

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Rafael Studart Mattos Di Piero

**MULTI-AGENT GRAPH EXPLORATION WITHOUT
COMMUNICATION**

Bachelor's Thesis
2024

Course of Computer Engineering

Rafael Studart Mattos Di Piero

**MULTI-AGENT GRAPH EXPLORATION WITHOUT
COMMUNICATION**

Advisor

Prof. Dr. Luiz Gustavo Bizarro Mirisola (ITA)

Co-advisor

Prof. Dr. Vitor Venceslau Curtis (ITA)

COMPUTER ENGINEERING

SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Cataloging-in Publication Data
Documentation and Information Division

Studart Mattos Di Piero, Rafael
Multi-Agent Graph Exploration Without Communication / Rafael Studart Mattos Di Piero.
São José dos Campos, 2024.
42f.

Bachelor's Thesis – Course of Computer Engineering– Instituto Tecnológico de Aeronáutica,
2024. Advisor: Prof. Dr. Luiz Gustavo Bizarro Mirisola. Co-advisor: Prof. Dr. Vitor Venceslau
Curtis.

1. Graph. 2. Search. 3. Multi-agent. 4. Mixed Radix. I. Instituto Tecnológico de Aeronáutica.
II. Title.

BIBLIOGRAPHIC REFERENCE

STUDART MATTOS DI PIERO, Rafael. **Multi-Agent Graph Exploration Without Communication**. 2024. 42f. Bachelor's Thesis – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Rafael Studart Mattos Di Piero

PUBLICATION TITLE: Multi-Agent Graph Exploration Without Communication.

PUBLICATION KIND/YEAR: Bachelor's Thesis (Undergraduation study) / 2024

It is granted to Instituto Tecnológico de Aeronáutica permission to reproduce copies of this final paper and to only loan or to sell copies for academic and scientific purposes. The author reserves other publication rights and no part of this final paper can be reproduced without the authorization of the author.

Rafael Studart Mattos Di Piero
Rua do H8A, Ap. 125
12.228-460 – São José dos Campos–SP

MULTI-AGENT GRAPH EXPLORATION WITHOUT COMMUNICATION

Bachelor's Thesis approved in its final version by the signatories below:

Rafael Studart Mattos Di Piero
Author

Luiz Gustavo Bizarro Mirisola (ITA)
Advisor

Vitor Venceslau Curtis (ITA)
Co-advisor

Prof. Dr. Marcos Ricardo Omena de Albuquerque Máximo
Course Coordinator of Computer Engineering

São José dos Campos: Nov 20, 2024.

To my family, for their love and constant support. To my friends, for standing by me through all moments.

Acknowledgments

I want to express my gratitude to:

My family, for their unwavering support and love throughout my studies.

My friends from H8 and Fortaleza, for their constant encouragement and positive energy.

Prof. Dr. Luiz Gustavo Bizarro Mirisola for his guidance and patience as my advisor.

"The journey of a thousand miles begins with one step."

— LAO TZU

Resumo

A teoria dos grafos, um campo fundamental da matemática e da ciência da computação, oferece estruturas robustas para modelar relacionamentos e percorrer estruturas de grafos. Esta pesquisa se aprofunda na exploração de grafos, cruciais para aplicações como roteamento de redes, robótica e geração procedural. Como as aplicações no mundo real frequentemente envolvem ambientes complexos com restrições de tempo, este estudo foca em sistemas multiagentes para a exploração de grafos, onde múltiplos agentes autônomos colaboram para otimizar a distribuição de tarefas.

O objetivo deste estudo é propor um algoritmo eficiente de exploração de grafos por múltiplos agentes sem comunicação, uma necessidade crucial em cenários onde a comunicação é impraticável ou impossível, como exploração em águas profundas ou ambientes com restrição de energia. Com base no método para exploração de labirintos perfeitos detalhado por Rodrigues (2024), este estudo estende a abordagem para grafos mais gerais, que podem incluir ciclos. Além disso, os algoritmos foram reimplementados utilizando uma biblioteca geral de grafos e não uma biblioteca específica para labirintos perfeitos (NAEEM, 2021). Ao estruturar a implementação em classes bem definidas e extensíveis, a pesquisa oferece uma estrutura versátil para aplicações mais amplas em sistemas multiagentes, abrindo caminho para novos avanços na exploração autônoma de grafos.

Abstract

Graph theory, a pivotal field in mathematics and computer science, offers robust frameworks for modeling relationships and traversing graph structures. This research delves into graph exploration, crucial for applications like network routing, robotics, and procedural generation. As real-world applications often involve complex environments with time constraints, this study focuses on multi-agent systems for graph exploration, where multiple autonomous agents collaborate to optimize task distribution.

This study's objective is to propose an efficient multi-agent algorithm for graph exploration without communication, a crucial need in scenarios where communication is impractical or impossible, such as deep-sea exploration or energy-constrained environments. Building on the method for perfect maze exploration detailed by Rodrigues (2024), this study extends the approach to general graphs, that may include cycles. Several algorithmic variations, combining the proposed approach and Tarry's variation (KIVELEVITCH; COHEN, 2010), were implemented in an effort to compare how different algorithms perform in various scenarios and to identify efficient use cases. Furthermore, the algorithms were reimplemented using a generic graph library instead of a specialized perfect maze library (NAEEM, 2021).

By structuring the exploration into well-defined and extensible classes, the research offers a versatile framework for broader applications in multi-agent systems. This framework was used to evaluate different datasets, containing graphs with varying properties, paving the way for further advancements in autonomous graph exploration.

List of Figures

FIGURE 2.1 – Simplified Class Diagram of the Multi-Agent Maze Exploration Model	22
FIGURE 2.2 – Three agents (Red, Blue, and Yellow) disperse from each other in a graph. Source: Rodrigues (2024)	26
FIGURE 2.3 – Three agents (Red, Blue, and Yellow) disperse in a non-uniform graph.	27
FIGURE 2.4 – Exploration of an imperfect maze(graph) by three agents.	36
FIGURE 2.5 – Trees constructed by agent post-exploration: Agent 1 .	37
FIGURE 2.6 – Trees constructed by agent post-exploration: Agent 2 .	38
FIGURE 2.7 – Trees constructed by agent post-exploration: Agent 3 .	39

List of Algorithms

1	Simulation - simulate()	23
2	Agent - move_one_step()	25
3	Agent - define_agent_next_step()	29
3	Agent - define_agent_next_step()	30
4	Agent - get_neighbors_intervals()	32
5	Agent - define_next_first_phase_tarry_step()	34
6	Agent - define_next_second_phase_tarry_step()	34

List of Abbreviations and Acronyms

DFS Depth First Search

BFS Breath First Search

LCL Last Common Location

List of Symbols

G Graph

v_n Vertex

v_nv_m Edge

Contents

1	INTRODUCTION	15
1.1	Motivation	15
1.2	Objective	16
1.3	Definitions	16
1.3.1	Graph	17
1.3.2	Agent	18
1.3.3	Mixed Radix	18
2	METHODOLOGY	21
2.1	Modeling	21
2.1.1	Simulation	21
2.1.2	Graph	22
2.1.3	Agent	24
2.2	Graph Exploration Algorithm	26
2.3	Mixed Radix Path Representation	31
2.4	Alternative Exploration Algorithms	32
2.4.1	Exploration Algorithm without Communication	33
2.4.2	Exploration Strategies with Communication: Tarry Algorithm Variations	33
2.5	Graph Visualization	35
2.6	Agent Tree Visualization	36
3	RESULTS AND DISCUSSION	40
3.1	Perfect Maze Results	40

3.1.1	Metrics for Analysis	40
3.1.2	No-Communication Algorithms Performance	40
3.1.3	Tarry's Variants Performance	40
3.2	Random Tree Results	40
3.2.1	Metrics for Analysis	40
3.2.2	No-Communication Algorithms Performance	40
3.2.3	Tarry's Variants Performance	40
4	CONCLUSIONS AND FUTURE WORKS	41
	BIBLIOGRAPHY	42

1 Introduction

1.1 Motivation

Graph theory is a fundamental field of mathematics with significant relevance in computer science due to its ability to model relationships between objects. One of the key challenges in computer science is graph traversal and exploration, which involves systematically navigating through the nodes of a graph with a specific purpose. This topic has practical applications across diverse fields, such as network routing, robotics, procedural generation, electronics design, and more.

When considering the complexities and time constraints imposed by real-world environments, graph exploration must be extended to consider multi-agent systems. In such systems, multiple autonomous agents collaborate to explore graphs, aiming to distribute tasks to achieve optimal efficiency. However, communication strategies on these systems are challenging, as they need to balance the trade-off between the amount of information shared among agents versus the quality of the solution. In other words, while communication can allow agents to speed up their exploration, it may cost time and/or energy.

Various versions of this problem have been explored in research. One such approach, discussed by Kivelevitch and Cohen (2010), proposed a generalization of Tarjan's Algorithm with significant emphasis on minimizing data transfer. Nevertheless, this solution is limited to mazes and does not extend to general graphs.

Despite the previously mentioned research, the concept of zero-communication exploration has not been concretely established in the literature. We believe this gap is significant, as practical situations might involve scenarios where communication is limited or impossible due to bandwidth limitations or energy consumption, such as deep-sea exploration, search with energy-limited agents, or high-efficiency state space search.

This work aims to address this gap by presenting a continuation and expansion of a method initially developed for perfect maze exploration (NAEEM, 2021), as discussed by Rodrigues (2024). Building upon this groundwork, the research extends the method to accommodate diverse graph structures by removing dependencies on specific projects.

Additionally, it seeks to compare and establish a lower bound on the time complexity for graphs with varying properties.

Alongside this main objective, this research seeks to incorporate heuristics and alternative strategies proven effective in zero-communication scenarios to enhance Tarry’s algorithm. As outlined by Kivelevitch and Cohen (2010), Tarry’s algorithm relies on a random selection of neighbors for exploration, which may not always yield optimal results for different graph structures. Our goal was not only to analyze how these choices are made but also to investigate whether a secondary algorithm or heuristic could guide the selection process, thereby improving the overall efficiency of exploration.

1.2 Objective

This study aims to propose a viable algorithm for graph exploration by multi-agents, specifically for the search of a goal node. A previous paper by Rodrigues (2024) discussed an approach for perfect maze exploration, which can be modeled as a tree, a connected acyclic undirected graph. Expanding the proposed algorithm to connected cyclic graphs enables the definition of a lower bound for exploration algorithms. This provides a benchmark for evaluating and comparing exploration strategies, allowing researchers to quantify the impact of communication on exploration efficiency relative to a baseline no-communication algorithm. This comparison is crucial for assessing the cost of implementation and the value brought by communication in contrast to a simpler, zero-communication strategy.

Additionally, this research aims to improve the existing Tarry’s algorithm by leveraging insights gained from the development of the no-communication exploration strategy. The objective is to integrate these strategies as heuristics or enhancements to Tarry’s original approach, aiming to optimize performance while preserving the algorithm’s core logic.

1.3 Definitions

This section aims to define key concepts in graph theory and multi-agent exploration that are essential for understanding the context and methodology of our research, ensuring clarity and consistency in the subsequent discussions.

1.3.1 Graph

Graphs are structures in mathematics and computer science used to represent relationships between pairs of objects. According to Manber (1989), a graph $G = (V, E)$ is defined as a set V of vertices (or nodes) and a set E of edges, that connect pairs of vertices.

1.3.1.1 Common Nomenclature

In graph theory, the basic terminology includes the following components:

- Vertex (Node): The fundamental unit of a graph, representing an object or a point.
- Edge: A connection between a pair of vertices, representing a relationship between them.
- Adjacency: Two vertices v and w of a graph G are adjacent if there is an edge vw joining them, and the vertices v and w are then incident with such an edge. (WILSON, 1996)
- Degree: Degree of a vertex v of G is the number of edges incident with v . (WILSON, 1996)
- Path: A path in a graph G , as defined by Wilson (1996), is a finite sequence of edges of the form v_0v_1, v_1v_2, \dots in which any two consecutive edges are adjacent, all edges are distinct and all vertices v_0, v_1, \dots, v_m are distinct (except, possibly, $v_0 = v_m$).
- Cycle: A path where $v_0 = v_m$ and with at least one edge. (WILSON, 1996)

1.3.1.2 Key Properties

A few key properties of graphs that interest us are:

- Connectivity: A graph is connected if and only if there is a path between each pair of vertices. (WILSON, 1996)
- Acyclicity: A graph is acyclic if it has no cycles. A tree is an acyclic connected graph.

1.3.1.3 Traversal

Graph traversal is the process of visiting all the vertices in a graph in a systematic manner. Two fundamental algorithms for graph traversal are:

- Breadth-first search (BFS): Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. (MANBER, 1989)
- Depth-first search (DFS): Explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. (MANBER, 1989)
- Tarry's Algorithm: Tarry's Algorithm is used to traverse the entire maze, or graph, in a depth first fashion, where each edge is travelled twice, and the agent finishes the motion at the same initial vertex it started from. (KIVELEVITCH; COHEN, 2010)
This algorithm is described in the context of a physical maze, where the agent must physically move between adjacent nodes. It was first described over botanical mazes built as amusements for the XIX century aristocracy.

1.3.2 Agent

An agent in maze exploration, as defined by Kivelevitch and Cohen (2010), is an entity able to move within the maze in any direction as long as it does not encounter obstacles.

In the context of graph exploration, this concept extends to an agent's ability to traverse edges between vertices within the graph structure. Each agent acts independently, making decisions based on its individual memory and the local information available at its current vertex and its adjacent edges.

1.3.3 Mixed Radix

A mixed radix system is a positional numerical system in which the numerical base for each digit varies. This allows for more flexible representation of numbers, accommodating various scales within the same numerical framework.

One mathematical approach for representing mixed radix numbers can be found in Arndt (2011), which establishes a comprehensive arithmetical method for manipulating them. However, for the specific subset of real numbers within the interval $[0, 1]$, a more specialized approach is utilized, as described by Rodrigues (2024).

In this context, the mixed radix representation $A = [a_0, a_1, a_2, \dots, a_{n-1}]$ of a number x with respect to a radix vector $M = [m_0, m_1, m_2, \dots, m_{n-1}]$, where $x \in [0, 1]$, is given by:

$$x = \sum_{k=0}^{n-1} a_k \prod_{j=0}^k \frac{1}{m_j} \quad (1.1)$$

where a_j are non-negative integers, m_j are integers such that $m_j \geq 2$, and $0 \leq a_j \leq m_j$. (RODRIGUES, 2024)

For instance, if x is represented by $0.2_3 1_2 1_4 0_3 1_5$, and considering that a_i is on the left of a_{i+1} , x might be transformed to the decimal base by the following steps:

$$x = 2_3 1_2 1_4 0_3 1_5 \quad (1.2)$$

$$A = [2, 1, 1, 0, 1] \quad (1.3)$$

$$M = [3, 2, 4, 3, 5] \quad (1.4)$$

$$n = 5 \quad (1.5)$$

$$x = \sum_{k=0}^4 a_k \prod_{j=0}^k \frac{1}{m_j} \approx 0.8777778 \quad (1.6)$$

1.3.3.1 Extending Mixed Radix with Unary Digits

In our study, we extended the mixed radix system to include unary digits, which are used to represent steps in a sequence where no meaningful decisions are made. These unary digits do not contribute to the numerical value and can be omitted in the conversion to standard numerical forms.

Within our framework, the mixed radix representation $A = [a_0, a_1, a_2, \dots, a_{n-1}]$ of a number x with respect to a radix vector $M = [m_0, m_1, m_2, \dots, m_{n-1}]$, where $x \in [0, 1]$, is given by:

Let $U = \{k \mid m_k \neq 1\}$ be the set of indices where the base m_k is not unary.

$$x = \sum_{k \in U} u_k \prod_{j=0}^k \frac{1}{m_j} \quad (1.7)$$

where a_k are non-negative integers, m_j are integers such that $m_j \geq 1$, and $0 \leq a_j \leq m_j$.

Consider the mixed radix representation $x = 0.2_3 I_1 1_2 I_1 I_1 1_4 0_3 1_5$, where I_1 represents the unary digit and each a_i is on the left of a_{i+1} . We can transform x to the decimal base using the following steps:

$$x = 2_3 I_1 1_2 I_1 I_1 1_4 0_3 1_5 \quad (1.8)$$

$$A = [2, I, 1, I, I, 1, 0, 1] \quad (1.9)$$

$$M = [3, 1, 2, 1, 1, 4, 3, 5] \quad (1.10)$$

$$U = [0, 2, 5, 6, 7] \quad (1.11)$$

$$x = \sum_{k \in U} a_k \prod_{j=0}^k \frac{1}{m_j} \approx 0.8777778 \quad (1.12)$$

2 Methodology

This section intends to discuss specific design choices that support our multi-agent search algorithm. Section 2.1 introduces the models and classes developed to improve our algorithm. Section 2.2 describes the core concepts of our algorithm and provides pseudocode to guide its implementation. Section 2.3 presents a mixed radix numerical representation used for the agent’s path and examples to demonstrate its functionality. Section 2.4 presents alternative exploration strategies, including Backward Interval Filling and four Tarry algorithm variations to improve efficiency and coordination. Section 2.5 highlights the algorithm’s capability to handle complex graphs, as shown through an imperfect maze exploration. Finally, Section 2.6 visualizes individual exploration trees for each agent, demonstrating their paths and handling of graph structures.

2.1 Modeling

In this section, we detail the modeling approach chosen to structure our problem, emphasizing the use of classes to achieve greater flexibility in our solution. The main classes include **Simulation**, **Graph** and **Agent**, each of which will be defined in subsequent sections. Figure 2.1 presents a simplified class diagram that illustrates the overall structure of our model.

2.1.1 Simulation

The **Simulation** class acts as a central controller in our multi-agent graph exploration problem. It is primarily responsible for managing shared information, including the graph structure. Additionally, it orchestrates the agents and the overall execution of the exploration process.

In more detail, its responsibilities include:

- **Shared Information:** Storing and managing all information that is common between agents.

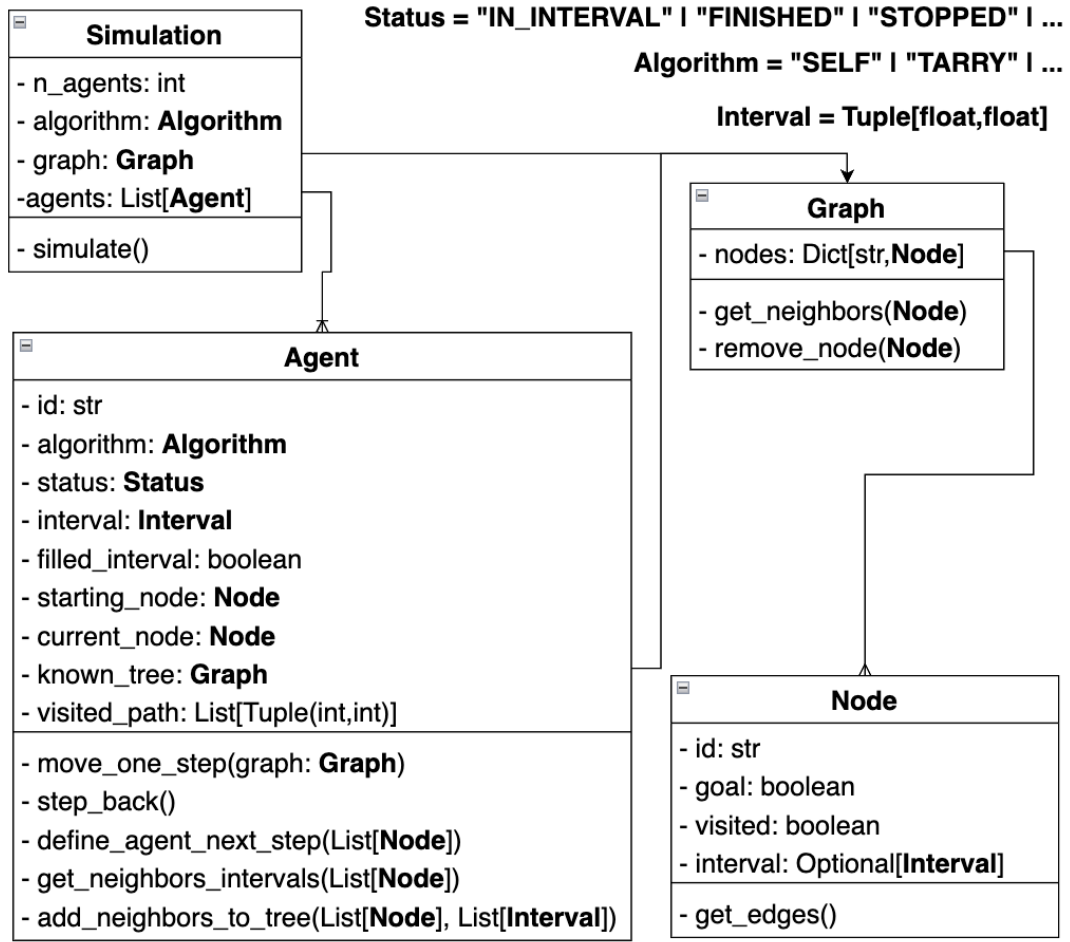


FIGURE 2.1 – Simplified Class Diagram of the Multi-Agent Maze Exploration Model

- **Orchestration:** Coordination agent activities and possible interactions during the exploration.
- **Visualization:** Providing tools for tracking the progress and visualizing the result of the exploration.
- **Metrics:** Collecting and analyzing performance metrics of the exploration.

As shown by Figure 2.1, the core method of the **Simulation** is *simulate* that is responsible for starting the exploration and managing each agent in parallel, until all agents have reached the goal or have no more moves available. The pseudocode in Algorithm 1 provides a breakdown of the function’s steps and logic.

2.1.2 Graph

The **Graph** class, as introduced in Section 1.3.1, serves as a basic representation of the graph structure for our multi-agent exploration. This class leverages existing libraries to

Algorithm 1 Simulation - simulate()

```

/* Graph to be explored */
graph ← Graph()
/* Agents */
agents ← [Agent(0), Agent(1), Agent(2), ...]

/* Agents that stopped already */
agents_stopped ← []
for all agent in agents do
    append(agents_stopped, FALSE)
end for

/* Exploration */
while False in agents_stopped do
    for all agent in agents do
        /* Moving each agent in parallel */
        agent.move_one_step()
        if agent.status == "FINISHED" or agent.status == "STOPPED" then
            agents_stopped[agent.id] ← TRUE
        end if
    end for
end while

/* Calculing Metrics */
for all agent in agents do
    /* Simplified function for calculating metrics */
    /* How each specific metric is calculated is not relevant for our study */
    calculate_metrics(agent)
end for

/* Simulation is complete */
/* Showing graphical result */
/* Simplified function for displaying */
display_demo_exploration(agents)

```

efficiently manage nodes and edges, facilitating the agents' traversal of the graph.

A key method within this class is *get_neighbors*. This method returns a list of neighboring nodes in a specific order, aiding in the consistent traversal of the graph.

2.1.3 Agent

The **Agent** class represents an agent, as defined in Section 1.3.2 and illustrated in Figure 2.1.

An **Agent** is uniquely identified by its *id* field and possesses the *status* and *algorithm* fields, which define its behavior during each step of the exploration process.

Additionally, an **Agent** records its path using its own graph, defined by the *known_graph* field, and a mixed radix representation of it in the *visited_path* field.

Furthermore, the **Agent** possesses the *current_node* field, which stores its current position in the graph, and the *interval* field, specific to our graph exploration algorithm, aiding in its traversal strategy.

The core method of the **Agent** class is *move_one_step*, which is responsible for determining the agent's next action during the exploration, as demonstrated in Algorithm 1. While the detailed logic of how the agent decides on its actions will be discussed in Section 2.2, we provide a pseudocode representation of this method in Algorithm 2.

Algorithm 2 Agent - move_one_step()

```

procedure MOVE_ONE_STEP(self, graph):
  /* Checking if the agent has already stopped or finished */
  if self.status == "FINISHED" or self.status == "STOPPED" then
    return
  end if

  /* Get current position */
  current_position ← graph[self.current_node]

  /* Taking new step */
  /* Neighbors are returned following a specific ordering */
  neighbors ← graph.get_neighbors(current_position)
  /* Get only non visited neighbors */
  non_visited_neighbors ← [ ]
  for all neighbor in neighbors do
    if self.known_tree[neighbor].visited == FALSE then
      append(non_visited_neighbors, neighbor)
    end if
  end for
  /* This decision will be explained in Section 2.2 */
  next_step ← self.define_agent_next_step(non_visited_neighbors)
  /* If invalid next_step, just move back */
  if next_step == "-1" then
    self.step_back()
    return
  end if

  /* Updating after step */
  self.current_node ← next_step
  self.known_tree[self.current_node].visited ← TRUE

  /* Checking if in goal */
  if current_position.goal == TRUE then
    self.status ← "FINISHED"
    return
  end if
end procedure

```

2.2 Graph Exploration Algorithm

As mentioned in Section 1.2, the aim of this study is to develop an effective algorithm for graph exploration using multiple agents without communication between them. A significant challenge in this context is avoiding the redundant exploration of nodes already visited by other agents, as it consumes resources without contributing new information to the overall exploration.

To address this problem, we adopt the solution proposed by Rodrigues (2024), that consists on dividing the graph into distinct intervals, which are then split among the agents. This approach ensures that each agent explores a unique section of the graph, thereby minimizing overlap and maximizing efficiency. The intervals are determined based on the number of agents, ensuring a non-overlapping distribution of the graph's nodes. Figure 2.2 illustrates this concept, showing a graph with intervals assigned to different colored agents.

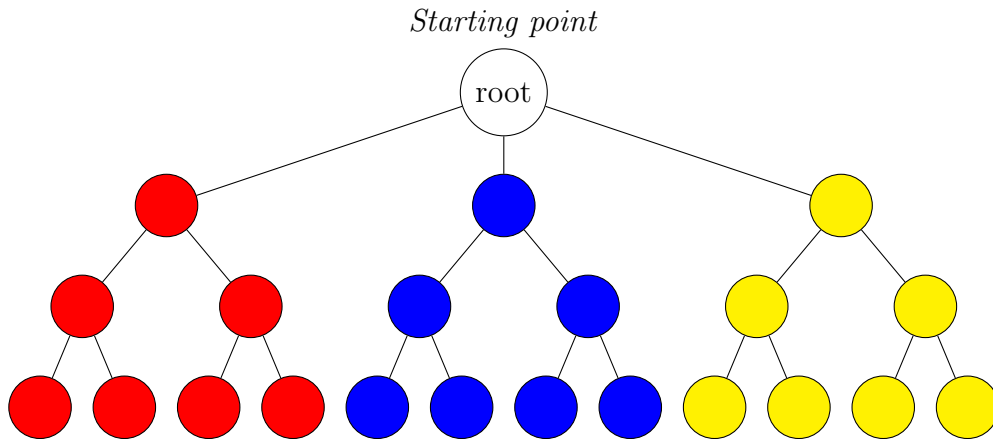


FIGURE 2.2 – Three agents (Red, Blue, and Yellow) disperse from each other in a graph. Source: Rodrigues (2024)

It is important to note that while the numerical intervals are non-overlapping, the dispersion may not always be fair or uniform. This is evident in Figure 2.3, which illustrates three agents dispersing in a non-uniform graph, contrasting with the more uniform dispersion shown in Figure 2.2. The graph shows the nodes colored based on the agent intervals they fall into. If a node has more than one color, it indicates that the node is within the intervals of multiple agents.

Mathematically, this dispersion can be expressed by a set of equations. Assume there are k agents, denoted as $a_1, a_2, a_3, \dots, a_k$. The distribution of intervals can be represented as shown in Equation 2.1, based on the approach proposed by Rodrigues (2024).

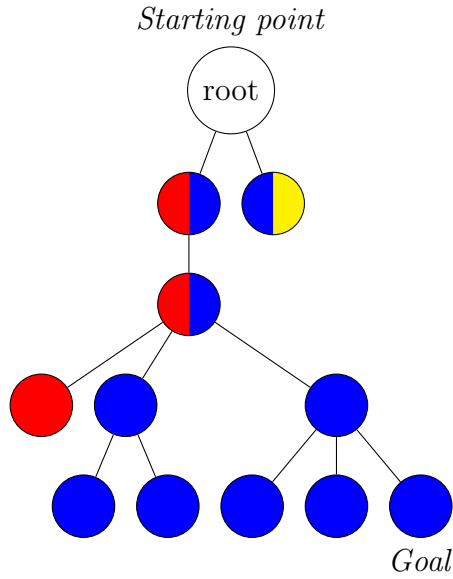


FIGURE 2.3 – Three agents (Red, Blue, and Yellow) disperse in a non-uniform graph.

$$\begin{aligned}
 d &= 1/k \\
 a_1 &: [0, d[\\
 a_2 &: [d, 2d[\\
 a_3 &: [2d, 3d[\\
 &\dots \\
 a_k &: [(k-1)d, 1]
 \end{aligned} \tag{2.1}$$

With the division of intervals explained, we now describe the step-by-step process of our algorithm for multi-agent graph exploration. This algorithm extends the work of Rodrigues (2024), incorporating modifications to handle previously visited nodes and potential cycles.

1. Initialization

- Each agent is assigned a unique identifier and a specific interval based on the Equation Set 2.1.
- Each agent begins with an empty tree representation of the graph, representing the knowledge it has accumulated during the exploration. Since each agent independently explores the graph, their trees may differ, reflecting the order in which nodes are visited.
- All agents start at the same node, although the specific starting node is arbitrary.

- The starting node is assigned the interval $[0, 1]$ and is added to the agent's tree.

2. Exploration Strategy

- At each step, an agent examines the adjacent nodes that have not been previously visited by it. To maintain consistency in exploration, these nodes are sorted by their unique identifiers.
- The agent dynamically calculates convergence intervals for each unvisited adjacent node. The calculation method is as follows:
 - **Single Child:** If there is only one child node, it inherits the convergence interval of the current node.
 - **Multiple Children:** If there are multiple children, the current node's convergence interval is uniformly divided among them.
- Each adjacent node is then added to the agent's tree along with the corresponding edge and its assigned interval.
- Since we are analyzing generic graphs, when adding a node, referred to as the target node, to the tree, it is possible to encounter a scenario where the target node has not been visited yet but already appears in the tree. This can happen in situations:
 - **Backtracking:** If the current node is the father of the target node, it indicates backtracking. In this case, the interval remains unchanged and there are no changes to the agent's tree.
 - **Cycle:** If it is not a backtrack situation, the added edge would be a return edge, as identified by Tarjan's Algorithm, indicating a cycle. To address this, the previous occurrence of the target node is removed from the tree to prevent cycles and is then re-added with its new edge and interval.
- The agent then chooses the first node whose convergence interval intersects with its own. If no such node exists or if all adjacent nodes have been visited, the current node is marked as explored, and the agent backtracks to the parent node.

3. Completion Criteria

- The agent continues the exploration process until it either finds the goal or fills its assigned interval. If the goal is not found within its interval, it must be within the interval of another agent.
- After finishing its interval, the agent may stop or adopt an alternative strategy to accelerate the search. The default behavior implemented is to do a depth-first search (DFS), ignoring nodes already explored in the initial attempt.

The steps described in the **Initialization** topic are implemented in the initialization procedures of the **Simulation** and **Agent** classes. The pseudocode presented in Algorithm 3 illustrates the *define_agent_next_step* method, which implements the step-by-step process described in the **Exploration Strategy** topic.

Algorithm 3 Agent - *define_agent_next_step()*

```

procedure DEFINE_AGENT_NEXT_STEP(self, unvisited_neighbors):
    /* Calculating the convergence intervals */
    /* This will be explained in Section 2.3 */
    intervals  $\leftarrow$  self.get_neighbors.intervals(unvisited_neighbors)

    /* Checking for backtracking or cycle */
    for all neighbor  $n_j$  in unvisited_neighbors do
        if self.known_tree[ $n_j$ ].interval then
            if self.current_node in self.known_tree[ $n_j$ ].get_edges() then
                /* Backtrack */
                intervals[ $n_j$ ]  $\leftarrow$  self.known_tree[ $n_j$ ].interval
            else
                /* Cycle */
                /* The agent removes the previous node from it's tree
                /* so it doesn't cause a cycle */
                self.known_tree.remove_node( $n_j$ )
            end if
        end if
    end for

    /* Adding neighbors to tree */
    self.add_neighbors_to_tree(unvisited_neighbors, intervals)

```

Algorithm 3 Agent - define_agent_next_step()

```

    /* Checking for intersecting intervals */
    /* Is important to take notice that as the neighbors are ordered
    /* And the interval of a neighbor is proportional to its position
    /* As we will see in Algorithm 4, the neighbor are visited in
    /* incresing order of intervals */
    for all neighbor  $n_j$  in unvisited_neighbors do
        max_neighbor  $\leftarrow$  intervals[ $n_j$ ][1]
        min_neighbor  $\leftarrow$  intervals[ $n_j$ ][0]
        max_agent  $\leftarrow$  self.interval[1]
        min_agent  $\leftarrow$  self.interval[0]

        if max_agent < min_neighbor then

            /* If the node's interval is on the right of the agent's interval, surely the
            agent has finished its interval, since the nodes are in increasing order of
            intervals as previously mentioned */
            finished_interval  $\leftarrow$  TRUE
            break
        else min_agent < max_neighbor and max_agent > min_neighbor
            /* Found intersecting intervals */
            return  $n_j$ 
        end if
    end for

    /* If got here, didn't find any intersecting intervals */
    /* If back on root, the agent filled its interval */
    if self.current_node == self.starting_node then
        self.filled_interval  $\leftarrow$  TRUE
    end if

    /* Just step back if interval isn't filled */
    if self.filled_interval == FALSE then
        return "-1"
    end if

    /* If interval is filled, change strategies */
    self.status  $\leftarrow$  "DFS_SEARCH"
    return
end procedure

```

2.3 Mixed Radix Path Representation

This section details how the mixed radix representation described in Section 1.3.3.1 is used to represent the path taken by the agent, including the decision taken in each node as used in Algorithm 3.

Our approach utilizes the same mixed radix representation technique as described by Rodrigues (2024). This method allows an agent to record each decision made along the path without the need to retrospectively analyze previous choices. Instead, it only requires the current node’s convergence interval and its edges to make decisions, reducing computational overhead. This is based on the following logic:

- The path starts at “0.”, and the starting node is assigned the interval $[0, 1]$.
- If the current node has a single child, the agent appends the unary digit I_1 to the path. This indicates that there is only one possible path to take from this node.
- If the current node has multiple children (e.g., n children), the agent chooses the i th child and appends $(i - 1)_n$ to the path. This notation denotes the specific choice among the available children, with i representing the child’s position in an ordered list and the subscript n denotes the number of children or the length of the ordered list.
- When the agent needs to return to a parent node, it removes the last value appended to the mixed radix representation, effectively backtracking to the previous decision point.

To illustrate this concept, we present the mixed radix path representation between the *Starting Point* and *Goal* from the Figure 2.3.

$$x = 0_2 I_1 2_3 2_3 \tag{2.2}$$

Following the conversion as described by Equation 1.7, we can transform this mixed radix notation into a numerical value:

$$x = 0_2 I_1 2_3 2_3 \quad (2.3)$$

$$A = [0, I, 2, 2] \quad (2.4)$$

$$M = [2, 1, 3, 3] \quad (2.5)$$

$$U = [0, 2, 3] \quad (2.6)$$

$$x = \sum_{k \in U} a_k \prod_{j=0}^k \frac{1}{m_j} \approx 0.4444444 \quad (2.7)$$

The function *get_neighbors_intervals* used in Algorithm 3 is defined by this logic and the Equation 1.7 as can be seen in Algorithm 4

Algorithm 4 Agent - *get_neighbors_intervals*()

```

procedure GET_NEIGHBORS_INTERVALS(self, unvisited_neighbors):
    /* Fetching current node interval */
    current_node_interval ← self.known_tree[self.current_node].interval
    current_interval_size ← (current_node_interval[1]-current_node_interval[0])
    interval_chunk ← current_interval_size / unvisited_neighbors.length()

    /* Calculating interval for each neighbor */
    intervals ← []
    for all neighbor  $n_j$  in unvisited_neighbors do
        interval_start ← current_node_interval[0] + interval_chunk *  $j$ 
        interval_end ← interval_start + interval_chunk
        intervals.append((interval_start, interval_end))
    end for
    return intervals
end procedure

```

2.4 Alternative Exploration Algorithms

In our implementation, modifying and experimenting with variations of the exploration algorithm is simple. By simply altering the *define_agent_next_step* function in Algorithm 2, we can easily incorporate different strategies based on the agent's current status or algorithm.

In this work, we have explored five alternatives: one variation of our core algorithm and four adaptations of Tarry's algorithm.

2.4.1 Exploration Algorithm without Communication

2.4.1.1 Backward Interval Filling

The **Backward Interval Filling** algorithm is based on the primary zero-communication strategy developed in this work. Instead of initiating a new Depth-First Search (DFS) traversal after completing a designated interval, agents continue exploring by filling the next interval in reverse order. This modification ensures that agents remain active, even after finishing their primary interval, enhancing the overall graph coverage and reducing idle time.

2.4.2 Exploration Strategies with Communication: Tarry Algorithm Variations

This section presents several adaptations of Tarry’s algorithm that utilize communication between agents. Each variation builds upon the foundational logic of Tarry’s approach but introduces different heuristics and optimizations aimed at improving performance and accommodating various graph structures.

2.4.2.1 Extended Tarry Algorithm

The **Extended Tarry Algorithm**, proposed by Kivelevitch and Cohen (2010), is an adaptation of Tarry’s algorithm. In this approach, agents share information about visited nodes, enabling them to identify unvisited cells and determine the Last Common Location (LCL)—the most recent shared position among agents before diverging paths. The algorithm consists of two phases:

- **Exploration Phase:** Each agent moves to unvisited cells, or to cells unvisited by itself, choosing arbitrarily among multiple options. If all cells have been visited, agents retreat to the last unvisited cell, marking cells as dead ends.
- **Cooperative Backtracking Phase:** Once an agent finds the goal and is defined as the pioneer, other agents backtrack to the Last Common Location (LCL) with it and follow the pioneer’s path to the goal.

The functions *define_next_first_phase_tarry_step* and *define_next_second_phase_tarry_step*, as shown in Algorithms 5 and 6, implement this logic.

Algorithm 5 Agent - define_next_first_phase_tarry_step()

```

procedure DEFINE_NEXT_FIRST_PHASE_TARRY_STEP(non_visited_neighbors):
  new_neighbors ← []
  already_visited_neighbors ← []
  /* Splitting Neighbors */
  for all neighbor in non_visited_neighbors do:
    if neighbor not visited and not deadEnd then:
      new_neighbors.append(neighbor)
    else if neighbor visited then:
      already_visited_neighbors.append(neighbor)
    end if
  end for
  if new_neighbors is not empty then:
    /* Choosing a random non-visited neighbor */
    return random_choice(new_neighbors)
  else if already_visited_neighbors is not empty then:
    /* Choosing a random non-visited by this agent neighbor */
    return random_choice(already_visited_neighbors)
  end if
  return "-1"
end procedure

```

Algorithm 6 Agent - define_next_second_phase_tarry_step()

```

procedure DEFINE_NEXT_SECOND_PHASE_TARRY_STEP(non_visited_neighbors,
  pioneer_id, pioneer_effective_path, lcl)
  if self.status == "TARRY_FIRST_PHASE" then
    self.status ← "TARRY_SECOND_PHASE"
  end if
  if self.status == "TARRY_SECOND_PHASE" then
    if self.current_node_id != lcl[pioneer_id][self.id] then
      /* Step back until the last common location */
      return "-1"
    else
      self.status ← "TARRY_FOLLOW_PIONEER"
    end if
  end if
  current_node_index ← pioneer_effective_path.index(self.current_node_id)
  /* Follow pioneer's path */
  return pioneer_effective_path[current_node_index + 1]
end procedure

```

2.4.2.2 Tarry Tie-Breaker Variation

In this variation, agents employ Tarry’s algorithm but modify the selection process described in Algorithm 5. Instead of randomly choosing among valid neighbors, agents prioritize their choices based on interval assignments derived from mixed radix path intervals. This enhancement aims to improve exploration efficiency by leveraging structured knowledge of the graph’s structure to better disperse the agents.

2.4.2.3 Tarry Interval Priority Variation

Each agent attempts to fill its assigned interval as described in Algorithm 3, but rather than using depth-first search (DFS) or backward filling once the interval is complete, it begins with Tarry’s algorithm, as described in Algorithms 5 and 6. This strategy seeks to leverage the additional information gathered during the interval search to enhance exploration efficiency and reduce redundant backtracking.

2.4.2.4 Delayed Tarry Variation

In this approach, agents begin by navigating their assigned intervals for a predetermined number of steps to ensure an initial spread across distinct branches of the graph. After this dispersal phase, they switch to Tarry’s algorithm to continue exploration. This method aims to reduce collisions by increasing the probability that agents explore separate regions early on, leading to improved coverage and efficiency in the overall exploration.

2.5 Graph Visualization

One significant enhancement is the capability to visualize graphs beyond perfect mazes, expanding our algorithm’s applicability to general graphs.

To demonstrate this capability, we applied our algorithm to explore an imperfect maze, which is a cyclic graph, using three agents. The overall exploration of the maze is depicted in Figure 2.4.

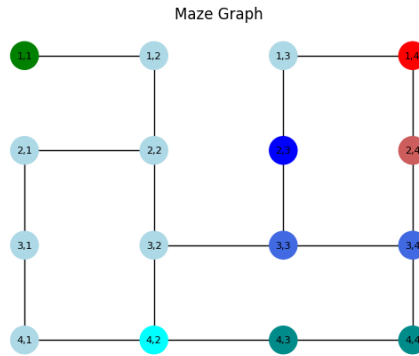


FIGURE 2.4 – Exploration of an imperfect maze(graph) by three agents.

This visualization showcases how our algorithm can handle graphs with cycles, illustrating the adaptability of our approach.

2.6 Agent Tree Visualization

In addition to overall graph exploration, we developed a method to visualize the individual exploration trees constructed by each agent during the process. These trees represent the paths traversed by the agents and provide insight into the structure of their independent exploration, including how cycles are managed and intervals are assigned to each node.

The Figures 2.5, 2.6 and 2.7 show the trees constructed by the 3 agents for the exploration displayed in Figure 2.4.

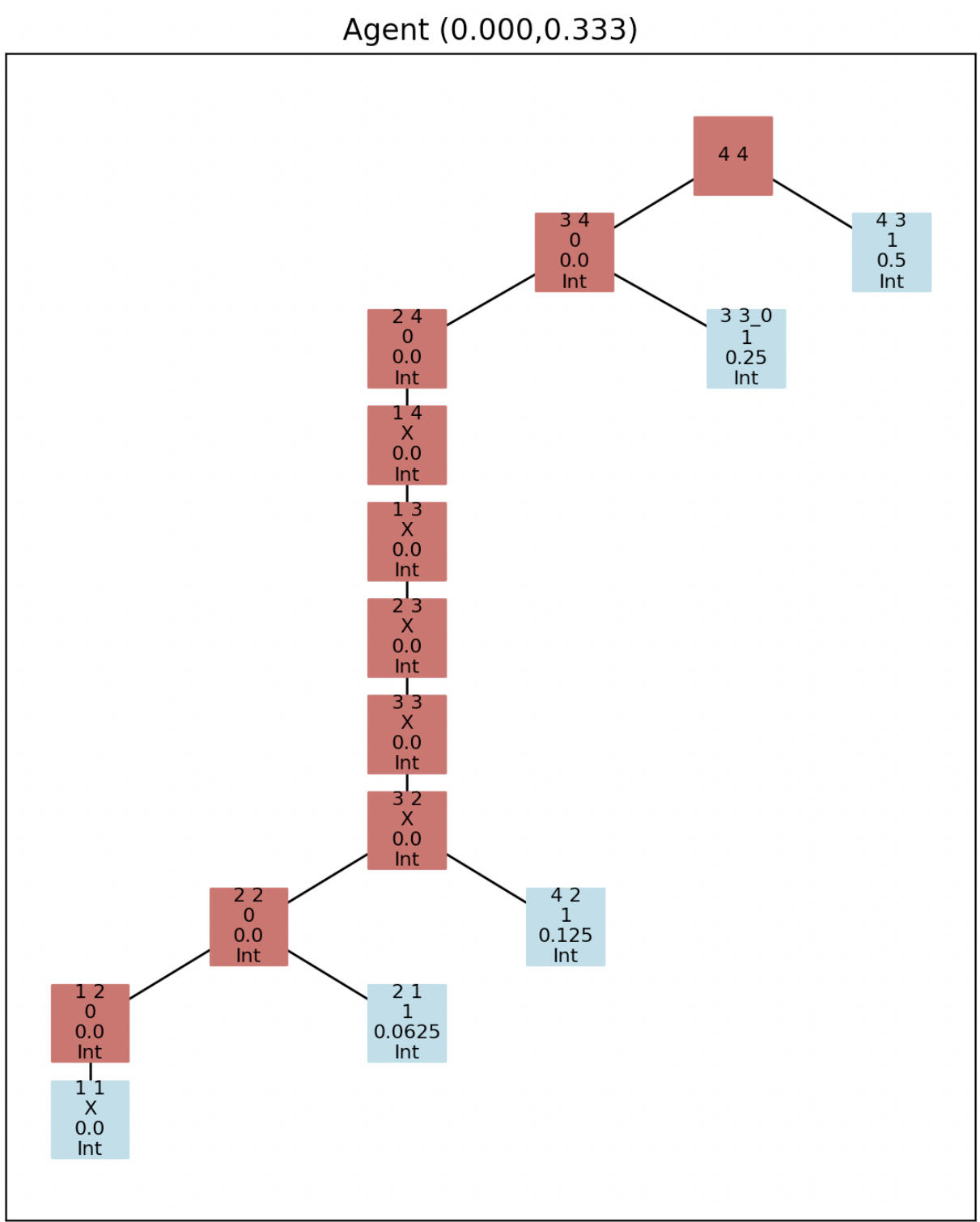


FIGURE 2.5 – Trees constructed by agent post-exploration: **Agent 1.**

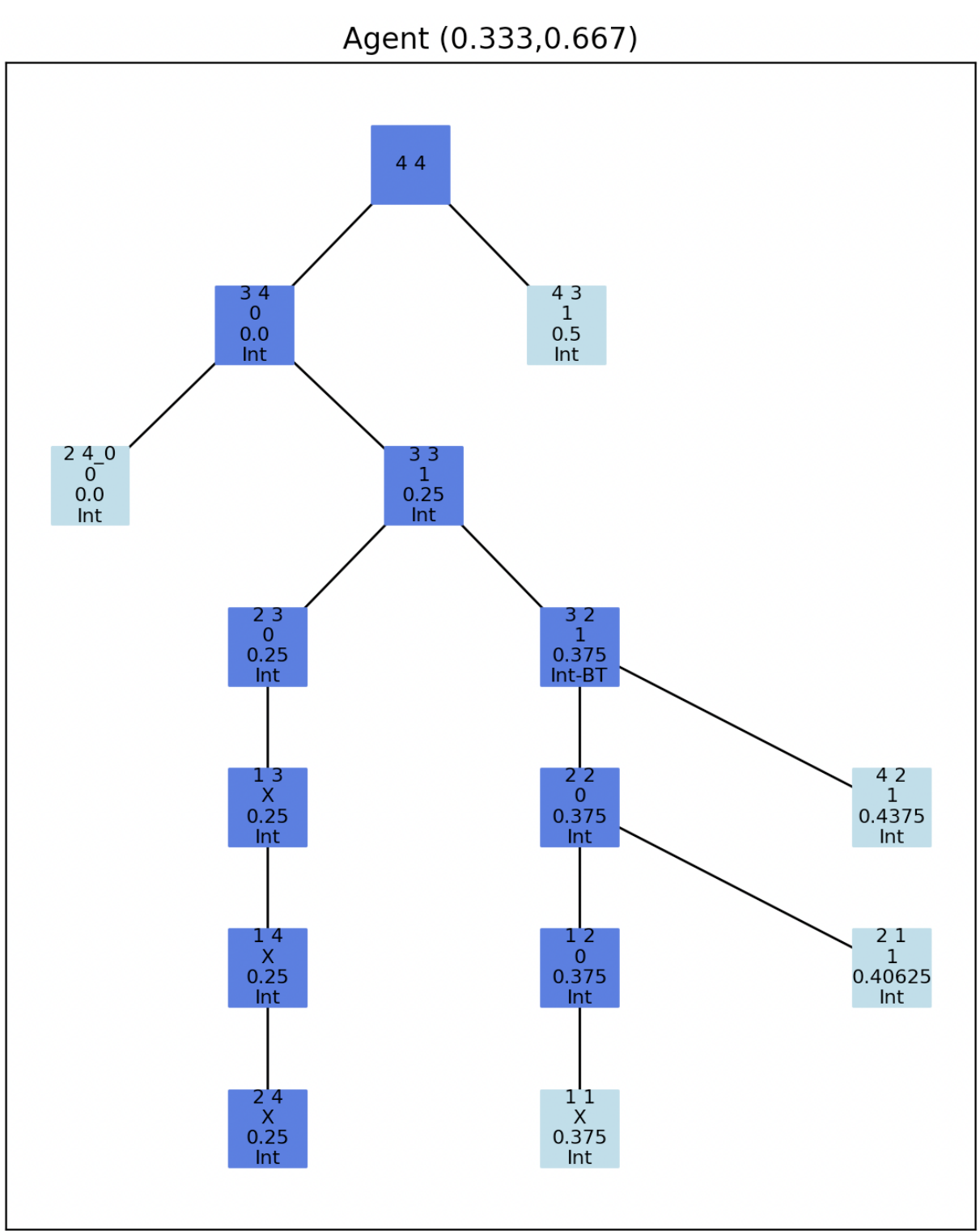


FIGURE 2.6 – Trees constructed by agent post-exploration: **Agent 2**.

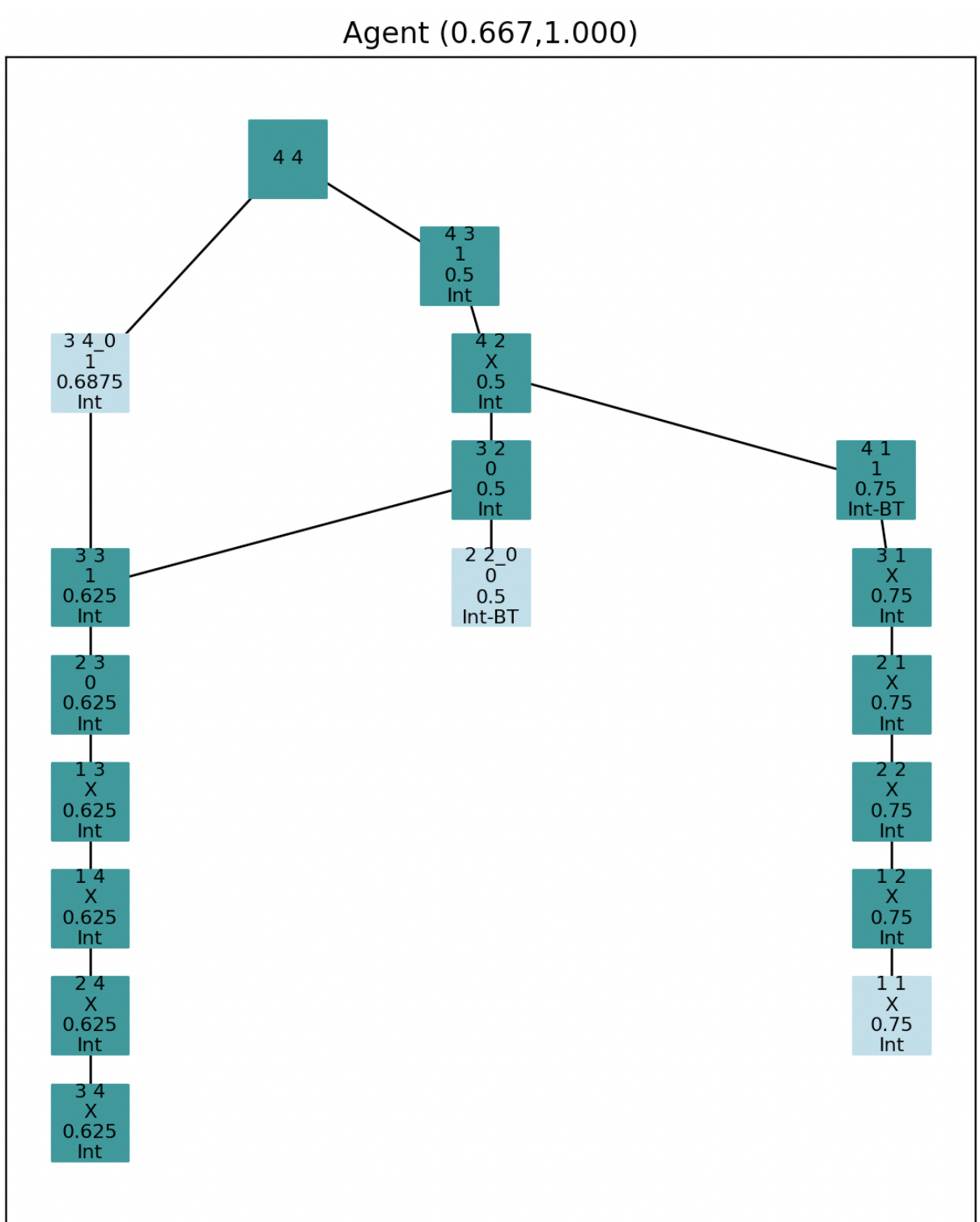


FIGURE 2.7 – Trees constructed by agent post-exploration: **Agent 3**.

These visualizations illustrate how each agent independently constructs its own representation of the graph, adapting to the imperfections and resulting in different exploration paths.

3 Results and Discussion

This section presents the results of comparing the proposed no-communication algorithm with Tarry’s algorithm, as well as analyzing Tarry’s performance against its variants. We assess these approaches on different datasets to observe how distinct graph properties influence the efficiency of the solutions. Section 3.1 provides results for perfect mazes using the dataset from Rodrigues (2024). Section 3.2 shows outcomes for random trees generated using a specific method from NetworkX (HAGBERG *et al.*, 2008).

3.1 Perfect Maze Results

3.1.1 Metrics for Analysis

3.1.2 No-Communication Algorithms Performance

3.1.3 Tarry’s Variants Performance

3.2 Random Tree Results

3.2.1 Metrics for Analysis

3.2.2 No-Communication Algorithms Performance

3.2.3 Tarry’s Variants Performance

4 Conclusions and Future Works

This work presented an algorithm for graph exploration in a multi-agent system without communication, expanding on the solution proposed by Rodrigues (2024). The flexibility provided by our framework, which no longer relies on the open-source perfect maze generator by Naeem (2021), allows for expansion to various exploration policies and algorithmic variations. Additionally, the modular design of our system enables integration of new exploration strategies, paving the way for future advancements.

We conducted tests on 100 randomly generated 40x40 perfect mazes and compared the results across three algorithms: our own, a backward interval variation, and an extended version of Tarry’s algorithm. The results were validated against previous work (RODRIGUES, 2024), confirming the correctness and reliability of our approach.

Looking forward, we aim to validate our algorithm against established graph generation algorithms, assessing its efficiency on a larger scale. Exploring the implementation of more advanced algorithms and optimizing the agents’ decision-making processes will be essential for enhancing the applicability of our approach.

By advancing multi-agent graph exploration without communication, this research paves the way for innovations in autonomous systems, with potential applications across various domains to be explored in future research.

Bibliography

ARNDT, J. Mixed radix numbers. In: **Matters Computational**. Berlin, Heidelberg: Springer, 2011. p. 183–209.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using networkx. In: VAROQUAUX, G.; VAUGHT, T.; MILLMAN, J. (Ed.). **Proceedings of the 7th Python in Science Conference (SciPy2008)**. Pasadena, CA USA: [s.n.], 2008. p. 11–15.

KIVELEVITCH, E.; COHEN, K. Multi-agent maze exploration. **Journal of Aerospace Computing, Information, and Communication**, v. 7, p. 391–405, 2010.

MANBER, U. **Introduction to Algorithms: A Creative Approach**. 1st. ed. New York: Addison-Wesley, 1989.

NAEEM, M. A. **pyamaze**. 2021. <https://www.github.com/MAN1986/pyamaze>. Accessed on: June, 2024.

RODRIGUES, A. J. D. S. **Multi-agent graph exploration without communication**. Bachelor's Thesis — Instituto Tecnológico de Aeronáutica,, São José dos Campos, 2024.

WILSON, R. J. **Introduction to Graph Theory**. 4th. ed. Boston, MA: Longman Group Ltd, 1996.

FOLHA DE REGISTRO DO DOCUMENTO			
1. CLASSIFICAÇÃO/TIPO TC	2. DATA X de Novembro de 2024	3. DOCUMENTO Nº DCTA/ITA/?/2024	4. Nº DE PÁGINAS 42
5. TÍTULO E SUBTÍTULO: Multi-Agent Graph Exploration Without Communication			
6. AUTOR(ES): Rafael Studart Mattos Di Piero			
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica – ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Graph, Search, Multi-Agent, Mixed-Radix			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: ?			
10. APRESENTAÇÃO: (X) Nacional () Internacional ITA, São José dos Campos. Curso de Graduação em Engenharia de Computação. Orientador: Prof. Dr. Luiz Gustavo Bizarro Mirisola; coorientador: Prof. Dr. Vitor Venceslau Curtis. Publicado em 2024.			
11. RESUMO: <p>Graph theory, a pivotal field in mathematics and computer science, offers robust frameworks for modeling relationships and traversing graph structures. This research delves into graph exploration, crucial for applications like network routing, robotics, and procedural generation. As real-world applications often involve complex environments with time constraints, this study focuses on multi-agent systems for graph exploration, where multiple autonomous agents collaborate to optimize task distribution.</p> <p>This study's objective is to propose an efficient multi-agent algorithm for graph exploration without communication, a crucial need in scenarios where communication is impractical or impossible, such as deep-sea exploration or energy-constrained environments. Building on the method for perfect maze exploration detailed by Rodrigues (2024), this study extends the approach to general graphs, that may include cycles. Several algorithmic variations, combining the proposed approach and Tarry's variation (KIVELEVITCH; COHEN, 2010), were implemented in an effort to compare how different algorithms perform in various scenarios and to identify efficient use cases. Furthermore, the algorithms were reimplemented using a generic graph library instead of a specialized perfect maze library (NAEEM, 2021).</p> <p>By structuring the exploration into well-defined and extensible classes, the research offers a versatile framework for broader applications in multi-agent systems. This framework was used to evaluate different datasets, containing graphs with varying properties, paving the way for further advancements in autonomous graph exploration.</p>			
12. GRAU DE SIGILO: (X) OSTENSIVO () RESERVADO () SECRETO			