UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Renato Fernando dos Santos

Parallel Multi-speed Pursuit-Evasion Game Algorithms

Belo Horizonte
2023

Renato Fernando dos Santos

Parallel Multi-speed Pursuit-Evasion Game Algorithms

Versão Final

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador: Marcos Augusto Menezes Vieira

Belo Horizonte
2023

© 2023, Renato Fernando dos Santos.
Todos os direitos reservados

Santos, Renato Fernando dos.

S237p Parallel multi-speed pursuit-evasion game algorithms [recurso eletrônico] / Renato Fernando dos Santos – 2023.
1 recurso online (87 f. il, color.): pdf.

Orientador: Marcos Augusto Menezes Vieira.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 82 -87.

1. Computação – Teses. 2. Jogo de perseguição e fuga – Teses. 3. Sistemas Multi-agentes – Teses. 4. Robôs heterogêneos I Vieira, Marcos Augusto Menezes. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*82(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

PARALLEL MULTI-SPEED PURSUIT-EVASION GAME ALGORITHMS

RENATO FERNANDO DOS SANTOS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Marcos Augusto Menezes Vieira - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Luiz Chaimowicz
Departamento de Ciência da Computação - UFMG

Prof. Douglas Guimarães Macharet
Departamento de Ciência da Computação - UFMG

Prof. Leandro Soriano Marcolino
Universidade de Lancaster

Prof. Denis Fernando Wolf
Instituto de Ciências Matemáticas e de Computação - USP

Belo Horizonte, 02 de junho de 2023.

Documento assinado eletronicamente por **Denis Fernando Wolf**, **Usuário Externo**, em 02/06/2023, às 16:55, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcos Augusto Menezes Vieira**, **Professor do Magistério Superior**, em 03/06/2023, às 00:23, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Douglas Guimaraes Macharet**, **Professor do Magistério Superior**, em 06/06/2023, às 10:06, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Luiz Chaimowicz**, **Professor do Magistério Superior**, em 06/06/2023, às 12:29, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Leandro Soriano Marcolino**, **Usuário Externo**, em 12/07/2023, às 12:47, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2356562** e o código CRC **4CE58528**.

Dedico este trabalho aos meus pais.

Acknowledgments

Ao longo do doutorado, tive a ajuda de muitas pessoas, direta ou indiretamente, às quais gostaria de agradecer. Agradeço primeiramente ao meu orientador, professor Marcos, pelas contribuições indispensáveis para o sucesso deste trabalho e a defesa desta tese, bem como pelo suporte acadêmico e seus conhecimentos. Agradeço também aos coautores dos artigos, Ragesh e Gaurav. Aos colegas do VerLab, laboratório da UFMG, ao qual estive vinculado durante todo esse processo. Agradeço à UFMG e em especial ao DCC pela formação. Aos colegas do Dinter com os quais compartilhamos angústias e aflições. Agradeço à Capes, à UFMS e ao professor Nalvo pela proposição do Dinter e pelo suporte ao longo do processo. Agradeço ao pessoal da secretaria acadêmica, em especial à Sônia, sempre muito solícita. Agradeço à minha instituição, o IFMS, pelo afastamento. Agradeço também a Belo Horizonte, que me acolheu tão bem e marcou a minha vida. Sempre será um dos meus lugares no mundo.

Agradeço aos meus pais por sempre me darem suporte e ajudar nas mais diversas situações. Aos meus cãezinhos, Billy e Marie, por sempre me darem tanto carinho e companhia, especialmente ao Billy, que pode ser considerado coautor da tese e dos artigos. Billy esteve presente nas pequenas vitórias, nos momentos de angústia, no isolamento da pandemia e nas diversas noites em claro. Ele testemunhou tudo, o tempo todo, até o ato final: os ajustes da versão final desta tese. Agradeço ao amor da minha vida, Lorrainy, pelo apoio e compreensão na reta final dessa caminhada.

Meu muito obrigado a todos.

“*Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.*”

(Albert Einstein)

Resumo

Pursuit-Evasion Game (PEG) consiste de um time de perseguidores tentando capturar um ou mais fugitivos. PEG é importante devido à sua aplicação em vigilância, busca e salvamento, robótica de desastres, defesa de fronteiras e assim por diante. Em geral, PEG requer tempo exponencial para calcular o número mínimo de perseguidores para capturar um fugitivo. Para mitigar isso, criamos um algoritmo paralelo ótimo para minimizar o tempo de captura no PEG. Dada uma topologia discreta, esse algoritmo também gera o número mínimo de perseguidores para capturar um fugitivo. Também foi estendido o algoritmo paralelo para outras versões como: heterogênea/jogadores multi-velocidade; a técnica pac-dot para aumentar a longevidade do evader em um jogo, e; uma estratégia de poda para a técnica pac-dot, para aumentar a sua escalabilidade. Além disso, descrevemos um algoritmo para alocação de recursos entre redes de agentes heterogêneos, para disponibilizar todos os recursos para cada agente heterogêneo da equipe, por meio do compartilhamento de recursos na vizinhança do nó. Cada equipe de agentes com todos os recursos disponíveis é associada a um fugitivo no jogo de perseguição e fuga. A abordagem é tolerante a falhas nos casos em que um recurso/agente quebra, sendo capaz de alocar um recurso compatível quando disponível. As principais contribuições desta tese são: Primeiro, foi descrito um algoritmo para alocação de recursos para múltiplas equipes de agentes com todos os recursos disponíveis. Em segundo lugar, aplicamos nossa abordagem de alocação de recursos para substituir um agente em tempo real se ele falhar. Em terceiro lugar, avaliamos nossa técnica aplicando-a ao PEG, desde a composição das equipes até a introdução e correção de falhas ao longo das simulações da instância. O desempenho do algoritmo paralelo foi avaliado pela métrica speedup. O algoritmo e suas extensões foram simulados e avaliados em diversas topologias, para validar a sua viabilidade a partir da discussão e avaliação de um conjunto de resultados. O algoritmo paralelo nos permite escalar até 8,13 vezes com 32 núcleos em comparação com o estado da arte. Considerando a complexidade do espaço de estados, a técnica de poda para o algoritmo pac-dot minimiza o espaço de estados e transições geradas, podendo lidar com um grande número de estados ($\approx 830Mi$) e transições ($\approx 11Bi$). Em geral, nossos algoritmos aumentam a escalabilidade e tornam viável o cálculo da estratégia ótima PEG para casos mais realistas. As simulações para estratégia de alocação de recursos foram realizadas para treze jogadores, simultaneamente. A partir da avaliação das simulações, a abordagem se mostrou eficiente para manter uma trajetória ótima até a captura dos fugitivos nos casos em que ocorre falha. Além disso, a abordagem escala para que muitos jogos ocorrerem simultaneamente.

Palavras-chave: Jogo de Perseguição e Fuga, Algoritmo Paralelo, Sistemas Multi-agentes, Caminho Ótimo, Robôs Heterogêneos, Alocação de Tarefas para Multi-grupos.

Abstract

Pursuit-Evasion Game (PEG) consists of a team of pursuers trying to capture one or more evaders. PEG is important due to its application in surveillance, search and rescue, disaster robotics, boundary defense, and so on. In general, PEG requires exponential time to compute the minimum number of pursuers to capture an evader. To mitigate this, we have designed a parallel optimal algorithm to minimize the capture time in PEG. Given a discrete topology, this algorithm also outputs the minimum number of pursuers to capture an evader. We also extended the parallel algorithm to consider other versions as: heterogeneous/multi-speed players; the pac-dot technique to increase evader lifetime in a game; and a pruning strategy for pac-dot technique to increase the scalability. Additionally, We describe an algorithm for resource allocation between networks of heterogeneous agents to make all resources available to each heterogeneous agent of the team via resource sharing in the node neighborhood. Each team of agents with all resources available is associated with an evader to play pursuit-evasion games. Our approach is fault tolerant in cases where a resource/agent breaks, being able to allocate a compatible resource when available. The main contributions of this thesis are as follows. First, we describe an algorithm for resource allocation for multi-teams of agents with all resources available. Second, we apply our resource allocation approach to replace an agent in real time if it fails. Third, we evaluate our technique by applying it to the PEG, from the composition of the teams to the introduction and fixing of failures throughout the instance simulations. The parallel algorithm performance was evaluated by speedup metric and this algorithm and its extensions were simulated and evaluated on many different topologies to validate the viability of our algorithms by discussing and evaluating a set of simulation results. The parallel algorithm enables us to scale up to 8.13 times with 32 cores compared to the state-of-the-art. Considering the complexity of the state space growing up, the pruning technique to the pac-dot algorithm minimizes the state space and generated transition and can handle a large number of states ($\approx 830\ M$) and transitions ($\approx 11\ G$) generated. In general, our algorithms increase the scalability and make it feasible to compute the PEG optimal strategy for more realistic cases. The simulations for allocation resources strategy were carried out for thirteen players, simultaneously. From the evaluation of the simulations, the approach proved to be efficient to maintain an optimal trajectory until the capture of evaders in cases in which failure occurs. Furthermore, the approach scales for many games being played simultaneously.

Keywords: Pursuit-Evasion Game, Parallel Algorithm, Multi-agent Systems, Optimal Path, Heterogeneous Robots, Multi-Team Resource Allocation.

List of Figures

3.1 A screenshot of the Pac-Man game. The yellow colored pie shaped object is the Pac-Man. The four entities at the center of the maze are the ghosts. 29

3.2 On the right, we can observe the step-by-step game flow, depicting the progression from the player's starting position to capture. On the left, we have the optimal strategy represented in the game graph, which consists of a path from the initial state to the capture state. 35

3.3 In Figures 3.3a and 3.3b, we show the execution of the parallel algorithm after the first iteration. The left table displays the states and their costs, with columns representing the ID and cost for pursuer and evader states. Capture states have a cost of zero, while other states initially have infinity cost. The right table shows transitions, indicating the pursuer and evader state IDs and corresponding transitions. The topology and players are shown at the top, and the minmax tree is depicted at the bottom. In the left table, a red "W" denotes writing flow over pursuer states, and a green "R" indicates reading of evader state costs. The "W" and "R" are inverted when calculating pursuer costs. Downward-pointing arrows in the right table represent the flow of transitions, aligned with the "W" in the left table. Threads (e.g., $P1, P2, P3, P4$) correspond to available cores, and their positions in the right table indicate the current state being processed. 36

3.4 Graph G where pursuer is on node 3 and evader is on node 5. G' is a resulting graph, after removing the pursuer position. 43

3.5 A Power Set for $k = 3$, that represents the sequence of pac-dot that can be reached in a game. Side arrows indicate the Game flow and inverting arrows the Calculation flow. 46

3.6 Flow of the pruning algorithm from the pac-dot position to $t = 3$. 48

3.7 Evaluated topologies. 52

3.8 Serial execution time performance for execution in one core, equivalent to the previous algorithm in [56] and performance of parallel algorithm execution for the number of cores equal powers of 2 up to 32 (Ours). The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). . 53

3.9 Traces execution of the Pac-Man topology. In both simulations, depicted in Figure 3.9a and Figure 3.9b, the evader's path is represented by the yellow trace, while the pursuers' paths are shown in red and green. The accompanying numbers, which match the color of each trace, indicate the game step number and the corresponding evolution steps. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 56
3.10 Pac-Man topology evaluated for multi-speed pursuers. The simulations were performed on Computer 2, which had 12 cores and 64 GB of RAM (refer to subsection 3.7.1). 57
3.11 Evaluation of Building 1 and Building 2 topologies for multi-speed players. Each chart depicts the execution in a specific topology, with the evader speed on the upper X-axis, pursuer speed on the lower X-axis, and capture time on the Y-axis. (a) shows the results for Building 1 topology, with evader speed = 3. Two cases are tested: the first with two pursuers, both with speed 1 and the second with two pursuers, one with speed 1 and the other with speed 2. The execution times were 11 game steps and 7 game steps, respectively. The same pattern applies to (b), (c), and (d). The simulations were performed on Computer 3, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 58

4.1 Resource Allocation algorithm flow: The input consists of a set of agent networks, each with an adjacency matrix (A) and a color distribution matrix (C). In the Classify stage, networks are analyzed and classified as Provider, Sufficient, or Receiver based on available resources and the minimum agent requirement. In the Transfer stage, agents are transferred from Provider to Receiver networks when compatibility exists. In the Add/Remove Edges stage, necessary edge modifications are made to transform networks into CHCNs. The resulting CHCNs are the algorithm's output. 65
4.2 Example of resource allocation and stages of the Algorithm 5. 65
4.3 This example shows network redistribution in G1, a Sufficient network with four agents. Node $1,2$ breaks down, but the network remains a CHCN with $c(T) = 2$ and all resources intact. The redistribution involves adjusting the connectivity by adding and removing edges. The transition from Sn = 4 to Sn = 3 ensures network stability. 70

4.4 Networks G_1 and G_2 start with the status of Sufficient. In Figure 4.4a, agent $1,2$ in G_1 is broken, causing G_1 to change its status to Receiver. Both features are not maintained: resource 1 is no longer available, and $C(T) = 1$. After capturing its evader, G_2 changes its status to Provider. In Figure 4.4b, during the Redistribution of Resources, agent $1,3$ from G_2 is transferred to G_1 to replace the broken agent and provide resource 1, resulting in $c(T) = 2$. The total cost N is the sum of the costs: the time before the agent breaks, the waiting time for the resource, and the time required to capture the evader after restarting. 72
4.5 Building 1. 74
4.6 Building 2. 74
4.7 (Re)distribution of resources by Algorithm 5. 75

List of Tables

2.1 Categorization of PEG Related Work. 22

2.2 Categorization of Related Work. 27

3.1 Speedup for executing the parallel algorithm in relation to the version equivalent to the previous one (core = 1) [56]. *The execution with a single core and a duration of 4304 minutes refers to the parallelizable sections of the algorithm being executed sequentially. The remaining results of this column (Parallel Time (min.)) were executed in parallel. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 54

3.2 Topologies and simulation results - Parallel Algorithm (32 cores) X Serial Execution (1 core - non-parallelized version). The simulations were performed on a Computer 3, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). . 54

3.3 Topologies and simulation results - Parallel Algorithm without pac-dot. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 54

3.4 Topologies and simulation results - Parallel Algorithm with pac-dot. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 59

3.5 Topologies and simulation results - Parallel Algorithm with pac-dot and pruning. The simulations were performed on Computer 3, which had 32 cores and 138 GB of RAM (refer to subsection 3.7.1). 59

4.1 List of resources/sensors. 74

4.2 Topology characteristics. 76

4.3 Fault Tolerance Evaluation for Building 1 Topology. 76

4.4 Fault Tolerance Evaluation for Building 2 Topology. 76

List of Algorithms

1 Parallel algorithm. 37
2 Parallel algorithm functions. 38
3 Multi-speed player transitions 44
4 Pac-dot Algorithm. 47
5 Resource Allocation Algorithm 67

Contents

1 INTRODUCTION **18**

1.1 Motivation . . . 18
1.2 Objectives . . . 19
1.3 Contributions . . . 19
1.4 Thesis Organization . . . 20
1.5 Summary . . . 21

2 LITERATURE REVIEW **22**

2.1 Pursuit-Evasion Game . . . 22
2.2 Resource Allocation for Heterogeneous Multi-agents Teams . . . 25
2.3 Conclusion . . . 27

3 PURSUIT-EVASION GAME FRAMEWORK **29**

3.1 Background . . . 30
3.2 Definitions, taxonomy, and assumptions . . . 31
3.3 Optimal Algorithm . . . 33
3.3.1 Algorithm Strategy . . . 34
3.3.2 Parallel Algorithm . . . 34
3.4 Multi-speed Players . . . 42
3.4.1 Multi-speed Pursuers . . . 42
3.4.2 Multi-speed Evader . . . 42
3.4.3 Multi-speed Player Algorithms . . . 44
3.5 PEG with Pac-dot . . . 45
3.5.1 Problem Modeling . . . 45
3.5.2 Pac-dot Algorithm . . . 46
3.5.3 Practical Applications to the Pac-dot Problem . . . 49
3.6 Pruning . . . 49
3.7 Simulation, Evaluation, and Results for PEG Framework . . . 51
3.7.1 Setup . . . 51
3.7.2 Simulation, Evaluation, and Results . . . 53
3.7.3 Parallel Algorithm Speedup . . . 54
3.7.4 Parallel Algorithm . . . 56
3.7.5 Heterogeneous / Multi-speed Player . . . 57

3.7.6 PEG with Pac-dot . . . 59
3.7.7 Pruning . . . 60
3.8 Conclusion . . . 60

4 RESOURCE ALLOCATION FRAMEWORK 61
4.1 Background . . . 63
4.2 Completely Heterogeneous Network for PEG . . . 64
4.2.1 Resource Allocation . . . 64
4.2.2 Fault Tolerance . . . 70
4.3 Simulation, Evaluation, and Results for Resource Allocation Framework . . . 73
4.3.1 Setup . . . 73
4.3.2 Evaluation and Results . . . 73
4.4 Conclusion . . . 78

5 CONCLUSION 79
5.1 Conclusion . . . 79
5.2 Future Work . . . 80

BIBLIOGRAPHY 82

Chapter 1

Introduction

Pursuit-evasion games (PEGs) and resource allocation for heterogeneous multi-agent systems are two prominent areas of research that have garnered significant attention in recent years. PEGs involve the interaction between pursuers and evaders in a dynamic environment, where the pursuers aim to capture or intercept the evaders while the evaders strive to avoid capture. PEGs have found applications in diverse domains such as surveillance, search and rescue operations, and network deployment, where effective strategies for pursuing and evading entities are crucial.

Heterogeneous multi-agent systems, on the other hand, involve the coordination and collaboration of agents with diverse capabilities, resources, and objectives to accomplish complex tasks. The allocation of resources among these agents plays a vital role in optimizing task performance and ensuring fault tolerance. Resource allocation for heterogeneous multi-agent systems requires efficient mechanisms to distribute resources effectively, maintain task resolution in the presence of failures, and adapt to dynamic environments.

1.1 Motivation

The effective resolution of pursuit-evasion games and resource allocation in heterogeneous multi-agent systems pose several challenges that demand innovative approaches and algorithms. Traditional methods for solving pursuit-evasion games often suffer from scalability limitations, hindering their applicability to large-scale and complex scenarios. Similarly, resource allocation in heterogeneous multi-agent systems requires mechanisms that can handle the dynamic nature of tasks and agents, ensuring efficient utilization of resources and fault tolerance.

The motivation behind this thesis is to address these challenges and contribute to the fields of pursuit-evasion games and resource allocation for heterogeneous multi-agent systems. By developing scalable algorithms for solving pursuit-evasion games and efficient resource allocation techniques for heterogeneous multi-agent teams, we aim to enhance task performance,

optimize resource utilization, and improve the fault tolerance of multi-agent systems.

1.2 Objectives

The primary objective of this thesis is to propose and validate algorithms and techniques that enhance pursuit-evasion game strategies and resource allocation in heterogeneous multi-agent systems. Specifically, we aim to achieve the following objectives:

1. Develop a parallel algorithm for solving pursuit-evasion games that can handle large-scale scenarios and improve computational efficiency.

2. Extend the pursuit-evasion game framework to incorporate multi-speed players and strategies such as the pac-dot strategy, enhancing evader lifetime and optimal strategy computation.

3. Design a resource allocation algorithm for heterogeneous multi-agent teams that enables efficient resource sharing and real-time agent replacement to improve task performance and fault tolerance.

4. Evaluate and validate the proposed algorithms and techniques through extensive simulations and comparisons with existing approaches, demonstrating their effectiveness and scalability.

1.3 Contributions

This thesis makes significant contributions to the fields of pursuit-evasion games and resource allocation for heterogeneous multi-agent teams. The key contributions of this research are as follows:

1. A parallel algorithm for solving pursuit-evasion games: We propose a novel parallel algorithm that improves the scalability and computational efficiency of solving pursuit-evasion games on graph-based environments. The algorithm enables the analysis of games with a large number of states and transitions, providing more comprehensive insights into pursuit and evasion strategies.

2. Extension of pursuit-evasion game strategies: We extend the pursuit-evasion game framework to support multi-speed players and incorporate the pac-dot strategy. These extensions enhance the evader lifetime and enable the computation of optimal strategies that consider

different player speeds and objectives.

3. Resource allocation algorithm for heterogeneous multi-agent teams: We develop a resource allocation algorithm that enables the construction of teams composed of heterogeneous agents and ensures efficient resource sharing. The algorithm incorporates real-time agent replacement to address failures, facilitating seamless task resolution and fault tolerance.

4. Evaluation and validation: Extensive simulations and evaluations are conducted to validate the proposed algorithms and techniques. The performance and effectiveness of the parallel algorithm for pursuit-evasion games and the resource allocation algorithm for heterogeneous multi-agent teams are compared with existing approaches, demonstrating their scalability, efficiency, and practical applicability.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 provides an in-depth literature review of pursuit-evasion games, highlighting their relevance and applications in various domains. It explores existing algorithms and approaches for solving pursuit-evasion games and discusses their limitations.

- Chapter 3 presents the Pursuit-Evasion Game framework and introduces the parallel algorithm developed for solving pursuit-evasion games. It discusses the implementation details, scalability improvements, and simulation results, demonstrating the effectiveness and efficiency of the proposed approach.

- Chapter 4 focuses on resource allocation for heterogeneous multi-agent teams. It presents the resource allocation algorithm, discusses its key features, and evaluates its performance through simulations and comparisons with existing approaches.

- Chapter 5 summarizes the key findings and contributions of this thesis and provides guidelines for future work in pursuit-evasion games and resource allocation for heterogeneous multi-agent systems.

By addressing the challenges associated with pursuit-evasion games and resource allocation in heterogeneous multi-agent systems, this research aims to advance the fields and pave the way for improved strategies, algorithms, and applications in team-based task performance and resource optimization.

1.5 Summary

In this chapter, we have provided an introduction to the research area of pursuit-evasion games and resource allocation for heterogeneous multi-agent systems. We have discussed the motivation behind this thesis, outlined the objectives, and highlighted the contributions of this research. The subsequent chapters will delve into the details of the proposed algorithms, their implementation, and the evaluation results, ultimately leading to advancements in team coordination, task accomplishment, and resource management in multi-agent systems.

Chapter 2

Literature Review

In this chapter, we present the related work for the Pursuit-Evasion Game framework (Chapter 3) and Resource Allocation for Heterogeneous Multi-agents Teams and Fault Tolerance the second framework (Chapter 4).

2.1 Pursuit-Evasion Game

In this section, we present the related work of pursuit-evasion game framework, discussed in Chapter 3.

Table 2.1: Categorization of PEG Related Work.

Criterion	[56]	[29]	[23]	[62]	[32]	[65]	our work
Pursuer to evader ratio	> 1	$= 1$	> 1	> 1	$>= 1$	$= 1$	$>= 1$
Players Configuration	MPME	SPSE	MPSE	MPSE	MPME	SPSE	MPME
Environment	Graph	Graph	Polygon	Polygon	Polygon	Graph	Graph
Environment Type	Discrete	Discrete	Discrete	Continuous	Continuous	Discrete	Discrete
Pursuer Visibility	Full	Local	Full	Full	Full	Local	Full
Evader Visibility	Full	Local	Full	Full	Full	Full	Full
Multi-Speed	Yes	Yes	Not	Not	Yes	Not	Yes
Capture Guaranteed	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Optimal	Yes	Yes	Not	Not	Yes	Yes	Yes
Parallel Algorithm	Not	Not	Not	Not	Not	Not	Yes

Pursuit-Evasion Game (PEG) is a well-studied topic in the robotics literature [2, 5, 7, 37, 39, 46, 56]. The huge interest for PEGs in robotics comes from their application to multi-robot problems such as surveillance, search and rescue, boundary defense, network deployment [53, 57], etc. In addition, PEGs are used to obtain results on the worst-case performance of robotic systems [7]. Common approaches for solving and analyzing PEGs are based on game theory; these approaches can be traced back to the seminal work of Isaacs on differential games [24]. Since then, various versions of PEGs have been introduced in the robotics literature: continuous time PEGs [61], discrete-time PEGs [5], discrete PEGs [40], etc. There are two primary approaches for solving PEGs [7]: differential and combinatorial. Usually, differential games are modeled by differential equations that determine the motion of players in a continuous environment [7, 46, 60]. In general, in the combinatorial approach, the environment

is discrete and modeled by a graph [2, 7, 37]. In this thesis, we focus on discrete PEGs and we base mainly on the formulations presented in [56].

There are a lot of approaches to solving PEGs designed using the different approach as: Voronoi Cells [23, 67], Manhattan Grid [26, 28, 29], Apollonius Circle [13, 30, 32, 65], Cartesian Ovals [13, 38], minmax tree and pruning technique [65, 66], Markov Decision Processes (MDP) [4, 63], Partially Observable MDP (POMDP) [22, 63], Stochastic Games [19, 25], Perimeter Defence [48, 49, 50, 51], Reinforcement Learning [58, 59, 64], and so on.

To situate our work in the literature, we used nine criteria to classify the type of PEG: the ratio of the number of pursuers in relation to the number of evaders; players configuration indicates whether the game deals with one or a set of pursuers, and one or a set of evaders; environment describes whether the environment is modeled as a graph or polygon; type of environment indicates whether the environment is discrete or continuous; whether pursuers and evaders have full (global) or partial (local) vision for other players; multi-speed is a game in which players play at different speeds during the game or with each other; whether the capture of the evader is guaranteed, and; if the game strategy is optimal. The criteria and main related work are shown in Table 2.1.

In [56] was proposed the design and implementation of an exact and optimal algorithm to compute discrete pursuit-evasion games, but it does not scale beyond a small number of robots. This work is the main basis of this thesis, which proposes a parallelization of the exact optimal algorithm to expand its scalability for larger topologies and/or a larger number of robots. In addition, the Multi-speed and pac-dot algorithms were proposed as contributions. In this work, the main contributions are the pruning algorithm to the pac-dot technique, multi-player speed for pursuers and evaders, and speedup of the parallel algorithm proposed in the previous work.

A minimax algorithm to guarantee optimal capture time in the worst case, in a Manhattan grid, with a pursuer and an evader was proposed in [29]. The pursuer has partial vision and the evader has any knowledge of the pursuer's position. In each node, the pursuer is notified if the evader passed through there and the time since it passed. The evader's actions are assumed to be unknown. The speed of the pursuer is twice the speed of the evader. In [26] and [28] extensions of the above work were proposed. In these works the game environment is a directed acyclic graph (DAG) and the evader can reach an exit of the graph without being captured, it is deemed to have escaped. Evader's capture is guaranteed. In our proposal, it is possible to play PEG with only one pursuer as long as it is at least twice the speed of the evader, although its speed may be higher. The vision is global for all players and the evader executes an optimal strategy, which always determines the worst for the capture.

In [23] and [67] a multi-pursuers single-evader (MPSE) method, in a polygon, convex, and bounded environment was proposed. This method is discrete and uses Voronoi cells from the evader's position in relation to the pursuers. The objective is to start the game with pursuers encircling the evader, and then they progressively reduce the encirclement until the capture. The evader capture is guaranteed, but optimality is not. Although our work supports MPME, we only

evaluate it under MPSE. We model the environment as a topological graph and guarantee the optimal time to capture.

A formulation for MPSE in a continuous environment and unstable speeds (varying in an interval) was proposed in [62]. Pursuers are randomly arranged forming an imaginary circle around the evader. The evader calculates the degree between each pair of pursuers and moves along the angular bisector of the maximum angle. If the maximal angle changes, the direction of the evader also changes. Evader is captured if the distance between it and a pursuer is small enough. The formulation does not guarantee capture or optimality. Our proposal is similar to the formulation above in three points: it supports MPSE; pursuers have multi-speeds, and; the capture occurs by approximation of the robots. The main differences are that our proposal works in discrete environments and at stable speeds.

A multi-pursuer multi-evader (MPME) strategy, which uses the Apollonius circle in relation to the evader to define which pursuers are enabled/disabled to pursue, was proposed in [32]. The evaders have speeds equal to or less than pursuers. The speed is equal to one of the three predefined values. The strategy has a task allocation algorithm that uses Voronoi cells to calculate the distribution of the pursuers in the environment and to assign the evaders to the pursuers. The strategy guarantees capture at the optimal time. Our proposal work with multi-evaders but not with task allocation, only with resource allocation. It's similar to providing multi-speeds and optimal time to evader's capture. Capture is guaranteed if the speed of at least one pursuer is greater than the speed of the evader or there is a minimum number [35] of pursuers for a given topology if all players have the same speed.

In [52] was proposed a model based on Apollonius' circle to define the capture region. Pursuers working cooperatively with a hunter to pursue and capture the evader. The evader has a speed higher than the speed of each pursuer and lower than the hunter's speed. Pursuers and the hunter follow an algorithm that predicts the capture points. The evader strategy is based on improved artificial potential fields. Our work has a lot of similarities with this work, such as the use of multi-speed players and strategy to maximize the evader lifetime. In [13], the authors used Cartesian Ovals to delimit the distance of areas reachable by pursuers. The Apollonius circle delimits the area when the distance is zero. This strategy is used for a group of slow pursuers that try to capture a superior evader. In our strategy, we can deploy a superior evader capable of assuming the role of the hunter, as well as another evader with a higher speed than the pursuers and a speed lower than that of the superior evader. In addition, we proposed the pac-dot strategy to maximize the evader lifetime, that scale for more robust topologies if used with pruning technique.

In [48, 49, 50, 51], the authors presented several approaches to address the perimeter defense problem. In general, the perimeter defense problem consists of a convex or circular region called the target region, whose defenders patrol the perimeter to avoid intruders from reaching the target. In the first work [48] the authors proposed a solution to the two player game, a decomposition method for multi-players, and a team strategy for the team of intruders

that gives a lower bound. In [50] was proposed a strategy for perimeter defense where defenders are under parameter uncertainties as sensing constraints, the relation between the team size, and the overall performance. In a third approach [51] was proposed a polynomial-time algorithm to compute the defense strategy. The lower bound provided by the intruder team matches with the upper bound provided by the defender team. The optimality is reached over certain initial conditions such as the number of defenders greater than the number of intruders, and so on. Although the above works are addressed to continuous environment models and the PEG happens in a convex or circular region, our work shares relevant aspects, as it highlights the necessity of having a greater number of pursuers than evaders (assuming all players have the same speed) to ensure an optimal strategy and favorable outcomes.

As a contribution of the work, in [65] a pruning strategy to minimax search tree applying classical alpha-beta pruning was proposed. A node is pruned if it is clearly dominated by another. The strategy finds and prunes redundant nodes before the terminal level is reached. The agent (evader) is aware (global vision) of the initial position of the guard (pursuer with partial vision). The initial positions are generated randomly for both players. The solution is optimal. We propose a pruning strategy (Section 3.6) for the pac-dot algorithm, in which case we have redundant nodes (we refer to it as unreachable). The parallel algorithm generates the optimal strategy from the starting position to the capture for all states. The set of all states represents the universe of all possible combinations of the starting positions of the players. Any state or node in a game minimax tree is able to be the starting position.

2.2 Resource Allocation for Heterogeneous Multi-agents Teams

In this section, we present the related work of Resource Allocation for Heterogeneous Multi-agents Teams Framework, discussed in Chapter 4.

In [1] the work presents a framework that analyzes the assignment of multiple resources to nodes and its effect on the overall network heterogeneity, specifically focusing on resource allocation among nodes and their neighborhoods within a single network. This work [1] serves as the foundation for the Resource Allocation framework, which extends the techniques outlined in Chapter 4.1 and provides further details in Chapter 4. The difference between our proposal for the work [1] is that our focus is to work with several teams simultaneously, expanding the cooperativeness to support fault tolerance, whose exchange of resources between the teams is necessary previous or during the PEG.

In work [31], among the main contributions has proposed an algorithm to identify the

instantaneous position of all the players in MPSE settings. The pursuers and evaders are homogeneous. The algorithm performs a dynamic task allocation of the pursuers that ensures the evader's capture in minimum time for any strategy of the evader. It then extends the algorithm to MPME settings to solve the problem by enforcing a dynamic divider-and-conquer approach. In our proposal, the new algorithm is scalable for any number of pursuers and evaders. The basis of this algorithm is similar to [32] described above, which is a derivative work of this. In addition to the similarities and differences mentioned above, the main difference here is that our proposal is aimed at heterogeneous robots.

The authors of [36] proposed an algorithm to optimally allocate tasks for heterogeneous robots. Each robot chooses the tasks considering the energy consumed when executing them and also the global specifications in task allocation. The algorithm takes into account the suitabilities of certain robots toward certain task allocation, for a team of robots with heterogeneous capabilities. The algorithm was formulated as a quadratic complexity time program and its output provides the robots task priorities and the control inputs required to execute the prioritized task. In our approach, considering we work with several groups of heterogeneous robots, our priority is to make them completely heterogeneous to cooperatively perform the specific task of pursuing and capturing the evader in a PEG.

In [55] based in [15, 16, 17] was proposed an experimental validation of methodology to improve the network connectivity and robustness of multi-robot systems. They introduced a fault-tolerant controller that ensures connectivity maintenance even in the presence of faults. The effect of faults was studied by injecting an experimental setup with two types of failure (communications-related and robotic hardware). Finally, they proposed a methodology to automatically explore and optimize the controller parameters. Real experiments were performed. Although the work above is situated in an area of the control systems, the objective is similar to our proposal in terms of improving the robustness and overcoming certain types of failures. The difference is that our focus on overcoming failures is aimed at keeping the group of robots operational, rather than just the robot that experienced a failure. This is necessary due to the interdependence among the robots within a group.

In [42] was proposed a method and a framework in [43] for resilience in a networked heterogeneous multi-robot team that shares resources with each other. If a specific robot's resource failure occurs, the team will reconfigure the network so that the affected robot can continue its tasks. The difference in our work is that we work with several teams simultaneously and, when it is not feasible for the team to reconfigure itself, exchange resources between them.

In [47] was proposed a framework for multi-robot coordination and task allocation. In this framework, when a sensor fails, only the capabilities or actions that rely on the output of that sensor are affected. Moreover, if another robot is able to provide the same information as the failed sensor, then higher-level actions will still be able to be executed after the proper reconfiguration. In our work, the idea of reconfiguration and utilizing alternative resources or shared resources to maintain functionality through cooperation is a key aspect of CHCNs.

Table 2.2: Categorization of Related Work.

Criteria	[1]	[31]	[36]	[15, 16, 17, 55]	[42, 43]	Our work
Types of Agents	Heterogeneous	Homogeneous	Heterogeneous	Heterogeneous	Heterogeneous	Heterogeneous
Fault Tolerant	Not	Not	Yes	Yes	Yes	Yes
Static/Dynamic	Static	Dynamic	Dynamic	Static	Dynamic	Dynamic
Multi-team	Not	Yes	Not	Not	Not	Yes

In Table 2.2 we categorized the related work by characteristics. In line 1 the types of agents can be heterogeneous (Het.) or homogeneous (Hom.). Next, if the work is fault tolerant or not. In the third line, whether the allocation of resources is static (prior to the start of the task execution) or dynamic (before and during the task execution). Next, if the approach is for multi-teams of agents (yes) or if it is for only one team (no). Our work is for heterogeneous, fault-tolerant agents, with dynamic resource allocation, and multi-teams.

2.3 Conclusion

In this chapter, we reviewed the related work in two key areas: the Pursuit-Evasion Game (PEG) framework and Resource Allocation for Heterogeneous Multi-agent Teams.

Regarding the Pursuit-Evasion Game, various approaches have been explored, including Voronoi Cells, Manhattan Grid, Apollonius Circle, Cartesian Ovals, minmax tree, Markov Decision Processes (MDP), Partially Observable MDP (POMDP), Stochastic Games, Perimeter Defence, and Reinforcement Learning. We classified these approaches based on criteria such as the number of pursuers and evaders, the type of environment, vision capabilities, multi-speed dynamics, capture guarantee, and optimality. We also discussed notable works in this field, such as the parallelization of an exact optimal algorithm, minimax algorithms for guaranteed capture, and strategies using Voronoi cells and Apollonius circles.

In the context of Resource Allocation for Heterogeneous Multi-agent Teams, we examined existing frameworks and studies. Abbas and Egerstedt presented a framework focusing on resource allocation within a single network, which served as the foundation for our proposed framework. We differentiated our work by considering multiple teams simultaneously and emphasizing cooperativeness and fault tolerance. Other works discussed dynamic task allocation, optimizing connectivity and robustness, and resilience in the face of resource failures.

By reviewing the literature, we identified gaps and opportunities for our research. Our work contributes to the PEG framework by addressing scalability, multi-speed dynamics, task allocation, and the inclusion of heterogeneous agents. In the Resource Allocation framework, we extend the techniques to multiple teams and emphasize fault tolerance and resource exchange between teams.

In conclusion, this literature review establishes the foundation for our research and positions our proposed frameworks within the existing body of work. By building upon and extending previous approaches, we aim to make significant contributions to the fields of Pursuit-Evasion Games and Resource Allocation for Heterogeneous Multi-agent Teams.

Chapter 3

Pursuit-Evasion Game Framework

Pursuit-Evasion Game (PEG) consists of a team of pursuers trying to capture one or more evaders. PEG is important due to its application in surveillance, search and rescue, disaster robotics, boundary defense, and so on. In general, PEG requires exponential time to compute the minimum number of pursuers to capture an evader. To mitigate this, we have designed a parallel optimal algorithm to minimize the capture time in PEG. Given a discrete topology, this algorithm also outputs the minimum number of pursuers to capture an evader.

Figure 3.1: A screenshot of the Pac-Man game. The yellow colored pie shaped object is the Pac-Man. The four entities at the center of the maze are the ghosts.

Pac-Man is a popular maze arcade game developed and released in 1980 [54]. Basically, the game is all about controlling a "pie or pizza" shaped object to eat all the dots inside an enclosed maze without being apprehended by the four "ghosts" patrolling the maze. Figure 3.1 shows a screenshot of the Pac-Man game. A classic example of PEG is the popular arcade game, Pac-Man. Pac-Man is an instance of a Pursuit-Evasion Game (PEG) in which an evader (Pac-Man) is pursued by four pursuers (ghosts). Therefore, the evolution of the Pac-Man game can be studied by analyzing the associated PEG problem. Pac-Man belongs to a subclass of PEGs

commonly referred to as the Multi-Pursuer Single-Evader (MPSE) [31] pursuit evasion problem. Due to its characteristics, the Pac-Man game was the main case study to the formulation of the PEG parallel framework.

In this chapter, we focus on discrete PEGs based on the formulations presented in [56, 57] and [40]. In essence, a discrete pursuit evasion game (DPEG) amounts to solving a PEG on a graph. The graph in this setting can be interpreted as a topological map of a domain representing the connectivity of various components in the domain of interest.

Since our framework is anchored on DPEG, we envision the domain of DPEG to be a graph that models any complex bounded environment. The nodes of the underlying DPEG graph represent regions (e.g. rooms in buildings) in the environment under consideration. The edges in the graph describe the links among the various regions represented by nodes. In our formulation, the pursuers attempt to capture the evaders in the graph as a team. A capture happens if and only if at least one pursuer is on the same node as the evader. [37] Similar to [56], our work also aims at computing the minimum number of steps to capture all the evaders. However, in this work, we focus on the parallelization of the optimal strategy to compute the capture time delineated in [56]. This parallelization is important since PEGs are, in general, EXPTIME-complete [18]. From the parallel algorithm onwards, we are able to analyze games with a large number of states and transitions. In addition, our parallelized algorithm is shown to be effective in computing the capture time of the game, played by robots with different speeds, or played by robots when the evader can reach pac-dots and increases its lifetime.

3.1 Background

In this section, we describe a framework to compute PEG from a topological map and a set of agents that play an optimal strategy.

PEG Optimal Strategy

[56] describes an optimal strategy capture time algorithm to minimize the time of capture in a PEG with both pursuers and evader playing the optimal strategy. This algorithm also indicates the minimum number of pursuers necessary to capture evaders in a given topology, denoted by $c(G)$, where $G = \{V, E\}$ is a graph (topological map), V is the set of locations or vertices and E is the set of links or edges between locations. The inputs are a set of agents

(pursuers and an evader) and a topological map. Agents have a global vision, that is, each player knows the current position of the other players before its turn to move. The pursuer's goal is to capture the evader as fast as possible, whereas the evader's goal is to escape as long as possible. From the input of the topological map and the number of agents (players), all states are generated, followed by the transition generation. A state is the position on the topological map at a time of the game. Capture states are those in which the pursuer and evader occupy the same node and their costs are zero. All possible position combinations are generated. A transition is a viable movement from the current state to another. The solution is computed in a bottom-up approach, building a minimax tree [45] from the capture states, based on min and max equations. The algorithm ends when all states have their cost calculated. The algorithm presented in this study generates optimal solutions or paths for pursuit-evasion games from any combination or set of start positions for the agents. These solutions continue until the evader is captured, ensuring the capture is achieved efficiently and effectively. By considering various start positions and combinations of agents, the algorithm provides comprehensive coverage and flexibility in addressing different scenarios and maximizing the chances of successful evader capture.

3.2 Definitions, taxonomy, and assumptions

In this section, we describe the various assumptions, terminologies, and definitions used in the PEG framework. Firstly, we outline the framework used in the work and specify the sensing, communication, and computational capabilities of the robots used in our framework. In addition, we enumerate the resources which aid us in the improved scalability of the optimal strategy computation algorithm presented in [56]. Finally, we detail the terminologies and definitions used to describe our problem and its algorithmic solution.

We consider games that are played on domains composed of a considerable number of regions, such domains include but are not restricted to urban areas, indoor regions of a building, any disaster precinct. We assume that each participant (pursuer or evader) has access in real time to the current position of all other participants, including its own. That is to say, the participants are equipped with global vision, i.e. they have full knowledge of the game. This is the hardest case to capture evaders since evaders know before moving the exact positions of all pursuers on the map at that instant and this gives the opportunity for evaders to play optimally. Our contribution enables us to scale the computation of minmax games with full visibility. Moreover, it is possible to have full visibility in many cases, such as with Wireless Sensor Network [56]

This section initially focuses on the parallelization of the optimal strategy algorithm

[56], to compute the least cost to PEG. In our problem setting, we identify the cost of PEG with the capture time of the evader. We propose an approach that uses parallelism to enable computing the required massive processing of the PEG. Furthermore, we extend our approach to incorporate heterogeneous robots. It is worth noting that the pursuers can have distinct speeds, which contribute towards the reduction in capture time and/or the reduction in the number of pursuers needed to capture an evader. Next, we extend the parallel algorithm to consider the case of pac-dot that increases the lifetime of evaders. Finally, we proposed a pruning technique for the parallel algorithm to the pac-dot.

As stated in the Introduction (Section 1), the domain of a PEG is defined as a discrete space bounded and mapped by a topological graph, which is further discretized into a grid representing the environment. In this grid, each node corresponds to a coarse-grained region or grid cell, while the edges or links connect neighboring regions or interconnected cells within the grid. We consider that all paths (links) are wide enough so that there are no collisions between agents. This discretization allows for a structured representation of the environment, enabling the formulation and analysis of the pursuit-evasion problem in a discrete framework. A topological graph is represented as an adjacency matrix, that is provided as input to the algorithm. An adjacency matrix is a square matrix of order n, where n is the number of regions (or occupied cells in discretized ambient), and each region denoted by matrix line i ($0 \leq i < n$) can be linked or unlinked for each region (of the map) denoted by matrix column j ($0 \leq j < n$). If two regions (i, j) are linked, then the cell value is 1, otherwise is 0.

We use a discrete-event simulation where the state of the system changes after each step of the game. In our context, this discrete sequence of events is a pursuer's turn to move followed by an evader's turn to move, this represents one step of the game or one time unit. At each step of the game, the participant pursuer or evader occupies one region and will move to an adjacent neighboring region. Based on the speed (hops), in a single game step, multi-speed pursuers can move to multiple regions connected according to the underlying topological graph. We are interested in the class of games in which there exist enough pursers to guarantee game termination in topological graphs. The algorithm supports Multi-Pursuers Multi-Evaders (MPME) players. To illustrate the effectiveness of our algorithm, we test it on graphs considerably larger than the ones used in [56]. Initially, we consider that pursuers and evaders move with the same speed, one hop in the topology at each time step. We later consider the case where the pursuer can move multiple hops at each time step, moving faster than the evader. Finally, we consider the case where pursuers and the evader have multi-speed and the evader is a superior evader, with higher speed among all players. Now we lay down the terminologies required for describing our problem.

The game is played on an undirected connected graph $G = (V, L)$, where a node $v \in V$ represent a region and a link between two regions u and v is represented using the edge $\{u, v\} \subset V \times V$. We consider two types of players in PEG: pursuers and evaders, indicated by P and E sets respectively. Let P_i be the position of the ith pursuer on G. Similarly, E_i gives the position

of the ith evader. Now, we define a tuple $a =< P_1, P_2, ..., P_n, E_1, E_2, ..., E_m >$ containing the positions of all the PEG participants. We assume that, during the evolution of the game, the pursuers and evaders make alternate moves, and the move of the pursuers followed by the move of the evaders (or vice versa) costs one game step.

Players move from their current vertex u to an adjacent vertex v. If we use the boolean variable T (turn) to encode whether it's the pursuers turn or evaders turn to move, then the tuple $< a, T >$, can be used to represent a state of the game. The pursuers win the game if they capture the evader. Otherwise, if the evader can avoid indefinitely, then the evader wins the game, i.e. pursuers lose and evader wins. If the evader wins the game then it implies that the number of pursuers required to capture the evader is insufficient for the given graph. The minimum number of pursuers required to terminate a DPEG on G is denoted by $c(G)$ in the literature [3].

From the above formulation, it is clear that a DPEG has at least $|V|^{|P|+|E|}$ states. Now, if we consider the fact that there are two turns (pursuer or evader transitions) associated with each state in the $|V|^{|P|+|E|}$ states, then the state space of the game would contain $2 * |V|^{|P|+|E|}$ states. Thus, the game's states are exponential in the number of players [18]. A sequence of transitions can represent the execution of a game.

We define the game as follows. The input is the number of pursuers, the number of evaders, and a topological graph of the environment. The output is the optimal sequence moves for all configurations of the initial position of the players to the capture of the evader. The selected optimal sequence is the motion commands for robots P and E. The goal is to minimize the maximum (worst-case) capture time of the evaders.

The game terminates when all the evaders are captured. An evader is captured when it resides on a vertex of the graph occupied by one or more pursuers. We refer to this state as the capture state of the evader. Consequently, the game terminates if and only if all the evaders are captured. Thus, a non-terminating game implies that more pursuers are required to capture all the evaders.

3.3 Optimal Algorithm

In this section, we describe our scalable algorithm to compute the optimal strategy for the pursuers and evaders participating in a PEG.

3.3.1 Algorithm Strategy

In this section, we focus only on equipotent players. Multi-speed players will be detailed later. We consider that pursuers and evaders have the same speed, so they move one hop in the topological graph at each time step.

We remind ourselves of our assumption that the positions of all participants are known for both pursuers and evaders at each time step. The optimal strategy for the pursuers and evaders is a zero-sum game [45], where the gains or losses of the pursuers are exactly balanced by the losses or gains of the evaders. The optimal strategy proposed by [56], uses a minimax algorithm [45] which minimizes the maximum possible loss for each player in the game. In a PEG, the pursuer's goal is to capture the evader as fast as possible whereas the evader's goal is to escape from the pursuers as long as possible.

Recall that, the evolution of our PEG occurs on a state space with $2 * |V|^{|P|+|E|}$ states. We construct a game graph/tree with the states as nodes and edges as all possible transitions between the states. It is easy to infer that the resulting graph is a connected and directed one. We refer to this graph as *game graph*. Please, do not confuse the *game graph*, which is a direct graph and represents all the player's moves in the game with the *topological map*, which is an undirected graph and represents the environment where the game is played. The start configuration of the game is the initial positions of pursuers and evaders. The game starts with the pursuers turning to move and in sequence the evaders turn to move, both moves account one step at a time. In turn, there is a directed edge to all possible next states. The cost function $C(s)$ assigns a cost to every state in the state space. For each state, a cost function $C(s)$ is assigned, which represents the minimum distance from state s to a capture state achievable through a pursuer's move, and in an evader's move denotes the maximum distance from state s to a capture state.

For a pursuer, the algorithm determines the policy ρ that selects the neighboring state that has the smallest $C(s)$. On the contrary, the neighboring state with the largest $C(s)$ is chosen as the policy ε for an evader. The game is essentially a traversal on the game graph that ends if the traversal ends in the capture states implying that the pursuers won the game. Whereas, if some states recur constantly during the traversal of the game graph, then the game is considered to be non-terminating, thereby favoring victory for the evaders.

3.3.2 Parallel Algorithm

The complete game graph, represented by the minmax tree in Figure 3.3b, provides the optimal strategy for the pursuer and evader from any starting position to capture. Figure 3.2

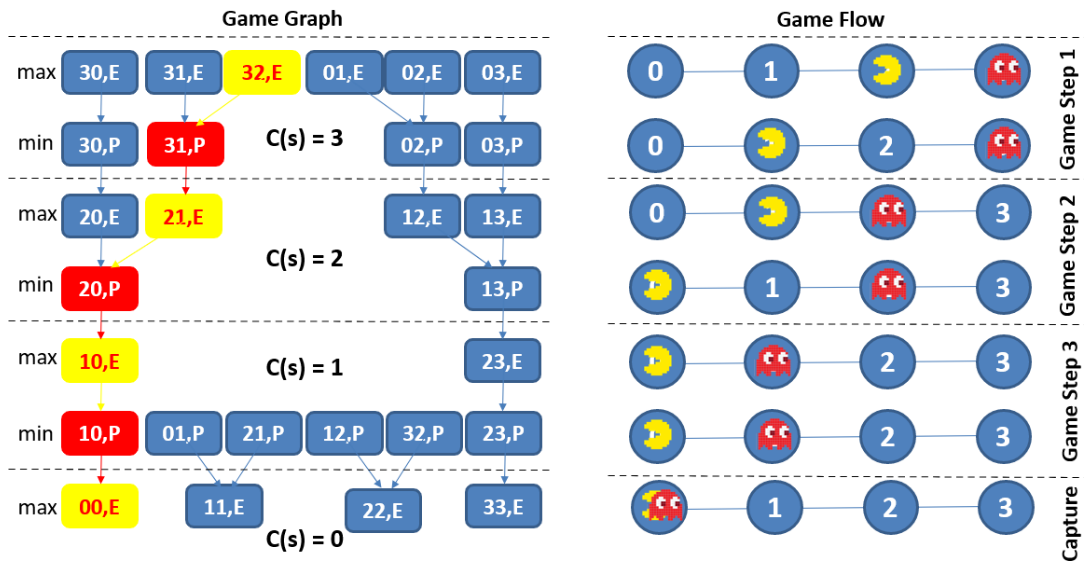
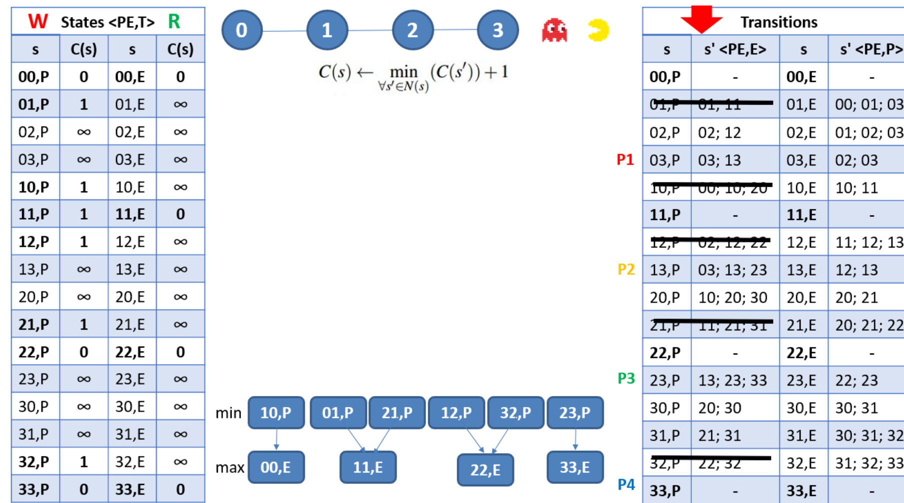


Figure 3.2: On the right, we can observe the step-by-step game flow, depicting the progression from the player's starting position to capture. On the left, we have the optimal strategy represented in the game graph, which consists of a path from the initial state to the capture state.

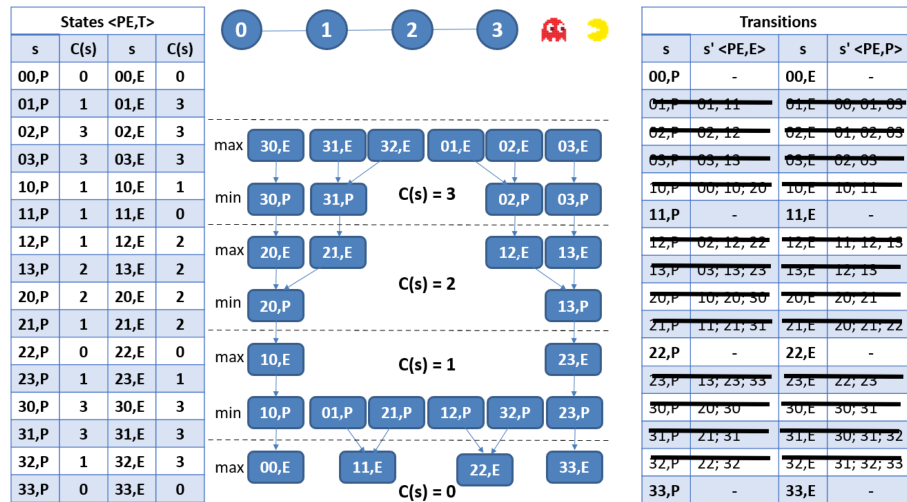
presents a step-by-step gameplay, illustrating the progress of the algorithm.

In Figure 3.2, the flow is depicted in a top-down manner, which is different from the calculation flow shown in Figure 3.3 that follows a bottom-up approach. The optimal strategy starts from the state representing the player's starting position and progresses through each state until reaching the capture state. On the right side of the figure, the game flow is presented from the perspective of the players and the topology. Each type of player (pursuers or evaders) makes a state transition, progressing step by step until the final move leading to the capture.

In Figure 3.3, we present two moments of the parallel algorithm execution, depicting the states, transitions, processors, and the minmax tree. Figure 3.3a and Figure 3.3b contain the same elements. On the left side, there is a table representing the states and their costs. The first and second columns denote the state ID and cost for the pursuer state, while the third and fourth columns represent the ID and cost for the evader state. Capture states start with a cost of zero, while the remaining states initially have a cost of infinity. After calculation, their costs become higher than zero and different from infinity. The table on the right side represents the transitions per state. The first column corresponds to the pursuer state ID, and the second column displays the transitions per pursuer state. The third and fourth columns indicate the evader state ID and the transitions per evader state. In Figure 3.3a, the calculation of the minmax tree, which represents the game graph, starts from the capture states with a cost of zero. These states are considered the first iteration and are labeled as *min*. The states one level up are calculated using the max function and have a cost of one. In the left table, the first line contains the red letter W, which represents the writing flow over the pursuer states. Only pursuer states can be written, while the evader state costs can only be read, represented by the green letter R. When the next level begins to calculate the pursuer cost using the *min* function, the W and R are inverted.



(a) After calculating the first iteration using the min and max functions.



(b) Post costs calculation.

Figure 3.3: In Figures 3.3a and 3.3b, we show the execution of the parallel algorithm after the first iteration. The left table displays the states and their costs, with columns representing the ID and cost for pursuer and evader states. Capture states have a cost of zero, while other states initially have infinity cost. The right table shows transitions, indicating the pursuer and evader state IDs and corresponding transitions. The topology and players are shown at the top, and the minmax tree is depicted at the bottom. In the left table, a red "W" denotes writing flow over pursuer states, and a green "R" indicates reading of evader state costs. The "W" and "R" are inverted when calculating pursuer costs. Downward-pointing arrows in the right table represent the flow of transitions, aligned with the "W" in the left table. Threads (e.g., $P1, P2, P3, P4$) correspond to available cores, and their positions in the right table indicate the current state being processed.

The downward-pointing arrows in the right table represent the current flow of transitions. They always align with the W in the left table. $P1, P2, P3$, and $P4$ represent the threads (the number of threads is equivalent to the number of available cores), and their line positions in the right table indicate the current state being processed. At the same line as $P1$ in the right table, the value $03, p$ represents the state ID, and $03; 13$ represents its respective transitions. The state

Algorithm 1 Parallel algorithm.

```
 1: Generate all States
 2: for all state s do "in parallel"
 3:     INITIALIZATION(s)
 4: for all no capture state s do "in parallel"
 5:     GENTRANSITIONS(s)
 6: i ← 0
 7: repeat
 8:     i ← i + 1
 9:     change ← false
10:     U_p ← set of all unmarked pursuer states
11:     for all s in U_p do "in parallel"
12:         change ← change or PURSUERCALCCOST(s, i)
13:     U_e ← set of all unmarked evader states
14:     for all s in U_e do "in parallel"
15:         change ← change or EVADERCALCCOST(s, i)
16: until not change
```

$03, p$ was assigned to $P1$ for processing, while the states $13, p$, $23, p$, and $33, p$ were assigned to $P2$, $P3$, and $P4$ respectively. The processing order in the respective type of state (pursuer or evader) does not matter. The strikethrough state IDs and transitions in the right table represent transitions of a state whose cost has been calculated, and these states will not be visited again. In Figure 3.3b, we can see the completely calculated minmax tree, and the costs by level are shown. Note that all states have their costs calculated (left table - all non-zero costs), and all transitions have been used and are strikethrough.

The parallel algorithm is the parallelization of the optimal strategy algorithm (Subsection 3.3.1). The objective of the parallelization is to make optimal strategy algorithm execution time faster and large topological graphs tractable. As shown in Algorithm 1, the optimal strategy algorithm has been parallelized in four key areas.

The intuition behind the parallel Algorithm consists of a similarity between the four functions (Lines 3, 5, 12 and 15) in one aspect, that is lock-free (lockless) programming features. Lock-free programming is a technique for multi-threaded programs without using locks or mutex [20, 34]. A multi-thread implementation of our algorithm is lock-free because it satisfies three properties [20, 34]: it is multi-thread; threads access shared memory, and; threads can't block each other. A lock-free data structure can preserve a coherent internal state while being used concurrently by multiple threads without serializing it through locks [33]. If there is some way to schedule the threads which would lock up indefinitely like deadlock, livelock, starvation, etc, then it is not lock-free programming.

In Figure 3.3, it is possible to observe the lockfree structure of the parallel algorithm. The iterations (see Algorithm 1, line Algorithm 11) occur over pursuer states calculated by the *min* function (see Algorithm 2, line Algorithm 10), followed by iterations over evader states

Algorithm 2 Parallel algorithm functions.

```
1: function INITIALIZATION(s)
2:     if s is a capture state then
3:         add s to F_0
4:         C(s) ← 0 {cost function}
5:     else
6:         C(s) ← ∞
7: function PURSUERCALCCOST(s, i)
8:     if s has at least one transition to a marked state then
9:         add s to F_i mark s
10:        C(s) ← min_{∀s'∈N(s)} (C(s')) + 1
11:        add transition to ρ
12:        return true
13:    return false
14: function EVADERCALCCOST(s, i)
15:    if all transition from s reach a marked state then
16:        add s to F_i mark s
17:        C(s) ← max_{∀s'∈N(s)} (C(s'))
18:        add transition to ε
19:        return true
20:    return false
```

(see Algorithm 1, line Algorithm 14) calculated by the *max* function (see Algorithm 2, line Algorithm 17). Each state is processed only once per iteration until its calculation is done (attending the requirements of *min*/*max* functions), which only happens once. Writes and reads are performed alternately, as well as the iterations for cost calculation. There is no blocking in this process.

Initially, Algorithm 1 generates all possible states that the pursuers and evaders position configurations can assume throughout the game. Next step is to assign an initial cost to each state and classify them into two sets: one containing the capture states, and the other containing non capture states. Initial cost of states is zero for capture states and infinite otherwise. After, during game graph traversing, the costs of the states whose costs were initially infinite are updated to be the cost (distance) of each state to a capture state. There are two possible results that one can obtain once the loops terminate: all states were processed, and the output is the shortest possible cost to capture or there is at least one state that remains with cost equal to ∞. The latter means that the number of pursuers is insufficient for evaders' capture.

All states that are marked in the ith iteration are added to the set F_i. This means that F_0 is the set of all capture states, F_1 is the set of all states that can reach a capture state in a single pursuer move. Therefore inductively, F_n denotes the set of states that can transition to a capture state in n pursuer moves.

In Algorithm 1, Line 2 a thread/process is dispatched for each state that calls the initial-

ization function (next line). In the initialization function (Lines 1-6) the initial value is written in the memory position of the state. Each state is passed as a function argument, then it is written once. A similar issue occurs in Line 4 on Algorithm 1, which dispatches a thread/process for each no capture state. The getTransitions function is called and the state id and initialization value are read to generate all possible transitions for each state. Transitions are indexed respectively by their origin state id. For each no capture state, the transitions are generated once. In both cases mentioned above, threads can run concurrently, accessing shared memory, without mutexes or locks. For cost calculation, the Algorithm 1 iterates over the lists of states (Lines 7-16). In each iteration, the list of all unmarked (those states that still have the initialization value) pursuer states is processed, followed by the processing list of unmarked evader states. When the pursuer state list is processed, each pursuer state is a candidate for writing. The transitions of this state are pointers to evader states, evaluated by the min utility function (Line 10 on Algorithm 2). Evader states cost calculation occurs in the opposite way considering the max utility function (Line 17 on Algorithm 2). In Line 11 is dispatched a thread/process for each unmarked pursuer state that calls pursuerCalcCost function in the next line. Each unmarked state is called once by (repeat) iteration while its cost is not calculated and assigned to it when the min function is satisfied. In the first part of the iteration, at first for (Line 11) all pursuers states are candidates for writing, and all evader states are candidates for reading. In other words, there isn't more than one function call passing a pursuer state as an argument at the same iteration. All evader states are only read. Thus, we don't use mutexes and locks for write or read operations. In the second part of the iteration, evader states are computed (Line 14) similar to the pursuer states. Our goal is not to develop a parallel algorithm with optimal time complexity but rather to preserve the optimality of the output produced by the base algorithm. Furthermore, the parallel algorithm we propose is designed to be lock-free.

In Line 1 of Algorithm 1 all states of pursuers and all states of evaders are generated. In Lines 2-3 the initialization function is called to assign the initial cost of each state. The transitions from each state to an adjacent state are computed by genTransition function call (Lines 4-5). Next step, variable i is initialized with zero and represents the iteration number of the repeat structure. In the sequence, there is a repeat structure (lines 7-16) that runs a loop until there is no new cost of the states to be updated (the negation of *change* boolean variable becomes *true*). Within the repeat structure, *change* is assigned with $false$. In Line 10, all unmarked pursuer states are selected and added to collection U_p. At the next line, a loop (Lines 11-12) iterates over elements of U_p. Each pursuer state s and the variable i are passed as an argument to pursuerCalcCost function, which returns $true$ if state cost has been updated or $false$ otherwise. Likewise, in Lines 13-15 the same reasoning is used for evader states.

In Algorithm 1, there are the called functions defined in Algorithm 2. GenTrantitions function was omitted here. The Initialization function (1-6) checks if the argument s is a capture state, initializing it with zero if $true$ or ∞ otherwise. Note that in Line 3 all capture states are added to set F_0, which are the first updated cost states and the first marked states. At Lines 7-13,

is defined as the pursuerCalcCost function, with s parameter. On the next line, it is checked if state s has at least one adjacent state that has had its cost updated, If this condition is true, then the cost of state s is calculated and updated in the subsequent lines. In the sequence, the state s is added to the set F_i and marked. In Line 7, s' denotes an element of set $N(s)$, that denotes all adjacent states of the state s, then for all adjacent states its cost is checked by $C(s')$ function, min function selects the adjacent state with the lowest cost that is incremented by one and assigned to $C(s)$, updating s cost. In the sequence, the transition from s to s' (selected by min function) is added to the ρ policy. $true$ is returned. If the conditional structure is not satisfied, then $false$ is returned at Line 20. The evaderCalcCost function (Line 14-20) by the if structure, checks if all adjacent states of s have its costs updated, returning $false$ on Line 20 or if $true$, performs the calculation and updates the cost of s in sequence. At Line 16 the state is added to set F_i and marked. At Line 17, the max function selects the maximum cost between the cost of all adjacent states of s and updates its cost. In sequence, the transition is added to the ε policy, and $true$ is returned.

Thus, we can compute the cost $C(s)$ for each s state at time i in parallel. If at least one pursuerCalcCost or evaderCalcCost function called return $true$, repeat structure will be running at least one more time, as described in the algorithm.

To establish the optimality of the parallel algorithm's output and demonstrate the consistency between the original and parallel versions of the algorithms, it suffices to prove the following theorems:

Theorem 1. For any given topological graph G, and a state s, if $o(s)$ is the minimal number of steps needed to reach a capture state, then $C(s) = o(s)$.

Proof. We will prove the theorem by induction on i, that is the length of the optimal path in the number of steps.

Basis: if $i = 0$ then $o(s) = 0$ and the game terminates, once one or more pursuers occupy the same vertex of the evader and no movement is needed. This is the definition of capture state (defined in Lines 1-6, Algorithm 2) for which $C(s) = 0$.

Inductive Step: Assuming that for all marked states s, it was marked on i iteration and $o(s) = i$ and $C(s) = i$. We will demonstrate that for any s', $o(s') = i+1$ and $C(s') = i+1$ on $i+1$ iteration.

We divided it into two cases, when is a pursuer's move and when is an evader's move.

Case 1. Consider a pursuer move, from s', an unmarked state that is checked by the condition in Line 8 (Algorithm 2). If the condition is $false$, then all adjacent s' are still unmarked. If the condition is $true$, then one or more adjacent s', denoted by s'' was marked on i iteration, that is, on the last iteration. In this case, as defined in Line 10 (Algorithm 2), the minimal cost of all s'' is selected and incremented by one. Therefore, $o(s') = i+1$ and $C(s') = i+1$, by inductive hypothesis.

Case 2. Consider an evader move, from s', an unmarked state that is checked by a condition in Line 15 (Algorithm 2). If the condition is $false$, at least one adjacent to s' remains unmarked. If the condition is $true$, then one or more s'' was marked on $i+1$ iteration. In an evader move, all adjacent s'' are pursuer states, and the last s'' that had its cost calculated were calculated in the first part of the iteration $i+1$ by parallel for in Lines 11-12 (Algorithm 1), before calculating the cost of evaders. As defined in Line 17 (Algorithm 2), the maximum cost of all s'' is selected. Considering that i is incremented by one for each iteration, the last adjacent marked has a cost equal $i+1$. Therefore, $o(s') = i+1$ and $C(s') = i+1$, by inductive hypothesis. □

The optimality of the algorithm is preserved by the min function, ensuring the minimum cost, and the max function, guaranteeing the maximum cost (specifically, Lines 10 and 17), as demonstrated in Theorem 1. Additionally, the lock-free nature of the parallel algorithm enables concurrent execution of the cost calculation functions without data dependency. Theorem 2 will establish the preservation of optimality in the parallel algorithm version.

Theorem 2. The output of the parallel algorithm is identical to the output of the optimum strategy.

Proof. The parallel algorithm is designed as a multi-threaded system without the use of locks, allowing shared memory space among states. It preserves the original structure, which calculates state costs using min and max functions.

In the algorithm, pursuer state costs are calculated in parallel using the min function, and evader state costs are calculated in parallel using the max function. Each state is calculated by only one thread, ensuring that there are no concurrency issues.

The parallel algorithm, when executed serially (without parallelism), follows the same steps as the optimal strategy algorithm proposed in [56]. Consequently, the outputs of both algorithms are identical.

Suppose α represents the output of the parallel algorithm and β represents the output of the serial execution or the optimum strategy. We aim to prove that $\alpha = \beta$.

If $\alpha \neq \beta$, it would imply that there is a discrepancy between the outputs. However, considering that calculating the cost of a pursuer state requires reading all the evader states, and vice versa, both algorithms ensure that all states are calculated and satisfy the necessary requirements.

Hence, assuming $\alpha \neq \beta$ leads to a contradiction.

Therefore, $\alpha = \beta$, demonstrates that the parallel algorithm maintains the same output as the optimum strategy. This completes the proof. □

The space complexity is $O(|V|^{|P|+|E|})$, once we need to store all states. In the worst case, we might need to visit all other states at time complexity $O(|V|^{2(|P|+|E|)})$. The parallel algorithm complexity is $O(|V|^{2(|P|+|E|)}/N)$, where N denotes the number of processors.

3.4 Multi-speed Players

In this section, we show the approach for heterogeneous players. Heterogeneity is defined by the speed of each player, then players with different speeds are heterogeneous. Speed is defined as the number of hops (nodes) that a player can take in a game step.

More concretely, let a pursuer $P_i \in P$ that occupies a vertex $v \in G$ and if it can move with a maximum speed p_i, then when it turns comes, P_i can reach any node in the topological graph there if there is a path of length p_i between the node and v. Similarly, let an evader E_i that occupies a vertex v and can move with a speed e_i, E_i can reach any node from v if there is a path of length e_i. A path is generated by avoiding crossing or reaching a vertex occupied by a pursuer.

3.4.1 Multi-speed Pursuers

In a set of pursuers, the highest speed among all pursuers is selected. Next, all paths from each node of the topological graph with lengths less than or equal to the highest speed are calculated by a graph search (Breadth-First Search or Depth-First Search). The result is used to generate the transitions for the pursuers, considering their respective speeds (≥ 1). The topological graph is modeled as an adjacency matrix. A graph search is enough to compute all transitions for all pursuer states.

3.4.2 Multi-speed Evader

A multi-speed evader move must never occupy or cross a vertex occupied by a pursuer. The parallel algorithm (Algorithm 1 and Algorithm 2) can handle this constraint if the evader speed is $= 1$. The max function selects the maximum cost iff all adjacent states are calculated (marked)(see Algorithm 2, lines 15 and 17). It means that transitions to all reachable states are generated without constraints and states that evader occupies the same vertex as a pursuer are unreachable for the evader. The speed of the evader defines the length of the largest path. In Graph G (on the left side) of the Figure 3.4, the evader is positioned at node 5. Suppose the evader speed $= 2$, then evader could stay at the same node or move to nodes $1, 2$ or 3. There are paths of length 2 from node 5 to nodes 1 and 3. In this case, transitions to states $< 3, 1 >$,

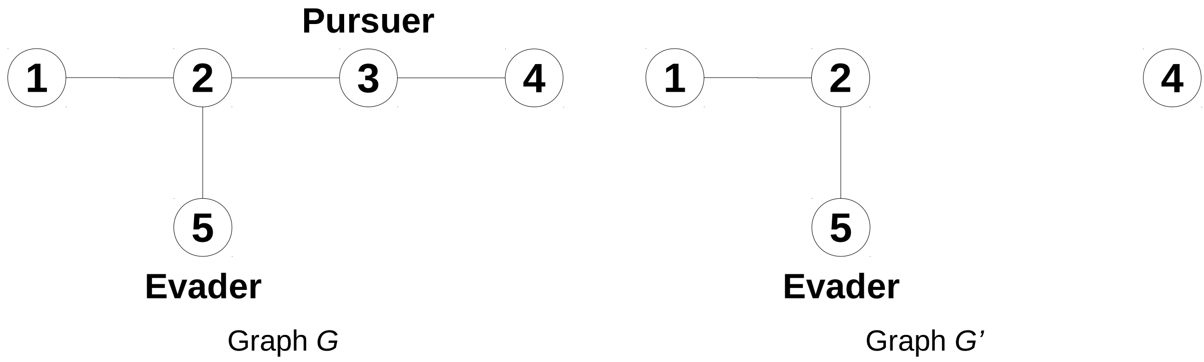


Figure 3.4: Graph G where pursuer is on node 3 and evader is on node 5. G' is a resulting graph, after removing the pursuer position.

$<3,2>$ and $<3,5>$ would be generated. The state $<3,3>$ is of capture, then transition would be not generated to it, once node 3 is ignored in this case, as illustrated in the right side of the Figure 3.4.

For multi-speed evaders, based on the reasoning presented above, we began trying to make several calls of max function, for each speed unit (e.g. if speed $= 3$ then max is sequentially called 3 times). Consider an evader with speed > 1, on the first call of max function, two cases can happen: 1) there is at least one reachable pursuer state that is unmarked, so the cost of the evader state is not calculated; 2) If all reachable pursuer states are marked, then the cost of the evader state is calculated. Next, we would call the max function again, but considering that all unmarked evader states were evaluated on the first call function, and nothing changed in the pursuer states (reachable for evader states), we can conclude that this approach does not work.

Another approach is to generate all transitions to reachable states, similar to generating transitions for pursuer states (Subsection 3.4.1). In contrast, there are some constraints for the movements of the evader in the topological map that reflects on the generation of transitions for the evader states. In Figure 3.4, consider the graph G as the topological map and assume that the evader speed $= 3$. The evader could move up to 3 nodes along the path. Node 3 is the position of the pursuer (as explained above), therefore the evader cannot move to the same node as the pursuer, as it would be captured. This defines the first constraint case. The evader cannot move to node 4 without crossing the pursuer on node 3. This defines the second constraint case. We need to take into consideration the two constraint cases and the fact that player positions are unique for each state, in order to generate transitions for evader states. For each evader state, we need to create a graph G', that is a subgraph of G after removing the constrained nodes. The graph G' (on the right side of the Figure 3.4), describes an unconstrained version of G, without nodes 3 and 4. The reachable nodes are $1,2$ and 5, and the reachable states are $<3,1>$, $<3,2>$ and $<3,5>$.

Algorithm 3 Multi-speed player transitions

```
 1: D_e ← evader speed
 2: D_p ← max(pursuer speed)
 3: DEPTH-FIRST SEARCH(G, D_p)
 4: for all no capture state s do "in parallel"
 5:     if s is an evader move then
 6:         if D_e > 1 then
 7:             W ← set of nodes where pursuers are located in s
 8:             G' ← G − W
 9:             DEPTH-FIRST SEARCH(G', s, D_e)
10:             generate transitions from s based on paths of G'
11:         else
12:             generate evader transitions from s based on paths of G
13:     else{s is a pursuer move}
14:         generate pursuer transitions from s based on paths of G
15:     add generated transitions from s to each reachable state
```

3.4.3 Multi-speed Player Algorithms

Algorithm 3 describes a step needed to generate transitions for multi-speed pursuers and/or evaders. Algorithm 1 can compute multi-speed players replacing lines 4 and 5 with Algorithm 3. In line 1 the evader speed is assigned to D_e and D_p is initialized by the fastest speed among all pursuers on line 2. A graph search is called in line 3 where the topological map G and D_p are the arguments. A graph search is enough to compute all transitions for all pursuer states. This function generates all paths from each node to nodes up to distance D_p. Next (on line 4), the for loop iterates through the no capture states list. In line 5, we check if state s is an evader move, and if it is, then we check in line 6 if the evader is multi-speed. Then the nodes in G occupied by the pursuers are assigned to W (line 8). In the next line, the set of nodes W are removed from G, resulting in an unconstrained subgraph G'. Next, a depth-first search is performed on G' from the evader position in state s up to distance D_e (line 9). After checking on line 6, if the evader is not a multi-speed player, then transitions are generated identically for the pursuers on line 12. If state s is not an evader move (in line 5), then states are generated based on paths calculated from G (in line 10). Whether s is an evader move or a pursuers move, all generated transitions are added in line 15.

Consider just one evader the total states is $|V|^{|P|+1}$. The time complexity for a graph search (BFS or DFS) is $O(|V|+|L|)$. In the worse case, we might need $|V|$ transitions for each evader state and $|V|^{|P|}$ transitions for pursuer states. The overall complexity after adding Algorithm 3 to Algorithm 1 in the worst case is still $O(|V|^{2*(|P|+1)}/N)$.

3.5 PEG with Pac-dot

Pac-dot is an item (pill) present in the Pac-Man game that when taken by the evader, allows the evader to be immune to being captured for an immunity time of t time units. During immunity time, pursuers and evader switch roles, so the evader can pursue and capture one or more pursuers. Capture by the evader results in a penalty for the captured pursuer. In this case, the captured pursuer is moved to a previously specified location. After the immunity time, pursuers come back to the game and two cases can happen: if the game still has one or more pac-dots, evaders can reach another pac-dot, and the immunity time restarts as explained above. If a pac-dot is not reached by the evader or there are no more pac-dots, pursuers pursue the evader until they capture it, and the game terminates. The locations of the pac-dots in the topological map have been known before the game starts.

3.5.1 Problem Modeling

Let k be the number of pac-dot in a game. The number and structure of sub-games generated are modeled as a Power Set [9]. We generate 2^k sub-games as shown in Figure 3.5 for $k = 3$. Each rectangle of the power set is a sub-game. The sub-game numbered 000 is the initial sub-game and the others (above the dashed line) are non-initial sub-games. The top-down arrow, labeled Calculation Flow, indicates the state's cost computation flow. Directed arrows, in the power set, indicate the following flow to obtain the optimal path after state costs computation. For each pac-dot is assigned a boolean bit, to identify the taken flow. If all bits are equal to zero, this represents the initial game sub-game. If a pac-dot is reached by the evader, then the matching bit is inverted to one and follows the arrow to the new corresponding sub-game, and so on, until the game terminates. On the left, the rectangle labeled Sub-Game (110) represents the sub-game in a game with $t = 3$. Each internal rectangle represents a frame, which is indexed by t'. Frames within the range $0 \leq t' < t$ cover the duration of the immunity time.

In Figure 3.5, we show the two types of sub-games. The initial sub-game is the sub-game where the game starts, it has only a frame and it does not have immunity time. A frame is composed of a set of states that corresponds to $2 * |V|^{|P|+|E|}$. The game can start and terminate in the initial sub-game. Non-initial sub-games are all subsequent sub-games that have t frames, where $t-1$ ($t' = [0, t-1]$) frames are relative to immunity time duration, and the last frame t is the frame after the immunity time. In the last sub-game frame, the game continues until it terminates or a new pac-dot is reached, and the sub-games flow follows ahead. For all other non-initial sub-games the behavior is similar. A non-initial sub-game has $t * 2 * |V|^{|P|+|E|}$ states.

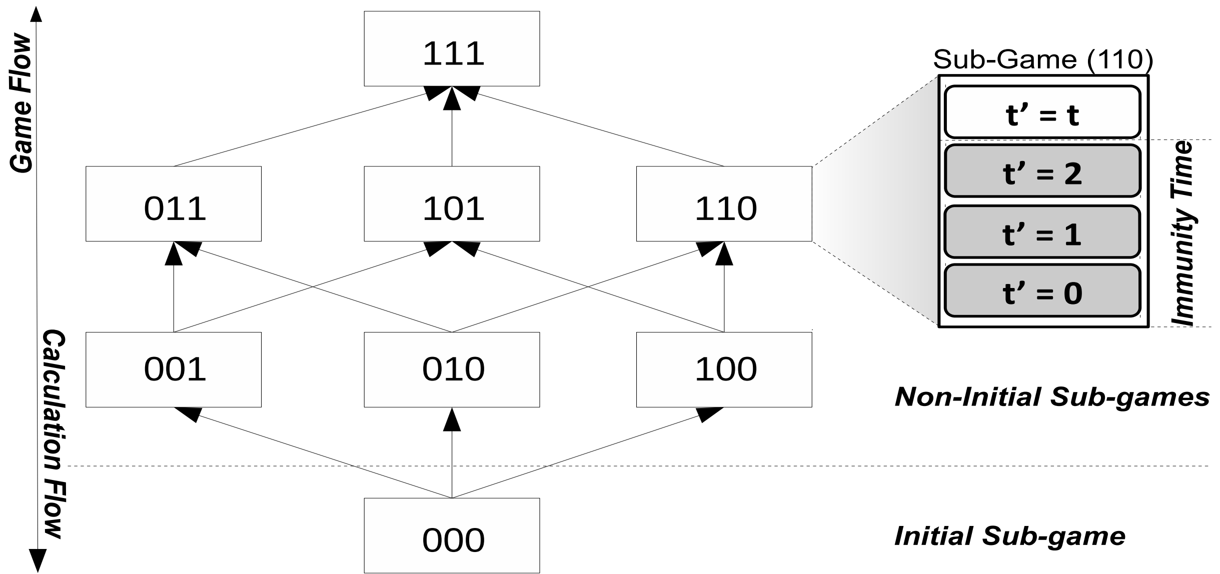


Figure 3.5: A Power Set for $k = 3$, that represents the sequence of pac-dot that can be reached in a game. Side arrows indicate the Game flow and inverting arrows the Calculation flow.

A game with k pac-dots consists of $[(2^k - 1) * t + 1] * |V|^{|P|+|E|}$ states, effectively doubling the number of state spaces. Let $s =< a', a, T >$ be the new state, where $a' =< b_1, .., b_k, t >$. Each b_i represents a bit that encodes the sub-games, allowing for indexing the state to its respective sub-game.

When an evader reaches a pac-dot position, the immunity time starts which in the game computation means starting the frame transitions. All possible movements are to states in the frame $t' = 0$, in the subsequent sub-game. In the context of states, the pac-dot position is those evaders occupy a pac-dot position and pursuer positions can be any of all possible combinations. Recall that each game step is a pursuers move followed by an evader move, during the immunity time pursuers move from the current frame (t') to the subsequent frame ($t' + 1$), and the evader moves in the same frame ($t' + 1$) that the pursuers reached in the last move, and so on until the immunity time expires.

3.5.2 Pac-dot Algorithm

This algorithm follows the same definitions and terminologies engaged for the parallel algorithm, except that all robots are homogeneous and they have speeds equal to one.

To be able to take into account the pac-dot during the game, we made three modifications between the parallel algorithm and the pac-dot Algorithm (Algorithm 3):

1. the initialization values are 0, α and ∞. States of the initial sub-game or states of the

Algorithm 4 Pac-dot Algorithm.

```
1:  i ← 0
2:  repeat
3:      i ← i+1
4:      change ← false
5:      for all k in range [2^k − 1..0] do
6:          for all t′ in range [t..0] do
7:              if k = 0 then
8:                  t′ ← 0
9:              U_p ← set of all unmarked pursuer states
10:             for all s in U_p do "in parallel"
11:                 change ← change OR PURSUERSCOST(s, i)
12:             U_e ← set of all unmarked evader states
13:             for all s in U_e do "in parallel"
14:                 change ← change OR EVADERCALCCOST(s, i)
15: until not change
```

last frame of any non-initial sub-game are initialized with zero if it is a capture state, ∞ if a state whose evader occupies a pac-dot position, and α remaining states (case 1). Otherwise, capture states are initialized with ∞, and remaining states s are initialized with α (case 2). The two cases of initialization can be seen in Figure 3.5.

2. transitions: consider the two cases mentioned above, in the first case the transition occurs locally (inside the same frame). However, if the evader's position corresponds to a pac-dot in the pursuer states, the transition will be made to the states in the next sub-game, based on the specific pac-dot reached. This can be better understood by referring to Figure 3.5, which provides a visual representation of the game flow perspective. In the second case, all pursuer state transitions are from t' frame to $t'+1$ frame, and the evaders state transition be in t frame.

3. The *repeat* structure in Algorithm 1 was redefined with the inclusion of two for structures at lines 10-15. The most external for in non crescent order iterates over the sub-games. The most internal for also in non crescent order, iterates over time t.

For a game with k pac-dot and t immunity time, we have $S = 2^k - 1$ sub-games and a total frame of $F = S * (t+1) + 1$. For a given graph with $|V|$ vertices, the space complexity is $O(F * |V|^{|P|+|E|})$, since we need to store all states. For each state, we might need to visit all other states of its frame or underlying frame, its potential adjacent states. The time complexity is $O(F * |V|^{2(|P|+|E|)}/N)$.

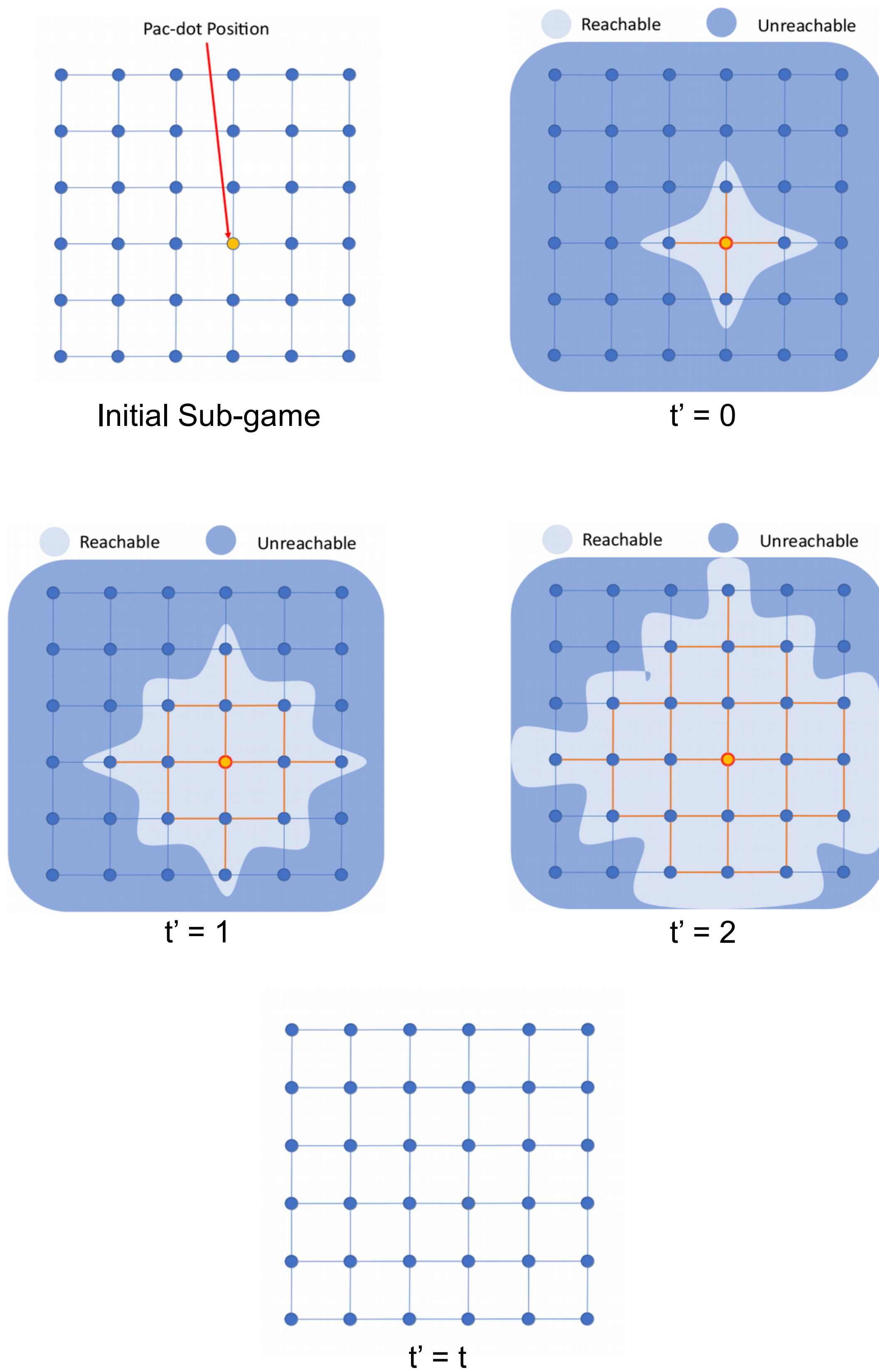


Figure 3.6: Flow of the pruning algorithm from the pac-dot position to $t = 3$.

3.5.3 Practical Applications to the Pac-dot Problem

The Pac-Dot problem and algorithm can be applied in the domain of robotics and multi-agent systems, particularly in scenarios involving coordinated movement and task allocation among multiple robots or agents. Here is a practical application:

Robotic Task Allocation and Cooperation: In a multi-robot system where robots are assigned specific tasks or areas to explore, the Pac-Dot algorithm can be used to optimize task allocation and cooperation among the robots. Each task or area can be represented as a Pac-dot, and the immunity time can represent the duration for which a robot is assigned to a specific task.

The algorithm can be utilized to dynamically allocate tasks to robots based on their proximity to Pac-dots and their current workload. When a robot reaches a Pac-dot, it becomes temporarily immune to task reassignment, allowing it to complete its current task or explore its assigned area. Meanwhile, other robots can pursue new tasks or areas that have not been assigned yet.

The algorithm's transitions and state representations can be adapted to incorporate factors such as task priorities, robot capabilities, communication, and resource constraints. This enables efficient task allocation, cooperative behavior, and adaptability in dynamic environments.

By applying the Pac-Dot algorithm in robotics and multi-agent systems, teams of robots can effectively coordinate their actions, allocate tasks, and adapt to changes in the environment. This can be particularly useful in scenarios such as search and rescue operations, collaborative mapping and exploration, or coordinated surveillance tasks.

3.6 Pruning

In this section, we present a technique of pruning to the pac-dot strategy (Section 3.5). The technique of pruning classifies the states as reachable (relevant) or unreachable (irrelevant). For all sub-games, reachable states are generated to calculate the game with pac-dot. Unreachable states are not considered in the state generation stage of the algorithm. Pruning unreachable states reduces significantly the number of generated states and transitions, which reduces the memory footprint and CPU time to compute the game. Pruning is applied only to immunity time frames ($t' = [0,t)$) in non-initial sub-games (see Figure 3.5).

Frames on which pruning is not applied can have pac-dots that can be reached by evader.

When the evader reaches a pac-dot, the immunity time starts. We will refer to this event only as a *pac-dot position*. In the pac-dot game structure, a frame holds all possible combinations of locations that can be occupied by players ($2 * |V|^{|P|+|E|}$ states), but only one game step is performed on it (see Section 3.5).

Based on the flow networks algorithm [27], we modeled the pruning algorithm. The pruning flow advances one hop per frame as shown in Figure 3.6. Reachable states are those that the evader position is up to $t'+1$ distance steps away from the pac-dot position, and the position of pursuers can be any. Distance is denoted by $d(s) \leq t'+1$. In a frame t', all possible paths from a pac-dot position up to $t'+1$ distance are the reachable states, and the remaining states are unreachable.

To demonstrate the effectiveness of the pruning technique, we will prove the Theorem 3.

Theorem 3. Let A be a set of reachable states and let B be a complementary set of A, where $A \cup B$ is a t' frame. $\forall a \in A$ and $\forall b \in B$, If $d(a) \leq t'+1 < d(b)$, then $C(a) < C(b)$ for all reachable states.

Proof. Consider a sub-game with t frames, where $0 \leq t' < t$. Let A be a set of reachable states, and B be the complementary set of A, such that $A \cup B$ forms a t' frame. Assume that all states for each frame have been generated, and distances and costs have been calculated.

We aim to prove that for any $a \in A$ and $b \in B$, if $d(a) \leq t'+1 < d(b)$, then $C(a) < C(b)$ for all reachable states.

Suppose, for the sake of contradiction, that there exist states $a \in A$ and $b \in B$ such that $d(a) \leq t'+1 < d(b)$ and $C(a) \geq C(b)$.

Since $d(a) \leq t'+1 < d(b)$, it implies that a is within reachable distance of the pac-dot position, whereas b is not reachable within the given frame t'.

However, the assumption $C(a) \geq C(b)$ contradicts the fact that the cost of a is equal to or greater than the cost of b. This contradicts the assumption that a is reachable and b is not reachable within the given frame.

Hence, the assumption that C$(a) \geq C(b)$ must be false, and therefore, we conclude that $C(a) < C(b)$ for all reachable states, given $d(a) \leq t'+1 < d(b)$.

Therefore, the theorem holds. □

As we have proved, unreachable states are unnecessary for computation and simulation of the game, without loss of generality.

We can calculate the state space for the pruning technique using the equation below:

$$S = 2 * \left(2^k * |V|^{|P|+|E|} + \sum_{b=1}^{2^k} \sum_{t'=0}^{t-1} a_{bt'} * |V|^{|P|} \right)$$

where S is the number of states, $a_{bt'}$ the number of reachable nodes in a frame t' of the sub-game b, V represents the set of nodes in the topological map, P the set of pursuers, E the set of evaders, t the immunity time, k the number of pac-dots, and 2^k the number of sub-games.

In a sub-game the flow occurs in the topology map $G = (V, L)$ hop by hop from the frame where the pac-dot position was reached for the evader up to frame $t-1$. The flow calculation time of a sub-game in the worst case is $O(|V| + |L|)$. The flow needs to be calculated for each non-initial sub-game if we have one or more pac-dot ($k > 0$), usually in different positions. The overall complexity of pruning is $O((2^k - 1) * (|V| + |E|))$.

3.7 Simulation, Evaluation, and Results for PEG Framework

In this chapter, we present the details of the implementation of the algorithm and simulator and we also describe the computer's configuration that we used to run the experiments. We also present the carried out simulations and results for the evaluated topologies.

3.7.1 Setup

We developed a discrete simulator to evaluate PEGs in different topologies. The algorithm and simulator were encoded in Python 3 language [12]. Besides the native resources of the Python language, we used the Joblib package [8]. More specifically, we used the class *joblib.Parallel* to implement parallel functions.

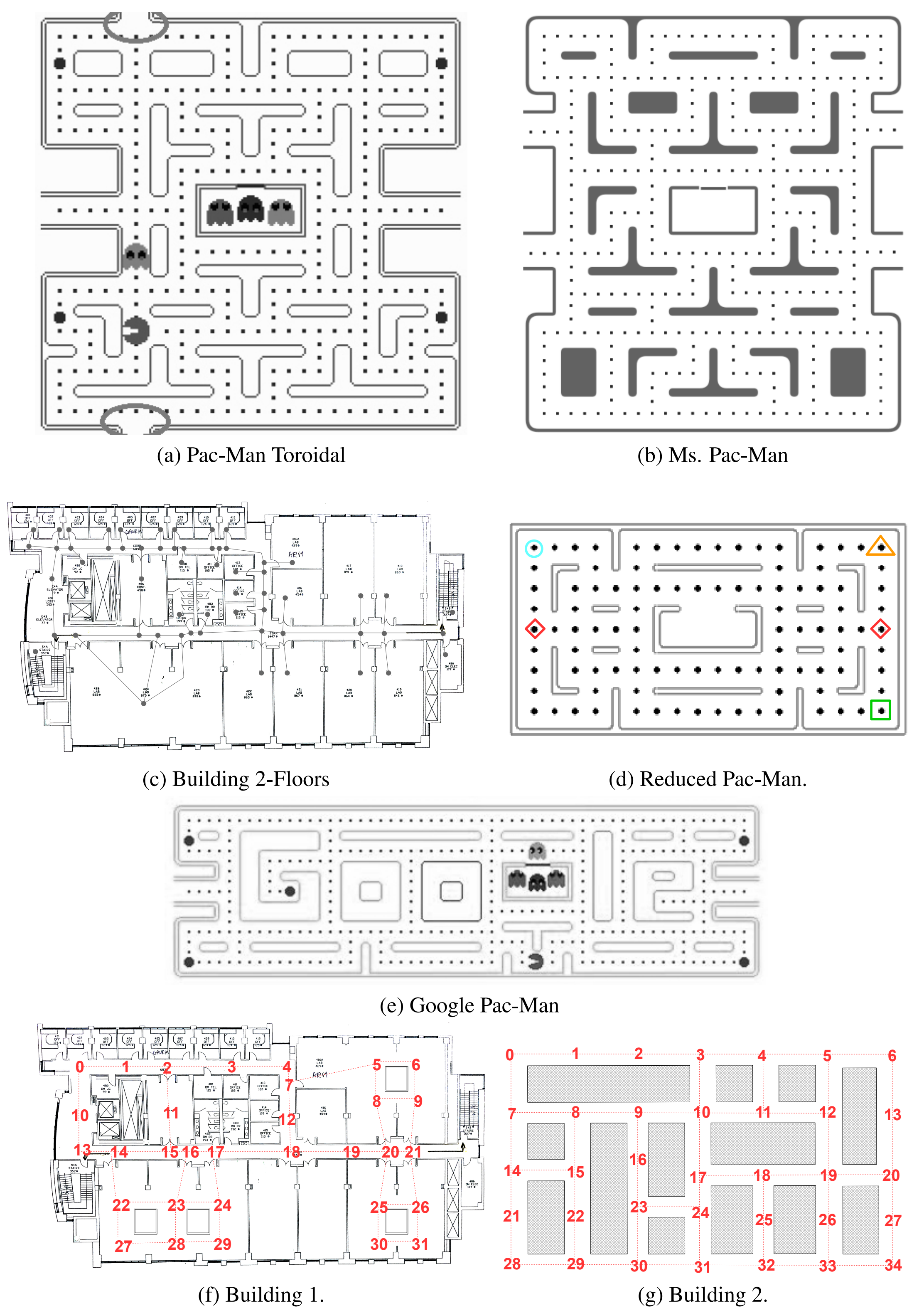
We used three computers to simulate the games:

1. Processor Intel XEON E5-2630 v3 2.4GHz 32 cores, 128GB RAM and 12TB HD;

2. Intel Core i7-6800K 3.4GHz 12 cores, 64GB RAM, 1TB HD.

3. Processor AMD EPYC 7282 2.80 GHz 32 cores, 128 GB RAM, and 240 GB SSD SATA.

All simulations of Table 3.3 and Table 3.4 were performed on computer 1. Pac-man simulations for multi-speed pursuers were performed on computer 2. All simulations of the Table 3.5 and speedup evaluations (Table 3.1) were simulated on computer 3.



(a) Pac-Man Toroidal (b) Ms. Pac-Man

(c) Building 2-Floors (d) Reduced Pac-Man.

(e) Google Pac-Man

(f) Building 1. (g) Building 2.

Figure 3.7: Evaluated topologies.

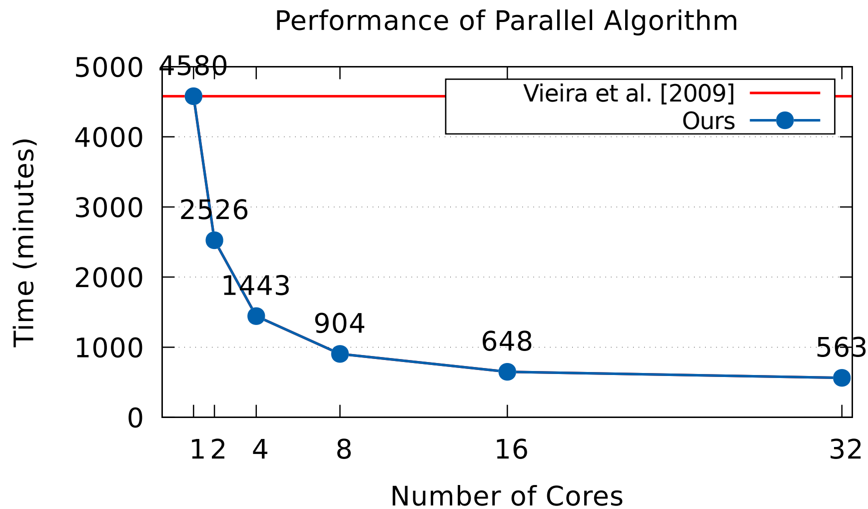


Figure 3.8: Serial execution time performance for execution in one core, equivalent to the previous algorithm in [56] and performance of parallel algorithm execution for the number of cores equal powers of 2 up to 32 (Ours). The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

3.7.2 Simulation, Evaluation, and Results

In this section, we present the speedup evaluation of the parallel algorithm. The remaining subsections show the results of the PEG simulations for the parallel algorithm (Subsection 3.3.2) and its extensions: the multi-speed algorithm (Section 3.4), the pac-dot algorithm (Subsection 3.5.2), and pac-dot algorithm with pruning (Section 3.6).

We evaluated the following topologies: Pac-Man game (Figure 3.7a), a Pac-Man topology in a torus (Figure 3.7a) where we add a north-south connection (marked in red ellipse), Pac-Man version from Google (Figure 3.7e), a two-story building (Figure 3.7c), a Reduced Pac-Man version (Figure 3.7d) where the red circles should be disregarded, and Reduced Pac-Man topology in a torus (Figure 3.7d) where we add an east-west connection (marked in red circle). Topologies Building 1 (Figure 3.7f) and Building 2 (Figure 3.7g) were used to simulate the multi-speed players.

The simulations of the pac-dot algorithm (see Subsection 3.5.2) and the pac-dot algorithm with pruning (see Section 3.6) involved a significant number of generated states and transitions. With nearly a billion states and reaching almost a dozen billion transitions, it was necessary to read and write data using text files to ensure the feasibility of computing the algorithms.

Table 3.1: Speedup for executing the parallel algorithm in relation to the version equivalent to the previous one (core = 1) [56]. *The execution with a single core and a duration of 4304 minutes refers to the parallelizable sections of the algorithm being executed sequentially. The remaining results of this column (Parallel Time (min.)) were executed in parallel. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

Cores	Parallel Time (min.)	Serial Time (min.)	Total Time (min.)	Parallel Speedup	Total Speedup
1*	4304*	276	4580	-	1
2	2267	259	2526	1.90	1.81
4	1162	281	1443	3.70	3.17
8	624	280	904	6.90	5.07
16	368	280	648	11.70	7.07
32	285	278	563	15.10	8.13

Table 3.2: Topologies and simulation results - Parallel Algorithm (32 cores) X Serial Execution (1 core - non-parallelized version). The simulations were performed on a Computer 3, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

Topology	Players	Vertices	States	Transitions	Cost/Steps	Parallel Version Execution Time (min)	Serial Version Execution Time (min)	Speedup
Building 2-Floors	3	118	1643032	20847208	17	10	104	10.4
Reduced Pac-Man	3	102	1061208	13654176	21	13	77	5.92
Reduced Toroidal Pac-Man	3	102	1061208	13803796	21	13	77	5.92

3.7.3 Parallel Algorithm Speedup

Table 3.3: Topologies and simulation results - Parallel Algorithm without pac-dot. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

Topology	Players	Vertices	States	Transitions	Time to generate Game Graph (min)	Cost Calculation Time (min)	Total Execution Time (min)	Cost/Steps	Diameter
Pac-Man	3	290	24389000	313285590	16	272	288	45	51
Pac-Man	3	290	24389000	313285590	16	272	288	65	51
Pac-Man Toroidal	3	290	24389000	402542566	23	355	378	73	38
Pac-Man Google	3	317	31855013	402542566	29	431	460	64	49
Building 2-Floors	3	118	1643032	20847208	1	7	10	17	17
Reduced Pac-Man	3	102	1061208	13654176	4	9	13	21	21
Reduced Toroidal Pac-Man	3	102	1061208	13803796	4	9	13	21	21
Ms. Pac-Man	3	308	29218192	376875135	39	831	870	69	44

Initially, we present the performance of the parallel algorithm (Algorithm 1 and Algorithm 2). In Figure 3.8 we have the time (in minutes) of serial execution for 1 core (the execution time in [56]) and the time of parallel executions for 2 cores up to 32 cores (the time of our executions). Executions were performed on the Pac-Man topology (Figure 3.7a) with 2 pursuers and an evader. All players with speed = 1. In Figure 3.8 we can see that as the number of cores doubles, the curve flattens (blue line points) in comparison to 1 Core line (red line). This happens with the growing number of cores because the overhead for creation and maintenance of the pool of threads/process grows too.

The speedup is computed based on Amdahl's law [21] to demonstrate the improvement

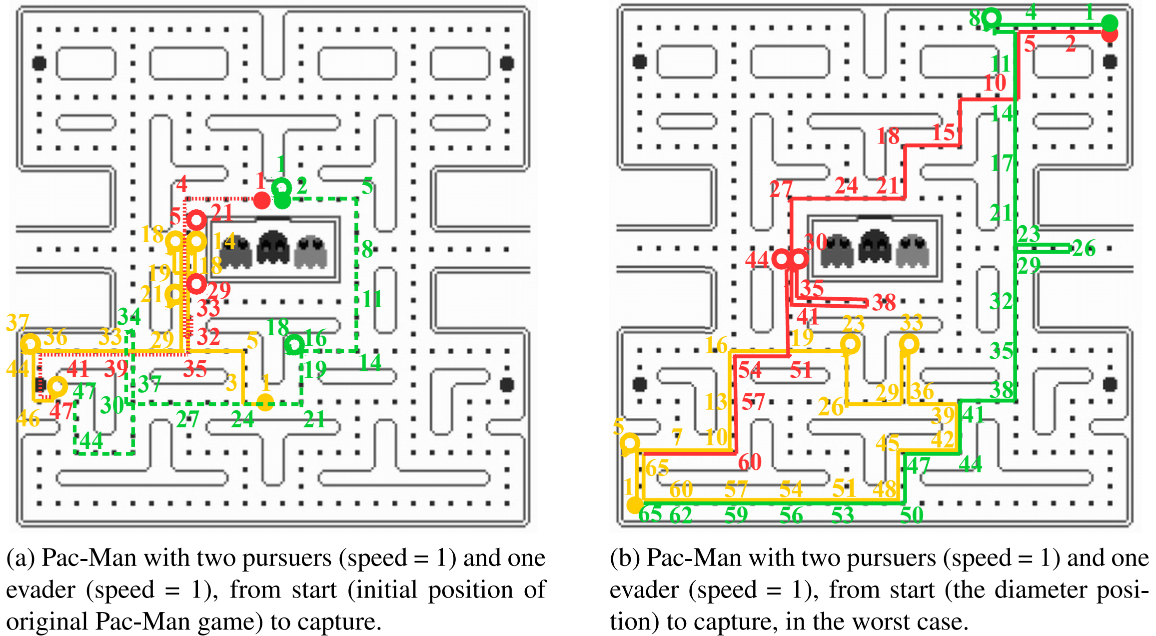
reached if a part of a system is improved. The speedup is computed using the following formula: $S = \frac{T(1)}{T(2)}$, where S is the speedup, $T(1)$ is the time of the program without improvement and $T(2)$ is the program with improvement. Speedup is the performance metric. In summary, a speedup = 1, is the baseline or the indication that no improvement was achieved with the improved version over the unimproved version ($T(1)$). If speedup > 1, then the resulting number represents how many times the performance of $T(2)$ has improved compared to $T(1)$. Finally, if the speedup $<$ 1, then $T(1)$ performs better than $T(2)$. $T(1)$ is equivalent to the execution time of the previous algorithm [56], which is completely serial and its speedup is 1.

In Table 3.1 we can see the time of execution detailed in Parallel Time ($T(2)$), Serial Time ($T(1)$), and Total. For one core the Serial Time represents approximately 6% of the total time, while for 32 cores represents more than 50%. In our case, the program not parallelized is $T(1)$ and the parallel version executed by N cores is $T(2)$. The last two columns (on the right) show the speedup for the parallel part and for the total execution (parallel part and strictly serial part). The speedup for 2 cores was 1.90 for parallel and 1.81 for total executions, while for 32 cores the speedup was 15.10 and 8.13 for parallel and total executions, respectively.

Another crucial aspect to consider is the necessity of reading and writing data from text files. This requirement resulted in substantial portions of the program remaining non-parallelized, thereby limiting the potential for achieving significant performance gains and higher speedups.

In Table 3.2, we present the execution times of the parallel (32 cores) version of the algorithm and the execution times of the serial (1 core) version of the algorithm for three different topologies: Reduced Pac-Man, Reduced Toroidal Pac-Man, and Building 2-Floors. Despite the overhead associated with the parallel algorithm, the parallel execution offers a significant advantage over the serial version in terms of execution time. For the Reduced Pac-Man topology, the execution time of the parallel part is 13 minutes, compared to 77 minutes for the serialized (non-parallel) time execution. Similarly, for the Reduced Toroidal Pac-Man topology, the parallel execution time is 13 minutes, while the serial execution time is 77 minutes. Finally, for the Building 2-Floors topology, the parallel execution time is 10 minutes, whereas the serial execution time is 104 minutes. The speedup achieved by the parallel algorithm is noteworthy. For Building 2-Floors topology, the speedup is 10.4, indicating that the parallel execution is approximately 10.4 times faster than the serial execution. For Reduced Pac-Man and Reduced Toroidal Pac-Man topologies, the speedup is 5.92 for both.

These results highlight the effectiveness of parallelization, leveraging multiple cores, in significantly reducing the overall computation time for the parallel algorithm. The speedup achieved demonstrates the efficiency and scalability of the parallel approach, making it a favorable choice for optimizing execution time in pursuit-evasion scenarios.



(a) Pac-Man with two pursuers (speed = 1) and one evader (speed = 1), from start (initial position of original Pac-Man game) to capture.

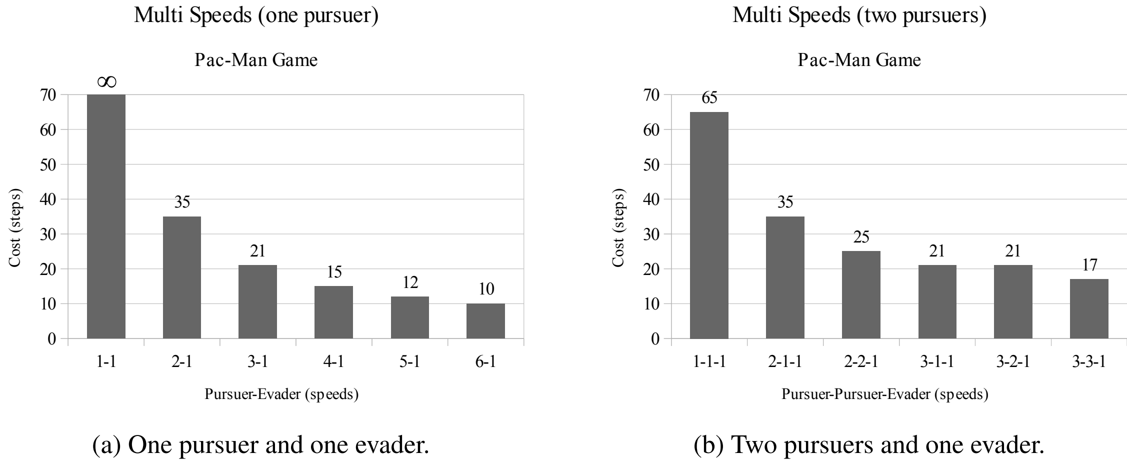
(b) Pac-Man with two pursuers (speed = 1) and one evader (speed = 1), from start (the diameter position) to capture, in the worst case.

Figure 3.9: Traces execution of the Pac-Man topology. In both simulations, depicted in Figure 3.9a and Figure 3.9b, the evader's path is represented by the yellow trace, while the pursuers' paths are shown in red and green. The accompanying numbers, which match the color of each trace, indicate the game step number and the corresponding evolution steps. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

3.7.4 Parallel Algorithm

Table 3.3 shows the results of the PEG simulation for the previously described topologies. The set of topologies was simulated on computer 1. Columns are defined from left to right. Column 1 identifies the topology. Column 2 indicates the number of players (pursuers and evaders). Column 3 shows the number of vertices for each topology. Columns 4 and 5 display the number of states and transitions needed to generate the game graph. Column 6 depicts the time to generate the game graph (all states and transitions). Column 7 shows the time to calculate the cost. Column 8 presents the total execution time (Column 6 + Column 7). Column 9 shows the number of steps (cost) to capture the evader in the worst case. Column 10 presents the diameter of the topology graph. The difference between cost and diameter is that the diameter is an invariable continuous path from beginning to end and the cost may vary due to the evader's attempts to prolong capture with each movement. For example, if the evader does not move during the entire game, the pursuer, in the worst scenario, would have to traverse the diameter of the graph, and the cost would be the same as the diameter.

Pac-man topology considering the original starting positions of the game (first line of Table 3.3), requires 45 steps to capture the evader and only needs 2 pursuers. Considering the



(a) One pursuer and one evader.

(b) Two pursuers and one evader.

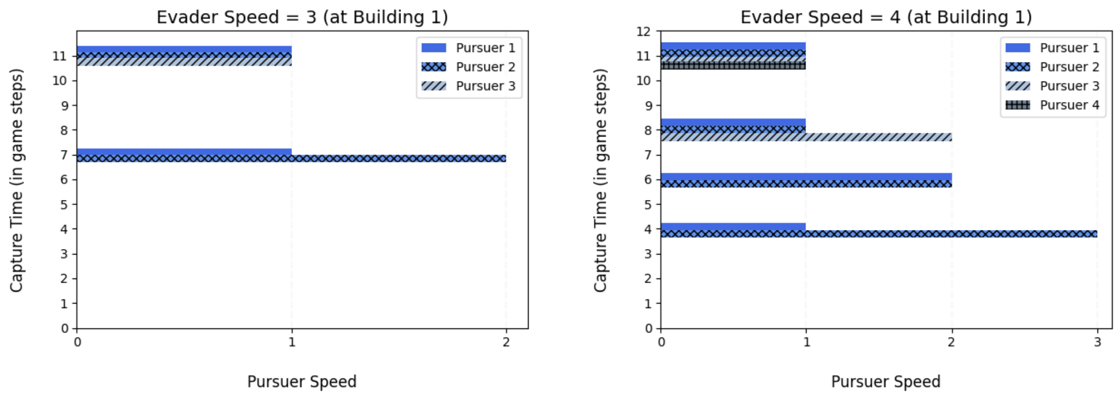
Figure 3.10: Pac-Man topology evaluated for multi-speed pursuers. The simulations were performed on Computer 2, which had 12 cores and 64 GB of RAM (refer to Subsection 3.7.1).

diameter distance between pursuers and the evader as the starting position (second line of Table 3.3) requires 65 steps to capture, it also only needs 2 pursuers to capture the evader. Trace execution of the Pac-Man game can be seen in Figure 3.9, for the two simulations aforementioned. The traces show the paths carried out to the players: red and green, the pursuers trace, and yellow the evader trace. These evaluations show that Pac-Man is overkill with 4 ghosts (pursuers) since it only needs 2 pursuers to capture an evader.

3.7.5 Heterogeneous / Multi-speed Player

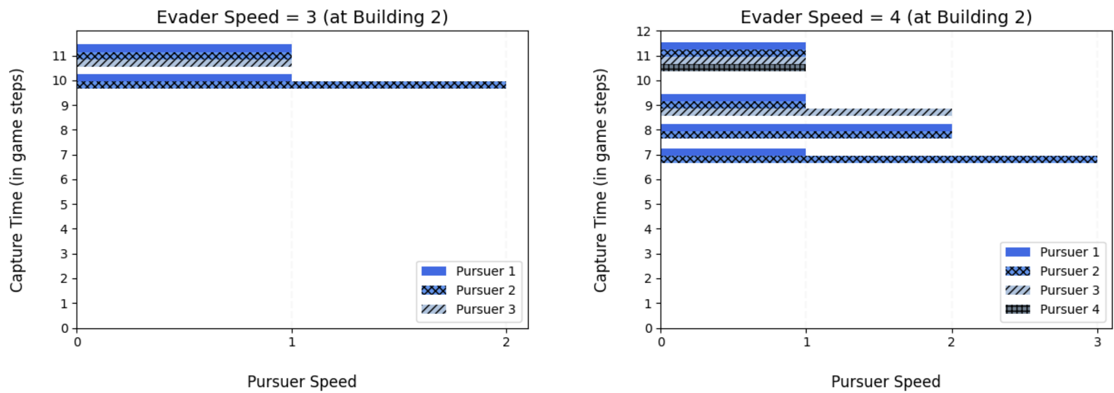
Next, we evaluate the Pac-Man topology for heterogeneous pursuers with different speeds (Algorithm 3 in Section 3.4). Figure 3.10a presents the number of steps for each configuration. Each configuration (labels on X-axis) has two numbers that represent the speeds of *pursuer-evader*. The evader's speed is always 1. If the cost is infinite, it means that the number of pursuers is insufficient to capture the evader. As the speed of the pursuer increases, the number of steps to capture decreases since the pursuer can travel more hops in one time step. In Figure 3.10a, we can note that with only one pursuer of speed equal 1, the game runs infinitely, because one pursuer is not enough to capture the evader in this topology. In the last column on the right, the pursuer's speed is 6 and the cost is 10.

Figure 3.10b shows the costs of the game with different speeds for games with 2 pursuers. The labels on the X-axis have three numbers to indicate the speeds of *pursuer-pursuer-evader*. In the first column on the left, the cost is 65 for two pursuers with a speed equal to 1. This result is the same as the Figure 3.9b. Note that in the fourth and fifth columns (from right



(a) Building 1: heterogeneous players (evader speed = 3)

(b) Building 1: heterogeneous players (evader speed = 4)



(c) Building 2: heterogeneous players (evader speed = 3)

(d) Building 2: heterogeneous players (evader speed = 4)

Figure 3.11: Evaluation of Building 1 and Building 2 topologies for multi-speed players. Each chart depicts the execution in a specific topology, with the evader speed on the upper X-axis, pursuer speed on the lower X-axis, and capture time on the Y-axis. (a) shows the results for Building 1 topology, with evader speed = 3. Two cases are tested: the first with two pursuers, both with speed 1 and the second with two pursuers, one with speed 1 and the other with speed 2. The execution times were 11 game steps and 7 game steps, respectively. The same pattern applies to (b), (c), and (d). The simulations were performed on Computer 3, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

to left), the use of a second pursuer doesn't result in a gain in comparison to the third column (from right to left) of Figure 3.10a, the cost is 21 for everyone.

We evaluated the topologies Building 1 (Figure 3.7f) and Building 2 (Figure 3.7g) for heterogeneous players. We simulated each topology for evader speed equal to 3 and 4. For all simulations, pursuers, and the evader started at the opposite end of the topology diameter. Pursuers follow two criteria: the minimum number of pursuers to guarantee the evader capture and the sum of the speed of all pursuers is equal to the evader speed. Figures 3.11a-3.11d show the results of the simulations mentioned above. Each bar represents a pursuer and a grouping of bars represents a team of pursuers. The label on X-axis is the cost to the pursuers capturing the evader and the label on Y-axis is the pursuers speed. Figure 3.11a the evader speed is 4, with

Table 3.4: Topologies and simulation results - Parallel Algorithm with pac-dot. The simulations were performed on Computer 1, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

Topology	Players	Vertices	States	Transitions	Number of Pac-dot	Immunity Time	Pac-dot Position	Cost/Steps	Diameter
Reduced Pac-Man	3	102	12734496	166569444	1	10	square	34	21
Reduced Pac-Man	3	102	36081072	472402512	2	10	triangle/square	39	21
Reduced Toroidal Pac-Man	3	102	12734496	168394616	1	10	square	36	21
Reduced Toroidal Pac-Man	3	102	36081072	477578800	2	10	circle/square	50	21

Table 3.5: Topologies and simulation results - Parallel Algorithm with pac-dot and pruning. The simulations were performed on Computer 3, which had 32 cores and 138 GB of RAM (refer to Subsection 3.7.1).

Topology		States			Transitions			Simulation Time		
	Pac-dot Number	Without Pruning	Pruning	Ratio (P/WP)	Without Pruning	Pruning	Ratio (P/WP)	Without Pruning	Pruning	Ratio (P/WP)
Reduced Pac-Man	1	12734496	3891096	30.56%	166569444	50159972	30.11%	03:02:55	00:45:56	25.11%
Reduced Pac-Man	2	36081072	10351980	28.69%	472402512	133623512	28.29%	09:04:36	02:31:37	27.84%
Reduced Toroidal Pac-Man	1	12734496	3891096	30.56%	168394616	59636060	35.41%	02:47:18	00:48:33	29.02%
Reduced Toroidal Pac-Man	2	36081072	12744900	35.32%	477578800	166293280	34.82%	07:21:01	03:27:07	46.96%
Reduced Toroidal Pac-Man	4	176160528	68125392	38.67%	2332709344	891087400	38.20%	43:59:28	17:20:52	39.43%
Pac-Man	1	292668000	60720238	20.75%	3781151956	777796844	20.57%	-	45:14:50	-
Pac-Man	2	829226000	145324876	17.53%	10716891864	1858045016	17.34%	-	129:36:55	-

two pursuers with 1 and 3 speeds respectively the cost to capture is 4 game steps and the cost is 11 to four pursuers with speed 1 each. We can observe in the simulations (Figures 3.11a- 3.11d) that a configuration where a small number of pursuers in which one pursuer has speed close to the evader speed performs better than configurations with more than two pursuers with greater equality in the speed distribution.

3.7.6 PEG with Pac-dot

Table 3.4 shows the results of the PEG with pac-dot simulation for Reduced Pac-Man (Figure 3.7d) and Reduced Toroidal Pac-Man topologies (Figure 3.7d with a link between the nodes inside the diamonds). This topology was simulated on computer 1 and computer 2. Columns are defined from left to right. Columns 1-5 have the same meaning as in Table 3.3. Column 6 shows the number of pac-dot of the simulation. Column 7 presents the evader immunity time duration. Column 8 depicts the position of each pac-dot in a simulation, differentiated by geometric shapes. Columns 9 and 10 have the same meaning as in Table 3.3. Reduced Pac-Man topology with 1 and 2 pac-dots, result in a cost of 34 and 39 respectively. Reduced Toroidal Pac-Man topology with 1 and 2 pac-dots, result in a cost of 36 and 50 respectively. Note in Table 3.3 that the cost is 21 for these two topologies without pac-dot. Although the immunity time is 10 time steps for both topologies, the resulting cost is not $21+10$ and $21+20$. This is due to the pursuers escaping the evader during the immunity time. Despite the immunity

time, all players performed an optimal strategy.

3.7.7 Pruning

Table 3.5 shows the results of the PEG with pac-dot in comparison to the PEG with pac-dot with pruning simulation. In the first line, the Table is divided in four groups: Topology; States; Transitions; and, Simulation Time. Topology has two columns, on the left is the topology name and at right the number of pac-dots configured. States and transitions took the number of states/transitions (respectively) generated by pac-dot without pruning simulation (PW) and pac-dot with pruning simulation (P), hence ahead, namely pruning. The third column of States and Transitions part took the ratio between P/PW. In the Simulation Time the three columns are the time to compute the game without pruning, with pruning, and the ratio of the pruning. Cells with - (hyphen) mean that it was not possible to complete the execution due to the lack of computational resources, such as primary or secondary memory space.

In Table 3.5, Pac-Man topology simulated with 2 pac-dots had a ratio of 17.53% in the number of states, a gain of 82.47%. The gain was similar to the number of transitions, which in absolute numbers exceeds 10.5 billion of transitions.

3.8 Conclusion

In this chapter, we presented a parallel algorithm to compute the optimal strategy to capture an evader in a PEG, and its extended versions of the multi-speed and pac-dot strategies. We also presented the pruning technique for the pac-dot strategy to reduce the number of states, transitions, and consequently the computational resources needed to compute the game. For all algorithms proposed, we provide the complexity time and proofs when necessary. We also present the simulation, evaluation, and results of the experiments for a set of topologies. These experiments validate the set of algorithms described.

Chapter 4

Resource Allocation for Heterogeneous Multi-agent Teams Framework

In this chapter, we present a comprehensive framework for resource allocation in heterogeneous multi-agent teams, specifically designed for a given set of heterogeneous teams of agents[2]. Our framework aims to create a completely heterogeneous environment with minimal resource allocation requirements, enabling efficient gameplay in the Pursuit-Evasion Game (PEG) within a target topology. Additionally, our resource allocation strategy leverages the output solutions of the PEG framework to evaluate execution under the presence of potential agent failures.

Motivated by the challenges of enabling networked agent teams to effectively accomplish tasks and achieve goals, our resource allocation framework focuses on organizing heterogeneous multi-agent teams topologically. Building upon the work [1], who proposed a resource allocation technique for networked heterogeneous agents, we extend their approach to cater to multiple teams. Our objective is to manage resource allocation for multi-teams, ensuring that all heterogeneous nodes within the teams have access to the required resources through resource sharing in their respective node neighborhoods [47].

Efficient resource allocation in multi-agent systems is comparable to the NP-complete conjunctive planning problem [6, 14] when dealing with a sequence of tasks. In our formulation, instead of addressing a sequence of tasks, we focus on the resource allocation problem for a set of teams, which shares similarities with the aforementioned problem.

In formulating our resource allocation framework, we draw motivation from the concept of multi-speed algorithms (as presented in Section 3.4). We recognize the critical role of resource allocation in achieving enhanced performance and speed. Specifically, we consider multi-speed agents as those equipped with multiple motors, highlighting the importance of efficient distribution and utilization of resources to maximize the capabilities and overall efficiency of the agents. By incorporating these principles into our framework, our goal is to optimize resource allocation and enable improved performance in heterogeneous multi-agent systems. In Figure 3.11b (Subsection 3.7.5), we present four simulations where we vary the resources (motors) to observe the effect on capture time. Assuming each speed unit is provided by a motor

[2]In the context of this thesis, consider the meaning of the following terms: robot, agent, and player are synonyms. Team, group, and network denote a group of robots, agents, or players. Networks, groups, and teams denote a set of teams, groups, or networks. The terms of each trio are interchangeable.

unit when the evader has a speed of 4 (using four motors) and two pursuers have speeds of 2 (using two motors each), the capture time is 6 game steps. However, if the pursuers have speeds of 1 and 3 (using one and three motors, respectively), the capture time reduces to 4 game steps. This demonstrates that intelligently allocating resources can significantly impact the capture time in the game.

By incorporating the principles of resource allocation and teamwork within our framework, we enable agents to dynamically adapt their resource sharing and allocation strategies based on their individual capabilities and the requirements of the tasks at hand. This flexibility ensures that agents efficiently utilize available resources, including motors or other relevant resources, to achieve optimal performance in completing tasks within the heterogeneous multi-agent teams.

To evaluate the effectiveness of our resource allocation framework, we apply it to the PEG in Chapter 3. The PEG scenario involves a multi-pursuer team of networked agents collaboratively working to capture an evader. By leveraging resource allocation and facilitating information sharing through resources [44] like GPS, sensors, and other neighborhood-provided inputs, our framework enhances the team's ability to capture the evader efficiently.

By integrating the PEG scenario with our resource allocation framework, we demonstrate the practical application and benefits of our approach in a specific task-oriented environment. The resource allocation algorithm constructs multi-teams of heterogeneous agents, ensuring all necessary resources are available through resource sharing within their respective neighborhoods. This allows agents to dynamically adapt their resource allocation strategies based on the real-time demands of the PEG scenario. Furthermore, the resource allocation approach enables the replacement of failed agents in real-time, mitigating disruptions in task resolution and ensuring uninterrupted gameplay.

The evaluation of our technique in the context of the Pursuit-Evasion Game involves analyzing team composition, introducing and fixing failures throughout different instances, and assessing the scalability and effectiveness of our framework by running multiple instances simultaneously.

Overall, our resource allocation framework for heterogeneous multi-agent teams, drawing motivation from the concept of multi-speed algorithms and their implications for resource usage, provide a practical and adaptable solution for optimizing resource allocation and enabling effective collaboration in diverse scenarios.

4.1 Background

In this section, we delve into the background that forms the foundation of our work, with a specific focus on a framework for analyzing heterogeneous networks and enabling their transformation into completely heterogeneous cooperative networks (CHCN).

Completely Heterogeneous Cooperative Networks

In [1], the authors present a framework for analyzing network topology and characterizing heterogeneity in cooperative networks consisting of diverse agents. The framework utilizes a graph coloring formulation to investigate the interactions among agents with different types of resources or capabilities distributed in a heterogeneous network. The network topology is represented as a graph, where agents are depicted as vertices and the links between them as edges. Each resource or capability in the network is assigned a unique color or label.

The main objective of the framework is to transform a given graph coloring into a completely heterogeneous graph by employing basic graph operations such as adding or removing edges. An agent is considered completely heterogeneous if all the colors present in the network are available within its immediate neighborhood, which encompasses both its own color(s) and the colors of adjacent nodes. If all nodes in the graph achieve complete heterogeneity, then the graph (or network) as a whole is classified as a completely heterogeneous cooperative (CHCN).

A heterogeneous network is transformed to a completely heterogeneous cooperative network (CHCN)performing some matrix operations. An adjacency matrix is denoted by A of dimension $n \times n$, where n is the number of agents. C denotes the color matrix, of dimension $n \times r$, where r is the number of different types of colors (resources/capabilities) in the network. If an agent i has the color j, then $C_{ij} = 1$, otherwise $C_{ij} = 0$. A and C are given to describe the network topology and agents, respectively. From A and C other two matrices are generated. The color distribution matrix is denoted by ϕ and calculated by the equation: $\phi = (A+I)C$, where I is the identity matrix of dimension $n \times n$. ϕ_{ij} represents the number of color j available in the closed neighborhood of node i. If $\phi_{ij} = 0$ means that it is needed to add an extra edge between node i and another node that has color j. The number of extra edges to turn a graph completely heterogeneous is $\left\lceil \frac{z(\phi)}{2s} \right\rceil \leq \varepsilon \leq z(\phi)$, where $z(\phi)$ is the number of zeros in ϕ, s is the maximum number of colors assigned to any vertex, and ε is the number of extra edges needed. The last

matrix is U, calculated based on the matrix C and ϕ as follow:

$$U_{i,k} = \begin{cases} |\{v_j : v_j \in N(v_i), \text{ and } \phi_{jk} = 1\}|, & \text{if } C_{ik} = 1 \\ 0 & \text{otherwise} \end{cases}$$

U has dimension $n \times r$. The U_{ij} value indicates that the number of vertices will become deficient of the color j if the vertex i is removed. The sum of the ith row of U indicates the increase of the deficiency of the network, the greater the row sum value, the greater the node deficiency. The framework offers other tools, but they are outside the scope of this work.

4.2 Completely Heterogeneous Network for PEG

In this section, we present the contributions in this chapter based mainly on the work [1], addressed specifically to the PEG domain based on work [56].

Based on work [1], we extend the technique to perform a prior analysis of the networks to classify them as to the possibility of transferring or the need to receive resources. This step is useful mainly along task resolution, for transferring resources between teams in case of failure. Then, agent transfers are performed to provide resources to deficit networks. After this step, the functionalities provided by the framework (described in the Section 4.1) are used and the heterogeneous networks are transformed into CHCNs. This extension has been integrated into the framework developed in our previous work ([10, 11]), that computes the optimal solution for PEG to MPSE (Section 3.1). For each CHCN an evader is associated and the PEG is solved to Multi-Pursuer Multi-Evader – MPME. Finally, with the extension of the frameworks, we turn the technique fault tolerant to cases described in Subsection 4.2.2. The same techniques for transferring agents are used to overcome agent or resource failures or context switches of PEG solutions. As well as the optimal strategy ensuring the optimal solution, the new technique also ensures an optimal solution in cases of agents/resources failure in real time.

4.2.1 Resource Allocation

Let G be the set of heterogeneous networks and each network is indexed by $i = \{1, ..., g\}$. For each G_i is given an adjacency matrix A and a color matrix C. From the given matrices, the ϕ matrices and U are generated, respectively distribution color matrix and deficiency color matrix. We consider that r (the number of resources/colors available in a network) is the needed number

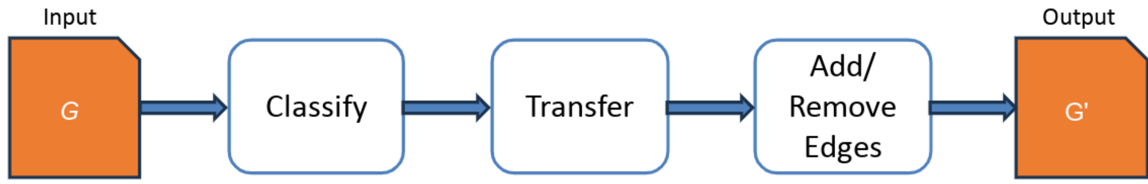


Figure 4.1: Resource Allocation algorithm flow: The input consists of a set of agent networks, each with an adjacency matrix (A) and a color distribution matrix (C). In the Classify stage, networks are analyzed and classified as Provider, Sufficient, or Receiver based on available resources and the minimum agent requirement. In the Transfer stage, agents are transferred from Provider to Receiver networks when compatibility exists. In the Add/Remove Edges stage, necessary edge modifications are made to transform networks into CHCNs. The resulting CHCNs are the algorithm's output.

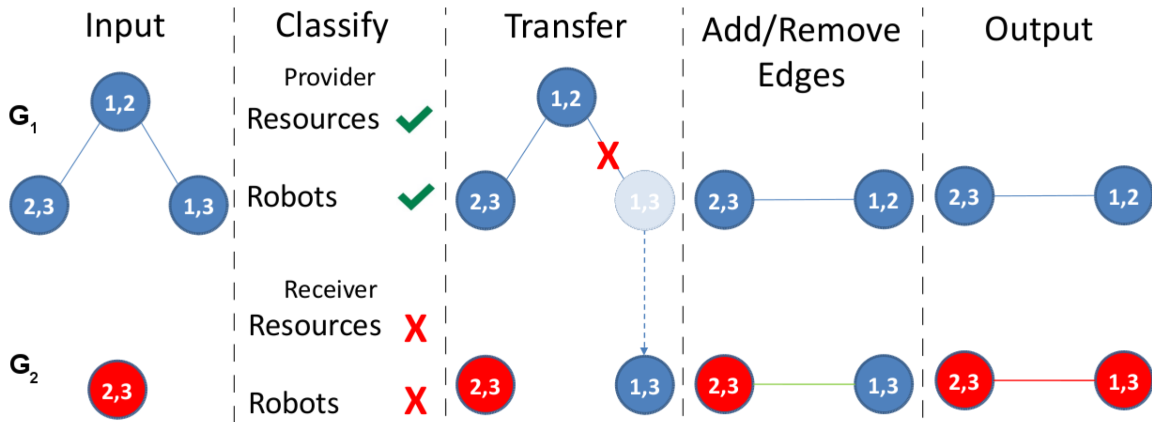


Figure 4.2: Example of resource allocation and stages of the Algorithm 5.

of resource types for each G_i, in this case, we assume that resource requirement is identical for all networks. Subsequently, all networks are evaluated and classified as sufficient, provider or receiver of agents with resources/colors. A sufficient network has available all r colors and the number of pursuers equal $c(T)$, the minimum number of agents required to capture (in the PEG) for a topology (denoted by T). A provider network is one that has one or more agents that have spare colors and it has a number of pursuers greater than $c(T)$, and it can transfer them to a receiver network, if compatible. A receiver network needs external resources to make it CHCN and/or to reach $c(T)$. Next, all agent transfers between networks are made. If necessary, edges are added to make the network a CHCN. In the next step, redundant edges are removed and then the networks are ready to run the PEG.

In Figure 4.1 the Resource Allocation algorithm follows a specific flow. The input to the algorithm consists of a set of agent networks, each represented by an adjacency matrix (A) and a color distribution matrix (C). The algorithm progresses through three main stages. In the Classify stage, each network is analyzed individually. Based on the available resources and the minimum number of agents required, each network is classified as Provider, Sufficient, or Receiver. This classification helps determine the role of each network in the resource allocation process. In the Transfer stage, agents are transferred from Provider networks to Receiver

networks. This transfer occurs when there is compatibility between the networks, meaning that the resources of the Provider network match the requirements of the Receiver network. This stage facilitates the redistribution of resources among the networks. The final stage is Add/Remove Edges. In this stage, the algorithm examines the networks and determines the necessary edge modifications. This involves adding or removing edges as required to transform the networks into completely heterogeneous cooperative networks (CHCNs). By modifying the network topology, the algorithm ensures that each network achieves complete heterogeneity. The resulting output of the Resource Allocation algorithm is a set of CHCNs, which represent the transformed agent networks with optimal resource allocation and cooperation.

In Figure 4.2 we show an execution example of the Algorithm 5 stages as described in Figure 4.1 step by step for networks G_1 and G_2. Consider the list of resource types $r = \{1,2,3\}$ and the smallest number of agents $c(T) = 2$. The input step represents the topologies before the resource allocation. The second stage classifies each network based on r and $c(T)$ criteria, labeling them as Provider, Sufficient or Receiver. In the third stage, a resource-compatible agent is transferred from G_1 to G_2. On the next stage edges are added or removed as needed. The last stage, output of the CHCNs.

In the Algorithm 5, we detail the steps to redistribute vertex and to make CHCNs. The algorithm starts at line 22, calling the function *classify* with argument G, the set of all networks. The repeat structure (lines 23-34) iterates while there exist possible transfers. The for structure (lines 25-33) iterates over each item of N, the set of receiver networks. The for structure (lines 26-29) iterates over each item of P, set of provider networks. On the line 27, the function *transfer* is called, passing the p and n networks as arguments, to evaluate the possible transfer of nodes. In the next two lines, if the transfer occurred, then a $True$ boolean value is returned and assigned to change variable and the inner for broken. On the line 30, if *change* is $True$ the *classify* function is called to N and then for P to reclassify the networks and break the external for. In line 36 the number of extra edges to the network g is calculated, being added in line 38 if $extra$ is greater than zero. The removal of redundant edges is performed on line 39. The function *classify* (lines 1-12) takes as a parameter a collection of networks iterated through for on line 2. At line 3 if the network g has all required colors, then three cases can happen: g is added to S (line 5) if it has the minimum number of pursuers but it doesn't have non-essential vertex (a vertex that, if transferred, the network would remain as CHCN, that is, classified as Provider or Sufficient); g is added to P set (line 8) if it has non-essential vertex and more than minimum number of pursuers; g is added to N set (line 10) if it needs more pursuers. Otherwise, g doesn't have all the required colors, then it is added to the N set (line 12). *Transfer* function (lines 13-19), take as parameters the networks p and n, candidates for provider and receiver, respectively. If there is compatibility between p and n, the compatible agent is removed from p and added to n (line 15). On the next two lines, the matrices ϕ and U are recalculated and $True$ is returned. If the networks are not compatible, $False$ is returned on line 19. Algorithm 5 outputs are the maximum number of CHCNs. The output is the input to the PEG algorithm.

Algorithm 5 Resource Allocation Algorithm

```
1: function CLASSIFY(collection)
2:     for all g in collection do
3:         if g has all r colors then
4:             if the number of vertices of g ≥ c(T) and g doesn't have non-essential vertex
           then
5:                 add g to S                                              ▷ Sufficient set
6:             else
7:                 if the number of vertices of g > c(T) and g has non-essential vertex then
8:                     add g to P                                          ▷ Provider set
9:                 else
10:                    add g to N                                          ▷ Receiver set
11:        else
12:            add g to N
13: function TRANSFER(p, n)
14:     if p is compatible with deficiency in n then
15:         remove compatible vertex from p and add in n
16:         calculate the matrices φ and U of p and n
17:         return True
18:     else
19:         return False
20:
21: function MAIN( )
22:     CLASSIFY(G)
23:     repeat
24:         change ← False
25:         for all n in N do
26:             for all p in P do
27:                 if TRANSFER(p, n) then
28:                     change ← True
29:                     break                                         ▷ Break the inner for
30:             if change then
31:                 CLASSIFY(N)
32:                 CLASSIFY(P)
33:                 break                                             ▷ Break the outer for
34:     until not change
35:     for all g in G do
36:         extra ← EXTRAEDGES(g)                                         ▷ ε value
37:         if extra > 0 then
38:             add the necessary extra edges in g
39:         analyze and remove redundant edges
```

The complexity of the Algorithm 5 is $O(G^2)$.

The maximum number of CHCNs that can be composed and the minimum number of agents per CHCN depends on the diversity of resources per agent and their number. Calculating precisely the optimal distribution of resources and the maximum number of CHCNs constructed

is impractical as it is an exponential time problem. The Theorem 4 prove the interval of the minimum number of agents in CHCN.

Theorem 4. Let m be the estimate of the number of agents per network to become a Complete Heterogeneous Cooperative Network (CHCN). Let r be the number of different types of resources necessary in a CHCN, and let s be the maximum number of resources per agent. We want to establish that m lies within the closed range $[b,w]$, where b is the minimum number of agents in the best case, and w is the minimum number in the worst case. In other words, $[b,w] = m \in \mathbb{N} : b \leq m \leq w$.

Proof. Calculating the optimal distribution of resources precisely is impractical due to its exponential time complexity. However, we can prove the best and worst cases that bound the distribution based on the network input.

We will divide the proof into two cases: when $m = b$ (best case) and when $m = w$ (worst case). Let q be the remainder of a division, where $q \geq 0$.

Case 1. Best Case: Assume that m is the minimum number of agents with at most s resources required to form a CHCN, where $r \geq s$, $r \geq m$, $r > 0$, and $s > 0$. Let's consider the scenario where there is only one resource of each type. The number of agents with at most s resources in the best case can be calculated as $m = \left\lfloor \frac{r}{s} \right\rfloor$, where $\lfloor \cdot \rfloor$ denotes the floor function. Let q be the remainder of the division $\frac{r}{s}$. Then, we have:

$$m = \left\lfloor \frac{r}{s} \right\rfloor = \frac{s-q}{s}$$

Note that s is a divisor of $r-q$, and the quotient is m. Therefore, $m = b$ represents the minimum number of agents with at most s resources in the best case.

Case 2. Worst Case: Assume that m is the minimum number of agents with at most s resources required to form a CHCN in the worst case, where $r \geq s$, $r \geq m$, $r > 0$, and $s > 0$. In this case, there can be repeated resources (if $s > 1$), meaning some resources exist in multiple agents, and each agent contributes at least one new resource. Let's assume that all m agents have s resources and contribute with exactly one new resource, except for the first agent, which contributes with s new resources. We will prove this case by contradiction. Suppose $m < r-s+1$. Considering that s is the number of resources per agent, the first agent contributes with s new resources. This leaves $r-s$ new resources to make the network a CHCN. According to our assumption, each additional agent contributes at least one resource, so the total number of resources contributed by m agents is $m+s-1$.

If $m < r-s+1$, then $m+s-1 < r$. However, this contradicts our assumption that there are r resources necessary in the CHCN. Hence, we conclude that $m = w$ represents the minimum number of agents with at most s resources in the worst case.

Therefore, combining Case 1 and Case 2, we can state that m lies within the closed range $[b,w]$, as desired.

□

Intuitively, we can estimate the maximum number of CHCNs from the set of all agents and their resources. Let us consider the following assumptions: each agent has at least one resource; each resource resides in a different agent (if an agent has more than one identical resource, then only one is considered); each agent has all r resources in its close neighborhood, a characteristic inherent to the CHCN. Considering the assumptions above, we can conclude that the maximum number of CHCNs is equivalent to the number of a resource type in the smallest number. More formally we can model as follow:

$$v = \sum_{k=1}^{|G|} \sum_{i=1}^{|G_k|} \sum_{j=1}^{r} G_k.C_{i,j} + v_j \tag{4.1}$$

In Equation 4.1, v denotes a row vector of size r, where r is the size of the set R, the list of all different types of resources. Each index i matches a resource of the set $R = \{1,2,...,r\}$. All positions of v are initialized with zero. The total number of each resource is counted and saved on row vector v. From left to right, the first summation iterates over G the set of all networks. The second summation iterates over the agents of G_k and the third summation iterates over the resources indexes. Each network G_k has a matrix C of dimension $n \times r$ associated, where $n = |G_k|$, and each cell $C_{i,j}$ has the value 1 if the agent i has the resource j or the value 0 otherwise. The sum of each resource j is accumulated in the position v_j.

$$(i,k) = min(v) \tag{4.2}$$

In Equation 4.2 the min function results in a tuple with values i and k, respectively the index/type of resource and the value in v_i. That is, the resource with the minimum number among all agents of all networks is $k = v_i$. Next, the tuple of values (i,k) is used to prove Theorem 5, where the hypothesis is that k is the maximum possible number of CHCNs.

Theorem 5. Consider the set of resources R, which consists of r different types of resources. Let $r_i \in R$ denote the resource with the minimum number of units among all resources in G, and let k be the number of units of r_i.

Proof. Consider the set of resources R, which consists of r different types of resources. Let $r_i \in R$ denote the resource with the minimum number of units among all resources in G, and let k be the number of units of r_i.

To form a CHCN, each network in G must possess at least one unit of each resource type. Therefore, for all resources other than r_i, there are k or more units available in G, since the maximum number of CHCNs that can be formed is bounded by k.

Now, suppose for the sake of contradiction that $q > k$, where q is the maximum number of CHCNs that can be composed from G. This implies that there are q CHCNs formed from G. However, since there are k units of the resource r_i in G and $q > k$, there must be at least one CHCN among the q CHCNs that does not possess the resource r_i, which contradicts the requirement for a CHCN to have at least one unit of each resource type.

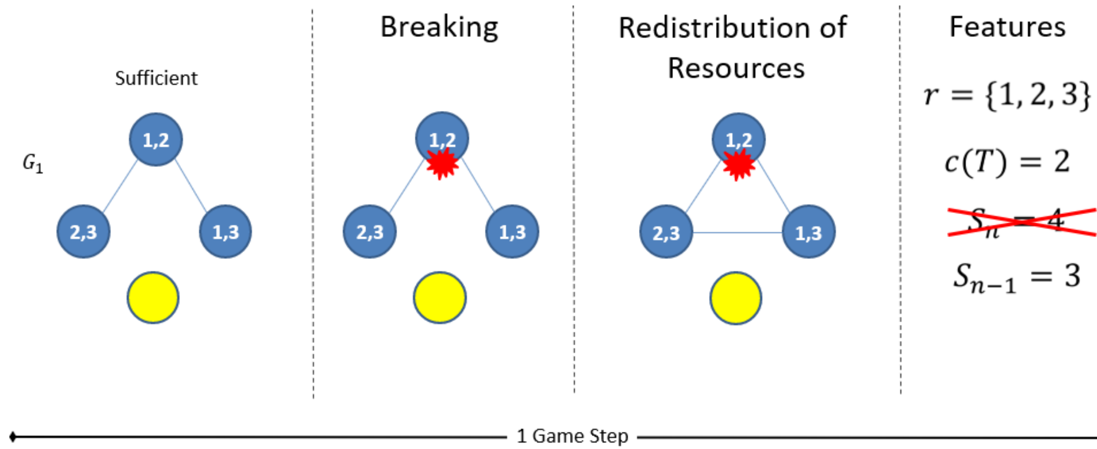


Figure 4.3: This example shows network redistribution in G1, a Sufficient network with four agents. Node 1,2 breaks down, but the network remains a CHCN with $c(T) = 2$ and all resources intact. The redistribution involves adjusting the connectivity by adding and removing edges. The transition from Sn = 4 to Sn = 3 ensures network stability.

Therefore, the assumption that $q > k$ leads to a contradiction. Thus, it must be true that $q \leq k$.

Hence, we have proved that if the maximum number of CHCNs that can be composed from G is bounded by k, then $q \leq k$.

□

A CHCN could potentially be reduced to a semi-CHCN by considering only a subset of the available resources (r) as a criterion, without significantly increasing the complexity. However, exploring this approach is beyond the scope of our work.

4.2.2 Fault Tolerance

We define failure in our fault tolerance domain as encompassing various types of issues, including damaged resources, communication interruptions, and locomotion problems. We categorize these types of failures as follows:

- If one or more of the agent's resources break or malfunction, where $n \geq 1$,
- If a keep-alive message is not being transmitted between a pair of agents;
- If an agent experiences any type of locomotion disruption.

If any of the aforementioned failures occur, we consider the agent to be broken and replace it if any of the CHCN criteria for PEG (discussed in Subsection 4.2.1) remain unfulfilled.

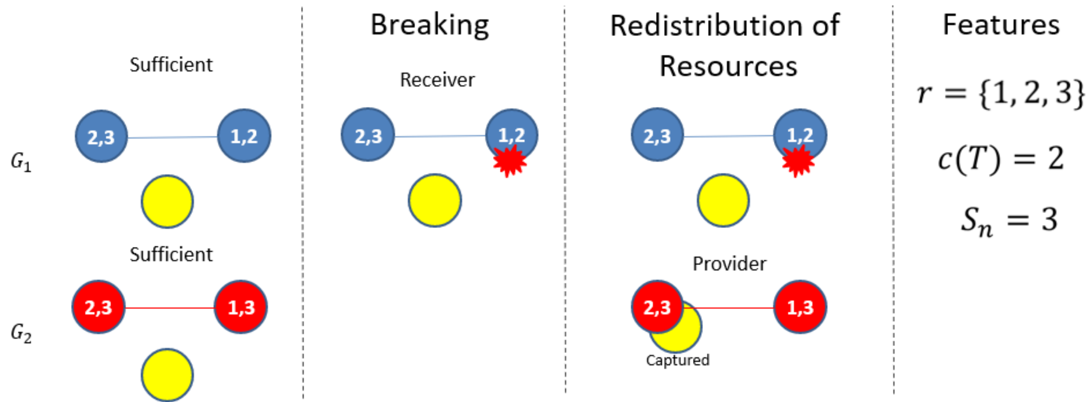
In the context of PEG, a PEG solution is calculated based on the topological map and the number of agents, which may vary when either the topological map or the number of agents changes. To handle networks with different numbers of agents playing PEG simultaneously, it is necessary to pre-calculate all the required PEG solutions. Let $c(T)$ represent the minimum number of pursuers required to capture on topology T and let the function $max(G)$ return the number of agents in the largest network. In the calculation of PEG solutions, we denote the minimum number of agents as $j = c(T)+1$, where one refers to the evader. Similarly, the maximum number of agents is denoted as $l = max(G)+1$, representing the number of pursuers in the largest network plus an evader. Consider a set S of PEG solutions for a given topology T, where $\forall s_k \in S$ with $j \leq k \leq l$, it is necessary to calculate a total of $|S|$ solutions.

After the evader assignment, the network $G_i \in G$ has n agents and plays PEG over the solution $s_n \in S$. If a pursuer in G_i has broken, $n \geq j$ and all resources/colors remain available for each pursuer's close neighborhood in the network, then in this case, the failure occurs but G_i remains as a CHCN. To address this issue, it is necessary to store the current state of G_i, including agent positions and the cost incurred up to that step. Additionally, a solution context switch is performed. The solution s_n for n players (agents) is switched to the solution s_{n-1} for $n-1$ players, and the state of G_i is restored to its previous state. The pursuers resume from the same positions they were in when the failure occurred. The stored cost is then aggregated with the ongoing cost as the PEG process continues.

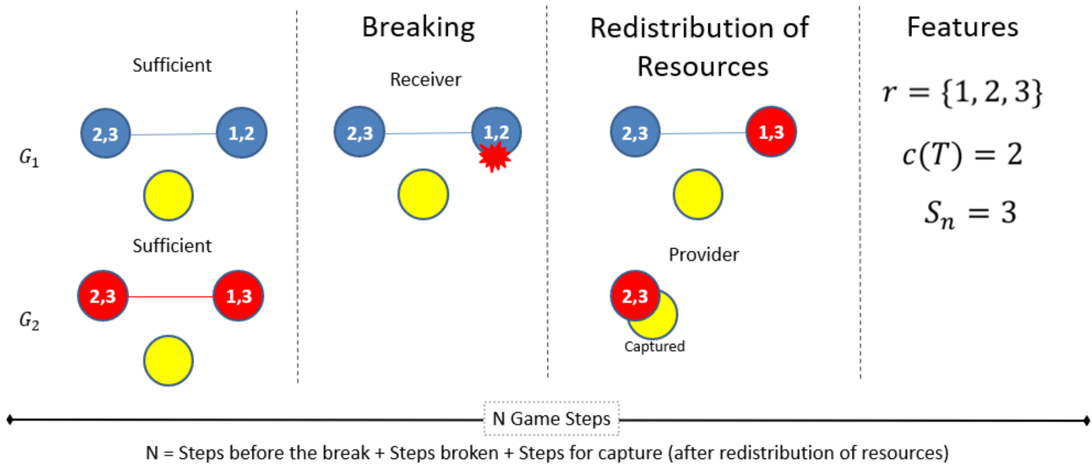
Figure 4.3 presents an example where the network G_1, classified as Sufficient, has four agents, three pursuers (blue circle), and an evader (the yellow circle). In Breaking the node labeled as $1,2$ is broken, but the Features $c(T) = 2$ and all r resources remain. The remaining nodes needed to be redistributed, in this case by adding and removing edges. The only change was the feature $S_n = 4$ to $S_n = 3$, once the network remains as a CHCN.

In Figure 4.4 is depicted the resource allocation after a failure happens. In the scenario depicted in 4.4b, both networks G_1 and G_2 initially start with the status of Sufficient. However, a failure occurs in G_1, causing it to transition to the Receiver status, while G_2 successfully captures the evader and becomes the Provider. This change in status reflects the broken agent in G_1 and the successful capture in G_2. After the Redistribution of Resources in Figure4.4b, G_1 is restored to its status as a CHCN. This restoration is achieved by transferring agent $1,3$ from G_2 to replace the broken agent $1,2$ in G_1, ensuring the provision of resource 1. As a result, G_1 attains the desired state of a CHCN, represented by the status $c(T) = 2$. Throughout this process, the total cost N is calculated by summing the costs associated with various factors, including the time leading up to the agent's failure, the waiting time for the resource, and the time required to capture the evader after restarting.

In simulations, the pursuits from starting positions until the capture can happen without failure or with the occurrence of failure. In the first case, the optimal strategy is running and the game is over when the last network ends. In the second case, a failure can happen either offline or online.



(a) G_1 breaks, G_2 capture the evader and the status of both changes to Receiver and Provider, respectively.



(b) $G1$ becomes a CHCN again, after the Redistribution of Resources, where the agent $1,3$ from G_2 replaced the broken agent $1,2$ in G_1.

Figure 4.4: Networks G_1 and G_2 start with the status of Sufficient. In Figure 4.4a, agent $1,2$ in G_1 is broken, causing G_1 to change its status to Receiver. Both features are not maintained: resource 1 is no longer available, and $C(T) = 1$. After capturing its evader, G_2 changes its status to Provider. In Figure 4.4b, during the Redistribution of Resources, agent $1,3$ from G_2 is transferred to G_1 to replace the broken agent and provide resource 1, resulting in $c(T) = 2$. The total cost N is the sum of the costs: the time before the agent breaks, the waiting time for the resource, and the time required to capture the evader after restarting.

Offline: after the redistribution of resources, there may be one or more networks that fail to become CHCNs or do not have the minimum required number of agents. In such cases, only the evader starts running, while the pursuers wait for a network to successfully complete the capture in order to transfer agents with the necessary resources.

Online: a CHCN that starts the simulation normally, and during execution, an agent with a single resource (in this CHCN) or an agent that just makes the number of agents less than the minimum (c(T)) fail, so the CHCN is not maintained. The pursuers remain in their respective positions at the failure moment, and the evader continues evading. When any network has a compatible agent and has been labeled a Provider, the transfer takes place immediately. If the network has been labeled Sufficient, then it will be necessary to wait for the capture of the

evader to perform the transfer.

The examples presented in Figure 4.3 and Figure 4.4 can in both online or offline cases.

4.3 Simulation, Evaluation, and Results for Resource Allocation Framework

In this section, we present the technologies for programming the simulator and experiments, as well as the computer that runs the simulations. We also present the simulations, evaluations, and results for resource allocation and multi-pursuer multi-evader PEG with fault tolerance.

4.3.1 Setup

In this section, we present the details of the implementation of the algorithm and simulators. We also describe the computer configuration that we used to run the experiments.

We develop a simulator to redistribute resources/colors among networks and to simulate PEG with fault tolerance in different topologies. The algorithm and simulator were encoded in Python 3 language [12]. Besides the native resources of Python language, we utilized the class *joblib.Parallel* of the Joblib package [8] to implement the parallel functions on PEG simulator.

The games were simulated on a computer with the following configuration: Processor AMD EPYC 7282 2.80 GHz 16 cores/32 threads, 128 GB RAM, and 240 GB SSD SATA.

4.3.2 Evaluation and Results

In this section, we present the evaluation and results carried out for some sets of heterogeneous agent networks to evaluate the distribution of resources to make them CHCNs, and then, these networks play PEG to evaluate the fault tolerance approach. The simulations are carried out for the topologies Building 1 (Figure 4.5) and Building 2 (Figure 4.6).

For the simulations with heterogeneous agents and their abilities or capabilities, consider

Table 4.1: List of resources/sensors.

Id	Resource/Sensor	Description
1	Wireless/4G/5G/LoRa	Communication
2	GPS/Beacon	Localization
3	Odometry/Accelerometer	Self-localization
4	Ultrasonic	Distance Measurement
5	Laser	Distance and sensing

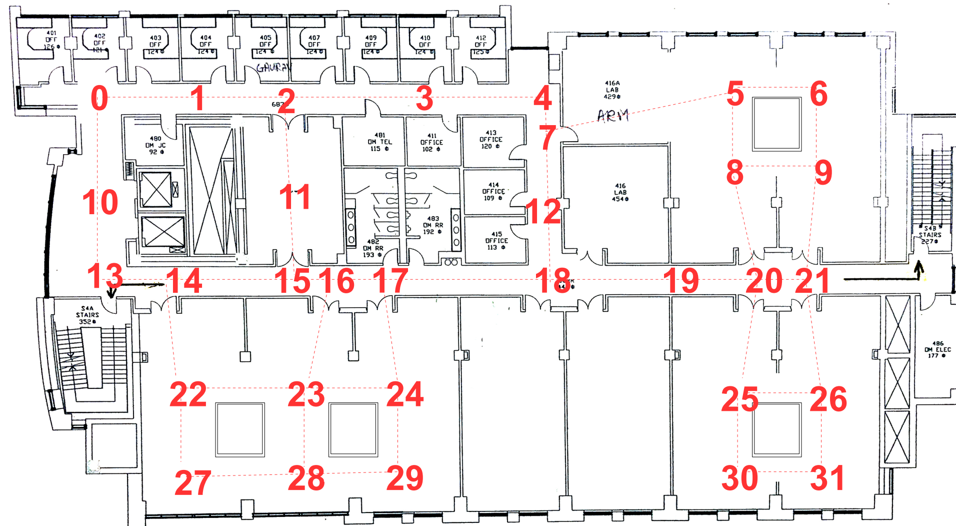


Figure 4.5: Building 1.

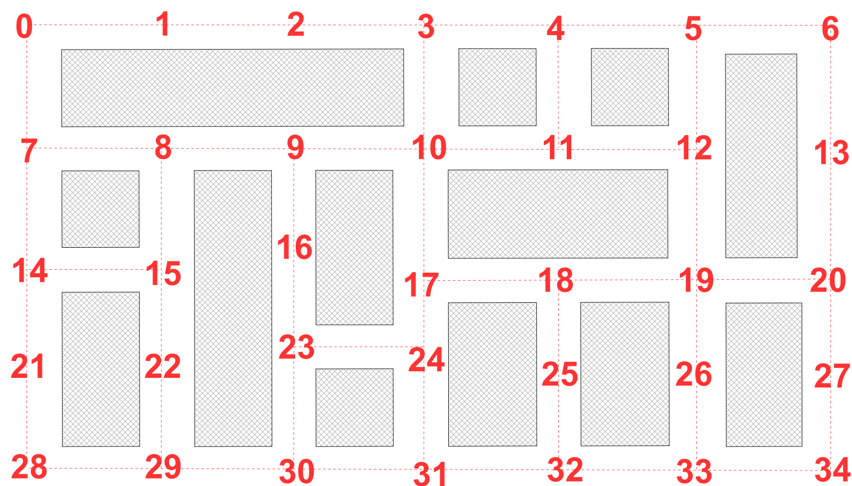


Figure 4.6: Building 2.

the Table 4.1 that depicts the resources and sensors that agents may have embedded. All agents have the same speed, one hop at a time. Agents that have communication resources communicate with other networks, to inform the current network status and which agent is broken, if this occurs. Resource two is used to inform the localization of this agent to the central controller, using Global Positioning System (GPS) for outdoor ambient or beacons to transmit location on the sensor network for indoor environments, for example. The localization of agents who

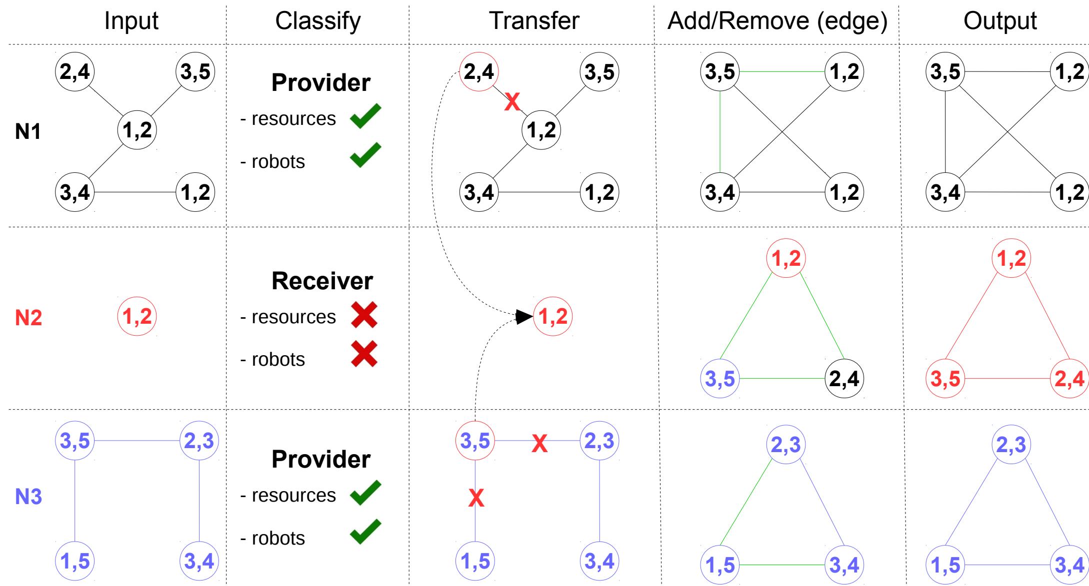


Figure 4.7: (Re)distribution of resources by Algorithm 5.

don't have resource two is relative to the agent who has. In this case, the agent has resources three, four, or five. Odometry sensor and accelerometer sensor to measure displacement. To the motion of the ambient, agents use resources four to avoid collisions or five to avoid collisions and measure distance.

In Figure 4.7 there are three networks namely $N1$, $N2$, and $N3$, modeled as graphs, where the circles denote agents, edges the links between agents and circle labels the number of resources embedded on agents. For this experiment, a network needs all resources in the Table 4.1 and at least two nodes, the minimum number for both topologies in the Figure 4.5 and Figure 4.6, to output CHCNs. Recall that all agents of the networks are pursuers in the PEG context. Following the stages on the Figure 4.7, we have the given networks (input column), which are the input to the Algorithm 5. Initially, the algorithm carried out the classification stage, where $N1$ and $N3$ have spare resources, and a number of agents greater than the minimum, being classified as Provider networks. $N2$ is classified as a Receiver network since it hasn't had all the necessary resources and it hasn't had the minimum number of agents. The next stage is to analyze and transfer the corresponding nodes from a provider to a receiver. In sequence, $N1$ transfers the node labeled as $2,4$, and $N3$ transfers the node labeled as $3,5$, both received by $N2$. Marked edges with red x represent their removal. At the add/remove stage, the added edges are green, the black edges remain for the previous stages. There was no need to remove any edges at this stage. Note that $N2$ received nodes, preserving the color of its source networks. The output is the CHCNs.

For PEG simulations an evader is associated to each output network (Figure 4.7). The simulations were performed on Building 1 (Figure 4.5) and Building 2 topologies (Figure 4.6). For both topologies we need to compute the PEG optimal strategy for 3, 4 and 5 agents (or players), where 3 is minimum number of pursuers and an evader, and 5 (four pursuers and an

evader) it is close to the PEG parallel algorithm computation limit (see Section 3.1). Although the PEG state explosion is a scalability limiter, the bottleneck here is the number of transitions [41], whose exponential growth curve is even greater (Table 4.2). The advantage is that once an instance of PEG is computed, it is not necessary to compute it again since the optimal solution from each state to the capture state has been computed.

Table 4.2: Topology characteristics.

Topology	Nodes	Players	States (S)	Transitions (T)	T/S ratio
Building 1	32	3	65536	515588	7.8
		4	2097152	48870424	23.3
		5	67108864	5211036548	77.6
Building 2	35	3	85750	798836	9.3
		4	3001250	91633892	30.53
		5	105043750	11821630076	112.5

Table 4.3: Fault Tolerance Evaluation for Building 1 Topology.

		Cost				
Network Id	Starting Positions	Without Failure	With Failure	Failure Step	Failure Node Label	Received Node
N1	31 26 31 31 24*	10	$11^1, 12^2, 17^3$	$5^1, 5^2, 10^3$	$(1,2)^1, (3,5)^2, (3,4)^3$	$-^1, N2(3,5)^2, N3(3,4)^3$
N2	13 27 28 25*	9	$13^2, 16^3$	$5^2, 9^3$	$(1,2)^2, (2,4)^3$	$N(1,2)^2, N3(3,4)^3$
N3	25 20 2 22*	7	$13^2, 16^3$	$5^2, 7^3$	$(3,4)^2, N2(2,4)^3$	$N1(3,4)^2, N1(3,4)^3$

Table 4.4: Fault Tolerance Evaluation for Building 2 Topology.

		Cost				
Network Id	Starting Positions	Without Failure	With Failure	Failure Step	Failure Node Label	Received Node
N1	5 6 5 9 24*	6	$8^1, 13^2, 14^3$	$5^1, 5^2, 6^3$	$(1,2)^1, (3,5)^2, (3,5)^3$	$-^1, N2(3,5)^2, N3(1,5)^3$
N2	27 25 16 21*	7	$12^2, 13^3$	$5^2, 7^3$	$(1,2)^2, (2,4)^3$	$N1(3,4)^2, N1(3,4)^3$
N3	16 16 22 32*	8	$11^2, 13^3$	$5^2, 8^3$	$(3,4)^2, (3,4)^3$	$N1(3,4)^2, N1(3,4)^3$

Table 4.2 presents the topology characteristics computed by PEG algorithm: the name of topology on the first column more left; the number of nodes on the right; the number of players for each PEG optimal strategy computation (3-5); in the column states, the number of states generated by the movement of the players ($2 * nodes^{players}$); the number of transitions in the penultimate column on the right; and, the ration between the number of transitions by the number of states. For Building 1 topology, with 5 players, the number of transitions is 77.6 times the number of states, while for the same number of players in topology Building 2, it is 112.5 times. The solutions for both topologies were performed on the same computer (see Subsection 4.3.1) and it took about 20 hours and 25 hours to calculate the solution set for Building 1 and Building 2, respectively.

Fault tolerance is evaluated initially to Building 1 topology (Figure 4.5). For simulations consider the three networks output by Algorithm 5 and respective evaders to play PEG. Table 4.3 presents the results of various simulations of PEG for networks $N1$, $N2$ and $N3$. Networks

that play PEG are identified in column *Network Id*; In the second column more left, the start positions of the first simulations for each network. Each number represents the position of the respective agent in the evaluated topology. The position marked with the star symbol represents the evader. The costs of the optimal PEG strategy are in column *Without Failure*. In the next four columns, superscript numbers represent the type of failure: 1- One agent broke, but the network remains with all necessary resources available for all agents and it has the minimum number of agents; 2-An agent broke and the network needs to receive resources from a provider network. 3-identical to case 2, but the network failed before starting the game. Cases 1 and 2 are considered *online* because they happen in execution/simulation time, and case 3 is *offline* because it occurs before starting the execution/simulation. The column *With Failure*, refers to the network simulation costs, to the failures superscript number. In column *Failure Step* is the game step in non crescent order, that the failure occurred. For example, network $N1$ failed at step 10 to case 3, this means that the network failed ten steps away from the goal, once the optimal solution without failure is 10 steps. The last two columns from left to right are the label of the broken agent and the received node to another network. The superscripts beside indicate the type of failure, that is used to link provider and receiver networks. In the last column, the network id provider accompanies the agent's label. In failure of case 1, $-^1$ denotes that there was no agent received, once in this case, there is only the change in the context of solution. Table 4.4 presents the results of the Building 2 topology. The meanings of notations and columns are identical to those explained above.

In case 1, when a failure occurs, such as in the case of $N1$, the total cost is increased by one at the step when the failure happens in Table 4.3. This cost can grow even more if the cost to capture from the current state (position of agents) is higher with the current solution, which has one less agent, the case of $N1$ in Table 4.4. In cases 2 and 3, the final cost can be the sum of steps from the start position to failure, the number of steps waiting to receive resources, and the steps to capture. However, the replacement agent needs to be in an equivalent position in the strategy, that is, the new state adding this new agent to the network has the same cost as the previous state. It occurs to $N1$ for both topologies. The new agent can be in a position that benefits or damages the final result, depending on its position. Case 2 for $N2$ and $N3$ in Table 4.4 are examples of gain due to the new agent position. When there is no concurrency to resources of other networks, the later the failure occurs, the less or no waiting time will be.

The main advantage of our approach is their scalability to significantly increase the number of networks and the ease of extending this model to include other features. The disadvantage is the limitation to expand the number of agents per network.

4.4 Conclusion

In this chapter, we described an algorithm to redistribute resources/agents among networks and turn them into completely heterogeneous cooperative networks. The networks were associated with the evaders to play the PEG for the same topology. Our approach is fault tolerant to the cases when a resource/agent breakdowns, reallocating the compatible resource as soon as it is available. In all cases prevail the optimal trajectory for capturing the evaders. Furthermore, our proposal is scalable for various networks playing the game simultaneously. We presented the simulation, evaluation, and results of the experiments for a pair of topologies. These experiments validate the algorithm and approach for fault tolerance cases on PEG as described.

Chapter 5

Conclusion

In this final chapter, we conclude the thesis by summarizing the key findings and contributions and providing guidelines for future work in pursuit-evasion games and resource allocation for heterogeneous multi-agent systems.

5.1 Conclusion

In this thesis, we have addressed the challenges associated with team-based task performance and resource allocation in pursuit-evasion games (PEGs) and heterogeneous multi-agent systems. Through an extensive literature review, we explored the concepts and research surrounding PEGs, highlighting their relevance and applications in various domains such as surveillance, search and rescue, and network deployment. We also delved into resource allocation for heterogeneous multi-agent teams, emphasizing the importance of effective resource management to optimize task accomplishment and agent replacement.

In Chapter 3, we presented the Pursuit-Evasion Game framework and its simulation, evaluation, and results. Our focus was on discrete PEGs played on graph-based environments, with the objective of capturing the evaders. We introduced a parallel algorithm that significantly increased scalability, enabling the analysis of games with a large number of states and transitions. The performance comparison with the previous serial approach showcased the efficiency of our parallel algorithm. Additionally, the extension of the algorithm to support multi-speed players and the incorporation of a pac-dot strategy demonstrated enhanced evader survival and the ability to fulfill objectives. Through extensive simulations, we validated the effectiveness and efficiency of our approach, achieving promising results in terms of computational performance and optimal strategy computation.

Chapter 4 centered around resource allocation for heterogeneous multi-agent teams. We developed a resource allocation algorithm that enabled the construction of teams composed of heterogeneous agents while ensuring resource availability through sharing with neighboring agents. Furthermore, we incorporated real-time agent replacement to address failures, allocating

compatible extra or idle agents from other teams to maintain task resolution. The simulation, evaluation, and results demonstrated the scalability and effectiveness of our resource allocation technique. The algorithm showcased the ability to efficiently allocate resources to multi-teams and seamlessly replace failed agents, thereby improving task performance and fault tolerance.

The findings from this research have practical implications for various domains where team-based task accomplishment and resource optimization are crucial. The insights gained from studying PEGs, particularly through the analysis of the Pac-Man game, provide valuable knowledge for addressing pursuit-evasion problems in real-world scenarios. Furthermore, the resource allocation approach presented in this thesis offers a viable solution for managing resources in heterogeneous multi-agent systems, enabling improved task performance and fault tolerance.

In conclusion, this thesis has made significant contributions to the fields of pursuit-evasion games and resource allocation for heterogeneous multi-agent teams. The proposed algorithms and techniques have demonstrated their effectiveness and scalability through extensive simulations and evaluations. The parallel algorithm for PEGs has showcased improved scalability and computational efficiency, while the resource allocation algorithm has facilitated the formation of multi-teams with shared resources, ensuring efficient task performance and seamless agent replacement. The results obtained in our simulations provide promising evidence of the practical applicability and effectiveness of our approaches. It is important to highlight that the published works [10, 11][1] are integral research results derived from this thesis.

We believe that the outcomes of this research will contribute to the advancement of team-based task performance and resource optimization, opening up new avenues for future research and applications in diverse domains. The findings and insights gained from this thesis lay a solid foundation for further exploration of pursuit-evasion games and resource allocation in multi-agent systems, ultimately leading to advancements in team coordination, task accomplishment, and resource management.

5.2 Future Work

While this thesis has made significant contributions to the fields of pursuit-evasion games and resource allocation for heterogeneous multi-agent teams, there are several avenues for future research and development that can further enhance and expand upon the presented work.

1. Advanced Pursuit-Evasion Game Strategies Future research can focus on the de-

[1] Articles from thesis research: [10] in 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), and [11] in Robotics and Autonomous Systems Journal.

velopment of advanced pursuit-evasion game strategies that incorporate more complex environments, dynamic obstacles, and multiple pursuers and evaders. Exploring the integration of machine learning techniques, such as reinforcement learning, could lead to the development of intelligent agents capable of adapting their strategies and behaviors in real-time based on environmental changes and game dynamics. Investigating the impact of uncertainty and imperfect information in pursuit-evasion games is another promising direction for future work.

2. Scalability and Optimization Scalability remains a significant challenge in pursuit-evasion games, particularly as the size of the environment and the number of agents increase. Future research can explore techniques to further enhance the scalability of the parallel algorithm presented in this thesis, enabling its application to even larger and more complex game scenarios. Additionally, the development of optimization algorithms and heuristics that can efficiently solve pursuit-evasion games in real-time while maintaining near-optimal performance is an area of interest.

3. Multi-Agent Task Allocation and Coordination In the context of heterogeneous multi-agent systems, there is scope for further research on advanced task allocation and coordination mechanisms. Investigating more sophisticated algorithms that consider agent capabilities, task requirements, and environmental constraints can lead to improved task assignment strategies. Furthermore, exploring decentralized coordination approaches, such as consensus algorithms or market-based mechanisms, can enhance the coordination efficiency and fault tolerance of multi-agent teams.

4. Real-World Applications and Validation Validating the proposed algorithms and techniques in real-world scenarios and practical applications is an important direction for future work. Conducting experiments and simulations in diverse domains, such as robotics, surveillance systems, and logistics, can provide valuable insights into the applicability and performance of the presented approaches. Additionally, conducting user studies and evaluations involving human participants can help assess the effectiveness and usability of the developed systems in real-world settings.

5. Extends the Resource Allocation Algorithm to Support Partial CHCNs Extending the resource allocation algorithm to support partial CHCNs, where different subsets of resources are assigned to different networks or agents based on their specific requirements. This would enhance resource utilization and optimize performance by allocating only the necessary resources. By accommodating partial CHCNs, the algorithm becomes more flexible and adaptable, paving the way for more efficient resource allocation in heterogeneous multi-agent systems.

By exploring these avenues for future work, researchers can further advance the fields of pursuit-evasion games and resource allocation for heterogeneous multi-agent systems, leading to improved strategies, more efficient algorithms, and practical applications in a wide range of domains.

Bibliography

[1] Waseem Abbas and Magnus Egerstedt. Characterizing heterogeneity in cooperative networks from a resource distribution view-point. *Commun. Inf. Syst.*, 14(1):1–22, 2014.

[2] Micah Adler, Harald Räcke, Naveen Sivadasan, Christian Sohler, and Berthold Vöcking. Randomized pursuit-evasion in graphs. *Combinatorics, Probability and Computing*, 12(3):225–244, 2003.

[3] Alessandro Berarducci and Benedetto Intrigila. On the cop number of a graph. *Advances in Applied Mathematics*, 14(4):389–403, 1993.

[4] Josh Bertram and Peng Wei. *An Efficient Algorithm for Multiple-Pursuer-Multiple-Evader Pursuit/Evasion Game*. American Institute of Aeronautics and Astronautics, 2021.

[5] Shaunak D Bopardikar, Francesco Bullo, and Joao P Hespanha. On discrete-time pursuit-evasion games with sensing limitations. *IEEE Transactions on Robotics*, 24(6):1429–1439, 2008.

[6] David Chapman. Planning for conjunctive goals. *Artificial intelligence*, 32(3):333–377, 1987.

[7] Timothy H Chung, Geoffrey A Hollinger, and Volkan Isler. Search and pursuit-evasion in mobile robotics. *Autonomous robots*, 31(4):299, 2011.

[8] Joblib Developers. Joblib running python functions as pipeline jobs, 2008.

[9] Keith J. Devlin. *Fundamentals of contemporary set theory.* Springer, Berlin, 1979.

[10] R. F. dos Santos, R. K. Ramachandran, M. A. M. Vieira, and G. S. Sukhatme. Pac-man is overkill. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11652–11657, 2020.

[11] Renato F. dos Santos, Ragesh K. Ramachandran, Marcos A.M. Vieira, and Gaurav S. Sukhatme. Parallel multi-speed pursuit-evasion game algorithms. *Robotics and Autonomous Systems*, 163:104382, 2023.

[12] Python Software Foundation. Python programming language, 2019.

[13] Eloy Garcia and Shaunak D. Bopardikar. Cooperative containment of a high-speed evader. In *2021 American Control Conference (ACC)*, pages 4698–4703, 2021.

[14] B.P. Gerkey and M.J. Mataric. Sold!: auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, 2002.

[15] C. Ghedini, C. Ribeiro, and Lorenzo Sabattini. A decentralized control strategy for resilient connectivity maintenance in multi-robot systems subject to failures. In *DARS*, 2016.

[16] C. Ghedini, C. Secchi, C. Ribeiro, and Lorenzo Sabattini. Improving robustness in multi-robot networks. *IFAC-PapersOnLine*, 48:63–68, 2015.

[17] Cinara Ghedini, Carlos Ribeiro, and Lorenzo Sabattini. Toward fault-tolerant multi-robot networks. *Netw.*, 70(4):388–400, December 2017.

[18] Arthur S. Goldstein and Edward M. Reingold. The complexity of pursuit on a graph. *Theoretical Computer Science*, 143(1):93 – 112, 1995.

[19] Yue Guan, Dipankar Maity, Christopher M. Kroninger, and Panagiotis Tsiotras. Bounded-rational pursuit-evasion games. In *2021 American Control Conference (ACC)*, pages 3216–3221, 2021.

[20] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[21] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[22] Karel Horák and Branislav Bošanský. Dynamic programming for one-sided partially observable pursuit-evasion games, 2016.

[23] H. Huang, W. Zhang, J. Ding, D. M. Stipanović, and C. J. Tomlin. Guaranteed decentralized pursuit-evasion in the plane with multiple pursuers. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 4835–4840, 2011.

[24] R. Isaacs. *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. Dover books on mathematics. Dover Publications, 1999.

[25] Olli Jansson, Matt Harris, and David Geller. A parallelizable reachable set method for pursuit-evasion games using interior-point methods. In *2021 IEEE Aerospace Conference (50100)*, pages 1–9. IEEE, 2021.

[26] Krishna Kalyanam and Meir Pachter. Pursuit of a moving ground target on a graph using partial information. 09 2015.

[27] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[28] K. Krishnamoorthy, D. Casbeer, and M. Pachter. Pursuit of a moving target with known constant speed on a directed acyclic graph under partial information. *SIAM J. Control. Optim.*, 54:2259–2273, 2016.

[29] K. Krishnamoorthy, S. Darbha, P. P. Khargonekar, D. Casbeer, P. Chandler, and M. Pachter. Optimal minimax pursuit evasion on a manhattan grid. In *2013 American Control Conference*, pages 3421–3428, 2013.

[30] Venkata Ramana Makkapati and Panagiotis Tsiotras. Optimal evading strategies and task allocation in multi-player pursuit–evasion problems. *Dynamic Games and Applications*, Jul 2019.

[31] Venkata Ramana Makkapati and Panagiotis Tsiotras. Optimal evading strategies and task allocation in multi-player pursuit–evasion problems. *Dynamic Games and Applications*, 9(4):1168–1187, 2019.

[32] Venkata Ramana Makkapati and Panagiotis Tsiotras. Apollonius allocation algorithm for heterogeneous pursuers to capture multiple evaders. *CoRR*, abs/2006.10253, 2020.

[33] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 01 2011.

[34] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA ’02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[35] S. Neufeld and R. Nowakowski. A game of cops and robbers played on products of graphs. *Discrete Mathematics*, 186(1):253–268, 1998.

[36] G. Notomista, S. Mayya, S. Hutchinson, and M. Egerstedt. An optimal task allocation strategy for heterogeneous multi-robot systems. In *2019 18th European Control Conference (ECC)*, pages 2071–2076, 2019.

[37] Richard Nowakowski and Peter Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Math.*, 43(2–3):235–239, jan 1983.

[38] Meir Pachter. Isaacs’ Two-on-One Pursuit-Evasion Game. In David M Ramsey and Jérôme Renault, editors, *Advances in Dynamic Games*, pages 27–57, Cham, 2020. Springer International Publishing.

[39] Selina Pan, Haomiao Huang, Jerry Ding, Wei Zhang, Claire J Tomlin, et al. Pursuit, evasion and defense in the plane. In *2012 American Control Conference (ACC)*, pages 4167–4173. IEEE, 2012.

[40] Torrence D Parsons. Pursuit-evasion in a graph. In *Theory and applications of graphs*, pages 426–441. Springer, 1978.

[41] U. Pferschy. Solution methods and computational investigations for the linear bottleneck assignment problem. *Computing*, 59(3):237–258, November 1997.

[42] R. K. Ramachandran, J. A. Preiss, and G. S. Sukhatme. Resilience by reconfiguration: Exploiting heterogeneity in robot teams. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6518–6525, Nov 2019.

[43] Ragesh K. Ramachandran, Pietro Pierpaoli, Magnus Egerstedt, and Gaurav Sukhatme. Resilient monitoring in heterogeneous multi-robot systems through network reconfiguration. *IEEE Transactions on Robotics*, 2020. Submitted to.

[44] Tiago Rodrigues, Miguel Duarte, Sancho Oliveira, and Anders Christensen. Beyond onboard sensors in robotic swarms: Local collective sensing through situated communication. volume 2, 01 2015.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[46] Mehdi Salimi, Gafurjan I Ibragimov, Stefan Siegmund, and Somayeh Sharifi. On a fixed duration pursuit differential game with geometric and integral constraints. *Dynamic Games and Applications*, 6(3):409–425, 2016.

[47] Pedro M. Shiroma and Mario F. M. Campos. Comutar: A framework for multi-robot coordination and task allocation. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4817–4824, 2009.

[48] Daigo Shishika and Vijay Kumar. Local-game decomposition for multiplayer perimeter-defense problem. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 2093–2100, 2018.

[49] Daigo Shishika and Vijay Kumar. A Review of Multi Agent Perimeter Defense Games. In Quanyan Zhu, John S Baras, Radha Poovendran, and Juntao Chen, editors, *Decision and Game Theory for Security*, pages 472–485, Cham, 2020. Springer International Publishing.

[50] Daigo Shishika, James Paulos, Michael R. Dorothy, M. Ani Hsieh, and Vijay Kumar. Team composition for perimeter defense with patrollers and defenders. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 7325–7332, 2019.

[51] Daigo Shishika, James Paulos, and Vijay Kumar. Cooperative team strategies for multiplayer perimeter-defense games. *IEEE Robotics and Automation Letters*, 5(2):2738–2745, 2020.

[52] Hao Tang, Weidong Zhang, Min Sun, Bin Lin, and Zhihuan Hu. A pe game with one superior hunter and multi-pursuer against an evader. In *2021 40th Chinese Control Conference (CCC)*, pages 5124–5130, 2021.

[53] Onur Tekdas, Wei Yang, and Volkan Isler. Robotic routers: Algorithms and implementation. *The International Journal of Robotics Research*, 29(1):110–126, 2010.

[54] Pac-man. `https://www.thoughtco.com/pac-man-game-1779412`, July 2019.

[55] V. Varadharajan, B. Adams, and G. Beltrame. Failure-tolerant connectivity maintenance for robot swarms. *ArXiv*, abs/1905.04771, 2019.

[56] Marcos A. M. Vieira, Ramesh Govindan, and Gaurav S. Sukhatme. Scalable and practical pursuit-evasion with networked robots. *Intelligent Service Robotics*, 2(4):247, Sep 2009.

[57] Marcos A.M. Vieira, Ramesh Govindan, and Gaurav S. Sukhatme. An autonomous wireless networked robotics system for backbone deployment in highly-obstructed environments. *Ad Hoc Networks*, 11(7):1963–1974, 2013.

[58] Bogdan Vlahov, Eric Squires, Laura Strickland, and Charles Pippin. On developing a uav pursuit-evasion policy using reinforcement learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 859–864, 2018.

[59] Maolin Wang, Lixin Wang, and Ting Yue. An application of continuous deep reinforcement learning approach to pursuit-evasion differential game. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 1150–1156, 2019.

[60] Isaac E. Weintraub, Meir Pachter, and Eloy Garcia. An introduction to pursuit-evasion differential games. In *2020 American Control Conference (ACC)*, pages 1049–1066, 2020.

[61] Hiroaki Yamaguchi. A Cooperative Hunting Behavior by Mobile-Robot Troops. *The International Journal of Robotics Research*, 18(9):931–940, 1999.

[62] Fuhan Yan and Yichuan Jiang. Pursuing a faster evader based on an agent team with unstable speeds. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, page 1766–1768, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[63] Sha Yi, Changjoo Nam, and Katia Sycara. Indoor Pursuit-Evasion with Hybrid Hierarchical Partially Observable Markov Decision Processes for Multi-robot Systems. In Nikolaus Correll, Mac Schwager, and Michael Otte, editors, *Distributed Autonomous Robotic Systems*, pages 251–264, Cham, 2019. Springer International Publishing.

[64] Sha Yi, Changjoo Nam, and Katia Sycara. Indoor Pursuit-Evasion with Hybrid Hierarchical Partially Observable Markov Decision Processes for Multi-robot Systems. In Nikolaus Correll, Mac Schwager, and Michael Otte, editors, *Distributed Autonomous Robotic Systems*, pages 251–264, Cham, 2019. Springer International Publishing.

[65] Zhongshun Zhang, Joseph Lee, Jonathon M. Smereka, Yoonchang Sung, Lifeng Zhou, and Pratap Tokekar. Tree search techniques for minimizing detectability and maximizing visibility. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8791–8797, 2019.

[66] Zhongshun Zhang, Yoonchang Sung, Lifeng Zhou, Jonathon M. Smereka, Joseph Lee, and Pratap Tokekar. Tree search techniques for minimizing detectability and maximizing visibility, 2019.

[67] Zhengyuan Zhou, Wei Zhang, Jerry Ding, Haomiao Huang, Dušan M Stipanović, and Claire J Tomlin. Cooperative pursuit with Voronoi partitions. *Automatica*, 72:64–72, 2016.