way (e.g., clockwise).
        if p_{last} (in memory) is the parent of the present node, **then**
            Move-to-Child towards the first child in the ordering;
        end if
        if p_{last} (in memory) is a gap that is not the last in the ordering, **then**
            Move-to-Child towards the next child in the ordering;
        end if
        if p_{last} (in memory) is a gap that is the last in the ordering and the current node is not the root, **then**
            Move-to-Parent towards p_{parent} via $\{g_{left}, g_{right}\}$ (that is a gap);
        end if
        if p_{last} (in memory) is a gap that is the last element in the order and the current node is the root, **then**
            Stop the algorithm, we have the full visibility on the polygon.
        end if
    end if
end if

V. A Simulator for Solving the Distributed AGP

We carried out some numerical experiments by testing and comparing the algorithms RS and DFS. For this purpose, we developed a self-contained simulator and tested the algorithms on benchmark instances as well as randomly generated non-convex polygons (without hole).

The self-contained platform [1], that has been developed in Java, allows for the generation of random simple (nonorthogonal) polygons based on 2-opt moves [2] and von Koch polygons based on a procedure presented in [11]. Furthermore, it is also possible to read vertices of polygons from a text file.

Once a polygon has been either generated or imported, the number of agents to deploy and the deployment policy can be defined via a GUI. The Algorithms RS and DFS have been

[1]The simulator is publicly available at https://sites.google.com/site/mahdimoeini2013/software-packages

implemented for exploring the tree and any of them can can be selected. One can choose a starting point $s \in Ve(Q)$ or a randomly selected vertex will be considered for starting the selected deployment procedure.

Additional settings can be adjusted as follows:

- *Movement Delay* of agents: can move 100 units per α seconds ($\alpha \in [0, 1]$),
- *Broadcast Delay* (in seconds) can be set to δ, where $\delta \in [0, 0.4]$,
- *Broadcast Range* R, which can be set to a number in the interval $[0, 1000]$,
- an initial *Startup Delay* (in seconds), that will make agents to randomly start after a delay that can be any number an the interval $[0, \beta]$, where $\beta \in [0, 20]$.

The simulation can then be started with the default or adjusted parameters. Then, the simulator presents a real time animation of the exploration of the polygon by the agents. Each agent continuously performs its autonomous actions, based on the predefined deployment policy. A graph highlights the time-discrete evolution of coverage (of the polygon), based on the number of covered (guarded) vertices.

As soon as all vertices are covered, a log file is created that contains all information about the polygon, the guard set, the selected settings, and runtime to achieve full coverage. If the algorithm (RS or DFS) fails (within a predefined time-limit) to find a guard set with the full cover of the polygon, the log file will also store the amount of currently visible vertices at the moment of the timeout. The current state of the network can also be stored as an image file. Figure 2 shows an overview of the developed platform.

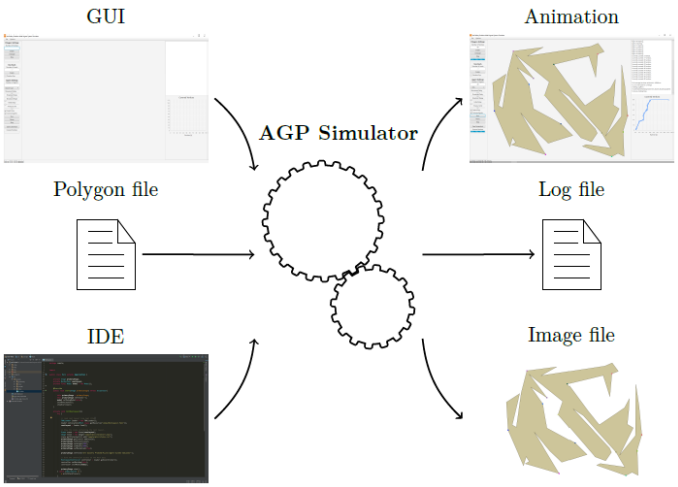

Fig. 2. Overview of the developed Platform.

VI. Numerical Experiments

In this section, we present a comparative analysis on the performance of DFS and RS methods for solving the AGP. For this purpose, the algorithms have been implemented in Java, inside the developed simulator, and the experiments have been carried out on a 3.5 GHz quad core laptop with 8GB of RAM.

A. Instances and Test Settings

For our experiments, we selected three types of polygons: simple (nonorthogonal), orthogonal, and von Koch. Some sample polygons are shown in Figures 3 - 5. Different types of polygons help us to gain deeper insight into the performance of each algorithm in solving AGP at different polygonal areas.

The test sets include benchmark instances as well as randomly generated polygons. The random polygons are generated by our Java platform and the benchmark instances (that have already been used in several research studies, e.g., [11], [12]) are publicly available at http://www.ic.unicamp.br/~cid/Problem-instances/Art-Gallery/AGPVG/index.html.

For a given polygon with n vertices, we deploy $\lfloor n/4 \rfloor$ agents for orthogonal and von Koch polygons and $\lfloor n/3 \rfloor$ agents for the others.

In total, we used 45 polygons and did 270 experiments:

- 9 types of polygons (orthogonal, von Koch, and simple (nonorthogonal): with 60, 100, and 200 vertices),
- 5 polygons of each type and 3 runs on each polygon (in each run, a different starting point was used),
- 2 deployment policies (i.e., RS and DFS) for each starting point.

For the polygons with either 60 or 100 vertices, the experiments were done under default settings, that is:

- Movement Delay is set to 0.5: i.e., the agents can move 200 units per second.
- Broadcast Delay is set to 0.2.
- Number of Broadcasts is 3: i.e., three broadcasts are sent before processing.
- Broadcast Range is 0: Only the agents that are located on a same vertex can communicate with each other.
- Initial Delay is 0: The agents start as soon as the simulation starts.

The higher dimensions need different configurations in order to have a better performance of the platform. For this purpose, following settings has been used for polygons with 200 vertices:

- Movement Delay is set to 0.4.
- Broadcast Delay is set to 0.4.
- Number of Broadcasts is set 3.
- Broadcast Range is set to 0.
- Initial Delay is set 5: This means that the agents have, at most, 5 seconds time for starting.

Finally, we set a time-limit of 5 minutes on each experiment. If no full visibility is achieved within the time-limit, the number of currently visible (covered) vertices is recorded in a log file.

B. Numerical Results

The results of our experiments are presented in Table I and Figures 6-8. In Table I, the *average* computation time (in seconds) of each algorithm (i.e., RS (s.) and DFS (s.)) are presented. In particular, each row of the table represents a polygon type and its corresponding number of vertices. The second column of Table I shows the number of deployed agents (# Agents) that, in fact, depends on the type and the size (i.e., number of vertices) of a given polygon. In our experiments,

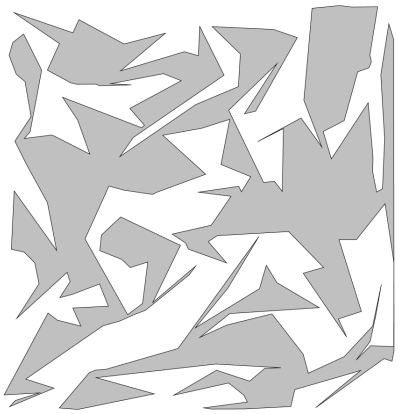

Fig. 3. A simple polygon with $n = 200$ vertices.

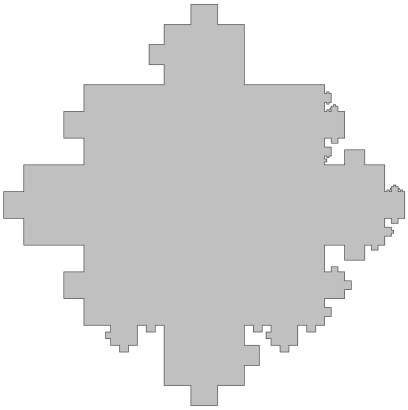

Fig. 4. A von Koch polygon with $n = 200$ vertices.

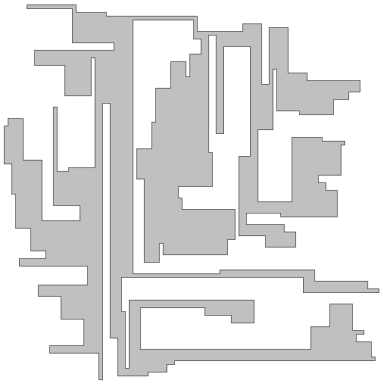

Fig. 5. An orthogonal polygon with $n = 200$ vertices.

we consider a time-limit of 300 seconds and we interrupt the algorithm if the time-limit is reached. In this case, it may happen that the polygon is not yet completely covered.

Figures 6–8 show more details on performance of the algorithms. Each of these figures is associated to a certain type of polygon (i.e., simple (nonorthogonal), orthogonal, and von Koch). On each figure, the horizontal axis is partitioned into 2 parts, associated to a deployment algorithm (i.e., random search and DFS). Furthermore, in each partition, we present the

Instances		# Agents	Average Computation Time (s.)	
Polygon Type	size		Random Search	DFS
Simple	60	20	115.6	69.7
Simple	100	33	169.2	106.9
Simple	200	66	291.6	240.8
von Koch	60	15	41.8	54.8
von Koch	100	25	71.5	94.8
von Koch	200	50	116.6	236.5
Orthogonal	60	15	123.1	46.2
Orthogonal	100	25	187.9	73.7
Orthogonal	200	50	300	136.5

TABLE I

THE AVERAGE COMPUTATION TIME (IN SECONDS) OF EACH ALGORITHMS FOR COVERING A GIVEN POLYGON Q. IF NO FULL COVERAGE IS ACHIEVED WITHIN 300 SECONDS, THE COMPUTATION TIME IS SET TO 300.

results for polygons of size 60, 100, and 200. More precisely, the minimum, average, and maximum computation time (in seconds) that each algorithm requires to find a guard set for polygons are presented. For some orthogonal polygons, the random search fails to cover the whole polygon within the time-limit.

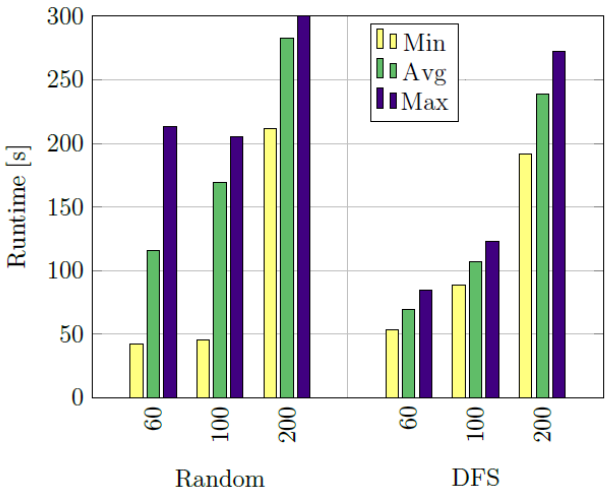

Fig. 6. Minimum, average and maximum runtime to find a guard set in simple polygons of various sizes by using DFS and random search.

C. Comments on the Results

First, it is interesting to note that the structure of the vertex-induced tree depends on shape of the underlying polygon, its number of vertices, and the choice of starting point. For example, the simple (nonorthogonal) as well as orthogonal polygons lead to a similar vertex-induced trees. In particular, the average degree of each node is usually smaller than the case of a tree for a von Koch polygon with a same size. However, if we consider the height of the vertex-induced trees of the polygons, the simple (nonorthogonal) and orthogonal polygons have (usually) trees with larger height than the case of von Koch polygons. Due to these facts, the deployment algorithms RS and DFS do not have the same performance in solving AGP on different types of polygons.

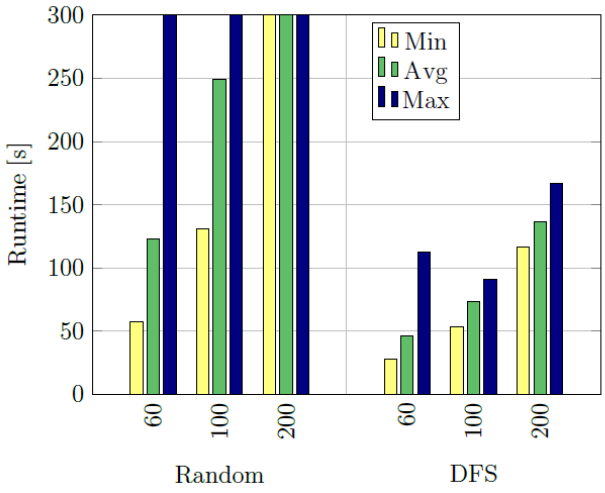

Fig. 7. Minimum, average and maximum runtime to find a guard set in orthogonal polygons of various sizes by using DFS and random search.

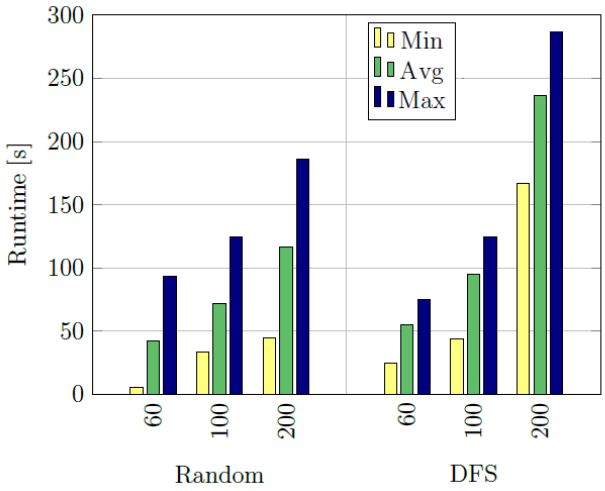

Fig. 8. Minimum, average and maximum runtime to find a guard set in von Koch polygons of various sizes by using DFS and random search.

According to Table I and Figures 6-8, we observe that, in solving simple (nonorthogonal) and orthogonal polygons, the RS algorithm is less efficient than DFS. However, random deployment performs surprisingly well in solving von Koch polygons. Indeed, in the case of von Koch polygons, most of the nodes of the vertex-induced tree have a relatively large degree but the tree has relatively small height. Due to this fact, if we scatter the agents randomly amongst children of a node, then we can explore quickly different nodes of the vertex-induced tree and, consequently, oversee the entire polygon. However, this is not the case for simple (nonorthogonal) and orthogonal polygons, where the tree has considerably large height and the nodes have small degree. In such a case, a more intelligent approach such as the DFS deployment algorithm performs considerably better than RS.

As an example, Figures 9 and 10 visualize the solutions provided, respectively, by DFS and RS for a von Koch polygon with 100 vertices. On these figures, the vertical axis shows the number of covered vertices and the horizontal axis represents the time. Furthermore, each circle indicates the moment of covering a new group of uncovered vertices. As we observe, both methods (DFS and RS) provide similar solutions. In fact, DFS needs 19 agents to cover the polygon and RS requires 18 agents. However, these methods are rather different in terms of the computation time. As it is depicted on Figures 11 and 12, the RS navigation algorithm has a better performance than DFS. Indeed, in less than 5 seconds, RS covers almost 90 % of the vertices versus 50 % for DFS. Finally, RS needs 16.145 seconds versus 57.242 seconds for DFS, in order to solve this instance.

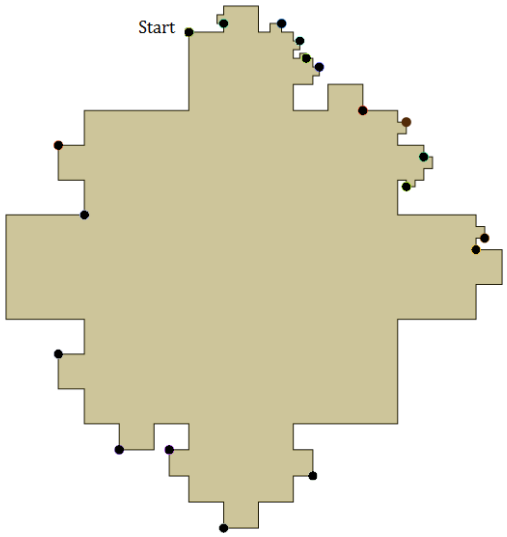

Fig. 9. The solution provided by DFS method for a von Koch polygon with 100 vertices.

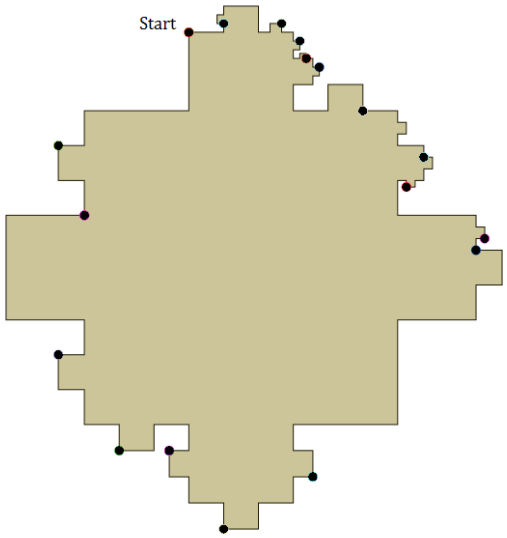

Fig. 10. The solution provided by Random Search method for a von Koch polygon with 100 vertices.

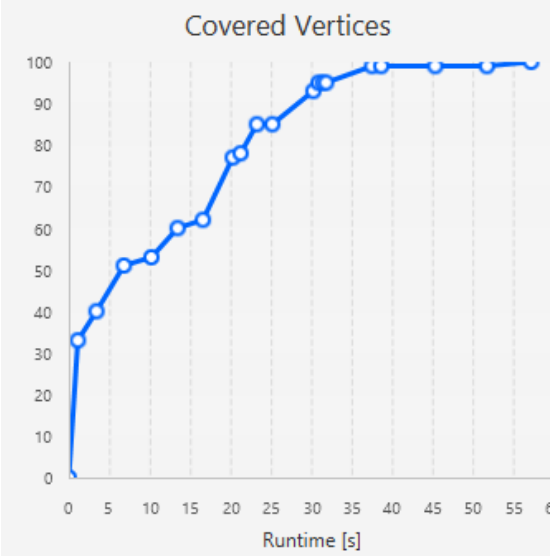

Fig. 11. The performance of the DFS method for covering a von Koch polygon with 100 vertices.

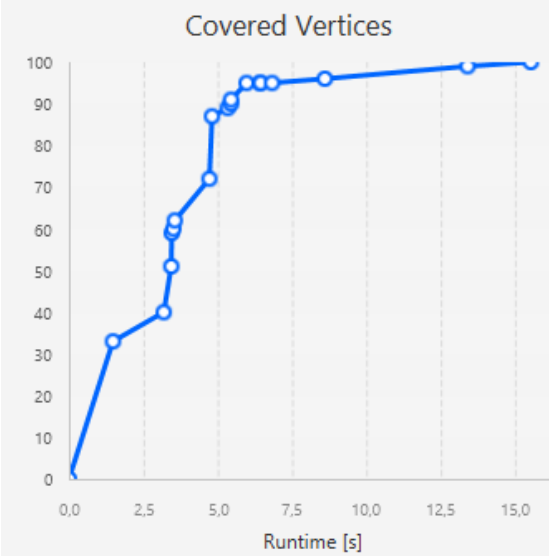

Fig. 12. The performance of the Random Search method for covering a von Koch polygon with 100 vertices.

VII. Conclusion

In this paper, we highlighted current research related to solving the Art Gallery Problem (AGP) using a multi-agent approach and present computational results of the relevant algorithms (DFS and RS) for solving the AGP. These solution methods are based on the vertex-induced partition and tree algorithm [5], [6], [7]. This approach allows us to deduce, as a multi-agent system, a graph exploring problem from the AGP. In particular, our experiments, that we carried out by a self-contained Java platform, confirm that the algorithm RS is more efficient than DFS in solving AGP on von Koch polygons. In the case of orthogonal and simple (nonorthogonal) instances, DFS is more efficient that RS.

The future research directions can be extension of the platform for processing more variants of polygons as well as developing and implementing more algorithms. Designing more efficient algorithms is also an interesting research topic to explore. The research in these directions is in progress and the results will be reported in future.

Acknowledgment

The authors acknowledge the Technical University of Kaiserslautern (Germany) for the financial support, through the research program “CoVaCo”.

References

[1] Asama, H., Tamio, A., Fukuda, T., Hasegawa, T. *Mobile Sensor Network Deployment using Potential Fields: A Distributed, Scalable Solution to the Area Coverage Problem*, Distributed Autonomous Robotic Systems, 299–308, 2002.

[2] Auer, T., Held, M. *Heuristics for the Generation of Random Polygons*. Proc. 8th Canad. Conf. Comput. Geom, 38–44, 1998.

[3] *CGAL: Computational Geometry Algorithms Library*, http://www.cgal.org

[4] de Rezende, P.J., de Souza, C.C., Friedrichs, S., Hemmer, M., Kröller, A., Tozoni, D.C. *Engineering Art Galleries*. arXiv:1410.8720, 2014.

[5] Ganguli, A., Cortes, J., Bullo, F. *Distributed Deployment of Asynchronous Guards in Art Galleries*, American Control Conference (ACC), 1416–1421, 2006.

[6] Ganguli, A., Cortes, J., Bullo, F. *Distributed Coverage of Nonconvex Environments*, Networked Sensing Information and Control, 289–305, 2007.

[7] Ganguli, A. *Motion Coordination for Mobile Robotic Networks with Visibility Sensors*, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, 2007.

[8] Ganguli, A., Cortes, J., Bullo, F. *Visibility-based Multi-Agent Deployment in Orthogonal Environments*, American Control Conference (ACC), 3426–3431, 2007.

[9] Howard, A., Matarić, M.J., Sukhatme, G.S. *An Incremental Self-Deployment Algorithm for Mobile Sensor Networks*, Autonomous Robots, Vol. 13, No. 12, 113–126, 2002.

[10] Huang, C.F., Tseng, Y.C. *The Coverage Problem in a Wireless Sensor Network, Mobile Networks and Applications*, Vol. 10, No. 4, 519–528, 2005.

[11] Kröller, A., Baumgartner, T., Fekete, S.P., Schmidt, C. *Exact Solutions and Bounds for General Art Gallery Problems*, J. on Experimental Algorithmics, Vol. 17, No. 1, 2012.

[12] Kröller, A., Moeini, M., Schmidt, C. *A Novel and Efficient Approach for Solving the Art Gallery Problem*, WALCOM: Algorithms and Computation, Lecture Notes in Computer Science (LNCS), Vol. 7748, 5–16, 2012.

[13] Lee, D.T., Lin, A. *Computational Complexity of Art Gallery Problems*, IEEE Transactions on Information Theory, Vol. 32, No. 2, 276–282, 1986.

[14] McLurkin, J., Smith, J. *Distributed Algorithms for Dispersion in Indoor Environments using a Swarm of Autonomous Mobile Robots*, Proc. 7th Internat. Sympos. Distr. Auton. Robot. Syst., 2004.

[15] Meguerdichian, S., Koushanfar, F., Qu, G., Potkonjak, M. *Exposure in Wireless Ad-Hoc Sensor Networks*, Proceedings of the 7th Annual International Conference on Mobile Computing and Networking, 139–150, 2001.

[16] Obermeyer, K.J. *Visibility Problems for Sensor Networks and Unmanned Air Vehicles*, Mechanical Engineering Department, University of California at Santa Barbara, 2010.

[17] Obermeyer, K.J., Ganguli, A., Bullo, F. *A Complete Algorithm for Searchlight Scheduling*, International Journal of Computational Geometry & Applications, Vol. 21, No. 1, 101–130, 2011.

[18] O’Rourke, J. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., 1987.

[19] Shermer, T.C. *Recent Results in Art Galleries*. Proceedings of the IEEE, Vol. 80, No. 9, 1384–1399, 1992.

[20] Urrutia, J. *Art Gallery and Illumnation Problems*, 2004.