ALGORITHMS

WITH APPLICATIONS

IN ROBOTICS

by

BOGDAN MUNTEANU

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2009

ABSTRACT

Many real world applications which involve computational steps are closely tied to theoretical computer science. In order for these systems to be efficiently deployed and used, a thorough analysis is required in advance. This dissertation deals with several real world problems related to the field of Robotics, which can be mathematically modeled and analyzed. One of these problems is known as the pursuit evasion problem and involves the use of independent automated robots to capture a fugitive hiding in a building or a cave system. This is an extensively studied game theory and combinatorics problem which has multiple variations. It can be modeled as a graph and the goal is to minimize the cost of capturing the evader. We deal with two completely different variations of this problem: a vision based variant, in which the robots have limited vision and thus can react when the fugitive is in line of sight; and a no-vision variant, in which the robots do not have any knowledge about the fugitive.

Another problem we deal with is the problem of neighbor discovery in wireless networks using directional antennas. This is another problem which received a growing interest in the last years. Our approach to solving this problem, as well as the model, is different from the other results that have been previously published in the literature.

Besides modeling and formally analyzing these problems, our focus in this dissertation is to design efficient algorithms that solve them either completely or partially.

DEDICATION

This dissertation is dedicated to my family. They have supported me, encouraged me, advised me. I have been separated from them by an ocean for many years in order to finish this research. It was as difficult for them as it was for me. Thank you.

ACKNOWLEDGMENTS

I would like to take this opportunity and thank, first and foremost, my adviser, Dr. Richard Borie for his extensive support, invaluable feedback, great ideas whenever I got stuck, suggestions for improvement, countless hours of reading my, sometimes, messy and poorly written work; for his belief in me, even at times when I thought I would not make it through; for teaching great algorithms and graph theory courses which, often being quite difficult, made me spend hours on homeworks and learn a lot as a consequence; for being available and helpful every time I needed; for being understanding when I fell behind with my research or congratulating me whenever I made good progress; for doubting every conjuncture or algorithm I came up with, forcing me to formally prove them. Thank you Dr. Borie.

I would like to thank Dr. Grzegorz Malewicz for giving me the chance to start this Ph.D. program. For helping me go through my first years at the University of Alabama, for introducing me to serious research, for being so dedicated to work and for inspiring me.

I am pleased to have the opportunity to thank the Computer Science Department at the University of Alabama, for offering me teaching or research assignments, work without which I would have not been able to complete my Doctoral Program.

I would also like to thank my friends for being supportive; for understanding when I had to work and lost touch with them; for making me have outdoor activities when I was spending all day in my office; for giving me advice and help; for being good friends. Thank you Ashley, Alex, Luigi, Raluca, Sebi and Charles.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 2

CHAPTER 3

CHAPTER 1

INTRODUCTION

Graph Theory and Algorithms are two of the most widely used areas of theoretical computer science since they typically involve formal proofs and results that can be later applied in a variety of computational areas. Completely different fields such as Bioinformatics, Robotics, Computational Chemistry, Databases or Computer Graphics, all use models and results from Graph Theory or Algorithms.

Some problems in the field of Game Theory and Combinatorics are of particular interest to us due to their applicability in Robotics and close connection to Graph Theory. The problem of finding a fugitive, trying to hide in a system of tunnels, is known in the literature as the pursuit evasion problem. It is related to Robotics because the pursuers can be pre-programmed automated robots behaving according to a predefined algorithm which guarantees their success. Even though this is an extensively studied problem, there are still many open questions and plenty of room for new variations or improvement to the existing ones.

Now let us consider a related problem: we want to find the fugitive in a remote system of caves and we first drop the robots near the entrance(s) of the cave. We assume that, initially, after deployment, the robots have no information about each other's location. Hence, a first step is to discover the other robots so communication can be established in a later phase. The problem of discovering all other robots, using directional antennas, is also presented in this dissertation

and it is called the spinning problem. The mathematical model we came up with to represent this problem is new and we hope it will lead to interesting results.

Next, we describe in detail the two problems and some of the previous work that has been done in the area. This chapter is organized as follows: Section 2 contains a detailed description of the Pursuit Evasion Problem, focusing on some variants that are closer to our model. In section 3 we present the Spinning Problem, including some of our results.

1.1 THE PURSUIT EVASION PROBLEM

Graph searching or the pursuit evasion problem (cops and robbers, pursuers and fugitives, searchers and evaders) encompasses a wide variety of combinatorial and game theory problems related to the goal of capturing a fugitive (robber, evader, intruder) residing in a graph, which represents a cave system, a building or a city map using the minimum number of searchers (cops, pursuers, robots).

There are many ways to represent this problem in term of real life applications. Besides trying to find a lost person in a cave system, we could consider the problem of catching a fugitive on a road system or decontaminating a pipe system which has been infected by a poisonous gas.

It can also be seen as a game in which the fugitives try to evade capture by the policemen. The fugitives reside in a system of rooms and corridors (a building). Each application can be represented as a graph, where the corridors, the caves or the streets are the edges and the rooms, the cave and street intersections are the nodes.

Different models can be obtained by modifying the parameters of this problem, either the dynamics of the robots and the fugitives, visibility or conditions of capture. Some variations are closer to the real world applications while others are pure theoretical, but still pose interesting questions from a Computer Science point of view.

In this section we present some of the variants of this problem, focusing on the ones that present more interest to us. The term pursuer or robot is sometimes used to refer to the searcher. Also, the terms evader and fugitive are used interchangeably.

Initially this problem was formulated by Torrence Parsons in 1976 [10] and was inspired by speleologists. They were interested in finding the minimum number of searchers that can come out with a strategy that allows them to rescue a lost explorer independent of his actions. They wanted to guarantee rescue, even if the explorer would unknowingly try to avoid rescue. This is the reason why this problem is widely known as the pursuit evasion problem, or pursuers and fugitives. The same problem was also independently discovered by Nikolai Petrov [20].

Both problems of Parsons and Petrov were proven to be equivalent to the following discrete game played on graphs. Suppose that G is a graph which represents a system of caves where an omniscient fugitive with infinite speed is hidden. Initially the whole graph is contaminated (i.e. the fugitive could be anywhere). The goal of the game is to clear all vertices and all edges, using one or more searchers. We can think of the same problem as having a dangerous gas in a cave system. In this case, the searchers would be the decontamination team.

The player can perform the following moves using the searchers:

- place a searcher on a vertex,
- remove a searcher from a vertex,

- slide a searcher on a vertex along an incident edge until its final placement to a neighboring vertex.

A search program is a finite sequence of moves. An edge $e = \{v, u\}$ of G is cleared if a searcher on v slides along $\{v, u\}$. A previously cleared edge can become contaminated at some moment of searching (recontaminated). That happens if at that moment it can be connected by a path without searchers with a contaminated edge. Intuitively, that means the fugitive can escape to an already cleared area. The goal of the searchers in a search game is to clear all vertices and all edges using a search program. The edge search number of a search program is the maximum number of searchers on the graph during any of its moves. The edge search number, $e(G)$, of a graph G is the minimum edge search number over all the possible search programs on G. Therefore, $e(G)$ is the minimum number of robots needed to clear graph G.

1.1.1 VARIATIONS OF THE PURSUIT EVASION PROBLEM

An important variant of this problem is to consider that an edge is cleared when both of its endpoint vertices are occupied by pursuers. Also, the pursuers cannot slide on the edges. This variant is called node search and the goal is to find the node search number $n(G)$. In mixed searching, the searchers can slide on the edges and an edge is cleared either by a slide or by occupation of its endpoints. It was shown in [3] that all these variants are very similar. In fact, they can only differ by a very small constant. We present this in more detail later in the current section.

Some other versions of the pursuit evader problem do not allow fugitives to be placed on edges, but only on the vertices. This can be useful when, for example, one is trying to find a program residing on a computer network. It does not make sense to search the program on the links between computers, but only on the actual machines (vertices). Sometimes, in the literature, the term searching is used for these variants, while the sweeping is used when fugitives can also reside on edges [9]. This can be confusing at times especially since this problem appears under different names, different conditions and restrictions in the literature.

Note that in our survey we only consider the case when the evaders can also hide on the edges, and we only use the term searching a graph.

In the previous variants, the fugitive's location is unknown to the pursuers (i.e. it is invisible). Hence the search strategy is independent of the fugitive's moves: it must work no matter what he does. The case when the evader is visible was also studied in [4] and was named the helicopter cop-and-robber game. Visibility can be tuned so we can have only some visible nodes, while in the rest the fugitive can be invisible.

Another case is when the evader moves only when a pursuer is going to occupy the vertex he currently resides in (i.e. lazy evader) [21]. Also, the case when we have multiple fugitives has been studied [22].

A variant that was not so extensively studied is when G is a digraph. The case when fugitives can move both ways on arcs, while the searchers must obey the orientation of the arc has been studied [24]. Also, the opposite case, or the case when both fugitives and pursuers must follow the arc orientation were analyzed [23]. The reader can see that combining node, edge or mixed searching with the 3 cases on digraphs result in multiple variants of the problem.

Even though it was proved there is an equivalence between lazy-but-invisible and notlazy-but-visible problems for unidirectional graphs, this equivalence does not hold for directed graphs [25].

The pursuit-evasion problem was also studied on hypergraphs, which is a graph where edges can connect any number of vertices, and was renamed the marshals and robbers game [26]. In this variant, the robbers are visible and the marshals can place their men on any hyperedge.

Another parameter that was introduced in [27] is the connectivity parameter. This means that clearing edges should be done in a way such that at all time, the cleared edges are connected. Cost of connectivity is defined as the ratio between the required number of searchers with the connectivity requirement and the number of searchers without.

In all the variants presented above, the goal was to minimize the number of searchers. Alternative costs have been analyzed: when we consider the maximum number of steps a vertex is occupied by a pursuer or the sum of all the pursuers present on the graph over all steps of a search strategy.

Even though they were not extensively analyzed, we could consider the total distance the pursuers have to move or the total time until the fugitive is found as alternative costs of a search strategy. In some variants, analyzed by Borie et al. [28], multiple searchers are needed to clear an edge or even to guard a node.

There are also a series of games, introduced by Petrov [29], in which the speeds are restricted for both the evader and the pursuers.

Variations in which the searchers have visibility radius have also been studied. In this case, the fugitive is caught if he is within a certain distance of a pursuer. Searching on grids has also been considered. Here a fugitive is visible if it is on the same "street" as a pursuer.

The pursuit-evasion problem on a grid was first analyzed by Sugihara and Suzuki both in the vision-based model and in the full vision model (the pursuers know the location of the fugitive at all times) [34, 35]. In [35] the authors show it is possible to capture an arbitrary fugitive using 4 pursuers with a maximum speed of 1. This result is improved in [36] by Dumitrescu et al. which show that 3 pursuers are sufficient. Also, in [36] the authors present several randomized algorithms for different variations of the problem, by using one pursuer with a maximum speed greater than 3, or two pursuers slightly faster than the fugitive. Dumitrescu et al. show that when the evader is "passive", a single pursuer slightly faster than the evader or two pursuers with the same speed as the evader can successfully capture the fugitive. However, these randomized algorithms have a probability of capture lower than 1. Dawes [37] showed that one pursuer with a maximum speed of n can see the evader, even if the evader has full knowledge of the pursuer's movement. Additional results for the vision-based pursuit evasion problem on a grid have been obtained by Tanaka [38] and Neufeld [39]. Our results in Chapter 2 are related to this variant.

Counteraction means that when a robber moves on a vertex occupied by a cop, the cop is killed. This variant was studied by Petrov in [5].

1.1.2 DEFINITIONS AND COMPLEXITY RESULTS

A search strategy in which an already searched node (and implicitly, an already searched edge) is never searched again is called monotonic (i.e. the fugitive cannot enter an area from where it has been previously expelled). Monotonicity is connected to some of the most complex results in the area of graph searching. This property ensures that the pursuit-evasion problem is in the class NP.

Monotonicity was first proven by LaPaugh [1, 2], after the proof that graph searching on general graphs is NP-Hard. LaPaugh's result was then used to prove monotonicity of node searching by Kirousis and Papadimitriou in [6]. A simpler and more important proof came from Bienstock and Seymour in [3]. They proved monotonicity for mixed searching and showed that node and edge searching can be reduced to mixed searching. Monotonicity was also studied in the case of visible evaders and on some variations of digraphs [31]. It was also proven that monotonicity does not hold for connected search, or for the Marshals game [30].

Next we present the definition of the pursuit-evasion problem as was initially introduced by Parsons followed by other definitions and results that we find interesting and that are related to our work.

Definition. Let G be a finite connected graph embedded in R^3 and let Γ be its straight line representation. A family $\Pi = \{x_i | \ 1 \leq i \leq k\}$ of continuous functions $x_i : [0, \infty) \rightarrow \Gamma$ is a search program for Γ if, for every continuous function $y : [0, \infty) \rightarrow \Gamma$ there exists a $t(y) \in [0, \infty)$ and an $i \in \{1, 2, \ldots, k\}$ such that $y(t(y)) = x_i(t(y))$. We can think of $y(t)$ as the position at time t of the evader in a cave represented by Γ, $x_i(t)$ is the position of the ith searcher, while $t(y)$ is the moment when the evader is caught. The minimum k that guarantees the existence of a search program on Γ was called the search number of Γ. [10, 11].

We remind the reader that a graph G with vertices $x_1,\ldots, x_n$ can be embedded in R^3 as follows: for each $k=1..n$, let y_k be the point (k, k^2, k^3) in R^3. y_i and y_j are connected by a straight line segment in R^3 if (x_i, x_j) is an edge in G. Also note that the curve $f(k) = (k, k^2, k^3)$ in R^3 cannot have 4 coplanar points which proves that no two straight line segments intersect. This allows us to embed G in R^3. For a more detailed proof of this result the reader may be interested in looking at [12].

However, the definition above does not make sense when we consider the case of reflexive multigraphs. A reflexive multigraph is a graph in which self loops and multiple edges are allowed. Since self loops and multiple edges cannot be embedded in R^3 as line segments. Loops can be represented as closed curves while multiple edges can be drawn as internally disjoint simple curves. Obviously, these will not intersect at any interior point. We need a new model for reflexive graph searching which is beyond the scope of this survey. However, such a model can be found in [9].

It is obvious that the previous definition refers to edge searching. Next we modify this definition to define the node searching strategy as described in [9].

Definition. Let G be a finite connected graph embedded in R^3 and let Γ be its straight line representation. A family $\Pi = \{x_i | \ 1 \leq i \leq k\}$ of functions $x_i : [0, \infty) \rightarrow \Gamma$ is a node search program for Γ if, for every continuous function $y : [0, \infty) \rightarrow \Gamma$ there exists a $t(y) \in [0, \infty)$ and $i, j \in \{1, 2, \ldots, k\}$ such that $y(t(y))$ lies on the edge joining $x_i(t(y))$ and $x_j(t(y))$.

One of the first results, which is quite trivial, in the area of graph searching is that for any finite graph, a finite number of searchers is needed.

Theorem 1.1.1 If G is a finite connected graph, then for any searching model, there exists a search strategy using a finite number of pursuers.

Proof.

Place one pursuer on every node of the graph. If we consider node searching, then the evader is found immediately. If we consider edge searching, we need an additional searcher to clear all the edges in the graph.

Note that the previous proof stands even if we consider reflexive multigraphs.

Also, given a graph G, we remind the reader that $e(G)$ represents the edge number of G, $n(G)$ the node search number while $m(G)$ represents the mixed search number of G.

The following inequalities have been previously proven in [9].

Theorem 1.1.2 If X is a reflexive multigraph, then

1. $e(G) - 1 \leq n(G) \leq e(G) + 1$
2. $e(G) - 1 \leq m(G) \leq e(G)$
3. $n(G) - 1 \leq m(G) \leq n(G)$

We can get the intuition behind this proof by looking at a couple of extreme cases. If G is a path of length 1, then $e(G) = m(G) = 1$, while $n(G) = 2$. If G is a multigraph with 2 nodes and more than 3 edges between them, then $e(G)$ is 3, while $n(G) = m(G) = 2$.

Definition. We say a graph G is k-searchable if $e(G) \leq k$.

Definition. If $H \subseteq G$ is the subgraph where the intruder cannot reside, then $G \backslash H$ is defined to be the intruder territory (or the contaminated area).

The following definitions were first introduced by Lapaugh [2] in 1993.

Definition. During the searching of any graph, for any time step, there are three types of vertices: clear, contaminated and partially clear. A vertex is called clear if all its incident edges

are cleared, contaminated if all its incident edges are contaminated, and partially clear if there are both contaminated and clear edges incident to it [2].

Definition. A clearing move clears an edge and reaches a state that satisfies the following two conditions [2]:

I. no clear or contaminated vertex contains a searcher;

II. no vertex contain more than one searcher.

LaPaugh stated that clearing moves are of the following two types for an edge (x, y):

1. $m^+(x, y)$: Vertex x has at least two incident contaminated edges. If vertex x contains no searcher, we first place a searcher on x. Then move a second searcher from x to y along the edge (x, y). Remove searchers from y if vertex y violates conditions (I) or (II).

2. $m^-(x, y)$: Vertex x has only one incident contaminated edge. If vertex x contain no searcher, we first place a searcher on x. Then move the searcher at x to y along the edge (x, y). Remove searchers from y if vertex y violates conditions (I) or (II).

Note that a clearing move clears exactly one edge and it does not cause any recontamination, meaning that any edge-search strategy composed of clearing moves has $|E|$ clearing moves. LaPaugh proved the following theorem which is one of the most famous results related the pursuit evasion problem. It implies that recontamination cannot help:

Theorem 1.1.3 There is an optimal edge-search strategy composed of clearing moves.

Even if trees are one of the simplest forms of graphs, finding the search number on trees is not trivial. We present next some of the most interesting results on clearing trees [7,9].

The authors use divide-and-conquer to find the minimum number of pursuers required to clear a tree *T*. They first divide *T* into two subtrees of smaller order, then recursively compute the search number and certain corresponding information for each subtree, and then they merge this information for the subtrees to produce a solution for *T*. Each time a tree is divided into two subtrees, the two subtrees share only one common vertex in *V*. Using the above ideas, Megiddo et al. [7] proved the following theorem.

Theorem 1.1.4 The search number of a tree can be computed in linear time.

However, knowing the number of searchers needed to clear a tree is not the same problem as producing a strategy to capture any intruder in the tree. The following result gives us information about what is required in terms of producing a strategy. Some of this was discussed in [7].

Theorem 1.1.5 A search strategy for a tree *T*, using *e(T)* searchers, can be computed in *O(n log n)* time.

The problem of edge searching on general graphs was proved to be NP-Complete by Megiddo et al. [7]. Also, the authors showed the problem is solvable in linear time on trees. Proving that node searching is also NP-Complete was done by reducing from edge searching provided in [6]. Since it was already shown that both edge searching and node searching can be reduced to mixed searching, it results that mixed searching is NP-Complete.

It was also shown that the edge searching problem remains NP-Complete on planar graphs with maximum degree three [13].

To prove the problem is NP-Hard, Megiddo et al. [7] used a reduction from the Min-Cut into Equal-Cardinality Subsets problem which is known to be NP-Complete.

Min-Cut into Equal-Cardinality Subsets Problem:

Input: Graph $G(E, V)$ of even degree, $k > 0$.

Question: Is there a partition of V into two subsets V_1 and V_2 with $|V_1| = |V_2| = |V|/2$ such that $|\{(u,v) \in E : u \in V_1, v \in V_2\}| \leq k$?

The Pursuit-Evasion Problem:

Input: Graph $G(E, V)$, $k > 0$.

Question: Can G be cleared using $\leq k$ searchers?

The main scheme of showing a certain variant is NP-Complete is by first analyzing its monotonicity and then find a graph parameter equivalent to its monotone version.

The following definitions are given in [32]:

Definitions.

A tree decomposition is the mapping of a graph G into a tree T. Specifically, given a graph $G = (V, E)$, a tree decomposition is a pair (X, T), where $X = \{X_1, ..., X_n\}$ is a family of subsets of V, and T is a tree whose nodes are $X_1, \ldots, X_n$ and satisfy the following properties:

1. The union of all sets X_i equals V. That is, each graph vertex is associated with at least one tree node.
2. For every edge (v, w) in the graph, there is a subset X_i that contains both v and w. That is, vertices are adjacent in the graph only when the corresponding subtrees have a node in common.

3. If X_i and X_j both contain a vertex v, then all nodes X_z of the tree in the (unique) path between X_i and X_j contain v as well. That is, the nodes associated with vertex v form a connected subset of T.

The width of a tree decomposition is the size of its largest set X_i minus one.

The treewidth of a graph G is the minimum width among all possible tree decompositions of G.

A path decomposition is a tree decomposition (X,T) where T is a path

Pathwidth of G is the least integer k such that G has a path decomposition of width k.

Pathwidth is one of these graph parameters which was shown to be equal to the node search minus one [32]. Arnborg et al. [32] also show that the problem of finding the pathwidth of a graph is NP-Complete on general graphs. Hence, pathwidth can be used to show certain variants of graph searching are NP-Complete.

Treewidth is another parameter used in graph searching. It was shown that the variant of notlazy-but-visible fugitives equals the treewidth minus one. Since finding the treewidth is NP-Hard [32], it results this variant of graph searching is also NP-Hard.

The monotone version of the Marshals game was proved to be equivalent to the hypertree-width in [8]. On graphs with vertex degree at most 3 the cutwidth and the edge searching number are equal [33]. Other parameters found to be related to graph searching are bandwidth, proper pathwidth, direct pathwidth, direct treewidth, DAG-width or tree span.

There have been many algorithmic and combinatorial results on various cases of graph searching. Some of these variants are too different from our approach and hence are beyond the scope of this survey.

Note that in order to find the search number of a graph G, one must first find a strategy of clearing G with k searchers and then prove it cannot be done using $k - 1$ searchers. The latter task is a lot more difficult because there are few results which give us good lower bounds on e(G).

We present next some complexity and lower bound results we find more interesting.

The following results were described in [9]:

Proposition 1.1.6 If the graph G has minimum degree $k \geq 3$ then $e(G) \geq k + 1$.

This results also implies that if the graph G is k-connected and $k \geq 3$, then $e(G) \geq k + 1$.

Theorem 1.1.7. If the graph H is a minor of the graph G, then $e(G) \geq e(H)$.

Corollary 1.1.8 If the graph G has clique number $k \geq 4$, then $e(G) \geq k$.

Lemma 1.1.9. For any clique K_n, $n \geq 4$, $e(K_n) = n$.

Lemma 1.1.10 Let K_n be a clique, $n \geq 4$. If $W \subseteq V(K_n)$, $|W| \geq 2$, and all the vertices in $V(K_n) \backslash W$ are guarded, then K_n can be cleared by additional $|W|$ searchers without moving any guarding searchers in $V(K_n) \backslash W$.

Proof. The authors use the following search strategy with additional $|W|$ searchers which does not move searchers guarding vertices $V(K_n) \backslash W$.

1. Place $|W| - 1$ searchers on W except one vertex v. Then clear all contaminated edges in K_n except those incident to v by the remaining free searcher.

2. Since $|W| \geq 2$ there exists a vertex $u \neq v$ in W. First apply a clearing move $m^{-}(u, v)$ to clear the edge (u, v). Then clear all contaminated edges incident to v by the remaining free searcher.

It was shown in [14] that the pursuit evasion problem is NP-Complete on chordal graphs, which are graphs in which each cycle of four or more nodes has a chord. A chord is an edge joining two nodes that are not adjacent in the cycle.

The authors also provide an $O(mn^2)$ – time algorithm for split graphs (graphs in which the vertices can be partitioned into a clique and an independent set), and an $O(m+n)$ – time algorithm for interval graphs.

Since the results on interval graphs present interest to us, we describe them in detail in the following section.

1.1.3 RESULTS ON INTERVAL GRAPHS

Definition. An interval graph is the intersection graph of a set of intervals on the real line. It has one vertex for each interval in the set, and an edge between every pair of vertices corresponding to intervals that overlap.

Lemma 1.1.11. If $G = (V, E)$ is an interval graph with maximum clique size $w \geq 4$, then $e(G) = w$ or $w+1$.

Definition. The neighborhood of u, $N(u)$ is the set of all vertices that are u's neighbors. $N[u]$ is $N(u)$ with the addition of vertex u.

Next we present the algorithm described in [14] for clearing interval graphs:

Input: A family of intervals F.

Output: $e(G)$

1. Sort F to W according to their right endpoints. If there are some intervals with the same right endpoints, then they are sorted according to their left endpoints. The set W contain all the remaining uncleared intervals.

2. *var_es = 0*; (Initially there is no searcher on G)

3. DO WHILE $W \neq \emptyset$

4. Let *u* be the first interval in *W*.

5. Let $U=\{v|\ v \in N(u) \cap W$ and *v* is not guarded$\}$

6. Case 1: $U \neq \emptyset$. Let $v \in U$. (Note that $N[u] \cap W$ is a clique)

7. *var_es = max (var_es, |N[u] ∩ W|)*

8. Place searchers on $U\backslash\{v\}$ and a searcher on *u* if *u* is not guarded.

9. Clear contaminated edges in $G(N[u] \cap W\backslash\{v\})$ by a free searcher.

10. Move the searcher on *u* to *v*.

11. Clear contaminated edges in $G(N[u] \cap W)$ by a free searcher.

12. Case 2: $U = \emptyset$ and *u* has a searcher. (*u* is cleared)

13. Remove the searcher on *u*.

14. Case 3: $U = \emptyset$ and *u* has no searcher.

15. *var_es = max(var_es, |N[u] ∩ W| + 1)*

16. Place one searcher on *u*.

17. Clear *u* by using a free searcher.

18. $W = W\backslash\{u\}$

19. ENDDO

20. *e(G) = var_es*

Definition. A perfect elimination ordering in a graph is an ordering of the vertices of the graph such that, for each vertex *v*, *v* and the neighbors of *v* that occur later than *v* in the order form a clique.

A sketch of the correctness proof for the interval graphs algorithm as described in [14] is presented next.

Since each interval u is scanned according to its right endpoint in non-decreasing order, such an ordering can be shown to be a perfect elimination ordering of G. This means that $N[u] \cap W$ forms a clique of G. Each time a vertex u is removed from W, it is cleared since its incident edges are all cleared and all its neighbors are cleared or guarded by searchers. This condition can be easily proved by induction on the ordering of u. The number of searchers used is recorded in the variable *var_es*. The only situation that the algorithm needs $w+1$ searchers occurs when Case 3 holds and $K = N[u] \cap W$ is a maximum clique of G.

In this case, there are two other vertices s and t such that the algorithm scans s and t before and after u, respectively, $N[s] \cap K = N[t] \cap K = K - \{u\}$, and s, t, u are not adjacent to each other. The existence of s results from the fact that $U = \emptyset$. If t does not exist, then there exists a vertex v in K such that u and v have the same right endpoints, and the left endpoint of v is smaller than that of u. It results that v is scanned (and thus removed from W) before u by the algorithm, which is a contradiction. Therefore $w+1$ searchers are necessary.

For maximum clique size $w \leq 3$, then in Case 3 of the above algorithm, $|N(u) \cap W| \leq 2$. u can be cleared by the following steps: First place a pursuer on one vertex in $N(u) \cap W$, then we move the pursuer to u. Next, move the searcher back to another vertex (if exists) in $N(u) \cap W$. Hence, w searchers are sufficient.

We have the following:

If $w = 2$, then $e(G) = 1$ iff G is a path, otherwise $e(G) = 2$.

If $w = 3$ then $e(G) = 2$ iff $\forall u, v\ |N(u) \cap N(v)| \leq 1$, otherwise $e(G) = 3$.

Theorem 1.1.12 The number of pursuers needed to clear an interval graph can be computed in linear time.

Similar results as the one presented here were independently discovered by us with slightly different characterizations. We show these results and point out the differences at the beginning of Chapter 3.

We stop here with the short survey on the pursuit-evasion problem results. If interested, the reader is encouraged to read about additional results or variations in the extensive reference list we provide at the end of this dissertation.

In chapter 3 of we present some results for graph searching on certain types of graphs. Chapter 2 deals with another variation of the problem, a vision based approach on grids.

1.2 THE SPINNING PROBLEM

Static wireless ad-hoc networks and sensor networks have received an increased interest in the past years, especially due to their applicability. Field operations, rescue operations, habitat monitoring and surveillance are just a few of the numerous applications ad-hoc and sensor networks can be used for. In most applications, after deployment, nodes must first independently discover their neighbors. After the localization phase, these nodes can start communicating among themselves or perform whatever task they were deployed for (in our case, capture the evader). The problem of discovering the location of other nodes is known as localization

problem or neighbor discovery and it is a very important first step in the establishment of a wireless network. Of course, neighbor discovery should be fast and energy efficient, in order to allow subsequent actions to take place in the network.

Omni-directional antennas and directional antennas may seem closely related, but in practice they are quite different. We focus on directional antennas rather than omni-directional antennas because the former have a stronger signal, a greater range, increased performance and reduced interference from unwanted sources.

We want to place an arbitrary number of robots (or battery-powered devices) on a two-dimensional plane. For example we want to drop these devices from an airplane over a field. We do not have control over how the devices, sometimes simply referred to as nodes, get placed, so we assume their location is arbitrary and unknown. The nodes are equipped with directional antennas. All antennas have the same beamwidth, transmission power, frequency channel and modulation technique. Once deployed, the antennas will start spinning at a predefined speed transmitting and receiving signals at the same time. We do not know and we cannot set the initial orientation of the antennas, since the robots are dropped and not carefully deployed. The goal is for each device to determine the location of every other device so that later they can communicate.

We assume that all devices are in each other's range. Every device has a unique ID. Once a node receives a successful transmission from a neighbor, it will record the identity and the location of that node. This can be done by using Angle-Of-Arrival information of the received signal, or by including direction information in the sent packet.

Before dropping the devices, we can set the rotation speed of the antennas. We want to minimize the energy consumption of the devices but also minimize the time until every device

discovers all his neighbors. If the beamwidth is wider and the antennas rotate faster then the energy consumption is higher but also the meeting time may be lower.

Our first question is: Is it possible to set the speeds such that the devices will discover one another? If yes, how can we optimize the total meeting time? Since in real life the speed is quite limited (by current technology), can we find a solution in the case we have an upper bound for the speed we may set? In this research we answer some of these questions.

Substantial work has been done in the area of discovery problems with directional antennas. In this section we present some of the work which is closest to the Spinning Problem.

In [15] the authors present several probabilistic algorithms for neighbor discovery in wireless networks. These algorithms are classified in two groups, Direct-Discovery Algorithms in which nodes discover their neighbors only upon receiving a transmission from them and Gossip-Based Algorithms in which nodes gossip about their neighbor's location information to enable faster discovery. Time is divided in time slots and in every time slot each node transmits in a random direction. The authors' goal is to maximize the probability of a node discovering its neighbors within a given amount of time

In [16, 17] a distributed algorithm for creating a multihop wireless network with a higher lifetime is presented. The lifetime depends on the battery power of the network and on the power consumption for communication. After creating the network using this algorithm, the power consumption will be close to the optimal. The basic idea of the algorithm is that a node u transmits with minimum power p required to ensure that in every cone of degree α around u, there is some node that u can reach with power p. In these papers, energy consumption depends on the range of transmission. The authors are not concerned with the time required to build the network but with the power consumption for the communication when the network is in place.

The idea in [18] is to equip only a small fraction of the nodes of the network with location determination hardware. These nodes, called “anchor nodes”, will act as reference points for location information. The rest of the nodes, called “target nodes”, can use the information from the anchor nodes to estimate their location. The sensor nodes, which are equipped with four directional antennas, will determine their own location by measuring the distance from each anchor node.

In our work we use new assumptions about the model which result in a totally different approach and we focus on minimizing the time until all nodes have discovered their neighbors.

Our results on this problem are presented in Chapter 4

CHAPTER 2

VISION-BASED PURSUIT-EVASION PROBLEM

Our focus in this section is the grid graph, which is an $n \times n$ undirected graph with n^2 vertices with integer coordinates in $[0, n-1] \times [0, n-1]$ and the connecting edges. Every edge of the graph is of unit length. The vertices of the grid G_n are counted from $(0, 0)$ to $(n-1, n-1)$, with the lower left corner being the vertex $(0, 0)$.

Both the pursuers and the fugitive are represented by a moving point on the grid. Two pursuers can occupy the same vertex at one time. We say a fugitive is captured if at some point in time it occupies the same vertex as a pursuer or same location along an edge. The vision of the pursuers is limited to a straight line. Also, the vision distance is n, meaning that the robots can see from any vertex to any other vertex on the same row or column. Nothing is known about the fugitive, it may have omniscient knowledge of the pursuers' location or it may be completely "blind". Our algorithm must work in either case.

We also assume the pursuers can communicate, execute the algorithm, accelerate and stop instantaneously, and their speed does not fluctuate (they either move at maximum speed or are stationary).

A pursuer is said to have *direction detection capability* if it can see in which direction the fugitive moves (left or right) when disappearing from the line of sight. [36]

Distance detection capability allows a pursuer to know the distance between it and the fugitive, whenever the fugitive is in sight. [36]

We show that in the discrete model, one pursuer with a speed of $s+1$ (where s is the speed of the fugitive) is sufficient to capture the fugitive and does that in $O(n)$ time. Or, two

pursuers with a speed of one and direction detection capability can capture a fugitive with speed one in $O(n)$ time.

For the continuous model we show that 2 pursuers with a maximum speed of 2 and direction detection capability are sufficient to capture the evader. In this case, the fugitive has a speed of one. The worst case running time for our algorithm is $O(n^2)$. This is an improvement on the previous 3-pursuers solution found by Dumitrescu et al. [4], with the drawback that in our solution, the pursuers have a speed of 2 instead of 1 and direction detection capability.

The rest of this chapter is organized as follows. The results for the discrete model are presented in the next section while the continuous model is dealt with in Section 2.2. Conclusions and future work are found in Section 3.4.

2.1 THE DISCRETE MODEL

The discrete model implies the pursuer(s) and the fugitive take turns to move. Also, at the beginning of each turn, the players are on vertices of the grid, and not on an edge. Before we move to our results, we present some of the notations used in the proofs.

Throughout Chapter 2, the letters *A* and *B* will be used to refer to pursuers and the letter *Z* for the fugitive.

Definition: The row-distance between *Z* and a pursuer *A*, $rd_t(Z, A)$, is the distance at time *t* between the row *Z* is on and the row *A* is on.

Definition: The column-distance between *Z* and a pursuer *A*, $cd_t(Z, A)$, is the distance at time *t* between the column *Z* is on and the column *A* is on.

We start this section by presenting a small result that we proved, but which we eventually found in a manuscript that was an earlier version of publication [36]. However, Dumitrescu et al. did not publish this theorem in the final version and we independently reached the same result without being aware of the unpublished manuscript. This is a very small Theorem, and we use it mostly as an introduction to our other results.

Theorem 2.1.1 In the discrete vision-based pursuit-evasion problem on a grid, 2 pursuers with a maximum speed of 1 are sufficient and necessary to successfully capture a fugitive with speed 1.

Proof:

Consider the following algorithm where the two pursuers *A* and *B* start at points *(0,0)* and *(1,0)*. The following algorithm will be repeated until capture.

```
If fugitive Z is in A's or B's line of sight on a vertical line
   Then Both A and B move one step to the north.
Else
   If A was the last pursuer to see Z on a vertical line,
        Then both A and B move to the west
   Else A and B move to the east
```

This algorithm will guarantee capture.

Claim a): Let $M = rd_t(Z, A) = rd_t(Z, B)$. After at most n iterations of the above algorithm, M will decrease by at least 1. Also, M never increases.

Proof:

The pursuers will move east or west until they see the fugitive. Since the grid is finite, and we are considering a discrete model, they will eventually see *Z*. Following from the algorithm, they will move north 1, which will decrease *M* by 1.

Claim b): *Z* will never be to the south of the pursuers.

Proof:

Initially, *Z* must be to the north of the pursuers since *A* and *B* start at the lower left corner of the grid. After they see *Z*, the fugitive cannot go to the south of *A* or *B*. For that, *Z* would have to make 3 moves before *A* or *B* makes 1 (see Fig. 2.1.1).

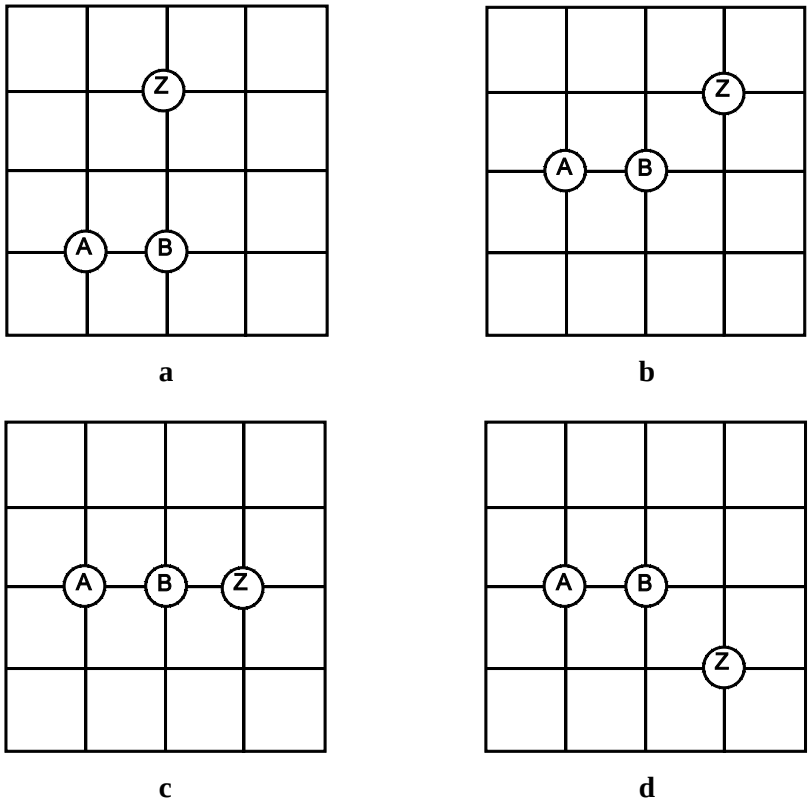


Figure 2.1.1 *Z* will never be south of the pursuers

Claim c): After the initial sighting of *Z*, if *Z* is on the same row as *A* and *B* (i.e. $M = 0$), then *Z* must be either on the same vertex as *A* or on the same vertex as *B* (but not to the east or west of *A* and *B*).

Proof:

After the initial sighting of *Z*, and before being 0, *M* should first be 1. That is because the maximum speed is 1, so the distance gradually decreases, it cannot drop directly to 0.

When $M = 1$, *Z* is either directly to the north of *A* or *B*, or immediately to the east or west (see Fig. 2.1.1b) When *Z* is directly north, the next move will either be the pursuers move north (so $M = 0$, capture) or the fugitive moves east or west, resulting in the position from Fig 2.1.1b.

That is because the pursuers and the fugitive alternate moves.

Next, it is pursuers' turn to move, and they will move east or west, with either *A* or *B* on the same column as *Z*, resulting in the same relative position with *Z* directly north of *A* and *B* and $M = 1$.

Z will keep moving east or west until it is captured, but it can never reach a position like in Fig. 2.1.1c.

From a), b) and c) it follows that the algorithm ends with capture.

Also, it is easy to see that only one pursuer with the same speed as the fugitive cannot guarantee capture. Consider the case of a square in which the fugitive and the evader are positioned like in Fig. 2.1.2. If *A* moves towards *Z* clockwise, then *Z* will also move clockwise and therefore maintain the same distance. Same result happens if *A* moves counterclockwise towards *Z*.

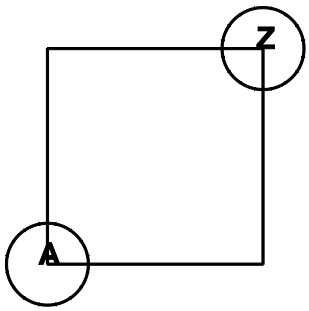


Figure 2.1.2 Cannot capture if both move same direction

Theorem 2.1.2 The algorithm above has a worst case running time of $O(n^2)$.

Proof:

In the worst case, the initial $M = n$. Since after every n steps, M will decrease by 1, after n^2 steps M will be 0. Also, from claim c) it follows that once M is 0, then we have capture. Therefore, the algorithm has a worst case running time of $O(n^2)$.

The following theorem will improve Theorem's 2.1.1 running time, but it will add an additional constraint on the pursuers, requiring that both pursuers are equipped with direction detection capability.

Theorem 2.1.3 In the discrete vision-based pursuit evasion problem on a grid, 2 pursuers with speed of 1 and direction detection capability can capture a fugitive with speed 1 in $O(n)$ time.

Proof:

Consider the following algorithms in which the pursuers A and B start at $(0, 0)$ on the grid:

Algorithm for *A*

```
1. move north until Z in sight east of A
2. IF Z in sight
3.   move east
4. ELSE
5.   move in the direction Z went
6. GOTO 2
```

Algorithm for *B*

```
1. move east until Z in sight north of B
2. IF Z in sight
3.   move north
4. ELSE
5.   move in the direction Z went
6. GOTO 2
```

For any two points *X* and *Y* on the grid, we define $d_t(X, Y) = rd_t(X, Y) + cd_t(X, Y)$

Let $\Delta d(X, Y) = d_{t+1}(X, Y) - d_t(X, Y)$

We call *Q* the point at the northeast corner of the grid located at coordinates $(n-1, n-1)$.

Let $M_t = d_t(A, Z) + d_t(B, Z) + d_t(Z, Q)$

We also define $\Delta M = M_{t+1} - M_t$.

Initially, $M_0 < 4n$

The maximum M_0 occurs when *Z* is at point *Q*.

The following table summarizes the possible deltas after each move of the algorithms:

Table 2.1 Delta values

	Z moves East	**Z moves West**	**Z moves North**	**Z moves South**	**Z is Stationary**
$\Delta d(A, Z)$	0	-2	0	0 or -2	-1
$\Delta d(B, Z)$	0	0 or -2	0	-2	-1
$\Delta d(Z, Q)$	-1	1	-1	1	0
ΔM	-1	-1 or -3	-1	-1 or -3	-2

The entries in rows $\Delta d(A,Z)$ and $\Delta d(B,Z)$ are obtained by analyzing each of steps 1, 3, 5 in the algorithms for *A* and *B*.

For every possible move that *Z* can choose, *M* will decrease by either 1 or 2 or 3. So *Z* must be captured within at most *4n* steps.

Therefore, the running time is *O(n)*.

Theorem 2.1.4 In the discrete vision-based pursuit evasion problem on a grid, 1 pursuer with a speed *s+1* can capture a fugitive with a speed of *s* in *O(n)* time.

Clarification: In this model the fugitive will take *s* steps after which the pursuer will take *s+1* steps.

Proof:

Consider the following algorithm where *A* starts at *(0, n/2)* and initial direction is east.

A will execute the following algorithm *s+1* times during a time unit:

```
1. A moves toward direction
2. IF Z in sight
3.  direction = the direction toward Z
4. GOTO 1
```

Also, during evader's turn, the direction value is updated if *Z* appears in *A*'s line of sight. Since *A* and *Z* move synchronously and take turns *A* will eventually see *Z*, either to the east, north or south.

After each round, the distance between *A* and *Z* decreases by *1*. No matter in which direction *Z* is going, since *A* is faster, it will get closer. Also, *Z* cannot "slip" by *A* undetected (i.e. if *A* moves in the direction where it saw *Z* last, it will either capture *Z* or see it again and change direction). Hence, *A* will eventually capture *Z*.

Analysis of the running time:

Consider the row-distance and the column distance between *A* and *Z* (*rd(A, Z)* and *cd(A, Z)*).

Initially, $rd(A, Z) + cd(A, Z) < 2n$

After every round, the distance decreases by at least 1. Hence, after at most *2n* rounds, $rd(A, Z) + cd(A, Z)$ will be zero.

2.2 THE CONTINUOUS MODEL

In the continuous model, there are no restrictions on the players' movement: they can move, change directions or stop anywhere (on an edge or on a vertex). Also, the players can move at the same time, without taking turns.

As we stated before, previous results for this model that guarantee capture require 3 pursuers. Using only one pursuer with speed *k*, where *k* is a constant cannot guarantee capture. This result was shown by Dawes [37], which means that a solution using 2 pursuers, if it exists, is optimal with regard to the number of pursuers needed. Of course, finding the fastest possible solution using 2 pursuers (optimal when considering time to capture), is an open problem.

The following theorem improves the number of pursuers needed, but at the cost of higher speed and direction detection capability.

Theorem 2.2.1 In the case of continuous vision-based pursuit-evasion problem on a grid, 2 pursuers with a maximum speed of 2 and direction detection capability are sufficient to successfully capture the fugitive.

Proof:

Before we present the algorithm, we first introduce the reader to our notations. Our algorithm has three phases. The goal of the first phase is for the pursuers to see the fugitive *Z*, and includes a particular case in which the pursuers will actually capture the evader. The goal of the second phase is to get one of the pursuers within distance of 1 and on the same row as *Z*. We will assume that *A* and *B* move at the same time.

Consider the following algorithm where the two pursuers *A* and *B* start at points *(0, 0)* and *(1, 0)*. The renaming of the two pursuers to *Chaser* and *Watcher* (steps 10-17) is done in preparation for Phase 2. If at any point during the execution of the algorithm, *Z* is on the same vertex or same place on an edge as *A* or *B*, then the algorithm ends with capture. Initial direction is north.

Phase 1 (end when either *A* or *B* sees *Z*, or if *Z* between *A* and *B*, end with capture)

```
0. IF at any point during the execution of the steps 1 to 8,  A
   or B (or both) see Z, then GOTO step 10.
1. A and B move towards direction
2. IF A and B reached the last row
3.   IF direction == north
4.    direction = south
5.   ELSE direction = north
6.   A moves east 1, B moves towards direction
7.   After A reaches the same column as B,
8.    B moves east 1, A moves direction
9. GOTO step 1.
10. IF A sees Z
11.   IF B also sees Z
12.    IF Z between A and B,
         both pursuers move towards Z // capture Z
13.    ELSE IF A is closer to Z,
14.      A becomes the Chaser, B becomes the Watcher
15.        ELSE A becomes Watcher, B becomes Chaser
16.  ELSE A becomes the Chaser, B becomes the Watcher
17. ELSE A becomes the Watcher, B becomes the Chaser
18. Start Phase 2
```

Before we present the second phase we will show the intuition behind Phase 1. We will formally prove the correctness of Phase 1 after we present the entire algorithm.

Phase 1 will basically end whenever one of the 2 pursuers sees the fugitive, and that pursuer will become the *Chaser*.

In the case when both pursuers see the evader at the same time, the one closest to the opponent will become the *Chaser*. Even though neither of the pursuers has the Distance Detection Capability, they know each other's location and in which direction the fugitive is, hence they can easily determine who is closer (Fig. 2.2.1a).

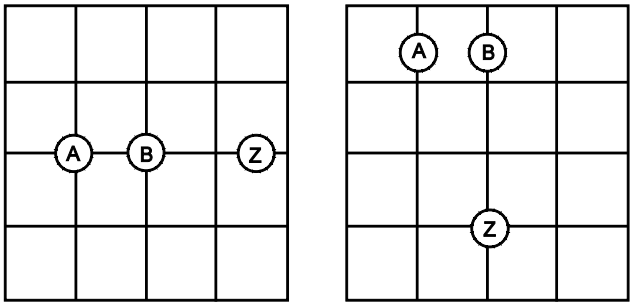


Figure 2.2.1 The pursuers see the evader (step 1).

The only case in which both pursuer see *Z* is when all 3 are on the same line. The case shown in Fig. 2.2.2 is not possible, at any time, during phase 1. That is because, during Phase 1, *A* and *B* either move synchronously north or south (step 1), or they move east while maintaining a distance of 1 (steps 6-8).

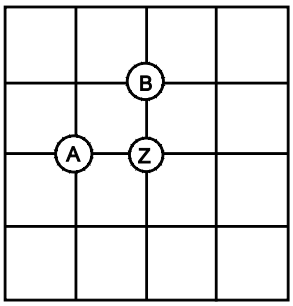


Figure 2.2.2 This case is not possible.

Whenever they see *Z*, the pursuers could be positioned like in Fig.2.2.1a, b (when executing step 1 of the algorithm), Fig. 2.2.3a, b (when executing step 6) or in Fig. 2.2.3c,d during the execution of steps 7-8.

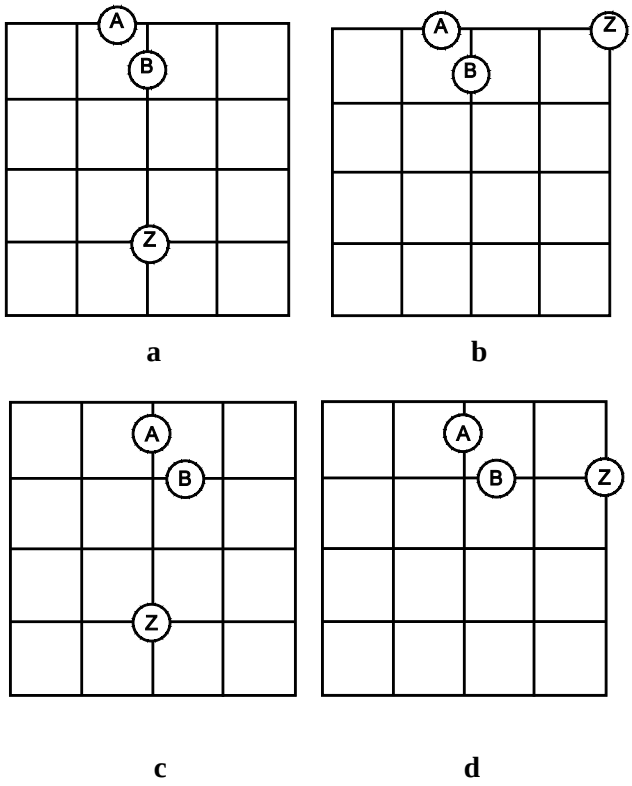


Figure 2.2.3 The pursuers see the evader (steps 6-8).

The particular case dealt with in step 12 of the algorithm would look like in Fig. 2.2.4. We can say that this is the “easy” case.

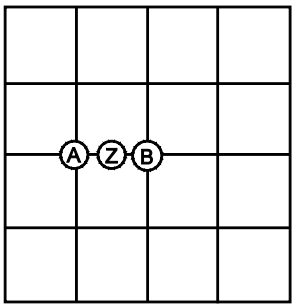


Figure 2.2.4 End with capture (step 12).

In Phase 2 of the algorithm, the two pursuers will be called *Chaser* and *Watcher*. Each will have a different role (and thus, execute a different algorithm) but when certain conditions are met, the *Chaser* can become the *Watcher* and the *Watcher* can become the *Chaser* (they switch roles and names).

Without loss of generality, assume the fugitive and the *Chaser* are on the same column (or at least were, at the time when the *Chaser* saw *Z*) and the direction where *Z* was last seen is north.

During Phase 2, the column of the *Chaser* will be called column *X*. The column to the east of the *Chaser* will be *X+1*, while the column to the west will be *X-1*.

The idea of the 2nd phase is to always have a *Chaser* advancing north, "chasing" *Z*, while the *Watcher* will be somewhere between *X-1* and *X+1*, "watching" the neighboring columns for any sight of the fugitive. The *Chaser* will always let the *Watcher* know in which direction *Z* went (either towards *X-1* or *X+1*). Whenever the *Watcher* sees *Z*, then the 2 pursuers switch names and roles.

We claim that by following our algorithm, the pursuers will eventually capture *Z*. The algorithm moves into Phase 3 when the *Chaser* sees the fugitive to its east or west.

Phase 2 (ends with the capture of *Z* or *Z* to east or west of *Chaser*)

Algorithm for the Chaser:

```
Move north
If Z is to the west or east on the same row
   Start Phase 3
```

Algorithm for the Watcher:

```
IF Z is in sight of Chaser
  IF Watcher not at column X, move towards column X
    ELSE                //both watcher and chaser see Z
     IF Watcher closer to Z than Chaser is
       switch roles     // Watcher becomes Chaser
                        // Chaser becomes Watcher
       ELSE move north
ELSE
  IF Z in sight of Watcher
    switch roles
  ELSE                  //Z is either between columns X-1
                        //and X or columns X and X+1
    IF Z was last seen going west towards column X-1
     IF Watcher not already at column X-1
       IF Watcher can go west
         Move west towards column X-1
       ELSE move north
      ELSE do nothing
    ELSE Z was last seen going east towards column X+1
     IF Watcher not already at column X+1
       IF Watcher can go east
         Move east towards column X+1
       ELSE move north
      ELSE do nothing
```

Phase 3 (end with capture of Z) – this phase involves only the Chaser

```
1. Move towards Z until Z out of sight or capture
2. Keep moving in the same direction until Z in sight again
3. GOTO 1.
```

Proof of correctness of Phase 1 (i.e. *A* or *B* will eventually see *Z*)

The area to the east of *A* and *B* will keep shrinking after each execution of steps 1 to 8, until *B* is on the last column. Also, if *Z* is initially located somewhere to the east of the *A* and *B*, then it cannot slip by pursuers' columns unseen. That is because at all times, *B*'s column is guarded by either *B* (steps 1-6, see Fig 2.2.3b), 3a), or A (steps 7-8, see Fig. 2.2.3c). Therefore, if *Z* is initially to the east of *B*, it will be eventually spotted.

Also, if *Z* is somewhere between *A*'s and *B*'s column, it will either be captured (Fig. 2.2.4) or it will be seen on one of the 2 columns (Fig. 2.2.1b).

Proof of correctness of Phase 2 (i.e. The *Chaser* will eventually see the fugitive to its west or east)

Claim a): *Z* will always be strictly between columns *X-1* and *X+1*. (Even though these columns change, *Z* will always be between what are currently named columns *X-1* and *X+1*)

Proof:

Assume *Z* disappears from the sight of *Chaser* and it starts moving east (the case when it moves west is similar). Since the speed of the pursuers is twice the speed of the fugitive, no matter where the *Watcher* is, it will reach column *X+1* before or at the same time as the evader.

If the *Watcher* reaches column *X+1* and it does not see *Z*, then it means *Z* is somewhere between *X* and *X+1*.

If *Watcher* sees *Z* at column *X+1*, then *Watcher* becomes *Chaser* and column *X+1* becomes column *X*, therefore *Z* is still between the (new) columns *X-1* and *X+1*.

Let $M_t = rd_t(Z, Chaser)+rd_t(Z, Watcher)$.

Claim b): $M_{t+1} \leq M_t - 1$

Proof:

CASE I. *Z* is moving east or west, then:

$rd_{t+1}(Z, Chaser) = rd_t(Z, Chaser) - 2$

$rd_{t+1}(Z, Watcher) \leq rd_t(Z, Watcher)$

So $M_{t+1} \leq M_t - 2$

In this case *M* will decrease, because *rd(Z, Watcher)* will stay constant, or decrease if *Watcher* is moving north, while *rd(Z, Chaser)* will decrease by 2.

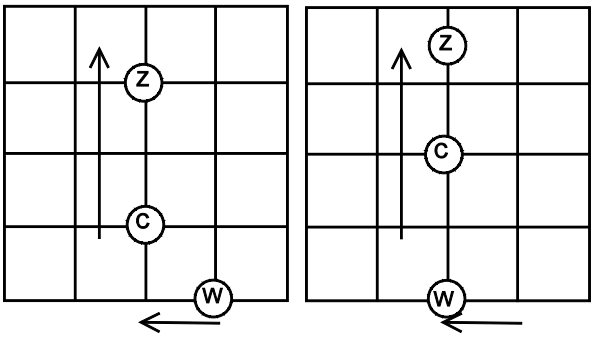
CASE II. *Z* is moving north (see Fig. 2.2.5a-c)

$rd_{t+1}(Z, Chaser) = rd_t(Z, Chaser) - 1$

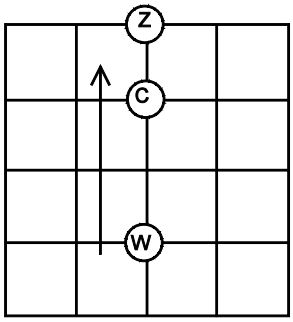
$rd_{t+1}(Z, Watcher) \leq rd_t(Z, Watcher)$

So $M_{t+1} \leq M_t - 1$

If *Z* moves north, then *rd(Z, Watcher)* will increase for at most half a time unit, until it gets to column *X*. After that, *rd(Z, Watcher)* will decrease until *Z* gets out of sight (moves east or west).



a) time *t* **b)** time *t* + ½



c) time *t+1*

Figure 2.2.5 *C* is the *Chaser, W* is the *Watcher*

CASE III. *Z* is moving south, is stationary or changes direction during 1 time unit.

After 1 time unit *Z* will be *q* distance north of its current position, $0 \leq q < 1$. If $0 > q$ that would mean Z moved south and ran into *Chaser*, so end with capture. We assume $0 \leq q < 1$.

$rd_{t+1}(Z, Chaser) = rd_t(Z, Chaser) - 2 + q$

If $0 < q \leq ½$, $rd_{t+1}(Z, Watcher) \leq rd_t(Z, Watcher) + q$

So $M_{t+1} \leq M_t - 2 + 2q$

But $2q \leq 1$

It results that $M_{t+1} \leq M_t - 1$

In this case the *Watcher* does not have time to move north, therefore after 1 time unit it will actually be further away from *Z*. But when we add up the row-distance between *Z* and the chaser, we can see that *M* is still decreasing.

If $½ < q < 1$,

$rd_{t+1}(Z, Watcher) \leq rd_t(Z, Watcher) + q - (q - ½) \times 2 = rd_t(Z, Watcher) - q + 1$

So $M_{t+1} \leq M_t - 2 + q - q + 1$

Therefore $M_{t+1} \leq M_t - 1$

In this case the *Watcher* will move north for $q - ½$ time (since it takes ½ time in the worst case to be on the same column as *Z*). In this time the row-distance between *Z* and the *Watcher* will decrease but after 1 time unit it will still be greater than 0. Again, when we add up the row-distance between *Z* and the *Chaser* we see that *M* is still decreasing.

From Claims a), b) it follows that *Chaser* and *Watcher* will get closer and closer to *Z*, while *Z* cannot get outside the boundary columns $X-1$ and $X+1$. Therefore, $rd(Z, Chaser)$ will eventually become zero, which means *Z* will be to its left or to its right.

Proof of correctness of Phase 3 (the pursuer will capture *Z*)

At the beginning of phase 3, the *Chaser* is at most distance of 1 from *Z*.

Since the pursuer is twice as fast, *Z* will be captured unless it moves out of the way in the next time unit.

Let *p* be the point where *Z* turns north or south (Fig. 2.2.6a).

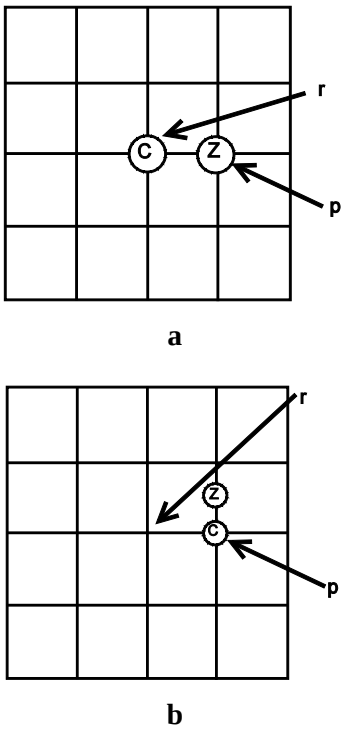


Figure 2.2.6 Phase 3.

After *Z* gets out of sight, the pursuer will reach point *r*, while Z will be somewhere in the middle of the edge (Fig. 2.2.6b). Now the pursuer is within distance of ½ from *Z*.

Capture occurs before *Z* reaches the next point.

Theorem 2.2.2 The algorithm above has a running time of $O(n^2)$ before *Z* is captured.

Proof:

Phase 1: $O(n^2)$

Phase 1 is the "slowest" of all the 3 phases. In the worst case, the pursuers have to move over all vertical edges in G_n plus some horizontal edges in order to see *Z*. This will take at most $(n(n-1) +2n) /2$ time (since the pursuers cover 2 edges in 1 unit of time) = $O(n^2)$.

Phase 2: $O(n)$

For Phase 2, in the worst case, the *Chaser* will move a distance of *2n* (*n* horizontal and *n* vertical). But, depending on *Z*'s moves, both pursuers can become the *Chaser* (but only one at a time), so the total possible distance would be *4n*. This can be done in *4n/2 = 2n* time.

Phase 3: *O(1)*

Phase 3 will only be executed at most twice in the worst case.

Therefore, our algorithm has a running time of $O(n^2)$ before *Z* is captured.

In this chapter we presented results for both the discrete and continuous models. We will discuss about the implications of these results and about future work in Chapter 5.

CHAPTER 3

PURSUIT EVASION ON DIFFERENT CLASSES OF GRAPHS

This Chapter will deal with the pursuit evasion problem on some graph classes that we find interesting. As we described in the Chapter 1, the pursuit evasion problem on general graphs is NP-complete. Therefore, researchers have focused on various graph classes, starting from simple graphs like paths and cycles, to series-parallel, chordal, or planar graphs. The pursuit evasion problem has been solved on many graph classes, proven to be either NP-complete or polynomial. For some, like the series-parallel graphs, the pursuit evasion is still an open problem. In the next sections we present our contribution to this area.

3.1 INTERVAL GRAPHS

In this Section we present our results for interval graphs. In Chapter 1, Section 1.1.3 we presented an algorithm by Peng et al. [14]. Our algorithm is similar, but it provides additional characterizations that cannot be found in [14].

We remind the reader that we can find the maximum clique of an interval graph in polynomial time using a simply Greedy algorithm [42].

Lemma 3.1.1 If G is an interval graph with maximum clique size $r > 3$, then $r+1$ robots are sufficient to clear G.

Proof:

(Greedy) Algorithm:

Assume we have n vertices and r available robots, with r = size of the largest clique in G. Consider any interval representation of the graph, each interval having a starting time s and a finish time f. We arrange in ascending order all vertices i (intervals $s_i f_i$) according to s and f.

Note: We assume the graph is connected, if not, apply the algorithm to each component separately.

We use the standard representation for interval graphs, where all s_i and f_i are distinct. The first starting time s_i is 0, the last finish time f_i is $2n - 1$. All intervals have at least length 1.

List: contains all vertices ordered ascending by start time.

set S: Contains the first vertex in the list, and all robots are on that vertex

Algorithm 1:

```
S = {vertex i with s_i = 0}
For t = 1 to 2n - 1
{
1.  If t is a finish time
        Remove vertex j from S that has f_j = t. Move all
        robots from j to any of the remaining vertices in S

2.  Else
        Remove vertex i from the list that has s_i = t
3.      Move one robot to vertex i, leaving at  least one
        robot on  each vertex from S.

4.      Use one robot from S (there is at least 1 left) to
        clear all edges between S and vertex i.
5.      Add vertex i to S.
}
```

Observation: There is always at least 1 vertex in S until time $t = 2n - 1$.

Proof: Immediately before iteration t of the loop, the set S contains each vertex j such that $s_j < t$ and $f_j \geq t$. S is not empty during this time because the interval graph is connected

Proof the previous algorithm clears any interval graph:

Invariant 1: At the start of each iteration, S forms a clique.

Proof:

Initialization: S has one vertex.

Maintenance: At the start of each iteration, S is formed by vertices j with $s_j \leq t - 1$ and $f_j > t - 1$. Therefore, they all overlap each other (i.e. they are pairwise adjacent).

Termination: Since S always forms a clique, and we know the maximum clique size is r, we can have S cleared as shown in steps 2 to 5 of the algorithm.

Invariant 2: At the start of each iteration, all vertices in S, and all edges between them, have already been cleared.

Proof:

Initialization: Obviously, the first vertex is already cleared since all robots are on that vertex.

Maintenance: As shown by invariant 1, S forms a clique. Therefore, during the previous loop, S was cleared as shown in steps 2 to 5, using at most $|S| + 1$ robots. We know the maximum clique size is r, so steps 2 to 5 require at most $r + 1$ robots. This means that at the start of the next loop, the invariant holds.

Termination: Once the loop terminates, all the remaining vertices from S and all edges between them have already been cleared. This, in conjunction with the next invariant, will prove our claim.

Invariant 3: At the start of each iteration, all vertices that have been removed from *S*, and all their incident edges have already been cleared.

Proof:

Initialization: No vertices removed yet, so the invariant holds.

Maintenance: We remove a vertex *j* if $f_j = t$. That means there are no new vertices in the list adjacent to *j*. Since *j* is in *S* before we remove it, it also means all edges between *j* and vertices in *S* are clear. If there are other vertices *i* that had been already removed from *S*, adjacent to *j*, obviously, all edges *(i, j)* were clear when *i* was removed.

Therefore, when we remove *j*, all its incident edges had been cleared.

Termination: Once the loop terminates, all vertices which were removed from *S*, and all their incident edges, have been cleared.

When the loop terminates, we have analyzed all vertices. That, plus Invariant 2 and Invariant 3, proves our claim.

Observation 1: We do not need to know the size of the clique in advance. We can apply the algorithm and add a new robot each time $|S_1 \cup \{i\}|$ increases. At the end, we have used $max(|S_1|) + 1$ robots.

Observation 2: Since clearing a clique of size *r* requires *r* robots, we cannot clear an interval graph with less than *r* robots, for $r > 3$.

Observation 3: There are interval graphs that require $r + 1$ robots to clear them.

Example:

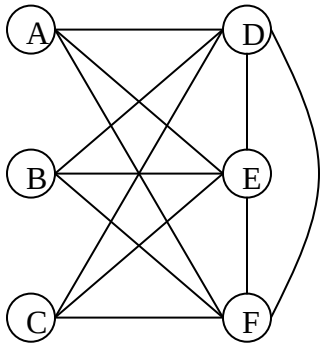


Figure 3.1.1 A graph that requires $r + 1$ searchers.

Here the maximum clique has a size of 4, but we cannot clear the graph with only 4 robots. We could start for example at vertex *A* (or similarly, *B* or *C*), then send robots towards *D*, *E* and *F* clearing edges *AD*, *AE*, *AF*. *A* does not need to be guarded so now we can clear the edges *DE*, *EF*, *FD* with the robot that was at *A*. Next, we can send this robot to *B* or *C*. We cannot move any robot after this, because any movement would result in recontamination.

Or, we could start at vertex *D* (or similarly, *E* or *F*) and clear clique *ADEF* first. As in the previous case, we must leave a robot on *D*, *E* and *F* while the 4^{th} robot can slide to *B* or *C*. We have deadlock again.

Claim 3.1.2 For $r = 2$, if *G* contains a $K_{1,3}$, we need 2 robots to clear the graph, otherwise 1 robot is sufficient.

Proof:

Obviously, $K_{1,3}$ cannot be cleared with less than 2 robots. We need 2 to clear it.

If *G* is just a path, then we can start in one end and move to the other end, effectively clearing the graph with just one robot

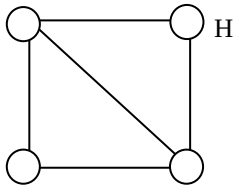


Figure 3.1.2 Graph with max clique size 3 that requires 3 robots

Claim 3.1.3 For $r = 3$, if *G* contains the subgraph *H* shown in Figure 3.1.2, we need 3 robots to clear the graph, otherwise 2 are sufficient.

Proof:

Obviously, *H* requires 3 robots for clearing. So if *G* contains *H*, *G* would require 3 too. If however *G* does not contain *H*, and $r = 3$, any 2 vertices belonging to a clique of size 3 cannot have any neighbors in common outside the clique (if they do, then *G* contains a subgraph *H*). But *G* is an interval graph, therefore *G* must be a "path" of vertices and triangles like in the following figure:

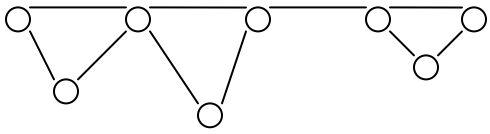


Figure 3.1.3 Graph with max clique size 3 that requires 2 robots.

Such a graph can easily be cleared with only 2 robots. Also, it is trivial to show that only one robot is not sufficient.

From the previous claims and Lemma 3.1.1 we have the following table:

Table 3.1 Maximum clique size and number of robots

Maximum Clique Size	Number of robots
1	1
2	1,2
3	2,3
4	4,5
5	5,6
6	6,7
…	…

Theorem 3.1.4 Given an interval graph G with a maximum clique size $r \geq 3$, if G contains at least 3 cliques of size r that share $r - 1$ vertices, then clearing G requires $r + 1$ robots. Otherwise, r robots are sufficient.

Proof:

The following algorithm clears G using r robots if and only if G does not contain at least 3 cliques of size r that have $r - 1$ common vertices.

As for Algorithm 1, we again use the standard representation for interval graphs, so that all s_i and f_i are distinct.

We say vertex x is cleared if all edges incident to x are cleared.

Vertices are sorted in ascending order by finish time.

Algorithm 2:

```
1.   For each vertex x in ascending order by finish time
{
2.   If there is no robot on x, place one robot on x
3.   If fewer than r robots are currently placed, use one robot
     to clear all contaminated edges (x, k) where k is a
     neighbor of x and there already is a robot on vertex k.
     //if already using r robots, do nothing
4.   For each j such that j is a contaminated neighbor of x and
     there is no robot already on j
5.        Place one robot on x then move it from x to j
6.   Remove the robot from x
7.   If x is cleared, clear all contaminated edges (j, k) where
     j, k are neighbors of x and edge (j, k) exists
8.   Remove any remaining robots that are still on any
     previously cleared vertices
}
```

Invariant 0: At the end of each iteration, all neighbors of *x* with a higher finish time are guarded.

Initialization: During steps 4 and 5, we send one robot to each of *x*'s neighbors.

Maintenance: During steps 4 and 5, we send one robot to each of *x*'s contaminated neighbors. So all neighbors with a higher finish time will be guarded.

Termination: After the loop is executed, there are no vertices left.

Invariant 1: At the beginning of each iteration, all vertices that are guarded are neighbors of x.

Initialization: Initially no vertices are guarded.

Maintenance: At steps 4 and 5, we send robots from x to its neighbors. At the end of the iteration, the robot on x is removed. The next iteration will choose a vertex k with the next lowest finish time. Since G is an interval graph, all neighbors of x that have a higher finish time, are also neighbors of k. However, x and k are not necessarily neighbors. So during the next iteration, only neighbors of k are guarded.

Termination: After the loop is executed, there are no vertices left.

Invariant 2: If fewer than r robots are placed after step 2 of an iteration, then all incident edges to x will be cleared during steps 3, 4 and 5.

Initialization: Initially, there are no placed robots. We place one robot on x during step 2. During steps 4 and 5, we clear all edges incident to x, by sliding robots from x to its neighbors. This is possible even if x has $r - 1$ neighbors, since we have $r - 1$ available robots. Therefore x will be cleared.

Maintenance: From Invariant 1 we know only neighbors of x are guarded, so we have at least as many available robots as there are unguarded contaminated neighbors of x. Step 3 will be executed because we have at least one robot available, clearing all edges between x and guarded neighbors. During steps 4 and 5, we slide robots from x to all contaminated neighbors k, and we have at least one robot for each k. Therefore x is cleared.

Termination: After the loop, all vertices are cleared.

Invariant 3: If r robots are placed after step 2 of an iteration, then all incident edges to x will be cleared during step 7 of the next iteration.

Initialization: Initially there are no robots placed.

Maintenance: Let current iteration be iteration i. If we have r robots placed during iteration i, then x will not be cleared during steps 3, 4 and 5 of the algorithm, because there are no available robots. The robot on x will be removed at step 7.

Assume vertex x_{i-1} was cleared before vertex x during iteration $i - 1$. We know that x cannot be a neighbor of x_{i-1}, because otherwise, from invariant 0, it follows that at the end of the iteration i, x is guarded while the robot on x_{i-1} would be removed. So during iteration i, only $r - 1$ robots would be placed. Contradiction.

Since x_{i-1} and x are not neighbors and we have r placed robots, it follows that we have a clique of size r, and x_{i-1} and x share $r - 1$ common vertices.

The vertex x_{i+1} chosen during the next iteration $i + 1$ will be a neighbor of x or the conditions in Theorem 3.1.4 are not met. If x_{i+1} and x are not neighbors, it results that x_{i-1}, x and x_{i+1} share $r - 1$ common neighbors.

We know that x_{i+1} is guarded, since we had r robots placed and from Invariant 1 results that all neighbors of x that have a higher finish time, including x_{i+1}, are guarded. After step 2 of the next iteration, there will be $r - 1$ placed robots. We can use one robot to clear all edges incident to x_{i+1}, during step 3 and then slide one robot from x_{i+1} to x. Now, during step 7, the robot on x_{i+1} will clear all edges between neighbors of x_{i+1}, clearing x in the process.

Termination: After the loop, all vertices are cleared.

From Invariants 1, 2 and 3 it follows that Algorithm 2 is correct.

This Greedy algorithm is similar to the algorithm described in [14]. However, it uses r robots when G does not contain at least 3 cliques of size r that share $r - 1$ vertices. When it encounters a clique of size r, it first chooses a vertex x that does not belong to another clique of size r that has not been previously cleared. Then, it clears x and all edges between x and the clique it belongs to. Now, we do not need to guard x anymore, so we can use that robot to clear the rest of the clique.

If G contains at least 3 r-size cliques that have $r - 1$ common vertices, like in Fig 3.1.1, then the algorithms still finds and clears the first x (for example A from Fig 3.1.1) then, as explained earlier, it cannot clear the next x (vertex B) unless we have $r + 1$ robots.

In this section we provided characterizations for when we need to use r robots, $r + 1$, or, in the case $r \leq 3$, $r - 1$ robots. We showed how an interval graph must look in order to be r-searchable. Even though an algorithm for interval graphs has already been published in [14], these characterizations have not. The following Theorem summarizes our results:

Theorem 3.1.6 Given an interval graph G with maximum clique size r, we have the following:

- If $r = 2$ and G does not contain a $K_{1,3}$ then clearing G requires 1 robot
- If $r = 2$ and G contains a $K_{1,3}$ then clearing G requires 2 robots
- If $r = 3$ and G does not contain a subgraph H like in Fig.3.1.2 then clearing G requires 2 robots
- If $r = 3$ and G contains a subgraph H like in Fig.3.1.2 then clearing G requires 3 robots
- If $r \geq 3$ and G does not contain 3 cliques of size r that have $r - 1$ common vertices, then G requires r robots to clear

- If $r \geq 3$ and G contains 3 cliques of size r that have $r - 1$ common vertices, then G requires $r + 1$ robots to clear

3.2 BICONNECTED OUTERPLANAR GRAPHS

We first define the class of 2-terminal series parallel graphs [40]:

A two-terminal graph is a graph with two special vertices s and t, called a source and a sink.

A parallel composition P = parallel(G, H) of two-terminal graphs G and H is created by merging the sources of G and H to create the source of P and the sinks of G and H to create the sink of P.

A series composition S = series(G, H) of two-terminal graphs G and H is created by merging the sink of G with the source of H. The source of G becomes the source of S and the sink of H becomes the sink of S.

A 2-terminal series-parallel graph is a graph that can be constructed by a sequence of series and parallel compositions from a set of copies of the single-edge graph K_2 with assigned terminals.

Next, we define the graph classes planar, outerplanar and biconnected outerplanar.

A planar graph is a graph that can be embedded in the plane (i.e. it can be drawn on the plane in such a way that its edges intersect only at their endpoints).

An outerplanar graph is a planar graph which has an embedding such that all vertices lie on the outermost face. For example, all trees are outerplanar.

A graph is biconnected if for all vertices x, y there exist two paths from x to y that are vertex-disjoint except at the endpoints. So removing any one vertex does not disconnect the graph.

A biconnected outerplanar graph has a planar embedding such that all vertices lie on a common cycle, and every edge either belongs to this cycle or forms a chord within the interior of the cycle (Fig. 3.2.1).

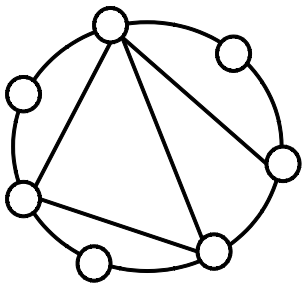


Figure 3.2.1 A biconnected outerplanar graph.

We have the following dynamic programming algorithm that finds the minimum number of pursuers needed to clear biconnected outerplanar graphs. This algorithm uses the fact that the biconnected outerplanar graph is a subclass of the series-parallel graphs. An eventual goal is to generalize this algorithm to the entire class of series-parallel graphs.

Define five properties to be computed for each subgraph, as follows. In each case, for each terminal t, either a pursuer is stationed at t, or a pursuer begins at t, or a pursuer ends at t.

a) $G.a$ = minimum number of pursuers needed to clear G if a pursuer is stationed at each terminal

b) $G.b$ = minimum number of pursuers needed to clear G if a pursuer is stationed at terminal 1 and a pursuer begins or ends at terminal 2

c) $G.c$ = minimum number of pursuers needed to clear G if a pursuer is stationed at terminal 2 and a pursuer begins or ends at terminal 1

d) $G.d$ = minimum number of pursuers needed to clear G if pursuers begin at both terminals (or end at both terminals)

e) $G.e$ = minimum number of pursuers needed to clear G if a pursuer begins at one terminal and a pursuer ends at the other terminal

In the cases b) through e) it does not matter whether we begin or end at these terminals due to time reversal in any solution. This means that if, for example, we have a solution X that starts at terminal 1 and ends at terminal 2, then the same number of pursuers can clear the graph if we start at terminal 2 and end at terminal 1 following the steps from solution X in reverse order.

We compute the given five properties for each subgraph in a bottom-up fashion, using formulas to be presented later in this section. We will have to construct all possible subgraphs that could be constructed during any series-parallel composition for our given graph. How we construct the subgraphs is the key of the algorithm. The number of subgraphs is not linear.

For example, the graph G = series (G_1, G_2, G_3) can be constructed as follows: either series (G_1, series (G_2, G_3)) or series (series (G_1, G_2), G_3). Compute properties for both

subgraphs: series (G_2, G_3) and series (G_1, G_2). No decomposition is specified, and we do not know yet which decomposition will yield an optimal solution. That is why we need to look at all possible decompositions.

Lemma 3.2.1 For any graph *G*, we have the following relations between the 5 properties:

a) $G.b \leq G.a \leq G.b+1$

Proof: $G.b \leq G.a$ because *G.a* is a particular case of *G.b*. That is, any solution for *G.a* also solves *G.b*.

Also, *G.a* cannot be larger than *G.b*+1, because we only need *G.b* pursuers to clear *G* while leaving one searcher stationed at terminal 1, and the additional pursuer can be stationed at terminal 2. So G.a ≤ G.b+1.

b) $G.c \leq G.a \leq G.c+1$

Proof:

Analogous to b)

c) $G.d \leq G.b \leq G.d+1$

Proof:

$G.d \leq G.b$ because *G.b* is a particular case of *G.d*, meaning that any solution for *G.b* also solves *G.d*. That is because when we clear *G* under the *G.d* criteria, we could station a pursuer at terminal 1, while the remaining ($G.b - 1$) pursuers will clear *G* under *G.b* criteria. So $G.d \leq G.b$.

G.b cannot be larger than *G.d*+1 because *G.d* pursuers can clear *G* while the remaining pursuer can stay at terminal 1, which is the definition of *G.b*. So $G.b \leq G.d+1$.

d) $G.d \leq G.c \leq G.d+1$

Proof:

Analogous to c)

e) $G.e \leq G.b \leq G.e+1$

Proof:

$G.e \leq G.b$ because $G.b$ is a particular case of $G.e$, meaning that any solution for $G.b$ also solves $G.e$. Like in the previous cases, when we clear G by starting at one terminal and ending at the other terminal, we could station one pursuer at terminal 1, and use the remaining $(G.b - 1)$ pursuers to clear G. So $G.e \leq G.b$.

Also, $G.b$ cannot be larger than $G.e+1$. That is because we can station the additional pursuer at terminal 1, while we clear G by starting at one terminal and ending at the other terminal using $G.e$ searchers. But that is the definition of $G.b$. So $G.b \leq G.e+1$.

f) $G.e \leq G.c \leq G.e+1$

Proof:

Analogous to e)

Corollary 3.2.2 The previous relations also imply that $G.d \leq G.e+1$, $G.e \leq G.d+1$, $G.b \leq G.c+1$, $G.c \leq G.b+1$.

We show this is true for $G.d \leq G.e+1$. The other 3 relations are similar.

From c) we have $G.d \leq G.b$ and from e) we have $G.b \leq G.e+1$ so $G.d \leq G.e+1$.

Theorem 3.2.3 For any graph that can be created during the series-parallel construction of a biconnected outerplanar graph *G*, the previous properties can be computed with the following dynamic programming algorithm:

I. If *G* has a single edge:

a) $G.a = 3$

Proof : Place one robot at each terminal, and another one to clear the edge

b) $G.b = 2$

Proof: Place one robot at terminal 1 plus another one to clear the edge

c) $G.c = 2$

Proof: Place one robot at terminal 2 plus another one to clear the edge

d) $G.d = 2$

Proof: Place one robot at each terminal and move them towards each other

e) $G.e = 1$

Proof: Start at one terminal and send the robot towards the other terminal

II. If *G* = series (G_1, G_2):

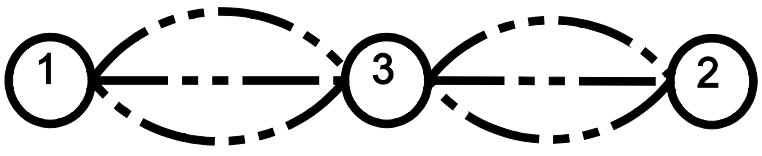


Figure 3.2.2 Vertices 1 and 2 are *G*'s terminals, while vertex 3 is the merged vertex.

a) $G.a = max\ (G_1.b, G_2.c) + 1$

Proof:

When we clear G, we can either clear G_1 first and G_2 second, or G_2 first and G_1 second, while keeping a pursuer at both terminals 1 and 2. It can never help to start in G_1, clear parts of G_1, followed by clearing G_2 and then finish clearing G_1.

After clearing G_1, but before clearing G_2, vertex 3 must be guarded. So clearing G_1 takes $G_1.b$ plus one pursuer stationed at vertex 2. Clearing $G_2.c$ takes $G_2.c$ plus one pursuer stationed at vertex 1. It results that, in order to clear both subgraphs, we must choose the maximum between the two.

So $G.a = max\ (G_1.b, G_2.c) + 1$

Due to symmetry, we reach the same conclusion when we clear G_2 first followed by G_1.

b) $G.b = min\ [max\ (G_1.b, G_2.d) + 1, max\ (G_1.b, G_2.e + 1)]$

Proof:

We can clear G either by clearing G_1 first and G_2 second, or vice versa. If we clear G_1 first, then we use at least $G_1.b$ pursuers to clear G_1 (by the definition of $G_1.b$). After G_1 is cleared, vertex 3 must be guarded. Next, we need to start at vertex 3, clear G_2 and end at terminal 2, which is exactly the definition of $G_2.e$. So we also need at least $G_2.e$ pursuers to clear G_2. While we clear G_2 using $G_2.e$ pursuers, an additional pursuer must be stationed at terminal 1.

Therefore $max(G_1.b, G_2.e + 1)$ pursuers are sufficient.

However, if we clear G_2 first, then, we can either use $G_2.d$ or $G_2.e$ pursuers to clear G_2 (plus one pursuer that stays at vertex 1). If we use $G_2.e$, then we have the same analysis as before and $G.b = max(G_1.b, G_2.e + 1)$.

But if we clear G_2 using $G_2.d$ searchers (we end at both vertex 3 and 2), then we need $G_2.d + 1$ pursuers to clear G_2 (one remains at terminal 1). After G_2 is cleared, we need only $G_1.b$ pursuers to clear G_1, but we also need to leave one additional searcher at terminal 2 (by the definition of $G.b$, one pursuer must end at terminal 2). Hence, now we need at least $G_2.d + 1$ pursuers and at least $G_1.b + 1$ pursuers, therefore we have $max (G_1.b, G_2.d) + 1$.

But we are looking for the minimum $G.b$ that can satisfy the definition, therefore

$G.b = min [max (G_1.b, G_2.d) + 1, max (G_1.b, G_2.e + 1)]$

c) $G.c = min [max (G_1.d, G_2.c) + 1, max (G_1.e + 1, G_2.c)]$

Proof:

Analogous to the previous case due to symmetry.

d) $G.d = min [max (G_1.d, G_2.e + 1), max (G_1.e + 1, G_2.d)]$

Proof:

We can clear G either by clearing G_1 first then G_2, or by clearing G_2 first then G_1.

If we clear G_1 first then we must start somewhere inside G_1 and end at vertices 1 and 3. For that, we need at least $G_1.d$ pursuers, by the definition of $G_1.d$. Next, we must clear G_2. We start at vertex 3, because vertex 3 is the border between G_1 and G_2, and we must end at vertex 2. But that is the definition of $G_2.e$.

So we can clear G_2 with $G_2.e$ pursuers, plus one pursuer that must remain at terminal 1. It results that we need at least $G_1.d$ pursuers and at least $G_2.e+1$ pursuers, which gives us max $(G_1.d, G_2.e + 1)$.

Also, if we clear G_2 first then G_1, following the same rationale and because of symmetry we need $max\ (G_1.e + 1, G_2.d)$ pursuers. But we are looking for the smallest number of pursuers that can clear both G_1 and G_2, so $G.d = min\ [max\ (G_1.d, G_2.e + 1), max\ (G_1.e + 1, G_2.d)]$

e) $G.e = max\ (G_1.e, G_2.e)$

Proof:

Like in the previous cases, we must either clear G_1 first and G_2 second or vice versa.

If we clear G_1 first, we must start at terminal 1 and end at terminal 3. By definition, $G_1.e$ is the minimum number of pursuers that can do that. Next, we clear G_2 by starting at vertex 3 and ending at vertex 2. Again, by definition, $G_2.e$ is the minimum number of pursuers that can do that. Therefore we need at least $G_1.e$ pursuers and at least $G_2.e$ pursuers. That is $max\ (G_1.e, G_2.e)$.

If we clear G_2 first followed by G_1 and following the same rationale, we have the same number if pursuers, $max\ (G_1.e, G_2.e)$.

Therefore $G.e = max\ (G_1.e, G_2.e)$.

III. If G = parallel (G_1, G_2) where G_2 is a single edge that forms a chord (that is, an edge not on the outer cycle). This case will apply for every parallel operation during the composition of G except for the final step.

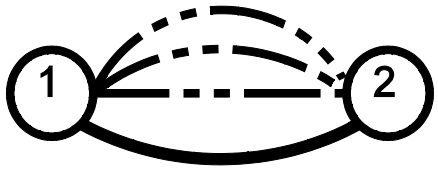


Figure 3.2.3 The thicker line is G_2.

a) $G.a = max\ (G_1.a, G_2.a) = G_1.a$

Proof:

If we clear G_1 first and G_2 second, we need $G_1.a$ pursuers to clear G_1 while terminals 1 and 2 are guarded. While we clear G_1, the terminals of G_2 are also guarded since we have a parallel composition. After we clear G_1, we can use one additional robot to clear G_2.

If we clear G_2 first, then we have the same analysis. Clear G_2 by having a pursuer at each terminal and a third pursuer clear the edge. After G_2 is cleared, we need to clear G_1, using $G_1.a$ pursuers.

So G.a = G_1.a, because $G_1.a \geq 3$ and $G_2.a = 3$.

b) $G.b = min\ [max\ (G_1.b, G_2.a), max(G_1.a, G_2.b)] = max\ (G_1.b, 3)$

Proof:

If we clear G_1 first, then, by definition, we must use $G_1.b$ pursuers. After $G_1.b$ is cleared, one searcher will be at terminal 1 (the stationary searcher) and another one at terminal 2 (the pursuer that ends at terminal 2, from the definition of $G_1.b$). Now, we need an additional pursuer to clear G_2. So in this case we have $max\ (G_1.b, G_2.a) = max\ (G_1.b, 3)$.

Similarly, if we clear G_2 first, then we need $G_2.b = 2$ pursuers to clear G_2. After G_2 has been cleared, we must leave one searcher at each terminal. Now we need $G_1.a$ pursuers to clear G_1. Which means that we have $max\ (G_1.a, G_2.b) = max\ (G_1.a, 2) = G_1.a$.

But we are looking for the minimum numbers of pursuers that satisfy $G.b$, so $G.b = min\ [max\ (G_1.b, G_2.a), max(G_1.a, G_2.b)] = min\ [\ max\ (G_1.b, 3), G_1.a]$.

But $G_1.a \geq 3$ and $G_1.b \leq G_1.a$ which means that $G.b = max\ (G_1.b, 3)$.

c) $G.c = min\ [max\ (G_1.c, G_2.a), max\ (G_1.a, G_2.c)] = max\ (G_1.c, 3)$

Proof:

Analogous to the previous case.

d) $G.d = min\ [max\ (G_1.d, G_2.a), max\ (G_1.a, G_2.d)] = max\ (G_1.d, 3)$

Proof:

If we clear G_1 first, then we must use $G_1.d$ pursuers and end at terminals 1 and 2. Next, we must clear G_2 but have terminals 1 and 2 guarded. That is the definition of $G_2.a$. It results that we need at least

$max\ (G_1.d, G_2.a) = max\ (G_1.d, 3)$ pursuers.

If we clear G_2 first, then we use $G_2.d = 2$ pursuers and end at terminals 1 and 2. We clear G_1 next, but by leaving 1 pursuer at each terminal, which is the definition of $G_1.a$. So, in this case, we need at least

$max\ (G_1.a, G_2.d) = max\ (G_1.a, 2) = G_1.a$ pursuers.

But we are looking for the minimum such value that satisfies $G.d$, which means that

$G.d = min\ [max\ (G_1.d, G_2.a), max(G_1.a, G_2.d)] = min\ [max\ (G_1.d, 3), G_1.a]$.

But $G_1.d \leq G_1.a$ and $G_1.a \geq 3$, which means that $G.d = max\ (G_1.d, 3)$.

e) $G.e = min\ [max\ (G_1.b, G_2.c), max\ (G_1.c, G_2.b)] = min\ (G_1.b, G_1.c)$

Proof:

Initially, suppose that G_1 is cleared before G_2.

We can start at terminal 1 and end at terminal 2. In this case we use $G_1.b$ pursuers to clear G_1 (terminal 1 must be guarded, since G_2 is contaminated). When we clear G_2, terminal 2 is guarded by the pursuer who ended at terminal 2, and the guard at terminal 1 can now move and clear G_2. Therefore, by definition, $G_2.c$ is the number of pursuers that can clear G_2.

We can apply the same idea if we choose to start at terminal 2 and end at terminal 1. We will need $G_1.c$ pursuers to clear G_1 (because terminal 2 must be guarded) then use the guardian stationed at terminal 2 to clear G_2.

$G.e = min\ [max\ (G_1.b, G_2.c), max\ (G_1.c, G_2.b)]$

But $G_2.c = G_2.b = 2$, $G_1.b \geq 2$ and $G_1.c \geq 2$. It results that $G.e = min\ (G_1.b, G_1.c)$.

By symmetry, the same formula holds if G_2 is cleared before G_1.

IV. If G = parallel(G_1, G_2) where G_2 is not a chord, which occurs only at the final step when G is the entire biconnected outerplanar graph.

All but one parallel operation will involve a chord and therefore will fall under case III. The last step will involve connecting two subgraphs in parallel, and looking at the $G.d$ property for both of them.

Before deriving the formula for case IV, we need the following definition:

Definition: A cut (G_1, G_2) of a graph G is a pair of subgraphs (G_1, G_2) such that $|V_1 \cap V_2| = 2$,

$|E_1 \cap E_2| = 0$ and $G_1 \cup G_2 = G$ (i.e. we "cut" the biconnected outerplanar graph in two).

We consider all possible cuts (G_1, G_2) such that G = parallel (G_1, G_2) and take the one that yields the minimum number for $max\ (G_1.d, G_2.d)$.

Let $Min(G_1, G_2)$ be the minimum number of pursuers for cut (G_1, G_2).

We have $Min\ (G_1, G_2) = max\ (G_1.d, G_2.d)$.

Proof:

Consider a 'cut' G_1, G_2 of G. G = parallel (G_1, G_2). We can clear G either by clearing G_1 first and G_2 second, or vice versa. If we clear G_1 first, we can use $G_1.a$, $G_1.b$, $G_1.c$ or $G_1.d$ pursuers. $G_1.e$ cannot be considered, since G_2 is contaminated, so after G_1 is cleared, both terminal 1 and 2 must be guarded. After we clear G_1, one pursuer must be stationed at each terminal, to separate cleared area from contaminated area.

Next, we clear G_2, and we can also use $G_2.a$, $G_2.b$, $G_2.c$ or $G_2.d$ pursuers. That is, because when we start clearing G_2, one pursuer is placed at terminal 1 and another pursuer is placed at terminal 2.

Therefore we can clear G by any combination of the above. However, we are looking for the minimum number of pursuers that can clear G, and since $G_1.a \geq \{G_1.b, G_1.c\} \geq G_1.d$ and $G_2.a \geq \{G_2.b, G_2.c\} \geq G_2.d$, it results that $Min\ (G_1, G_2) = max\ (G_1.d, G_2.d)$.

The minimum number of pursuers that can clear G is the minimum over all possible $Min(G_1, G_2)$ values.

Observation:

At any point during any graph searching algorithm, there will be a frontier between the cleared space and the infected space. At some points during an algorithm, that frontier will be on a cut. That is the focus of our algorithm. Of course, since we are looking at all possible cuts, some will be redundant. However, this does not significantly increase the running time, since we are using a dynamic programming approach.

Example:

Consider the following figure:

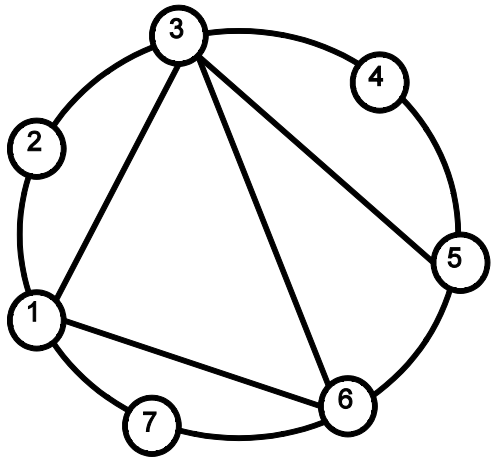


Figure 3.2.4 Example of optimal solution.

The graph in Fig. 3.2.4 can be cleared using 3 robots. We will show next one possible solution:

Assume the final step is G = parallel (G_1, G_2), where G_1 contains the vertices 1, 2, 3, 6, 7 and the edges between them including edge (3, 6). G_2 will contain vertices 3, 4, 5, 6. So the cutset consists of vertices {3, 6}.

So we start somewhere in G_1, clear G_1 using $G_1.d$ pursuers, then we clear G_2 using $G_2.d$ pursuers.

One possible solution for clearing G_1, which is not unique, is to split G_1 into 2 subgraphs G_1' and G_1'' containing vertices 1, 2, 3 and 1, 6, 7 respectively. Connect them in series then add the edge (3, 6) in parallel.

We can clear the G_1'' as follows: clear (1, 7) then add in series (7, 6). Finally, add in parallel (1, 6).

Initially (1, 7).d = (7, 6).d = (1, 6).d = 2 and (1, 7).e = (7, 6).e = (1, 6).e = 1. After the series composition we have series((1, 7), (7, 6)).d = 2 and series((1, 7), (7, 6)).e = 1. After the parallel composition we have $G_1''.d = max$ (2, 3) = 3, $G_1''.e = min$(2, 2) = 2 (according to the formulas for case III).

Similarly, we clear G_1' as follows: clear (1, 2) then add (2, 3) in series. Then add (1, 3) in parallel. $G_1'.d$ = 3, $G_1'.e$ = 2.

Series (G_1', G_1'').$d = min\ [max(3, 3)\ ,\ max(3, 3)] = 3$

Next we add (3, 6) in parallel so $G_1.d = max\ (3, 3) = 3$.

So we used 3 robots to clear G_1.

Similarly, we find that $G_2.d$ = 3.

So the min number of robots is $max\ (G_1.d, G_2.d)$ = 3.

Next we present the actual algorithm in pseudo code. Our algorithm yields one optimal solution, but not all possible optimal solutions.

The algorithm we present next is a dynamic programming algorithm that builds a table T in a bottom-up fashion, starting with the edges and moving up to subgraphs of increasing sizes. For each subgraph we compute the 5 properties. We only consider subgraphs G_1, G_2 that can be 'cuts' (i.e. G = parallel (G_1, G_2)).

If a subgraph G' = series (G_1,.., G_p), then we use an algorithm similar to the well-known matrix chain multiplication algorithm that will compute the optimal value for each property [63], and it will take into consideration all possible series decompositions.

The algorithm:

Definition: Let *{x, y}* be a cutset. Define the interior subgraph *I(xy)* as the induced graph on vertices *x*-..-*y* (moving clockwise). The interior subgraph *xy* may or may not contain the edge *(x,y).*

Define $I^-(xy) = I(xy)$ minus interior edge *(x, y)* and $I^+(xy) = I(xy)$ plus interior edge *(x,y)*, if it exists. If $\nexists$ interior edge *(x, y)* then $I^-(xy) = I^+(xy) = I(xy)$.

If there is an edge that is incident to an interior vertex of *I(xy)* and also incident to a vertex not in *I(xy)* (i.e. to an interior vertex of *I(yx)*) then that *I(xy)* will not be considered an interior subgraph. That is because it can never be considered a 'cut' that would split *G* in subgraphs G_1 and G_2 such that G = parallel(G_1, G_2).

We will generate subgraphs *G* bottom-up and store their properties (*G.a, G.b, G.c, G.d* and *G.e*) in a table *T*.

Let *Opt(G)* be the minimum number of pursuers needed to clear *G*. The following algorithm will compute *Opt(G)* using the properties and definitions described earlier in this section.

Algorithm BiconnectedOuterplanar:

```
1.  For each edge (x,y) store its properties in T at T((x,y))
2.  For i = 2 to n – 1
3.       For all paths I(xy) of length i
4.            If there is no chord (s, t), where s ∈ I(xy),
              t ∉ I(xy), s≠x and s≠y
5.                 ComputeProperties (I⁻(xy))
6.       If ∃ a chord (x, y)
7.            ComputeProperties (I⁺(xy))
8.  Opt(G) = min of all max(T(I⁺(xy).d), T(I⁻(yx).d)) for  all
    possible interior subgraphs I(xy)
```

ComputeProperties(*I(xy)*)

```
1.  If there is no chord (x,y) ∈ I(xy)
2.       Series-chain(G1..Gp) //I⁻(xy) = series (G1,.., Gp)
3.  Else
4.       I⁺(xy) = parallel(I⁻(xy), (x,y))
5.       Compute the properties for I⁺(xy) and store them at
         T(I⁺(xy))
```

Series-Chain(*G1..Gp*)

```
1.  for i=1 to p
2.    m[i, i].a = Gi.a
3.    m[i, i].b = Gi.b
4.    m[i, i].c = Gi.c
5.    m[i, i].d = Gi.d
6.    m[i, i].e = Gi.e
7.  for L = 2 to p do {
8.    for i = 1 to p-L+1 do {
9.      j = i + L - 1;
10.     m[i,j].a = INFINITY;
11.     m[i,j].b = INFINITY;
12.     m[i,j].c = INFINITY;
13.     m[i,j].d = INFINITY;
14.     m[i,j].e = INFINITY;
15.     for k = i to j-1 do {
16.       g1 = m[i, k]
```

```
17.       g2 = m[k+1, j]
18.       q.a = max (g1.b, g2.c) + 1
19.       q.b = min[max(g1.b, g2.d) + 1, max(g1.b, g2.e + 1)]
20.       q.c = min[max(g1.d, g2.c) + 1, max(g1.e + 1, g2.c)]
21.       q.d = min[max(g1.d, g2.e + 1), max(g1.e + 1, g2.d)]
22.       q.e = max(g1.e, g2.e)
23.       if (q.a < m[i, j].a)
24.             m[i,j].a = q.a;
25.       if (q.b < m[i, j].b)
26.             m[i,j].b = q.b;
27.       if (q.c < m[i, j].c)
28.             m[i,j].c = q.c;
29.       if (q.d < m[i, j].d)
30.             m[i,j].d = q.d;
31.       if (q.e < m[i, j].e)
32.             m[i,j].e = q.e;
33.     }
34.   }
35. }
36. T(I-(xy)) = m[1,p]
```

Next we analyze the running time of the preceding algorithm.

The Series-chain algorithm requires $O(|V|^3)$ time due to the nested loops at lines 7, 8 and 15.

ComputeProperties($I(xy)$) has a running time of $O(|V|^3)$ since it calls Series-chain.

Running time for the BiconnectedOuterplanar algorithm:

Step 1. Running time $O(|V|+|E|)$. But $|E| < 2*|V|$ for outerplanar graphs, so $O(|V+|E|) = O(|V|)$

Steps 2 to 7. BiconnectedOuterplanar considers $O(|V|^2)$ subgraphs, and for each it calls ComputeProperties, so the total time is $O(|V|^2) \cdot O(|V|^3)) = O(|V|^5)$

Step 8. Running time $O(|V|^2)$

Final running time: $O(|V|) + O(|V|^5) + O(|V|^2) = O(|V|^5)$

Observation: There exists an algorithm by Hu, T C. and M T. Shing [41] that solves the matrix chain multiplication problem in $O(n \log n)$ instead if $O(n^3)$. Even though it is more complex, it could be used in our Series-chain function, yielding a total running time of $O(|V|^3 \log |V|)$.

Observation: We have to use the matrix chain multiplication algorithm to compute G' = series (G_1,.., G_p) and take into consideration all possible series decompositions because the series operation is not associative under the 5 properties.

Here is a counterexample that illustrates non-associativity:

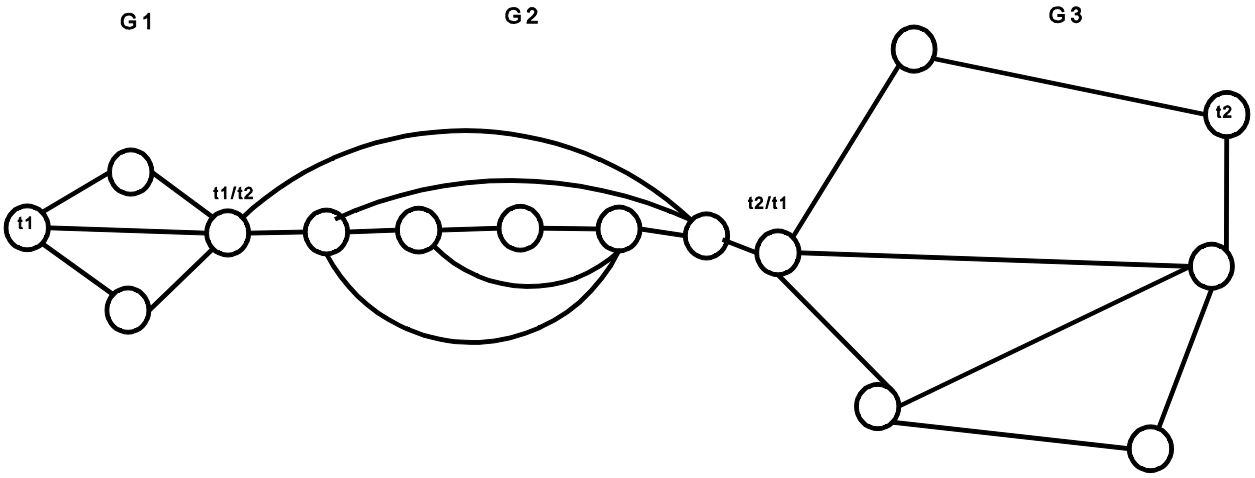


Figure 3.2.5 Example where the series operation is not associative.

In this example $G_1.b$ = 3, $G_2.d$=3, $G_2.e$ = 4 and $G_3.c$ = 4.

G = series (G_1, G_2, G_3)

If we do series(series(G_1, G_2), G_3) then $G.a$ = 5. However, if we use series (G_1, series(G_2, G_3)), then $G.a$ = 6.

So the order in which we use our series operation does matter. This means, in terms of our solution, that in some cases clearing in one direction is better than clearing in the reverse direction.

In the next section we will extend our analysis to graphs which contain biconnected outerplanar subgraphs as components.

3.3 EXTENSIONS TO BICONNECTED OUTERPLANAR GRAPHS

Our goal is to extend our biconnected outerplanar solution to the entire outerplanar class. We will advance gradually, analyzing graphs of increasing complexity. First we will consider a graph composed of 2 biconnected outerplanar graphs. Then, we will expand this solution to a path of biconnected outerplanar graphs followed by a graph shaped as a star which has biconnected outerplanar components. A last step, which we do not present in this dissertation, is to extend to tree-like graphs with biconnected outerplanar components.

What happens when we have 2 biconnected ourplanar graphs G_1, G_2 with a common vertex x like in the following figure?

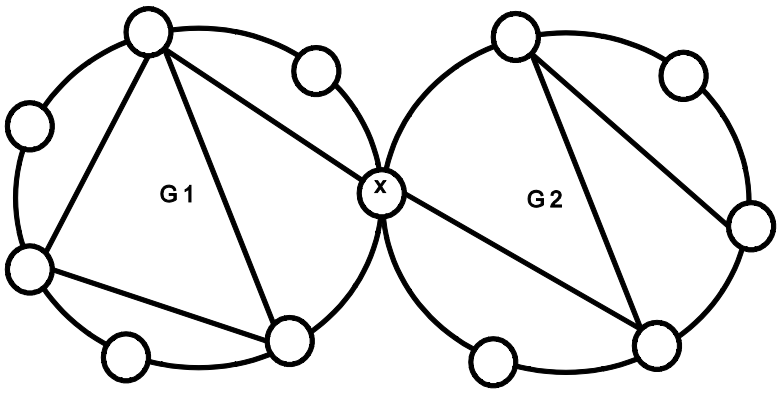


Figure 3.3.1 2 Biconnected outerplanar graphs that share a common vertex

First we would like to point out that, given the BiconnectedOuterplanar algorithm from section 3.2, it is easy to find a solution within 1 from the optimal solution: simply place one additional robot on x, and then compute the optimal solution for each component separately. However, finding the optimal solution for such a graph requires some additional modifications of the algorithm.

Observation 3.1 Any optimal solution would fall under one of the following three cases:

1. Start in G_1, clear G_1 ending at point x, then clear G_2 (Or its reverse)
2. Start in G_1, reach vertex x before being done clearing G_1, clear G_2, finish clearing G_1
3. Start in G_2, reach vertex x before being done clearing G_2, clear G_1, finish clearing G_2

Note that for the first case, taking the steps in reverse order would produce the solution that starts in G_2, clears G_2 first, then G_1. So we will only look at the solution that starts in G_1.
We will compute a candidate optimal solution for each case and choose the minimum as our optimal solution for G.

It can happen that clearing part of one subgraph first, then clearing the other subgraph, and then clearing the remainder of the first subgraph gives a better solution (cases 2 and 3). Therefore we need to look at all possible cases.

We will use the algorithm BiconnectedOuterplanar as follows:

First we compute the optimal solutions $Opt(G_1)$ and $Opt(G_2)$ for G_1 and G_2. Then we compute the optimal solutions for both G_1 and G_2 that start or end with a robot at x. Call these $Opt_x(G_1)$ and $Opt_x(G_2)$. Also, compute the optimal solutions for clearing G_1 and G_2 while a robot

remains guarding x from start to end. Call these $Opt_{xx}(G_1)$ and $Opt_{xx}(G_2)$. We can do all of the above using the BiconnectedOuterplanar algorithm presented earlier but with some modifications.

In order to compute Opt_x we add step 9 to the BiconnectedOuterplanar algorithm as follows

9. Opt_x = min of all max($T(I^+(xy).b)$, $T(I^-(yx).d)$), max($T(I^+(yx).c)$, $T(I^-(xy).d)$) and Opt + 1 for all possible interior subgraphs I(xy), where x is the vertex between the two biconnected components and $y \neq x$

Explanation:

We want to start at x. By looking at $I^+(xy).b$ and $I^+(yx).c$, we guarantee we start at x (since $I^+(xy).b$ and $I^+(yx).c$ imply x is guarded at all times). So we clear one part of the graph, while x is guarded, then we clear the rest of the graph (which is either $I^-(yx).d$ or $I^-(xy).d$) by starting at both terminals (x and y). Since we are looking at all possible cuts $I(xy)$, we guarantee that at least one of them will result in the optimal solution. The 3rd item in the solution, $Opt + 1$, bounds the solution by the optimal solution + 1 (we start anywhere, end anywhere, but x is guarded at all times).

In order to compute Opt_{xx} we add step 10 to the algorithm as follows:

10. Opt_{xx} = min of all max($T(I^+(xy).b)$, $T(I^-(yx).c)$) and Opt + 1 for all possible interior subgraphs I(xy), where x is the vertex between the two biconnected components and $y \neq x$

Explanation:

We want x to be guarded at all times. By looking only at $I^+(xy).b$ and $I^+(yx).c$, we guarantee x is guarded at all times. Any optimal solution would have to clear a section of the

graph first, while x is guarded, then clear the other section of the graph while x is still guarded. But we look at all possible cuts $I(xy)$, which means at least one of them must result in be the optimal solution.

Like in the previous case, $Opt + 1$, bounds the solution.

It is easy to see that $Opt_{xx}(G_i) \geq Opt_x(G_i) \geq Opt(G_i)$, i = 1, 2, because $Opt_{xx}(G_i)$ is more restrictive than $Opt_x(G_i)$ which is more restrictive than $Opt(G_i)$. Also, $Opt(G_i) + 1 \geq Opt_{xx}(G_i)$.

For case 1 in Observation 3.1, the candidate optimal solution for the entire graph is $max[\ Opt_x(G_1), Opt_x(G_2)]$.

For case 2, we want to start in G_1, clear parts of G_1 until reaching point x, clear G_2, then finish clearing G_1. Next we present the algorithm for this case.

Algorithm:

Compute the values $G.a$ through $G.e$ recursively as shown in the previous section for each subgraph of G_1. The only difference is when G has a single edge and G_2 is attached to one of the endpoints, as in the following figure:

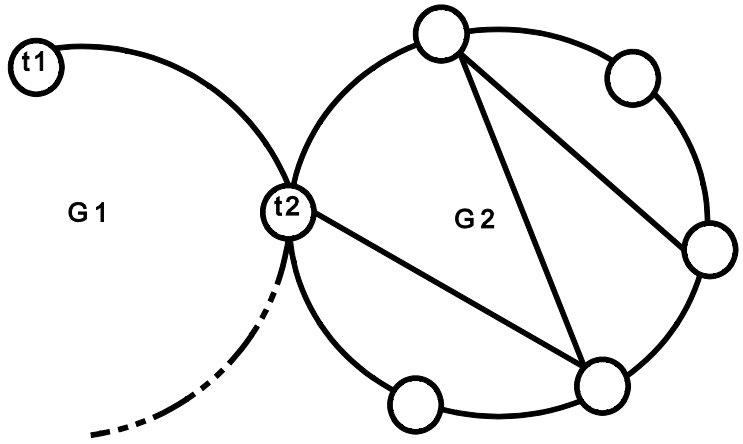


Figure 3.3.2 G_2 is connected to an edge in G_1

In this case we have:

a) $G.a = Opt_{xx}(G_2) + 1$

b) $G.b = Opt_{xx}(G_2)$ if G_2 is attached to terminal 1, and $Opt_{xx}(G_2) + 1$ if G_2 is attached to terminal 2 (Fig. 3.3.2).

c) $G.c = Opt_{xx}(G_2) + 1$ if G_2 is attached to terminal 1, and $Opt_{xx}(G_2)$ if G_2 is attached to terminal 2.

d) $G.d = Opt_{xx}(G_2)$

e) $G.e = Opt_{xx}(G_2)$

Next, we use the BiconnectedOuterplanar algorithm, with the modifications shown above, to compute the candidate optimal solution for the entire graph $G_1 \cup G_2$.

For case 3, we use the same modifications as for case 2, but we swap G_2 with G_1.

The optimal solution for G will be the minimum between the 3 candidate solutions, one for each case.

Next we consider a path of biconnected outerplanar components like the one shown in Fig. 3.3.3:

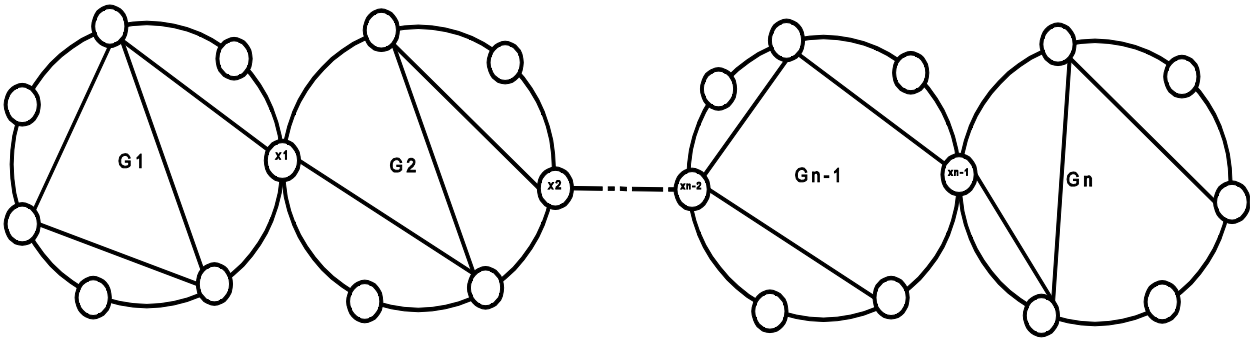


Figure 3.3.3 Path of biconnected outerplanar components

One trivial way to clear G, is to start at G_1, clear G_1, then clear G_2 and so on, until we finally clear G_n. This solution would require at most $max(Opt(G_i) +1), 1 \leq i \leq n$ pursuers. Next we prove this claim.

Theorem 3.3.1 The solution that starts at one end and finishes at the other end is at most within 1 from the optimal solution.

Proof:

Let $m = max \{Opt(G_i) \mid i = 1..n\}$

Obviously any solution to clear G requires at least m robots, and we will show that our approach clears G using at most $m + 1$ robots.

First we clear G_1 such that a robot ends at x_1, which requires $Opt_x(G_1)$ robots where $x = x_1$, and $Opt_x(G_1) \leq Opt(G_1) + 1 \leq m + 1$.

Next we clear G_2 such that we start at one point x_1 and end at x_2. We claim that this solution is at most $Opt(G_2) + 1$. Leave one robot at x_1 then move all robots to the starting point of the optimal solution $Opt(G_2)$. Clear G_2 as in the optimal solution until we reach vertex x_1. Then, move the robot from vertex x_1 to vertex x_2 and continue clearing G_2 as per optimal solution. If the optimal solution reaches vertex x_2 before x_1 then consider the reverse solution for G_2 (taking the steps of the optimal solution $Opt(G_2)$ in reverse order also produces an optimal solution). This reverse solution will reach x_1 first. As before, move the robot from x_1 to x_2. Therefore we used $Opt(G_2) + 1 \leq m + 1$ pursuers to clear G_2 by starting at x_1 and ending at x_2.

Using the same algorithm, we continue clearing all components up to G_n but no component will require more than $Opt(G_k) + 1 \leq m + 1$ pursuers, $k = 1..n$.

Next we consider a star arrangement of biconnected outerplanar components like in Fig. 3.3.4. We have G_1 in the middle, G_2, .., G_n on the outside. Our analysis is similar to the previous cases when we had 2 biconnected outerplanar components or a path of biconnected components.

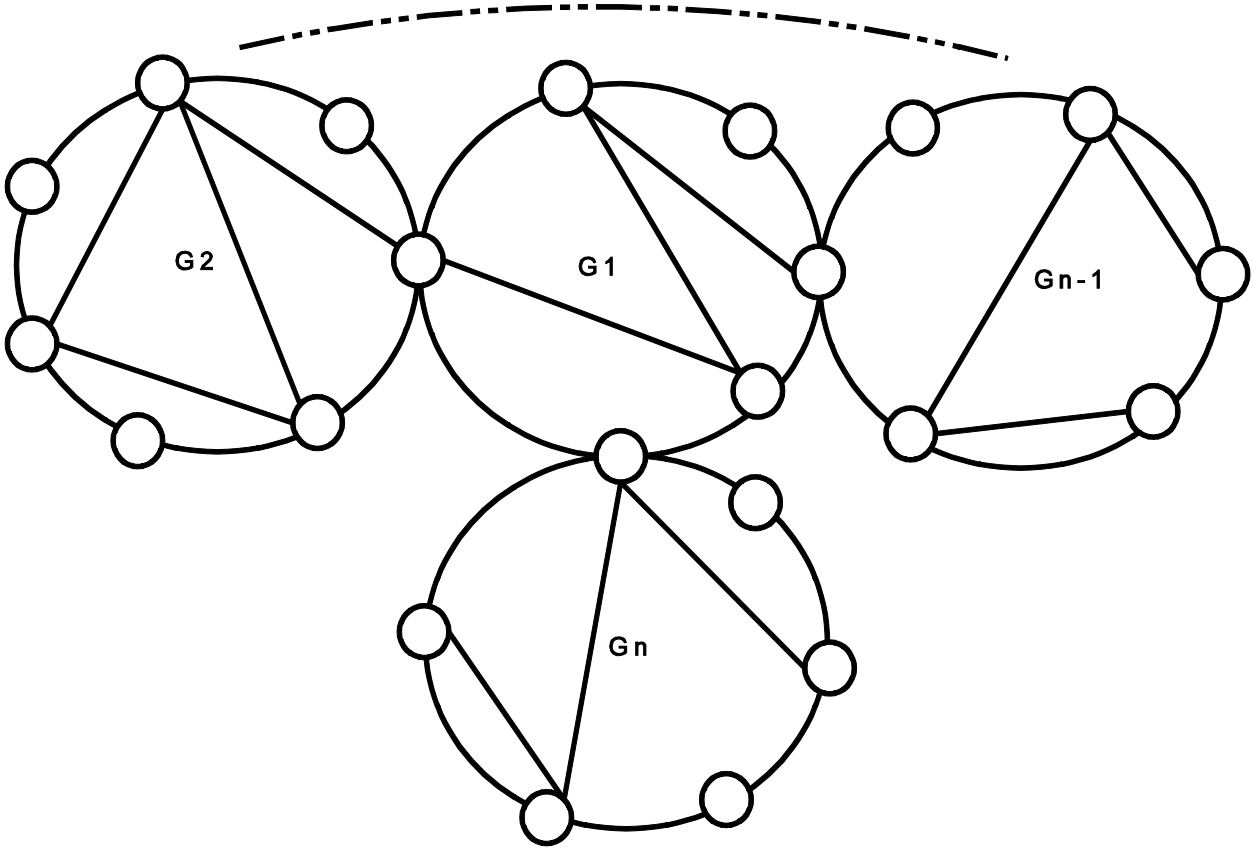


Figure 3.3.4 Biconnected outerplanar components arranged as in a star.

Algorithm BiconnectedStar

1. Find Opt(G_i) for all i = 1..n
2. Follow the optimal strategy for computing the optimal solution, Opt(G_1). During the optimal strategy for clearing G_1, whenever a pursuer reaches a vertex x that has a biconnected component G_i attached, use Opt(G_i) additional robots to clear G_i.
   //note that vertex x is guarded by the robot used during the optimal strategy for G_1

This algorithm will produce a solution that is at most $Opt(G_1) + max\{Opt(G_i) \mid i = 2..n\}$.

Theorem 3.3.2 The solution above is at most $Opt(G_1) + max\{Opt_{xx}(G_i) \mid i = 2..n\}$.

Proof:

We first compute the optimal solution for G_1 using the BiconnectedOuterplanar algorithm from section 3.2. The BiconnectedStar algorithm described earlier will follow the optimal solution for G_1, but whenever it reaches a vertex that has at least one biconnected component G_i attached, it will clear that component using an additional $Opt(G_i)$ robots. So this algorithm will use at most $Opt(G_1)$ robots plus an additional $max\{Opt(G_i) \mid i = 2..n\}$ robots required to clear the each component separately.

Theorem 3.3.3 $Opt(G_1) + max\{Opt(G_i) \mid i = 2..n\} \leq 2 \cdot Opt(G)$

Proof:

The optimal solution for G, $Opt(G)$, is at least $max\{Opt(G_i) \mid i = 2..n\}$.

Also, $Opt(G) \geq Opt(G_1)$. It follows that $Opt(G_1) + max\{Opt(G_i) \mid i = 2..n\} \leq 2 \cdot Opt(G)$.

Corolary 3.3.4 The BiconnectedStar algorithm is a 2-approximation algorithm.

Proof:

Follows immediately from Theorem 3.3.3

In this chapter we analyzed the original version of the pursuit evasion problem on 2 classes of graphs: interval graphs and biconnected outerplanar graphs. In Section 3.3 we also generalized our biconnected outerplanar solution to 2-biconnected outerplanar components, a path of biconnected outerplanar components and a star like arrangement of biconnected outerplanar components. These incremental steps get us closer to finding a solution to the outerplanar graphs class, which is a quite large and important graph class.

CHAPTER 4

THE SPINNING PROBLEM

This Chapter deals with a widely known problem which has applications in many areas, including wireless networks, sensor networks, military applications and robotics. Our version of the problem, which assumes the robots are equipped with directional antennas, has not been extensively studied. We first start with a description of our model, continue with some particular cases after which we present our results.

4.1 THE MODEL

We start by presenting the model for the problem.

1. We have a set of n static nodes at arbitrary locations on a plane and equipped with directional antennas;
2. Every node has a unique ID from 1 to n.
3. Each antenna has an unknown starting orientation (starting positioning angle);
4. All antennas have the same beamwidth α, $0 \leq \alpha \leq 2\pi$;
5. Two nodes meet (i.e. discover each other) if both emit in the other's direction simultaneously;
6. a) Every node is equipped with a device capable of providing AOA (Angle of Arrival);

   OR

   b) The direction information is included in the transmitted signal;
7. All nodes are in each other's range (the graph forms a clique);

8. Every antenna j, $j=1..n$, rotates clockwise at speed v_j, $Z \leq v_1 \leq v_2 \leq \ldots \leq v_n \leq V$, where v_j is expressed in the number of rotations per unit of time and Z,V are constants;

9. All antennas have the same transmission power, frequency channel and modulation technique.

We want to find a set $\{v_1, \ldots, v_n\}$, such that all n nodes will meet (i.e. each node will discover every other node).

Definition. We call a solution to the spinning problem with given n, α and V, a pair $(\{v_1, \ldots, v_n\}, t)$ where $\{v_1, \ldots, v_n\}$ is a set of speeds such that the n nodes will meet in at most t time, for any distribution of the nodes on the plane and any starting positioning of the antennas.

Definition. An optimal solution to the spinning problem with given n and α, is a solution $(\{v_1, \ldots, v_n\}, t)$ such that there exists no other solution $(\{v'_1, \ldots, v'_n\}, t')$ such that $t' < t$.

We assume that the system of coordinates is defined prior to the deployment.

Definition. The starting angle of an antenna is the angle formed by the beam and the positive x axis in the initial deployment of the device.

Problem: How to find the optimal solution to the spinning problem? In case the problem is proven to be NP-complete, how to find a solution that is close to the optimal solution?

We can see that if $\alpha \geq \pi$ then any 2 beams meet during a full spin of the slower beam. And that occurs no matter what starting positions and what speeds they have (even if they have the same speed). The optimal solution in this case would be to set maximum speed (V) for all beams such that the full spin occurs as fast as possible. Hence, this case does not interest us. From now on we assume that $\alpha < \pi$.

Also, it is not possible that $V = v_1$ because that would mean all speeds must be equal, so for $\alpha < \pi$ it is possible to place the antennas such that the beams never meet (in the next section we show this for the case of 2 antennas rotating at the same speed).

Why is the problem interesting? Naturally, we assume that spinning requires energy and the energy consumption rises proportionally with the speed. Hence, if we spin longer or faster the energy consumption grows. Also, small α is better because we consume less energy when sending signals and we have lower chances of interference.

We want to minimize the energy consumption by finding a trade-off between the beamwidth, speed and meeting time. This leads to the upper and lower bounds Z and V on speed. If Z or V are exceeded then the energy consumption would be too high and not worth considering. Now we want to minimize the meeting time for antennas rotating at speeds between Z and V. Even more, we want to determine the speeds such that all n antennas will eventually meet.

4.2 PARTICULAR CASES

If we do not choose the speeds carefully some antennas might never meet. For example, consider the case $n = 2$, $\alpha < \pi/2$ and a starting position like in the following figure where the second beam is very close to the meeting point:

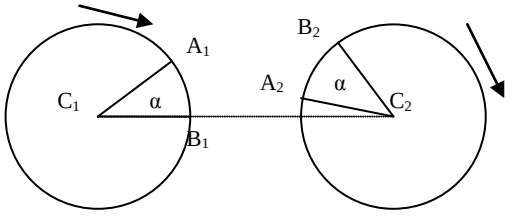


Figure 4.2.1 The two beams rotating clockwise are represented as sections of a circle.

For ease of representation, we consider C_1 and C_2 to be the circles described by the rotations. The sections (A_1, B_1) and (A_2, B_2) of the circles represent the beams.

Consider the segment C_1C_2 which connects the locations of the 2 beams. We say that beam 1 is available for meeting if points A_1 and B_1 are on opposite sides of the segment C_1C_2. In the Fig. 4.2.1, beam 1 is available, but beam 2 is not.

Next, we prove a trivial claim in order to introduce the reader to our notations.

Claim 4.2.1 It is possible that the 2 beams will never meet.

Proof:

Let us choose $v_2 = 2v_1$ and $\alpha = \pi/6$. By the time beam 1 ends being available (so it rotates α distance), beam 2 rotates 2α. No meeting occurs in this time.

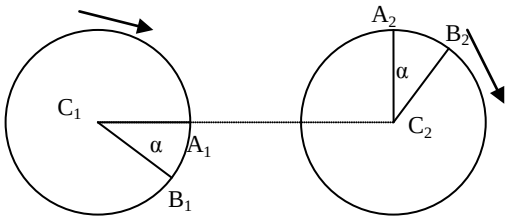


Figure 4.2.2 The two beams after beam 1 ends being available.

After $t_1/2$ time (where t_1 is the time it takes the first beam to make a full rotation) we have a placement where beam 2 has passed over the meeting point and beam one has moved half the distance:

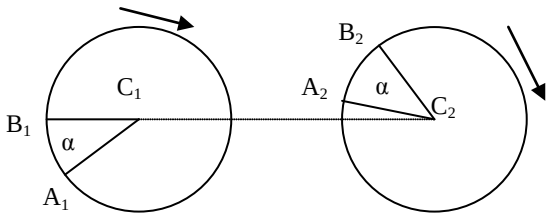


Figure 4.2.3 The two beams after $t_1/2$ time.

After t_1 time (since the initial deployment) the two beams will be in the starting positions. Beam 2 has completed two full circles while beam 1 has completed one circle. Now the scenario repeats, so the 2 beams will never meet.

So if we choose $v_2 = 2v_1$ the beams might never meet. In fact, if we choose $v_2 = k \cdot v_1$ the 2 beams might never meet ($v_2 = k \cdot v_1$ must be smaller than the upper bound V). Of course, this also depends on α. If α is bigger then the second beam might reach the meeting point before beam 1 ends being available, so they meet. The conclusion is that there is a chance (which depends on α, speeds and the starting positions) that the two beams will never meet.

Now let us consider the particular case in which $\alpha = 0$. This would correspond to having laser rays (which are straight lines with no angle) instead of antenna beams. We cannot guarantee that the antennas will always meet in this case; it depends on the starting positions. The following Lemma states this formally for $n = 2$:

Lemma 4.2.2 If $\alpha = 0$ then $\forall v_1, v_2 \exists \beta_1, \beta_2$ starting angles such that the spinning problem has no solution.

Proof:

We show that there is a starting position that will guarantee the 2 beams will never meet. The proof is non-constructive.

Let $\beta_1 = 0$, $\beta_2 = \pi$ (the antennas start in the meeting position) like in Fig. 4.2.4:

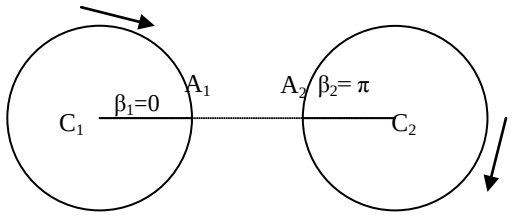


Figure 4.2.4 The starting position of 2 rays.

Now, there are two possibilities: the antennas will meet again, or they will never meet again.

Case 1: The antennas will never meet again.

In this case, we can choose the positioning of the 2 antennas at time t_1 (after one full rotation of the first beam) like in Fig 4.2.5.

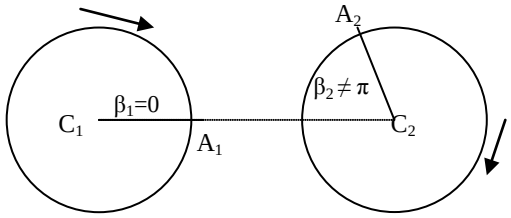


Figure 4.2.5 The starting position after one full rotation of C_1.

Since, by our initial assumption, the two beams never meet after t_1, the new β_2 will be different from π. So if we let the starting positions be these β_1, β_2, according to our assumption for this case, the 2 beams will never meet.

Case 2: The antennas will meet again after k rotations of beam 1, $k \in N$, $k \geq 2$ but not earlier (A_1 will be at the starting point only after a number of full rotations).

If the antennas started at the meeting position and they will meet again, it means we have a periodic pattern. That is, they will meet again after $2k$ rotations, $3k$ rotations, and so on.

Now, after (k - 1) rotations we will have a positioning like in Fig. 4.2.6. We will consider δ_1 the angle between the second beam and the meeting point after 1 full rotation of the first beam, δ_2 the angle after 2 rotations … δ_{k-1} after $k-1$ rotations. Let δ be the smallest of them, which occurs after rotation j, where $1 \leq j \leq k - 1$.

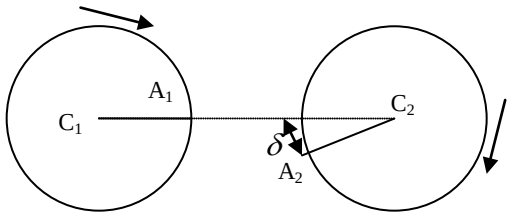


Figure 4.2.6 The 2 beams after $k - 1$ rotations.

Now, if we choose other starting positions, with $\beta_1 = 0$ and $\beta_2 = \delta/2$ it means that after j rotations beam one will be at the starting position, beam two will be $\delta/2$ "behind". After k rotations, the second beam will be $\delta/2$ "ahead".

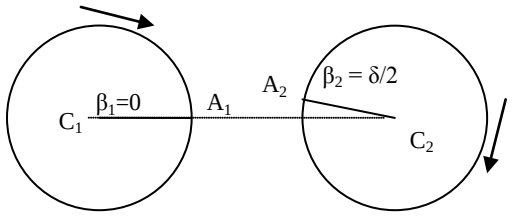


Figure 4.2.7 The 2 beams after *k* rotations.

As we stated earlier, we have a pattern. They will not meet after $i \cdot j$ rotations, nor after $i \cdot k$ rotations, with $i \in N$. But they will never be as close to meeting position as after $i \cdot j$ rotations or $i \cdot k$. That is because in the rest of the time the distance between beam 2 and the meeting point will always be greater than $\delta/2$. So the two beams will never meet.

Hence, in either case, we can construct a starting position such that the 2 beams will never meet.

4.3 PRELIMINARIES

In this section we make a few assumptions about the Spinning Problem which will enable us to formally represent all the parameters of the problem. That is because in practice there will always be diffractions at the edges of the beams thus making the signal at the edges weaker than closer to the center. Hence we can "approximate" the continuous model with a discrete one.

First we assume that α is a rational number, i.e. $\alpha = a/b$, where $a, b \in Z, b \neq 0$. This enables us to partition the circle formed by the rotation of the beam into *m* equal sections, where *m* is a

multiple of b. Furthermore, the starting angles will also be a rational number of the form c/m. Note that it is possible to choose $m = b$.

These assumptions allow us to reduce the starting position and the angle α to a natural number, representing the number of sections it covers. So an angle α of k means α covers k sections. Also we represent time as time units, without the concern of what a time unit really corresponds to in the real world. The above assumptions enable us to formalize the problem and solve it more rigorously.

Now we consider that the first beam rotates k_1 sections in one time unit and the second beam k_2, with k_1 and k_2 natural numbers. S_1, $S_2 \geq 0$ are the starting positions of the two beams.

We assume that the first beam starts in the meeting position, which will be true within some finite time after the initial deployment. S_2 varies, and we consider $S_2 = 0 = m$ to be the meeting position.

Next we present the particular case when $\alpha = 0$. The theory presented here is an introduction for the general case. However, it cannot be applied in the real world. We have already proved that for $\alpha = 0$, we cannot guarantee the 2 beams will ever meet. Now we have a few additional assumptions about our model which allow us to analyze this particular case: α is a rational number and the circle has m sections. We can see that as m tends to infinity, our discrete model allows more flexibility in the selection of parameters, and it “approaches” the continuous model. However, in order for meeting to occur, beam 2 must start at one of the m points on the circle, and not somewhere in-between. These points are imaginary, but they are not flexible: one point on each circle must be placed such that when the beams are positioned on them, they are aligned (i.e. we have meeting).

But the devices are dropped arbitrarily, and we cannot know in advance if the beams will start on one of the points. This may not matter when $\alpha > 0$, but when $\alpha = 0$, it is crucial the beams start on the points. So we just assume this happens, in order to provide the idea behind the proofs for the general case.

Given these assumptions, we can now present the case $\alpha = 0$.

Lemma 4.3.1 For $\alpha = 0$ and for given S_2, k_1, k_2 and m, we have meeting iff $\exists\, i, j \in N$ such that $i \cdot m \cdot k_2/k_1 = j \cdot m - S_2$

Proof:

Note that if $S_2 = 0$ then we have meeting already, so i and j will be 0. The first beam will rotate a full circle in m/k_1 time.

In m/k_1 time, the second beam will rotate $m \cdot k_2/k_1$. For $\alpha = 0$, the two beams will meet if and only if the second beam will be at the meeting point at the same time with the first beam. The first beam will be at the meeting point after every m/k_1 time units. But every m/k_1 time units the 2nd beam will rotate $m \cdot k_2/k_1$ distance.

In order to be at the meeting point after i rotations of the first beam, the 2nd beam will have to rotate a total distance of $j \cdot m - S_2$ (j full rotations minus the starting shift). Therefore, for given S_2, k_1, k_2 and m, if $\exists\, i, j \in N$ such that $i \cdot m \cdot k_2/k_1 = j \cdot m - S_2$, then we have meeting.

We are interested in the smallest natural i which satisfies this equation.

We can write the previous equation as $i \cdot m \cdot k_2 = j \cdot m \cdot k_1 - S_2 \cdot k_1$. And since S_2, k_1, k_2 and m are known, we can rewrite the equation as $j \cdot a - i \cdot b = c$ with $a, b, c \in N$. This is a linear Diophantine equation with the form $a \cdot x + b \cdot y = c$. There exists a polynomial time algorithm

for solving Diophantine equations which is not presented in this dissertation but the reader can find it at [19].

Corollary 4.3.2 For $\alpha = 0$ if beams meet then $k_1 = p \cdot m$, $p \in N^+$.

Proof:

We have solution iff $\exists\, i, j \in N$ such that $j \cdot m \cdot k_1 - i \cdot m \cdot k_2 = S_2 \cdot k_1$

We know from the Diophantine Equation algorithm that the *gcd(a,b)* must divide *c*, in other words $m \cdot gcd(k_1, k_2)$ must divide $S_2 \cdot k_1$, $\forall\ S_2 = 1..m$. In particular, for $S_2 = 1$, m must divide k_1, which means k_1 can be written as $k_1 = p \cdot m$, $p \in N^+$

Theorem 4.3.3 For $\alpha = 0$ the two beams meet after at most m-1 spins of the first beam in the worst case, or they never meet.

Proof:

After each rotation of the 1st beam, the second beam will visit (be positioned on) one of the m points of the circle. Let p_i be the point the second beam is positioned after i spins. After a number of rotations (let us say r rotations) the beam will be again positioned at point $p_0 = S_2$. After this, all positions become repetitive, meaning that $p_0 = p_r$, $p_1 = p_{r+1}$… and so on. If $r < m$ then there exist some points on the circle that have not been visited. Hence, there exists a starting position for which the meeting point will not be visited, therefore there is no meeting. So in order to have meeting, the second beam must visit all m points on the circle. In the worst case, the meeting point is last visited; therefore the two beams will meet after at most m-1 spins in the worst case.

We now extend our analysis to the case $\alpha > 0$.

Theorem 4.3.4 Given k_1, k_2, S_2, m, if $\alpha > 0$ then we have meeting iff $\exists\, i, j \in N$ such that

$0 \geq i \cdot m \cdot k_2/k_1 + S_2 - j \cdot m \geq -\alpha \cdot k_2/k_1 - \alpha$

Proof:

In the case $\alpha > 0$ we consider S_2 to be the distance between the meeting point and the rear margin (A_2) of the second beam, like in the Fig. 4.3.1:

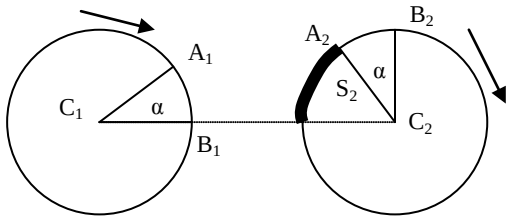


Figure 4.3.1 Starting position S_2.

Every m/k_1 time, the first beam will be in the same position as the initial position and it will be available for α/k_1 time. Where should the second beam be positioned on the circle in order to have meeting? If after m/k_1 time the rear margin (A_2) of the second beam is positioned on the meeting point, then we have meeting (Fig. 4.3.2).

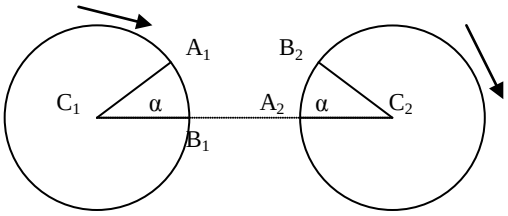


Figure 4.3.2 Meeting after m/k_1 time.

Or, if the front margin (B_2) is $\alpha \cdot k_2/k_1$ away from the meeting point we still have meeting (Fig. 4.3.3). That is because the first beam will be available for α/k_1 time, and in this time the

second beam will rotate a distance of $\alpha \cdot k_2/k_1$. In this case, the rear margin is $\alpha + \alpha \cdot k_2/k_1$ away from the meeting point.

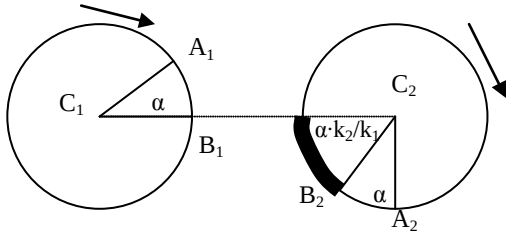


Figure 4.3.3 Meeting if B_2 is $a \cdot k_2/k_1$ away.

These are the 2 extreme cases. If the 2nd beam is anywhere between these points, we will have meeting. So, if the rear margin is between 0 (the meeting point) and $0 - (\alpha + \alpha k_2/k_1)$ (the distance beam 2 will rotate in α/k_1 time plus the distance α to the rear margin) then we have meeting.

The total distance the rear margin of the 2nd beam must rotate in $i \cdot m/k_1$ time should be smaller than $j \cdot m - S_2$ and bigger than $j \cdot m - S_2 - (\alpha + \alpha \cdot k_2/k_1)$.

Hence,

$i \cdot m \cdot k_2/k_1 \leq j \cdot m - S_2$ and $i \cdot m \cdot k_2/k_1 \geq j \cdot m - S_2 - \alpha \cdot k_2/k_1 - \alpha$

Or,

$i \cdot m \cdot k_2/k_1 - j \cdot m + S_2 \leq 0$

and

$i \cdot m \cdot k_2/k_1 - j \cdot m + S_2 \geq -\alpha \cdot k_2/k_1 - \alpha.$

Therefore, in the case $\alpha > 0$ we have meeting iff $\exists i, j \in N$ such that

$0 \geq i \cdot m \cdot k_2/k_1 + S_2 - j \cdot m \geq -\alpha \cdot k_2/k_1 - \alpha$

4.4 AN ALGORITHM FOR COMPUTING THE MEETING TIME

4.4.1 DESCRIPTION OF THE ALGORITHM

The inequality of Theorem 4.3.4 can be also written as:

$$-S_2 \cdot k_1 \geq i \cdot (m \cdot k_2) - j \cdot (m \cdot k_1) \geq -S_2 \cdot k_1 - \alpha \cdot (k_1 + k_2)$$

We need to solve all Diophantine Equations

$i \cdot (m \cdot k_2) - j \cdot (m \cdot k_1) = c$ where $c \in Z$

and

$$-S_2 \cdot k_1 \geq c \geq -S_2 \cdot k_1 - \alpha \cdot (k_1 + k_2).$$

Then, for every solution i, we must compute the exact meeting time. Because at this point we only know the two beams will meet after i full rotations of the first beam, but we do not know exactly when this meeting occurs. First we compute the position of the second beam after $i \cdot m/k_1$ time. Let p_i be the position of the rear margin of the second beam after $i \cdot m/k_1$ time.

$$p_i = (i \cdot m \cdot k_2/k_1) \bmod m$$

If $m - p_i \leq \alpha$ then we have the situation from Fig. 4.4.1:

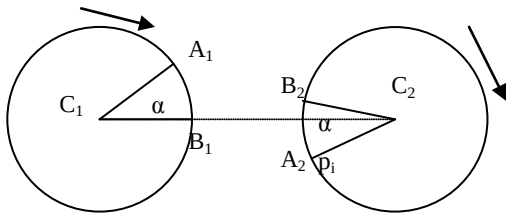


Figure 4.4.1 When $m\text{-}p_i \leq \alpha$.

Hence, after exactly $i \cdot m/k_1$ time, the 2 beams meet. But if $m - p_i > \alpha$, then after $i \cdot m/k_1$ time the two beams are positioned like in Fig 4.4.2:

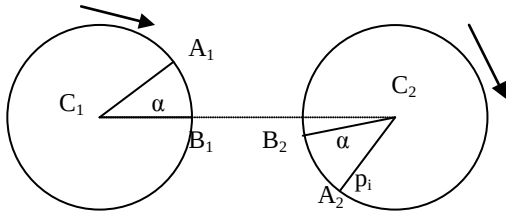


Figure 4.4.2 When $m\text{-}p_i > \alpha$.

Hence, it will take a while until the second beam will be in the meeting position. The second beam will have to rotate the distance between B_2 (the front margin) and the meeting point. That distance is $m - p_i - \alpha$. Hence, the exact meeting time is $i \cdot m/k_1 + (m - p_i - \alpha)/k_2$.

After solving all the Diophantine Equations, we can choose the smallest solution as the best meeting time for this choice of k_1, k_2, S_2, m.

We remind the reader that this is the meeting time if we consider the starting position of the first beam like in Fig. 4.4.3:

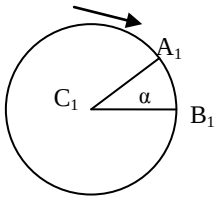


Figure 4.4.3 Starting position of beam 1.

Now we need to compute the meeting time for any starting positions.

Lemma 4.4.1 Assume t is the worst case meeting time of the 2 beams if $S_1 = 0$ and $0 \leq S_2 \leq m - 1$ (computed by our algorithm). Then the worst case meeting time for any S_1, S_2 is smaller than $t + m/k_1$.

Proof: For any S_1 the first beam will have to rotate less than a full spin to get into position $S_1 = 0$ (at most $(m - 1)/k_1$ time). So it will take less than m/k_1 to reach $S_1 = 0$ and once it is there, it will take t time to meet.

Theorem 4.4.2 Given k_1, k_2, S_1, S_2, m and $\alpha > 0$ we have meeting iff $\exists\, i, j \in N$ such that

$0 \geq i \cdot m \cdot k_2/k_1 + S_2 + (m - S_1) \cdot k_2/k_1 - j \cdot m \geq -\alpha \cdot k_2/k_1 - \alpha$ or

$(\alpha - S_1) \cdot k_2/k_1 + S_2 \geq m - \alpha$ and $S_1 < \alpha$.

Proof:

In Lemma 4.3.1 we proved the same result but for $S_1 = 0$. If $S_1 > 0$ we have two possibilities: either the two beams meet during the first full rotation of the first beam or the beams meet later.

The first case is a particular case, and may occur only if $S_1 < \alpha$ (so the first beam is already available) and the second beam rotates so fast, that it will reach the meeting point before

beam one ends being available. That means, in $(\alpha - S_1)/k_1$ time, beam two will rotate more than $m - \alpha - S_2$ distance.

Hence

$(\alpha - S_1) \cdot k_2/k_1 + S_2 \geq m - \alpha.$

In the second case for $S_1 = 0$, the equation was

$0 \geq i \cdot m \cdot k_2/k_1 + S_2 - j \cdot m \geq -\alpha \cdot k_2/k_1 - \alpha$

Since the beams do not meet during the first rotation of the first beam, then we can consider the problem with $S_1 = 0$ but with a different (later) S_2. S_2 will be S_2 plus the distance beam two rotates in $(m - S_1)/k_1$ time (the time it takes until beam one reaches position $S_1 = 0$). Therefore, the new S_2 will be $S_2 + (m - S_1) \cdot k_2/k_1$. Replacing S_2 in the equation we get

$0 \geq i \cdot m \cdot k_2/k_1 + S_2 + (m - S_1) \cdot k_2/k_1 - j \cdot m \geq -\alpha \cdot k_2/k_1 - \alpha$

How to compute the exact meeting time for any initial shift of both beams?

First we compute the meeting time t when $S_1 = 0$, using the previous algorithm. Then we try to find the exact snapshot of the meeting: exactly what point of the first beam will overlap the meeting point? In order to do this, we first compute the distance d between the front margin B_1 and the meeting point after t time.

$d = (k_1 \cdot t) \bmod m$

For any $S_1 < \alpha - d$ the meeting will occur at the same time as for $S_1 = 0$ because after t time the distance between the front margin and the meeting point will be smaller than $\alpha - d + d = \alpha$. Hence, the meeting point will be between the two margins, therefore we have meeting.

For any $S_1 \geq \alpha - d$ the meeting will not occur at the same time as for $S_1 = 0$ (same justification). Therefore, it will occur later. When? The beam will first have to get in position $S_1 = 0$. Once it is there, it will take t time to meet. Hence, the first beam will have to rotate $m - \alpha + d$ distance.

In this case, the worst case meeting time is $t + (m - \alpha + d)/k_1$.

4.4.2 ALGORITHM FOR TWO DEVICES

In this section we present the algorithm for finding the best choices of speed for 2 devices. The following algorithm computes a matrix which has rows and columns in the range $Z..V$. Every value at row x and column y represents the meeting time if $k_1 = x$ and $k_2 = y$. To find the best meeting time we need to compute the minimum meeting time inside the generated matrix (which can easily be done in $O((V - Z + 1)^2)$) additional time.

```
MatrixGeneration{
    for k1=Z to V-1 do
      for k2=k1+1 to V do
      {
         a = m*k2;
         b = -m*k1;
         g = gcd(a,b);
         max = 0;            // max is the longest time
                             // until we guarantee meeting
         for S=1 to m do
         {
           min = +∞;    // min is the soonest meeting time
           for c=-S*k1-alpha*(k1+k2)to -S*k1
              if c mod g = 0 then
              {
                a2 = a/g;
                b2 = b/g;
                c2 = c/g;
                solve Diophantine Equation a2*i+b2*j=c2;
                                    // return a solution (i,j)
                find minimum strictly positive solution (i,j)
                if (min > i)
                min = i;
              }
              if (max < min)
              max = min;
         }
         Matrix[k1][k2] = max + m/k1;
                                    //we add one more rotation
                                    //to the final meeting time
    }
}
```

For a better understanding of the MatrixGeneration algorithm, the reader should be familiarized with the algorithm for solving linear Diophantine Equations ([19]).

It is obvious that the matrix is generated in polynomial time. Also, finding the best meeting time for 2 devices can be done in polynomial time. But in order to determine the best time for *n* devices, we would have to check all possible solutions, because a good meeting time for devices 1 and 2 might result in a very bad meeting time between devices 2 and 3, etc. Hence, we have an exponential running time solution for the *n*-device Spinning Problem.

4.4.3 ALGORITHM FOR N DEVICES

Next, we present the algorithm for computing the best meeting time for *n* devices using the matrix generated by the previous algorithm. This algorithm is recursive and it has exponential running time. For every possible solution *A* (where *A* is an array of *n* speeds) we compute the worst case meeting time (compute_time function) then we find the minimum of all these meeting times.

```
spin()
{
  A[0] = Z-1;
  compute(1);
  min = +∞;
  for all solutions A
  {
     if min > time(A)  then
     {
      min = time(A);
      best_solution = A;
              // and the best solution (choice of speeds) is A
     }
  }
}
compute(int i)
{
  if i<n then
    for A[i]=A[i-1]+1 to V-n+i
      compute(i+1);
  else
    for A[i]=A[i-1]+1 to V-n+i
      time(A)  = compute_time(A);
}
int compute_time(array A)
{
  max=0;
  for j=1 to n-1
    for k=j+1 to n
      if max<Matrix[A[j]][A[k]] then
        max=Matrix[A[j]][A[k]]
  return max;
}
```

In this Chapter we presented and analyzed a model for neighbor discovery in static wireless ad-hoc networks using directional antennas. In Chapter 5 we discuss about future work related to the Spinning Problem.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this research we have analyzed problems and algorithms that can be applied in robotics. The pursuit evasion problem, either vision-based or non-vision based, can be used in practice by autonomous or human controlled robots to search an area. A system of caves, streets or buildings, are some examples that can be represented as graphs.

Current limitations in robotics might not allow a full search by autonomous robots, especially if the fugitive is human. However, remotely controlled robots could efficiently clear an area if the optimal solution exists. The human controller could simply follow the steps indicated by the algorithm, thus guaranteeing the capture (or detection) of the evader.

The pursuers do not have to be robots. They could be policemen (or police cars) chasing a fugitive in a system of streets. Given the vision-based algorithm or a similar variant, two police cars could corner the fugitive if the drivers will follow the optimal steps indicated by our algorithm. A chase that otherwise might take hours, could be shortened or at least capture will be guaranteed.

In Chapter 2 of this dissertation we analyzed the vision based pursuit evasion problem. In the discrete model, we presented three solutions, one that requires a pursuer faster than the fugitive and another two that requires 2 pursuers with the same speed as the evader and with or without direction detection capability.

Our main result, Theorem 2.2.1, is an improvement over the 3-pursuer strategy of Dumitrescu et al. [4] and Sugihara and Suzuki's 4-pursuer strategy [3]. Even though our pursuers

have twice the speed of the fugitive and direction detection capability, we only use 2 pursuers instead of 3 or 4.

In the continuous model, 1 pursuer with speed *k*, where *k* is a constant cannot guarantee capture. In that regard, our algorithm is optimal, since it requires the minimum number of pursuers.

However, an open question is: can we guarantee capture if the speed of the two pursuers is less than 2? Phase 1 of our algorithm, which ends when Z is spotted, does not require the pursuers to have speeds higher than 1. Phase 2 however, does not work unless *A* and *B* have a speed of 2. The challenge is to come up with an alternate phase 2 that would work for speeds of 1, even if that would require slightly modifying the model.

Another way to improve our results is to find an algorithm that requires two pursuers, but captures the fugitive in less than $O(n^2)$ time.

Other models in which the speed of the fugitive is *s* while the robots move at a speed of 1 or somewhere in between are also worth analyzing.

One question we find interesting is: is it possible to expand our algorithms to a 3D grid? Will adding another dimension result in a completely different algorithm?

In Chapter 3 we presented our version of the solution for the pursuit evasion problem on interval graphs. We provided some characterizations for the optimal solution that were not previously published.

Our main result in Chapter 3 is the biconnected outerplanar algorithm. Our algorithm has a running time of $O(n^5)$ but we believe that under a tighter analysis and with some modifications, including a more efficient algorithm for matrix chain multiplications, it can achieve a running time between $O(n^2 \log n)$ and $O(n^3)$. Some real-world models which could have a biconnected

outerplanar graph representation are stadiums or some other particular types of buildings. There is not a huge variety of real world models that can be represented using biconnected outerplanar graphs, which is why our goal is to expand these results to bigger classes of graphs.

However, even as it is right now, our algorithm represents a very important step towards finding a solution for outerplanar graphs or for series-parallel graphs. It shows that the pursuit evasion problem is complex even on very restricted classes of graphs. The difficulty increases significantly when we try to extend our analysis to two biconnected outerplanar components, a path or a star of biconnected outerplanar components.

An important problem for future research is finding the optimal solution for trees of biconnected outerplanar components. This would mean a solution for the entire class of outerplanar graphs. Of course, a first step is finding the optimal solution for a path or a star of biconnected outerplanar components (that is not just within 1 of the optimal solution, or a 2-approximation).

We have spent significant time trying to find a solution for series-parallel graphs. However, the problem seems very complex, and we were unable, so far, to either find a polynomial time algorithm or prove the problem is NP-complete on this class. Looking at series-parallel posets [64, 65] might provide new insight for the pursuit evasion problem on series-parallel graphs which could lead to a polynomial time algorithm.

Future research could also involve looking at other types of graphs. Circular-arc graphs are similar to interval graphs, therefore an algorithm might not be too difficult to find.

Neighbor discovery using omni-directional antennas is an important area of theoretical research and real world applications. Dropping robots on a plane has numerous military and non-

military applications. Many articles, related to this problem, have been published in the past years. In most of these papers, the robots use omni-directional antennas due to their simplicity.

However, the directional antennas variant, even though it has some advantages, has not been studied extensively. This is mainly due to limitations in technology, but also to the fact that good solutions already exist using omni-directional antennas. However, we believe that given the advantages of directional antennas, if an efficient *n*-device solution could be found, then the area of neighbor discovery would be dominated by the directional antennas.

In chapter 4 we have presented and analyzed a model for neighbor discovery in static wireless ad-hoc networks using directional antennas. We defined our variant of the problem and made some assumptions about our model after which we provided theoretical descriptions of the solutions and a simple algorithm.

The solution for two devices is intuitive and quite simple. Even though we have presented a method to determine the optimal solution to the Spinning Problem for *n* devices, this method requires exponential time. The next step in solving this problem is proving that it is NP-Hard, or finding a polynomial time algorithm.

Another approach could be to use random rotation speeds and compute the discovery probability or the expected meeting time.

Changing the model from a plane to a 3D environment is another direction this problem could be expanded.

This dissertation is the result of many years of research. In some cases our work reached dead ends, conjectures that after weeks of work proved to be incorrect, partial results that were not interesting or complex enough to be worth mentioning or even results that after months of

work, we discovered, have been previously published under a different name. It has been a long road, with ups and downs, but we have finally reached a point where we can say that we are satisfied with the quality of this work. Even though we cannot claim we have completely solved the problems presented in this dissertation, our results represent a step forward and a good contribution to the area. We intend to continue this research and improve or build on the algorithms presented here.

REFERENCES

[1] A. LaPaugh, “Recontamination does not help to search a graph”, tech. rep., Electrical Engineering and Computer Science Department, Princeton University, 1983.

[2] A. S. LaPaugh, “Recontamination does not help to search a graph”, J. Association for Computing Machinery, 40 (1993), pp. 224–245

[3] D. Bienstock and P. Seymour, “Monotonicity in graph searching”, J. Algorithms, 12 (1991), pp. 239–245.

[4] P. D. Seymour and R. Thomas, “Graph searching and a min-max theorem for tree-width”, J. Combin. Theory Ser. B, 58 (1993), pp. 22–33.

[5] V. Y. Andrianov and N. N. Petrov, “Graph searching problems with the counteraction, in Game theory and applications”, Vol. X, vol. 10 of Game Theory Appl., Nova Sci. Publ., Hauppauge, NY, 2005, pp. 1–12.

[6] L. M. Kirousis and C. H. Papadimitriou, “Searching and pebbling”, Theoretical Computer Science, 47 (1986), pp. 205–218.

[7] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, “The complexity of searching a graph”, J. Assoc. Comput. Mach., 35 (1988), pp. 18–44.

[8] G. Gottlob, N. Leone, and F. Scarcello, “Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width”, J. Comput. System Sci., 66 (2003), pp. 775–808.

[9] B. Alspach, “Searching and Sweeping Graphs: A Brief Survey”, in COMBINATORICS04, 2004

[10] T. D. Parsons, “Pursuit-evasion in a graph”, in Theory and applications of graphs, vol. 642 of Lecture Notes in Math., Springer, Berlin, 1978, pp. 426–441.

[11] F. V. Fomin, D. M. Thilikos, “An annotated bibliography on guaranteed graph searching”, Theoretical Computer Science, Vol. 399, issue 3 (June 2008), 236-245

[12] A. K. Dewdney, “Embedding Graphs in Euclidean 3-Space”, The American Mathematical Monthly, Vol. 84, No. 5 (May, 1977), pp. 372-373

[13] B. Monien and I.H. Sudborough, “Min cut is NP-complete for edge weighted trees”, Theoretical Computer Science, 58(1988), 209-229

[14] S.-L. Peng, M.-T. Ko, C.-W. Ho, T.-s. Hsu, and C. Y. Tang, “Graph searching on chordal graphs”, ISAAC’96, Lecture Notes in Computer Science, Vol. 1178, pp. 156–165, 1996.

[15] S. Vasudevan, J. Kurose, D. Towsley “On Neighbor Discovery inWireless Networks With Directional Antennas”. 24th Annual Joint Conference of the IEEE Computer and Communication Societies, Proceedings IEEE. Infocom 2005.

[16] R. Wattenhofer, Li Li, P. Bahl, Y. Wang ”Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad Hoc Networks”. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Anchorage, Alaska, April 2001.

[17] Li Li, J.Y. Halpern, P. Bahl, Y. Wang, R. Wattenhofer. “Analysis of a Cone-Based Distributed Topology Control Algorithm for Wireless Multi-hop Networks”. Twentieth ACM Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island, August 2001.

[18] N. Malhotra, M. Krasniewski, C. Yang, S. Bagchi, W. Chappell “Location Estimation in Ad-Hoc Networks with Directional Antennas”. The 25th International Conference on Distributed Computing Systems (ICDCS 2005), Columbus, Ohio, June 2005.

[19] Eric W. Weisstein, "Diophantine Equation." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/DiophantineEquation.html

[20] N. N. Petrov and N. N. Petrov, The “Cossack-robber” differential game, Differentsial'nye Uravneniya, 19 (1983), pp. 1366–1374

[21] N. D. Dendris, L. M. Kirousis, and D. M. Thilikos, “Fugitive-search games on graphs and related parameters”, Theoret. Comput. Sci., 172 (1997), pp. 233–254.

[22] D. Richerby and D. M. Thilikos, “Graph searching in a crime wave”, in Procceeding of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2007), vol. 4769 of Lecture Notes in Comput. Sci., Springer, 2007, pp. 21–32.

[23] B. Yang and Y. Cao, “Searching digraphs: Monotonicity and complexity”, Tech. Rep. TR 2006-06, Department of Computer Science, University of Regina, Canada, 2006.

[24] R. J. Nowakowski, “Search and sweep numbers of finite directed acyclic graphs”, Discrete Appl. Math., 41 (1993), pp. 1–11.

[25] P. Hunter and S. Kreutzer, “Kelly decompositions, games, and orderings”, in Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), New York, NY, USA, 2007, ACM Press, pp. 637–644.

[26] G. Gottlob, N. Leone, and F. Scarcello, “Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width”, J. Comput. System Sci., 66 (2003), pp. 775–808.

[27] L. Barri`ere, P. Fraigniaud, N. Santoro, and D. M. Thilikos, “Searching is not jumping”, in Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2003), vol. 2880 of Lecture Notes in Comput. Sci., Springer, 2003, pp. 34–45.

[28] R. Borie, C. Tovey and S. Koenig. “Algorithms and Complexity Results for Pursuit-Evasion Problems”. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2009.

[29] N. N. Petrov, “Some extremal search problems on graphs”, Differentsial′nye Uravneniya, 18 (1982), pp. 821–827.

[30] I. Adler, “Marshals, monotone Marshals, and hypertree-width”, J. Graph Theory, 47 (2004), pp. 275–296.

[31] I. Suzuki, Y. Tazoe, M. Yamashita, and T. Kameda, “Searching a polygonal region from the boundary”, Internat. J. Comput. Geom. Appl., 11 (2001), pp. 529–553.

[32] S. Arnborg, D. G. Corneil, and A. Proskurowski, “Complexity of finding embeddings in a k-tree”, SIAM J. Algebraic Discrete Methods, 8 (1987), pp. 277– 284.

[33] F. Makedon and I. H. Sudborough, “On minimizing width in linear layouts”, Discrete Appl. Math., 23 (1989), pp. 243–265.

[34] K. Sugihara, I. Suzuki, “On a pursuit-evasion problem related to motion coordination of mobile robots”. Proc. the 21st Hawaii Int. Conf. on System Sciences, pp. 218–226. Kailua-Kona, Hawaii (1988)

[35] K. Sugihara, I. Suzuki, “Optimal algorithms for a pursuit-evasion problem in grids”. SIAM Jorunal of Discrete Mathematics 2(1), pp.126–143 (1989)

[36] A. Drumitrescu, H. Kok, I. Suzuki, P. Zylinski, “Vision-based pursuit-evasion in a grid”. Proceedings of the 11th Scandinavian workshop on Algorithm Theory, pp. 53 – 64. Gothenburg, Sweden (2008)

[37] R. W. Dawes, “Some pursuit-evasion problems on grids”. Information Processing, Letters 43, pp. 241–247 (1992)

[38] K. Tanaka, “An Improved Strategy for a Pursuit-Evasion Problem on Grids”. Manuscript (1996)

[39] S. W. Neufeld, “A pursuit-evasion problem on a grid”. Information Processing Letters 58, pp. 5-9 (1996)

[40] E. David, "Parallel recognition of series-parallel graphs". Information and Computation 98 (1), 41–55, (1992)

[41] T. C. Hu, M T. Shing, "Computation of matrix chain products. Part II". SIAM Journal on Computing (Univ. of California at San Diego: Springer-Verlag) 13 (2): 228-251, (1984)

[42] U. I. Gupta, D. T. Lee, J. Y.-T. Leung, “Efficient algorithms for interval graphs and circular-arc graphs”. Networks, 12, 1982, pp 201-206

[43] S. W. Neufeld, R. Nowakowski, “A game of cops and robbers played on product of graphs”. Discrete Mathematics 186 (1998), 253–268.

[44] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, R. Motwani, “Visibility-based pursuit-evasion in a polygonal environment”. International Jorunal of Computational Geometry and Applications 9(4/5) (1999), 471–493.

[45] V. Isler, S. Kannan, S. Khanna, “Randomized pursuit-evasion with local visibility”. SIAM Journal on Discrete Mathematics 20(1) (2006), 26–41.

[46] M. Aigner, M. Fromme, “A game of cops and robbers”, Discrete Applied Mathematics 8 (1984), 1–12.

[47] T. Andreae, “Note on a pursuit game played on graphs”, Discrete Applied Mathematics 9 (1984), 111– 115.

[48] B. Yang, D. Dyer, and B. Alspach, “Sweeping graphs with large clique number”. Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC 2004), vol. 3341 of Lecture Notes in Comput. Sci., Springer, 2004, pp. 908–920.

[49] B. Yang, R. Zhang, and Y. Cao, “Searching cycle-disjoint graphs”. Proceedings of the 1st International Conference on Combinatorial Optimization and Applications (COCOA 2007), vol. 4616 of Lecture Notes in Computer Sci., Springer, 2007, pp. 32–43.

[50] X. Tan, “Sweeping simple polygons with the minimum number of chain guards”. Inform. Process. Lett., 102 (2007), pp. 66–71.

[51] J. D. H. Smith, “Minimal trees of given search number”. Discrete Math., 66 (1987), pp. 191–202.

[52] N. Robertson and P. D. Seymour, “Graph minors—a survey”. Surveys in combinatorics 1985 (Glasgow, 1985), vol. 103 of London Math. Soc. Lecture Note Ser., Cambridge Univ. Press, Cambridge, 1985, pp. 153–171.

[53] D. Richerby and D. M. Thilikos, “Graph searching in a crime wave”. In Procceeding of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2007), vol. 4769 of Lecture Notes in Comput. Sci., Springer, 2007, pp. 21–32.

[54] N. N. Petrov and I. Ture, “A pursuit problem on a graph”. Vestnik Leningrad. Univ. Mat. Mekh. Astronom., (1990), pp. 12–18.

[55] N. N. Petrov and M. A. Teteryatnikova, “Some problems of the search on graphs with retaliation”. Vestnik St. Petersburg Univ. Math., 37 (2004), pp. 37– 43.

[56] N. N. Petrov and S. A. Starostina, “Minimal graphs with a search number that is less than four”. Vestnik Leningrad. Univ. Mat. Mekh. Astronom., (1989), pp. 105–106.

[57] S.-L. Peng, C.-W. Ho, T.-s. Hsu, M.-T. Ko, and C. Y. Tang, “Edge and node searching problems on trees”. Theoret. Comput. Sci., 240 (2000), pp. 429– 446.

[58] R. Nowakowski and P. Winkler, “Vertex-to-vertex pursuit in a graph”. Discrete Math., 43 (1983), pp. 235–239.

[59] M. Moscarini, R. Petreschi, and J. L. Szwarcfiter, “On node searching and starlike graphs”. In Proceedings of the Twenty-ninth Southeastern International Conference on Combinatorics, Graph Theory and Computing, vol. 131 of Congr. Numer., 1998, pp. 75–84.

[60] F. Mazoit and N. Nisse, “Monotonicity of non-deterministic graph searching” .Proccedings of the 32th International Workshop on Graphs (WG07), vol. 4769 of Lecture Notes in Comput. Sci., Springer, 2007, pp. 33–44.

[61] S. V. Lunegov and N. N. Petrov, “Some graph search problems”. Vestnik St. Petersburg Univ. Math., 35 (2002), pp. 9–12.

[62] F. Luccio, L. Pagli, and N. Santoro, “Network decontamination in presence of local immunity”. International Journal of Foundations of Computer Science, 18 (2007), pp. 457–475.

[63] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, “Introduction to Algorithms (2nd ed.)”. MIT Press and McGraw-Hill (2001).

[64] K. Lodaya, P.Weil, “Series-parallel posets: Algebra, automata and languages”, Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 555–565. Springer, Heidelberg (1998)

[65] J. Neggers, H. S. Kim, “Basic Posets”, World Scientific Publishing Co., New Jersey, 1998.