Area Protection in Adversarial Path-finding Scenarios with Multiple Mobile Agents on Graphs

A Theoretical and Experimental Study of Strategies for Defense Coordination

Marika Ivanová[1], Pavel Surynek[2] and Katsutoshi Hirayama[3]

[1]*Department of Informatics, University of Bergen, Thormhlensgt. 55, 5020 Bergen, Norway*

[2]*National Institute of Advanced Industrial Science and Technology (AIST), 2-3-26, Aomi, Koto-ku, Tokyo 135-0064, Japan*

[3]*Kobe University, 5-1-1, Fukaeminami-machi, Higashinada-ku, Kobe 658-0022, Japan*

marika.ivanova@uib.no, pavel.surynek@aist.go.jp, hirayama@maritime.kobe-u.ac.jp

Keywords: Graph-based Path-finding, Area Protection, Area Invasion, Asymmetric Goals, Mobile Agents, Agent Navigation, Defensive Strategies, Adversarial Planning.

Abstract: We address a problem of area protection in graph-based scenarios with multiple agents. The problem consists of two adversarial teams of agents that move in an undirected graph. Agents are placed in vertices of the graph and they can move into adjacent vertices in a conflict-free way in an indented environment. The aim of one team - *attackers* - is to invade into a given area while the aim of the opponent team - *defenders* - is to protect the area from being entered by attackers. We study strategies for assigning vertices to be occupied by the team of defenders in order to block attacking agents. We show that the decision version of the problem of area protection is PSPACE-hard. Further, we develop various on-line vertex-allocation strategies for the defender team and evaluate their performance in multiple benchmarks. Our most advanced method tries to capture bottlenecks in the graph that are frequently used by the attackers during their movement. The performed experimental evaluation suggests that this method often defends the area successfully even in instances where the attackers significantly outnumber the defenders.

1 INTRODUCTION

We address an *Area Protection Problem* (APP) with multiple mobile agents moving in a conflict-free way. APP can be regarded as a modification of known problem of *Adversarial Cooperative Path Finding* (ACPF) (Ivanová and Surynek, 2014) where two teams of agents compete in reaching their target positions. Unlike ACPF, where the goals of teams of agents are symmetric, the adversarial teams in APP have different objectives. The first team of *attackers* consists of agents whose goal is to reach a pre-defined target location in the area being protected by the second team of *defenders*. Each attacker has a unique target in the protected area. The opponent team of defenders tries to prevent the attackers from reaching their targets by occupying selected locations so that they cannot be passed by attackers.

Another distinction between ACPF and APP is a definition of victory of a team. A team in ACPF wins if all its agents reach their targets and agents of no other team manage to do so earlier. In APP, the team of defenders wins if all attackers are kept out of their targets. Our effort is to design a strategy for the defending team, so the success is measured from the defenders' perspective. It is often not possible to prevent all attackers from reaching their targets, and so the following objective functions can be pursued:

1. maximize the number of target locations that are not captured by the corresponding attacker
2. maximize the number of target locations that are not captured by the corresponding attacker within a given time limit
3. maximize the sum of distances between the attackers and their corresponding targets
4. minimize the time spent at captured targets

The common feature of APP and ACPF is that once a location is occupied by an agent, it cannot be entered by another agent until it is first vacated by the agent which occupies it (opposing agent cannot push the agent out). This is utilized both in competition for reaching goals in ACPF, where agents may try to slow down the opponent by occupying certain locations, as

Ivanová, M., Surynek, P. and Hirayama, K.
Area Protection in Adversarial Path-finding Scenarios with Multiple Mobile Agents on Graphs - A Theoretical and Experimental Study of Strategies for Defense Coordination.
In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018) - Volume 1*, pages 184-191
ISBN: 978-989-758-275-2

well as in APP, where it is a key tool for the defenders for keeping attackers out of the protected area.

APP has many real-life motivations from the domains of access denial operations both in civil and military sector, robotics with adversarial teams of robots or other type of penetrators (Agmon et al., 2011), and computer games.

Our contribution consists in analysis of computational complexity of APP. In particular, we show that APP is PSPACE-hard. Next, we suggest several online solving algorithms for the defender team that allocate selected vertices to be occupied so that attacker agents cannot pass into the protected area. We identify suitable vertex allocation strategies for diverse types of APP instances and test them thoroughly.

1.1 Related Work

Movements of agents at low reactive level are assumed to be planned by some *cooperative path-finding - CPF* (*multi-agent path-finding* - MAPF) (Silver, 2005; Ryan, 2008; Wang and Botea, 2011) algorithm where agents of own team cooperate while opposing agents are considered as obstacles. In CPF the task is to plan movement of agents so that each agent reaches its unique target in a conflict free manner.

There exist multiple CPF algorithms both complete and incomplete as well as optimal and suboptimal under various objective functions.

A good trade-off between the quality of solutions and the speed of solving is represented by suboptimal/incomplete search based methods which are derived from the standard $A*$ algorithm. These methods include *LRA**, *CA**, *HCA**, and *WHCA** (Silver, 2005). They provide solutions where individual paths of agents tend to be close to respective shortest paths connecting agents' locations and their targets. Conflict avoidance among agents is implemented via a so called reservation table in case of *CA**, *HCA**, and *WHCA** while *LRA** relies on replanning whenever a conflict occurs. Since our setting in APP is inherently suitable for a replanning, the algorithm *LRA** is a candidate for underlying CPF algorithm for APP. Moreover, *LRA** is scalable for large number of agents.

Aside from CPF algorithms, systems with mobile agents that act in the adversarial manner represent another related area. These studies often focus on patrolling strategies that are robust with respect to various attackers trying to penetrate through the patrol path (Elmaliach et al., 2009).

Theoretical or empirical works related to APP also include studies on pursuit evasion (Hespanha et al., 1999; Vidal et al., 2002) or predator-prey (Benda et al., 1986; Haynes and Sen, 1995) problems. The Tileworld (Pollack and Ringuette, 1990) also provides an experimental environment to evaluate planning and scheduling algorithms for a team of agents. A major difference between these works and the concept of APP is that, unlike the previous works, we assume the agents in each team perform CPF algorithms, which provide a new foundation of team architecture.

1.2 Preliminaries

The environment is modeled by an undirected unweighted graph $G = (V, E)$. We restrict the instances to 4-connected grid graphs with possible obstacles. The team of attackers and defenders is denoted by $A = \{a_1, \ldots, a_m\}$ and $D = \{d_1, \ldots d_n\}$, respectively. Continuous time is divided into discrete time steps. Agents are placed in vertices of the graph at each time step so that at most one agent is placed in each vertex. Let $\alpha_t : A \cup D \to V$ be a uniquely invertible mapping denoting configuration of agents at time step t. Agents can wait or move instantaneously into adjacent vertex between successive time steps to form the next configuration α_{t+1}. Abiding by the following movement rules ensures preventing conflicts:

- An agent can move to an adjacent vertex only if the vertex is empty, or is being left at the same time step by another agent
- A pair of agents cannot swap along a shared edge
- No two agents enter the same adjacent vertex at the same time

We do not assume any specific order in which agents perform their conflict free actions at each time step. Our experimental implementation moves all attackers prior to moving all defenders at each time step. The mapping $\delta^A : A \to V$ assigns a unique target to each attacker. The task in APP is to find a strategy of movement for defender agents so that the area specified by δ^A is protected.

We state APP as a decision problem as follows:

Definition 1. *The decision APP problem: Given an instance* $\Sigma = (G, A, D, \alpha_0, \delta^A)$ *of APP, is there a strategy of movement for the team D of defenders, so that agents from the team A of attackers are prevented from reaching their targets defined by* δ^A.

In many instances it is not possible to protect all targets. We are therefore also interested in the optimization variant of the APP problem:

Definition 2. *The optimization APP problem: Given an instance* $\Sigma = (G, A, D, \alpha_0, \delta^A)$ *of APP, the task is to find a strategy of movement for the team D of defenders such that the number of attackers in A that reach their target defined by* δ^A *is minimized.*

2 THEORETICAL PROPERTIES

APP is a computationally challenging problem as shown in the following analysis. In order to study theoretical complexity of APP, we need to consider the decision variant of APP. Many game-like problems are PSPACE-hard, and APP is not an exception. We reduce the known problem of checking validity of Quantified Boolean Formula (QBF) to it.

The technique of reduction of QBF to APP is inspired by a similar reduction of QBF to ACPF, from which we borrow several technical steps and lemmas (Ivanová and Surynek, 2014). We describe the reduction from QBF using the following example. Consider a QBF formula in prenex normal form

$$\varphi = \exists x \forall a \exists y \forall b \exists z \forall c \\ (b \vee c \vee x) \wedge (\neg a \vee \neg b \vee y) \wedge \\ (a \vee \neg x \vee z) \wedge (\neg c \vee \neg y \vee \neg z) \quad (1)$$

This formula is reduced to the APP instance depicted in Fig. 4. Let n and m be the number of variables and clauses, respectively. The construction contains three types of graph gadgets.

For an **existentially** quantified variable x we construct a diamond-shape gadget consisting of two parallel paths of length $m+2$ joining at its two endpoints.

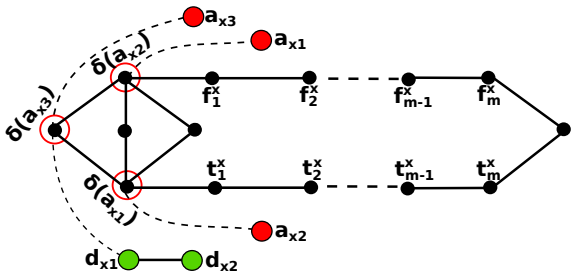


Figure 1: An existentially quantified variable gadget.

There are 4 paths connected to the diamond at specific vertices as depicted in Fig. 1. The gadget further contains three attackers and two defenders with initial positions at the endpoints of the four joining paths. The vertices in red circles are targets of specified attackers. The only chance for defenders d_{x1} and d_{x2} to prevent attackers a_{x3} and a_{x1} from reaching their targets is to advance towards the diamond and occupy $\delta_A(a_{x3})$ by d_{x2} and either $\delta_A(a_{x1})$ or $\delta_A(a_{x2})$ by d_{x1}.

For every **universally** quantified variable a there is a similar gadget with a defender d_{a1} and an attacker a_{a1} whose target $\delta^A(a_{a1})$ lies at the leftmost vertex of the diamond structure (see Fig. 2). The defender has to rush to the attacker's target and occupy it, because otherwise the target would be captured by the attacker.

Moreover, there is a gadget in two parts for each **clause** C depicted in Fig. 3. It contains a simple path

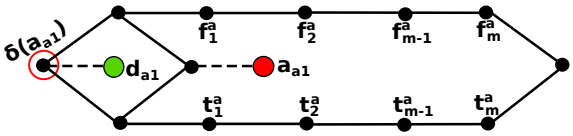


Figure 2: A universally quantified variable gadget.

p of length $\lfloor n/2 \rfloor + 1$ with a defender d_C placed at one endpoint. The length of p is chosen in order to ensure a correct time of d_C's entering to a variable gadget, so that gradual assignment of truth values is simulated. E.g., if a variable occurring in C stands in the second $\forall\exists$ pair of variables in the prefix (the first and last pair is incomplete), then p is connected to the corresponding variable gadget at its second vertex. The second part of the clause is a path of length k, with one endpoint occupied by attacker a_C whose target $\delta^A(a_C)$ is located at the other endpoint. The length k is selected in a way that the target $\delta^A(a_C)$ can be protected if the defender d_C arrives there on time, which can happen only if it uses the shortest path to this target. If d_C is delayed by even one step, the attacker a_C can capture its target. These two parts of the clause gadget are connected through variable gadgets.

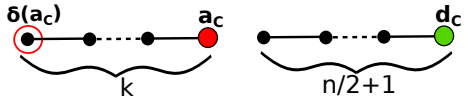


Figure 3: A clause gadget.

The connection by edges and paths between variable and clause gadgets is designed in a way that allows the agents to synchronously enter one of the paths of the relevant variable gadget. A gradual evaluation of variables according to their order in the prefix corresponds to the alternating movement of agents. A defender d_C from clause C moves along the path of its gadget, and every time it has the opportunity to enter some variable gadget, the corresponding variable is already evaluated.

If there is a literal in φ that occurs in multiple clauses, setting its value to true causes satisfaction of all the clauses containing it. This is indicated by a simultaneous entering of affected agents to the relevant path. Each clause defender d_C has its own vertex in each gadget of a variable present in C, at which d_C can enter the gadget. This allows a collision-free entering of multiple defenders into one path of the gadget.

Theorem 1. *The decision problem whether there exists a winning strategy for the team of defenders, i.e. whether it is possible to prevent all attackers from reaching their targets in a given APP instance is PSPACE-hard.*

Proof. Suppose φ to be valid. To better understand validity of φ we can intuitively ensure that variables

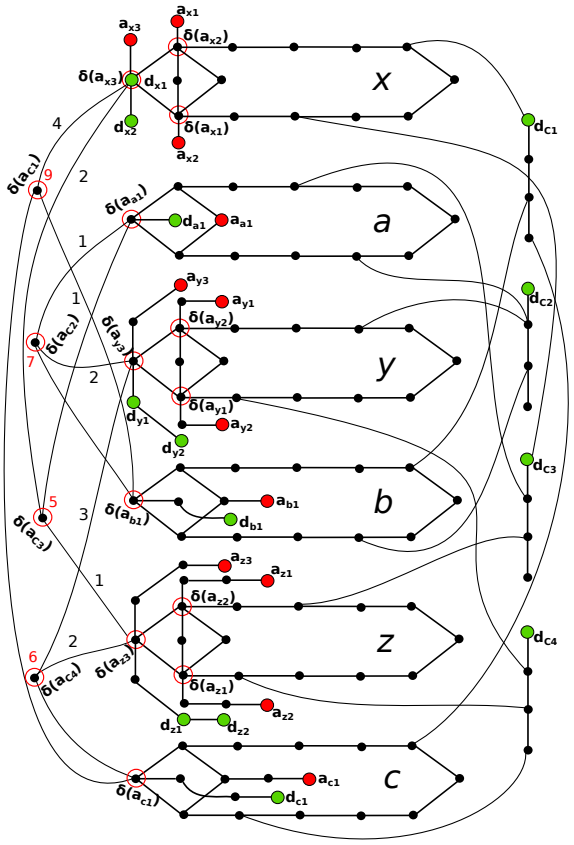


Figure 4: A reduction from TQBF to APP. Black points represent unoccupied vertices. If two points are connected by a line without any label, it means there is an edge between them. A line with a label k indicates that the two points are connected by a path of k internal vertices. Initial positions of attackers and defenders are represented by red and green nodes, respectively. A red circle around a node means that the node is a target of some attacker. For simplicity we do not fully display the second part of the clause gadget. Instead, there is a red number near the target of a clause gadget that indicates the distance of the attacker aiming to that target. A vertex with an agent is labeled by the agent's name. Labels of targets specify the associated agents.

are assigned gradually according to their order in the prefix. For every choice of value of the next ∀-variable there exists a choice of value for the corresponding ∃-variable so that eventually the last assignment finishes a satisfying valuation of φ. The strategy of assigning ∃ variables can be mapped to a winning strategy for defenders in the APP instance constructed from φ. Every satisfying valuation guides the defenders towards vertices resulting in a position where all targets are defended. Every time a variable is valuated, another agent in the constructed APP instance is ready to enter the upper path, if the variable is evaluated as true, or the lower path, otherwise. Note that the vertices on the paths are labeled as t and f indicating the truth value that is simulated by passing through a path. When the evaluated variable x is existentially quantified, the defender d_{x1} enters the upper or lower path. In case of universally quantified variable a, the entering agent is the attacker a_{a1}. Since the valuation satisfies φ, every clause C_j has at least one variable q causing the satisfaction of C_j. That is modeled by the situation where defenders d_{q1} and d_{Cj} meet each other in one of the diamond's paths, which enables either the defender d_{q2} (in case q is existentially quantified) or d_{q1} (in case q is universally quantified) to advance towards the target $\delta^A(d_{C1})$. The situation for an existentially quantified variable is explained by Fig. 5.

Whenever there exists a winning strategy for the constructed APP instance, the defenders must arrive in all targets on time. This is possible only if variable defenders and clause defenders meet on one of the paths in a diamond gadget, and only if all defenders use the shortest possible paths. The variable agents' selection of upper or lower paths determines the evaluation of corresponding variables. An advancement of variable and clause defenders that leads to meeting of the defenders at adjacent vertices, and a subsequent protection of targets indicates that the corresponding variable causes satisfaction of the clause. □

3 DESTINATION ALLOCATION

Solving APP in practice is a challenging problem due to its high computational complexity. Our solving approaches are based on a technique called *destination allocation*. The basic idea is to assign a destination vertex to each defender and subsequently use some CPF algorithm modified for the environment with adversaries to lead each defender to its destination. A defender may be allocated to any vertex, including the attackers' targets. Destination allocation can be divided into two basic categories: *single-stage*, where agents are allocated to destinations only once at the beginning, and *multi-stage*, where destinations can be reassigned any time during the agents' course. This work focuses merely on the single-stage destination allocation and uses the LRA* algorithm for controlling agents' movement.

The defenders are initially not allocated to any destination and do not have any information about the intended target of any attacker. However, the defenders have a full knowledge of all target locations in the protected area. The task in this setting is to allocate each defender agent to some location in the graph, so that via its occupation, defenders try to optimize a given objective function.

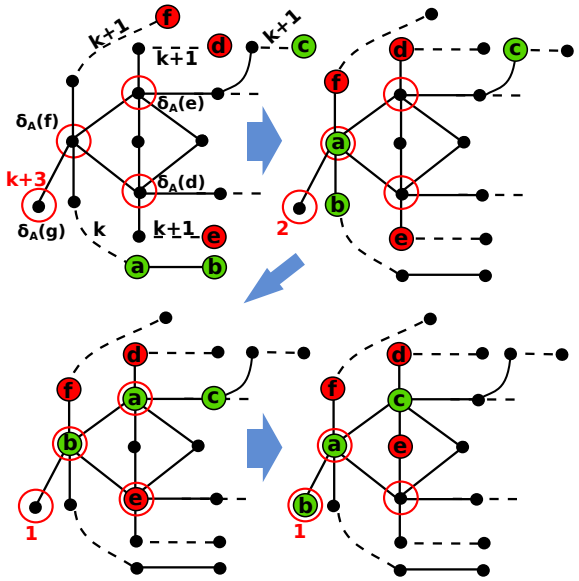


Figure 5: An example of agents' advancement at an existential variable clause. It is defenders' turn in each of the four figures. The top left figure shows the initial position of the agents. The value k depends on the order of the corresponding variable in the prefix. When agents reach the positions in the second figure, the corresponding variable is about to be evaluated which is analogous to defender a entering one of the paths, which prevents the attackers d and e from exchanging their position and reaching the targets. If a moves to the upper path, as happens in the third figure, the agent c from a clause gadget has the opportunity to enter the upper path where the two defenders meet. Attacker e can enter the target $\delta^A(d)$, which is nevertheless not its intended goal. Finally, the defenders can protect all targets by a train-like movement resulting in the position in the last figure. Also note the gradual approaching of the undisplayed attacker g to its target $\delta^A(g)$. The distance between g's current location and the target is indicated by the red number.

We describe several simple destination allocation strategies and discuss their properties. The first two methods always allocate one defender to some attacker's target. The advantage of this approach is that if a defender manages to capture a target, it will never be taken by the attacker. This can be useful in scenarios where the number of defenders is similar to the number of attackers. Unfortunately, such a strategy would not be very successful in instances where the attackers significantly outnumber the defenders. This issue is addressed in the last strategy, in which we attempt to exploit the obstacle structure and succeed even with a smaller number of defenders.

3.1 Random Allocation

For the sake of comparison, we consider the simplest strategy, where each defender agent is allocated to a random target of an attacker agent. Neither the locations of agents nor the underlying grid graph structure is exploited.

3.2 Greedy Allocation

The greedy strategy is a slightly improved approach. Defenders are one by one in an arbitrary order allocated to the closest available target of an attacker.

3.3 Bottleneck Simulation Allocation

Simple target allocation strategies do not exploit the structure of underlying graph. Hence, a natural next step is to occupy by defenders those vertices that would divert attackers from the protected area as much as possible with the help of graph structure. The aim is to successfully defend the targets even with small number of defenders. As our domains are 4-connected grids with obstacles, we can take advantage of the obstacles already occurring in the grid and use them as addition to vertices occupied by defenders. Figure 6 illustrates a grid where the defenders could easily protect the target area even though they are outnumbered by the attackers. Intuitively as seen in the example, hard to overcome obstacle for attacking team would arise if a bottleneck on expected trajectories of attackers is blocked.

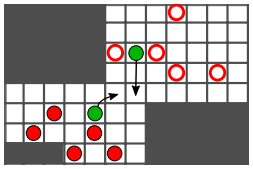


Figure 6: An example of bottleneck blocking. Solid red and green circles represent attackers and defenders, respectively. Empty red circles are the attackers' target locations.

In order to discover bottlenecks of a general shape, we develop the following simulation strategy. This method is based on the assumption that as attackers move towards the targets, vertices close to bottlenecks are entered by the attackers more often than other vertices. This observation suggests to simulate the movement of attackers and find frequently visited vertices. As defenders do not share the knowledge about paths being followed by attackers, frequently visited vertices are determined by a simulation in which paths of attackers are estimated.

There can be several vertices with the highest frequency of visits, so the final vertex is selected by another criterion. The closer a vertex is to the defenders, the better chance the defenders have to capture it before the attackers pass through it. On the grounds of

that, our implementation selects the vertex of maximum frequency with the shortest distance to an approximate location of defenders.

After obtaining such a frequently visited vertex, we then search its vicinity. If we find out that there is indeed a bottleneck, its vertices are assigned to some defenders as their destinations. Under the assumption that the bottleneck is blocked by defenders, the paths of attackers may substantially change. For that reason we estimate the paths again and find the next frequent vertex of which vicinity is also explored and so on. The whole process is repeated until all available defenders are allocated to a destination, or until no more bottlenecks are found. The high-level description of this procedure is expressed by Algorithm 1. The input of the algorithm is the graph G and sets

Data: $G = (V,E)$, D, A
Result: Destination allocation δ^D
$D_{\text{available}} = D$; // Defenders to be allocated
$F = \emptyset$; // Set of forbidden locations
δ'_A = Random guess of δ_A;
while $D_{available} \neq \emptyset$ **do**
  for $a \in A$ **do**
    $p_a = \text{shortestPath}(\alpha_0(a), \delta'_A(a), G, F)$;
  end
  $f(v) = |\{p_a : a \in A \wedge v \in p_a\}|$
  $w \in \arg\max_{v \in V} f(v)$;
  $B = \text{exploreVicinity}(w)$;
  if $B \neq \emptyset \vee |D| < |B|$ **then**
    D' such that $D' \subseteq D_{\text{available}}, |D'| = |B|$;
    assignToDefenders(B, D');
    $D_{available} = D_{\text{available}} \setminus D'$;
    $F = F \cup B$
  else
    break ;
  end
end
assignToRandomTargets($D_{available}$);

Algorithm 1: Bottleneck simulation procedure.

D and A of defenders and attackers, respectively. During the initialization phase, we create the set $D_{available}$ of defenders that are not yet allocated to any destination. Next, we create the set F of so called forbidden nodes. The following step takes attackers one by one and every time makes a random guess which target is an attacker aiming for, resulting in the mapping δ'. The algorithm then iterates while there are available defenders. In each iteration, we construct a shortest path from each attacker a between its initial position $\alpha_0(a)$ and its estimated target location $\delta'(a)$. A vertex w from among the vertices contained in the highest number of paths is then selected, and its surroundings is searched for bottlenecks. If a bottleneck is found, the set of vertices B is determined in order to block the bottleneck. The set D' contains a sufficient number of available defenders that are subsequently allocated to the vertices in B. Agents from D' are no longer available, and vertices from B are now forbidden. The pathfinding in the following iterations will therefore avoid vertices in B. If no bottleneck is found, it is likely that agents have a lot of freedom for movement, and blocking bottlenecks is not a suitable strategy. The loop is left and the remaining available agents are assigned to random targets from $T_{available}$.

The search of the close vicinity of a frequently used vertex w is carried out by an expanding square centered at w. We start with distance 1 from w and gradually increase this value[1] up to a certain limit. In every iteration we identify the obstacles in the fringe of the square and keep them together with obstacles discovered in previous iterations. Then we check whether the set of obstacles discovered so far forms more than one connected components. If that is the case, it is likely that we encountered a bottleneck. We then find the shortest path between one connected component of obstacles and the remaining components. This shortest path is believed to be a bottleneck in the map, and its vertices are assigned to the available defenders as their new destinations.

In order to discover subsequent bottlenecks in the map, we assume that the previously found bottlenecks are no longer passable. They are marked as forbidden and in the next iteration, the estimated paths will not pass through them. The procedure *shortestPath* returns the shortest path between given source and target, that does not contain any vertices from the set F of forbidden locations.

In this basic form, the algorithm is prone to finding "false" bottlenecks in instances with an indented map that contains for example blind alleys. It is possible to avoid undesired assigning vertices of false bottlenecks to defenders by running another simulation which excludes these vertices. If the updated paths are unchanged from the previously found ones, it means that blocking of the presumed bottleneck does not affect the attackers movement towards the targets, and so there is no reason to block such a bottleneck.

4 EXPERIMENTAL EVALUATION

Experimental evaluation is focused on competitive comparison of suggested destination allocation strategies with respect to the objective 2. - maximization

[1]Two locations are considered to be in distance 1 from each other if they share at least one point. Hence, a location that does not lie on the edge of the map has 8 neighbors.

of the number of locations not captured by attackers within a given time limit.

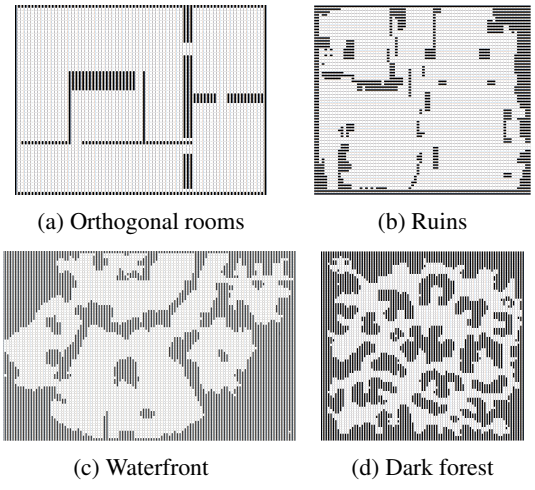
Our hypothesis is that the random strategy would perform as worst since it is completely uninformed. All the simple strategies are expected to be outperformed by more advanced bottleneck simulation.

We implemented all suggested strategies in Java as an experimental prototype. Our testing scenarios use maps of various structures and initial configurations of agents. Our choice of testing scenarios is focused on comparing studied strategies and discovering what factors have a significant impact on their success.

As the following sections show, different strategies are successful in different types of instances. It is therefore important to design the instances with a sufficient diversity, in order to capture strengths and weaknesses of individual strategies.

4.1 Instance Generation and Types

The instances used in the practical experiments are generated using a pseudo random generator, but in a controlled manner. An instance is defined by its map, the ratio $|D| : |A|$ and locations of individual defenders, attackers and their targets. These entries form an input of the instance generation procedure. Further, we select rectangular areas inside which agents of both teams and the attackers' targets are placed randomly. We use 4 maps with increasingly complicated obstacle structure depicted in Fig. 7. Each map size is in the order of thousands of vertices.



(a) Orthogonal rooms (b) Ruins

(c) Waterfront (d) Dark forest

Figure 7: Maps.

In the main set of experiments, each map is populated with agents of 3 different $|D| : |A|$ ratios, namely $1 : 1$, $1 : 2$ and and $1 : 10$, with fixed number of attackers $|A| = 100$. Each of these scenarios is further divided into two types reflecting a relative positions of attackers and defenders. The type *overlap* assumes that the rectangular areas for both teams have an identical location on the map, while the teams in the type *separated* have distinct initial areas. The maximum number of agents' moves is set to 150 for each team.

Note that the individual instances are never completely fair to both teams. It is therefore impossible to make a conclusion about a success rate of a strategy by comparing its performance on different maps. The comparison should always be made by inspecting the performance in one type of instance, where we can see the relative strength of the studied algorithms.

4.2 Results

The performed experiments compare random, greedy, and simulation strategy in different instance settings. Each entry in Table 1 is an average number of attackers that reached their targets at the end of the time limit. The average value is calculated for 10 runs in each settings, always with a different random seed. Random and greedy strategies have very similar results in all positions and team ratios. It is apparent and not surprising that with decreasing $|D| : |A|$ ratio, the strength of these strategies decreases. The simulation strategy gives substantially better results in all settings. Also note that in case of overlapping teams, the simulation strategy scores similarly in all $|D| : |A|$ ratios.

Table 1: Average number of agents that eventually reached their target in the map Orthogonal rooms.

Team position	$\|D\| : \|A\|$	RND	GRD	SIM
Overlapped	1:1	40.4	49.2	21.0
	1:2	56.7	56.5	20.8
	1:10	67.8	64.7	24.7
Separated	1:1	39.0	40.7	10.3
	1:2	57.7	50.1	13.3
	1:10	78.5	69.9	30.2

Table 2 contains results of an analogous experiment conducted on the map Ruins. The random strategy performs well in instances with many attackers. The dominance of the simulation strategy is apparent here as well.

Maps Waterfront and Dark forest contain very irregular obstacles and many bottlenecks, and are therefore very challenging environments for all strategies. In the Dark forest map, random and greedy methods are more suitable than the simulation strategy in instances with equal team sizes, as oppose to the scenarios with lower number of defenders, where the bottleneck simulation strategy clearly leads. In the separated scenario, the simulation strategy is even worse

Table 2: Average number of agents that eventually reached their target in the map Ruins.

Team position	$\|D\| : \|A\|$	RND	GRD	SIM
Overlapped	1:1	36.8	49.4	17.7
	1:2	80.0	63.5	33.0
	1:10	92.5	88.9	58.2
Separated	1:1	9.5	33.6	11.8
	1:2	47.6	34.4	11.8
	1:10	85.6	85.9	14.7

in all tested ratios (see Tab. 3 and Tab. 4). This behavior can be explained by the fact that occupying all relevant bottlenecks in such a complex map is harder than occupying targets in the protected area. In contrast, bottlenecks in the Waterfront map have more favourable structure, so that those relevant for the area protection can be occupied more easily.

Table 3: Average number of agents that eventually reached their target in the map Waterfront.

Team position	$\|D\| : \|A\|$	RND	GRD	SIM
Overlapped	1:1	32.0	41.7	37.1
	1:2	60.6	63.8	39.8
	1:10	77.8	72.9	51.7
Separated	1:1	15.8	19.3	10.7
	1:2	46.4	37.6	9.8
	1:10	75.3	65.5	14.9

Table 4: Average number of agents that eventually reached their target in the map Dark forest

Team position	$\|D\| : \|A\|$	RND	GRD	SIM
Overlapped	1:1	21.6	37.9	48.8
	1:2	53.7	42.6	37.8
	1:10	60.9	51.9	38.4
Separated	1:1	35.3	35.9	61.5
	1:2	40.6	41.3	59.6
	1:10	65.1	67.0	66.0

5 CONCLUDING REMARKS

We have shown the lower bound for computational complexity of the APP problem, namely that it is PSPACE-hard. Theoretical study of ACPF (Ivanová and Surynek, 2014) showing its membership in EXPTIME suggests that the same upper bound holds for APP but it is still an open question if APP is in PSPACE. In addition to complexity study we designed several practical algorithms for APP under the assumption of single-stage vertex allocation. Performed experimental evaluation indicates that our *bottleneck simulation* algorithm is strong even in case when defenders are outnumbered by attacking agents. Surprisingly, our simple *random* and *greedy* algorithms turned out to successfully block attacking agents provided there are enough defenders.

For future work we plan to design and evaluate algorithms under the assumption of multi-stage vertex allocation. As presented algorithms have multiple parameters we also aim on their optimization. Another generalization motivated by practical applications in robotics is APP with communication maintenance.

REFERENCES

Agmon, N., Kaminka, G. A., and Kraus, S. (2011). Multi-robot adversarial patrolling: Facing a full-knowledge opponent. *J. Artif. Intell. Res.*, 42:887–916.

Benda, M., Jagannathan, V., and Dodhiawala, R. (1986). On optimal cooperation of knowledge sources - an empirical investigation. Technical Report BCS–G2010–28, Boeing Advanced Technology Center.

Elmaliach, Y., Agmon, N., and Kaminka, G. A. (2009). Multi-robot area patrol under frequency constraints. *Ann. Math. Artif. Intell.*, 57(3-4):293–320.

Haynes, T. and Sen, S. (1995). Evolving beharioral strategies in predators and prey. In *Proc. of Adaption and Learning in Multi-Agent Systems, IJCAI'95 Workshop*, pages 113–126.

Hespanha, J. P., Kim, H. J., and Sastry, S. (1999). Multiple-agent probabilistic pursuit-evasion games. In *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*, volume 3, pages 2432–2437 vol.3.

Ivanová, M. and Surynek, P. (2014). Adversarial cooperative path-finding: Complexity and algorithms. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, pages 75–82.

Pollack, M. E. and Ringuette, M. (1990). Introducing the tileworld: Experimentally evaluating agent architectures. In *Proc. of the 8th National Conference on Artificial Intelligence*, pages 183–189. AAAI Press.

Ryan, M. R. K. (2008). Exploiting subgraph structure in multi-robot path planning. *J. Artif. Intell. Res.*, 31:497–542.

Silver, D. (2005). Cooperative pathfinding. In *Proc. of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference, 2005*, pages 117–122.

Vidal, R., Shakernia, O., Kim, H. J., Shim, D. H., and Sastry, S. (2002). Probabilistic pursuit-evasion games: theory, implementation, and experimental evaluation. *IEEE Trans. Robotics and Autom.*, 18(5):662–669.

Wang, K. C. and Botea, A. (2011). MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *J. Artif. Intell. Res.*, 42:55–90.