INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Rafael Studart Mattos Di Piero

MULTI-AGENT GRAPH EXPLORATION WITHOUT COMMUNICATION

Bachelor's Thesis
2024

Course of Computer Engineering

CDU CDU

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

2024

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.
39f.

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. 39f. Bachelor's Thesis – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

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: Jun 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. 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, paving the way for further advancements in autonomous graph exploration.

List of Figures

List of Algorithms

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
$v_n v_m$ Edge

Contents

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 expanded 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 Tarry'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.

General improvements, such as structuring the exploration into well-defined and extensible classes and incorporating built-in testing, strive to make the development process more versatile and robust, paving the way for broader applications in multi-agent systems.

1.2 Objective

This study aims to propose a viable algorithm for graph exploration by multi-agents, specifically targeting the identification 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 could allow further discussion and contribute to research in real-world applications such as robotics, as well as in the theoretical analysis of parallelism and communication between agents.

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, ... in which any two consecutive edges are adjacent, all edges are distinct and all vertices v_0, v_1, ..., 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 $A = [a_0, a_1, a_2, ..., a_{n-1}]$ of a number x with respect to a radix vector $M = [m_0, m_1, m_2, ..., m_{n-1}]$, where $x \in [0, 1]$, is given by:

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

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:

$$x = 2_3 1_2 1_4 0_3 1_5 \tag{1.2}$$

$$A = [2, 1, 1, 0, 1] \tag{1.3}$$

$$M = [3, 2, 4, 3, 5] \tag{1.4}$$

$$n = 5 \tag{1.5}$$

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

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 $A = [a_0, a_1, a_2, ..., a_{n-1}]$ of a number x with respect to a radix vector $M = [m_0, m_1, m_2, ..., m_{n-1}]$, 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.

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

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:

$$x = 2_3 I_1 1_2 I_1 I_1 1_4 0_3 1_5 \tag{1.8}$$

$$A = [2, I, 1, I, I, 1, 0, 1] \tag{1.9}$$

$$M = [3, 1, 2, 1, 1, 4, 3, 5] \tag{1.10}$$

$$U = [0, 2, 5, 6, 7] \tag{1.11}$$

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

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. Finally, 2.4 details how we can extend our solution to allow different exploration algorithms.

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.
- Orchestration: Coordination agent activities and possible interactions during the exploration.

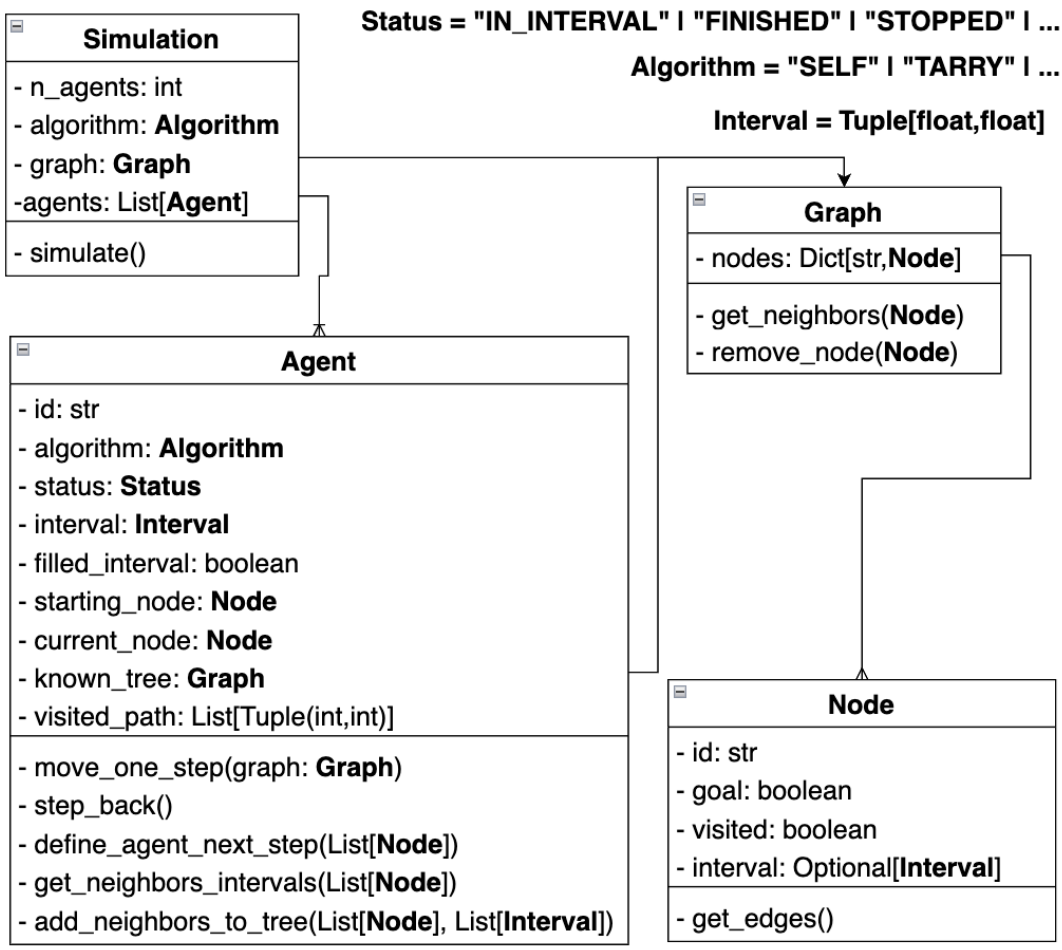

FIGURE 2.1 – Simplified Class Diagram of the Multi-Agent Maze Exploration Model

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

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)
```

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.

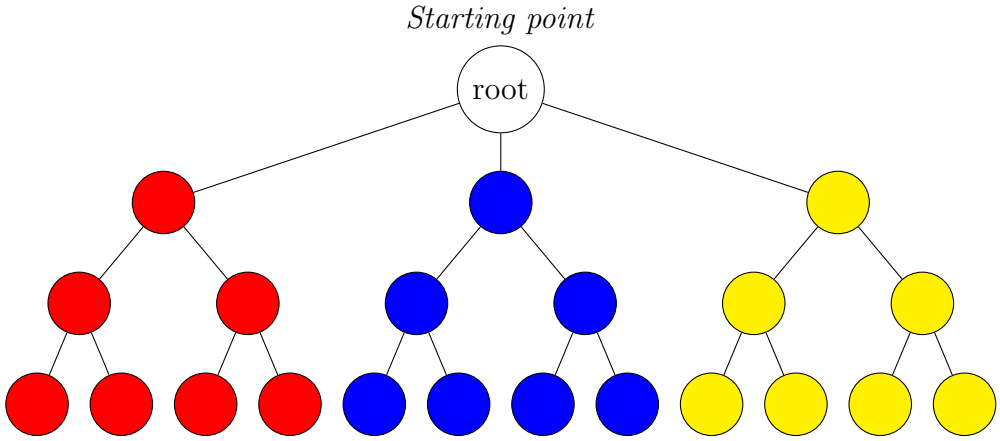

FIGURE 2.2 – Three agents (Red, Blue, and Yellow) disperse from each other in a graph. Source: Rodrigues (2024)

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 $a_1, a_2, a_3, ..., a_k$. The distribution of intervals can be represented as shown in Equation 2.1, based on the approach proposed by Rodrigues (2024).

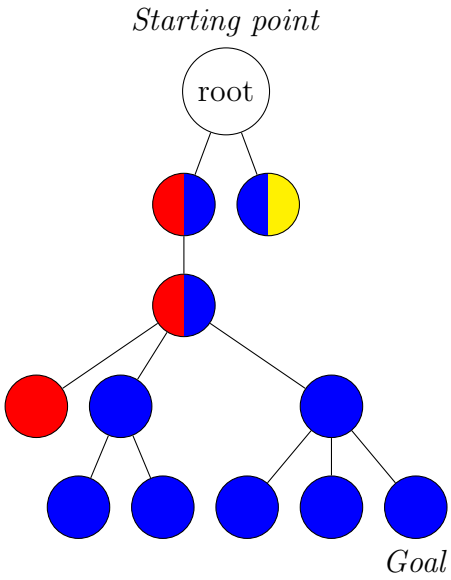

FIGURE 2.3 – Three agents (Red, Blue, and Yellow) disperse in a non-uniform graph.

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

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 ← 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] ← 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 ← intervals[n_j][1]
    min_neighbor ← intervals[n_j][0]
    max_agent ← self.interval[1]
    min_agent ← 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 ← 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 ← 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 ← "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 ith 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:

$$x = 0_2 I_1 2_3 2_3 \tag{2.3}$$

$$A = [0, I, 2, 2] \tag{2.4}$$

$$M = [2, 1, 3, 3] \tag{2.5}$$

$$U = [0, 2, 3] \tag{2.6}$$

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

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 changing the $define\_agent\_next\_step$ function in Algorithm 2, we can easily incorporate different strategies based on the status or algorithm of the agent.

We've explored two alternatives:

- **Backward Interval Filling:** Instead of starting a new DFS after finishing an interval, agents fill the next interval in reverse order. This modification aims to enhance exploration by keeping agents active and improving coverage.

- **Extended Tarry Algorithm:** The extended version of Tarry's algorithm proposed by Kivelevitch and Cohen (2010).

3 Results and Discussion

3.1 Initial Results

This section presents our initial results, aimed at replicating the findings from Rodrigues (2024) to ensure the integrity of our implementation. We tested our algorithm on 100 randomly generated 40x40 perfect mazes (NAEEM, 2021), incrementing the number of agents from 1 to 40. We evaluated:

- Steps taken by the pioneer (first agent to find the goal).
- Fraction explored before the pioneer found the goal.

These tests were conducted for three implemented algorithms:

- Our Algorithm
- Backward Interval Filling Variation
- Extended Tarry's Algorithm (KIVELEVITCH; COHEN, 2010)

Below, we provide comparative visuals for each algorithm's performance.

Figures 3.1 and 3.2 show a side-by-side comparison of our results with those from Rodrigues (2024), considering all algorithms implemented.

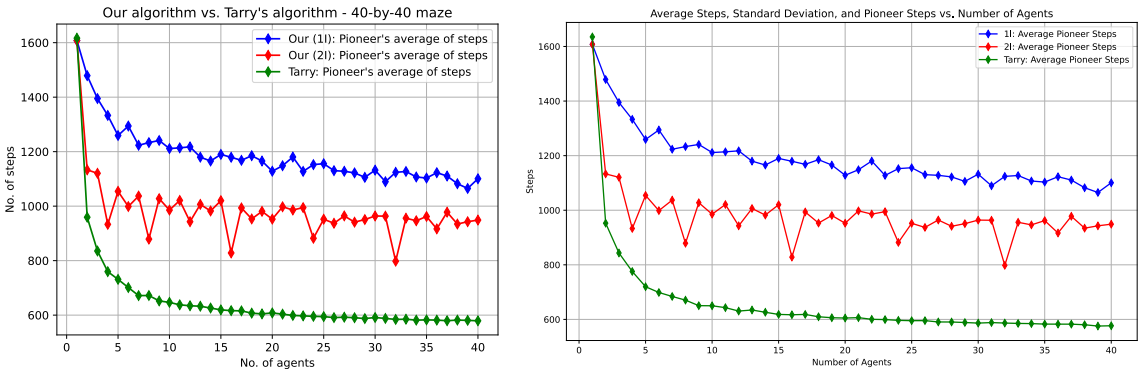

FIGURE 3.1 – Comparison of total steps: Left - Results from Rodrigues (2024); Right - Our results

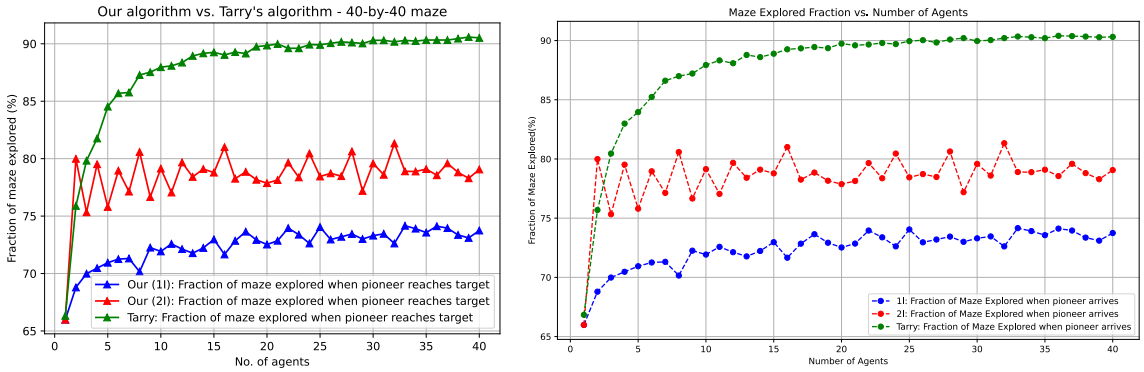

FIGURE 3.2 – Comparison of explored fraction: Left - Results from Rodrigues (2024); Right - Our results

As evident from the figures, our results closely match the original findings, demonstrating that our modifications have not altered the core functionality of the algorithm.

3.2 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 3.3.

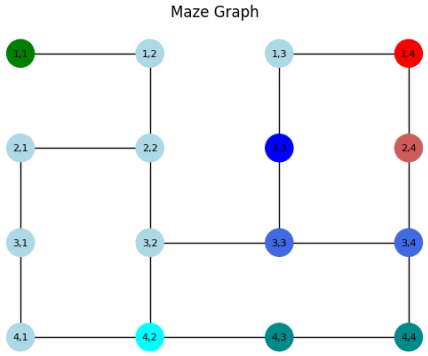

FIGURE 3.3 – Exploration of an imperfect maze(graph) by three agents.

This visualization showcases how our algorithm can handle graphs with cycles, illustrating the adaptability of our approach. To enable comparisons with our previous work on perfect mazes, we developed utilities to convert between the old and new graph formats.

3.3 Agent Tree Visualization

In addition to overall graph exploration, we enhanced our implementation to visualize the individual trees constructed by each agent during the exploration process. These trees reflect the paths taken by agents, incorporating features such as handling cycles and displaying intervals for each node.

The Figures 3.4, 3.5 and 3.6 show the trees constructed by the 3 agents for the exploration displayed in Figure 3.3.

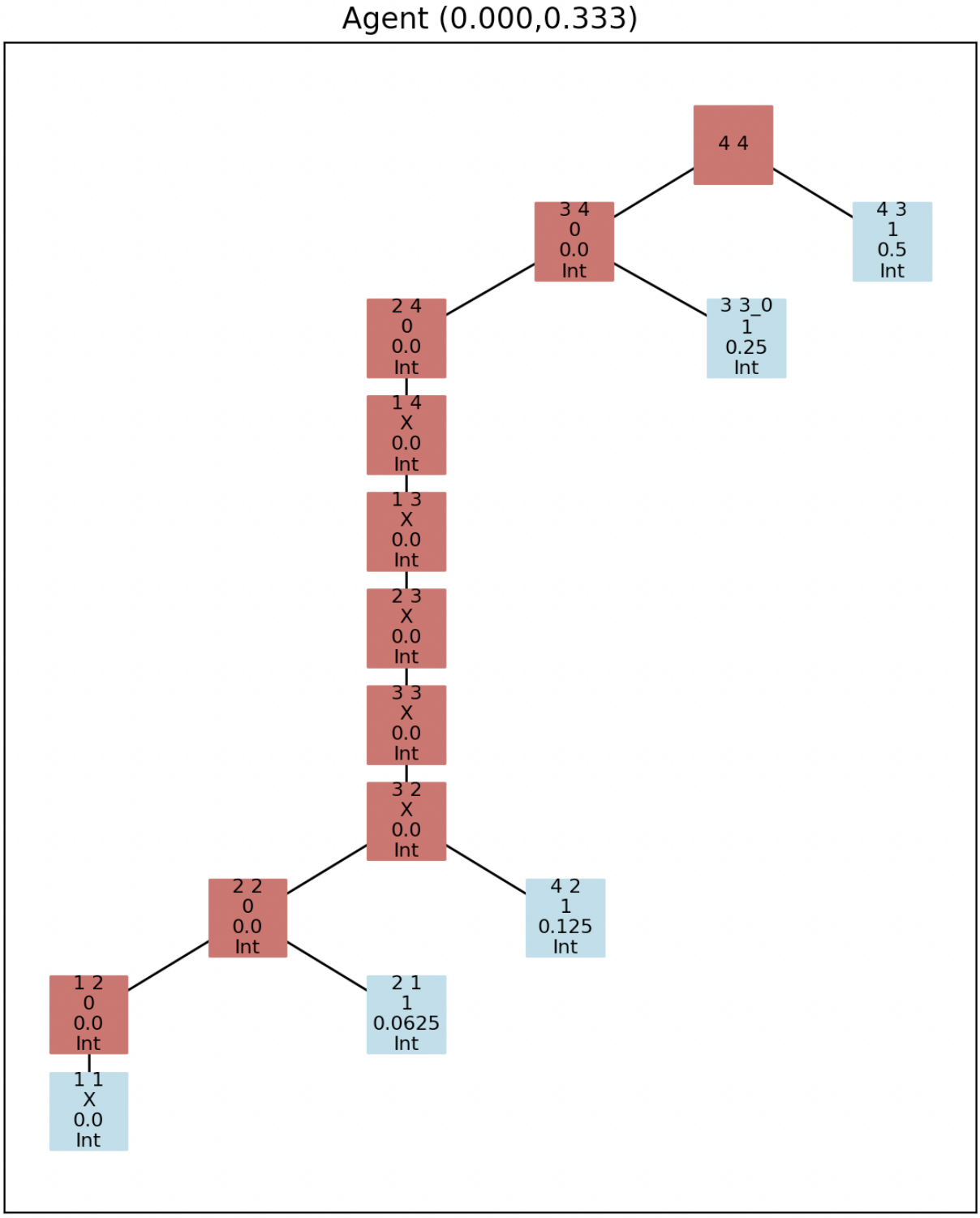

FIGURE 3.4 – Trees constructed by agent post-exploration: **Agent 1**.

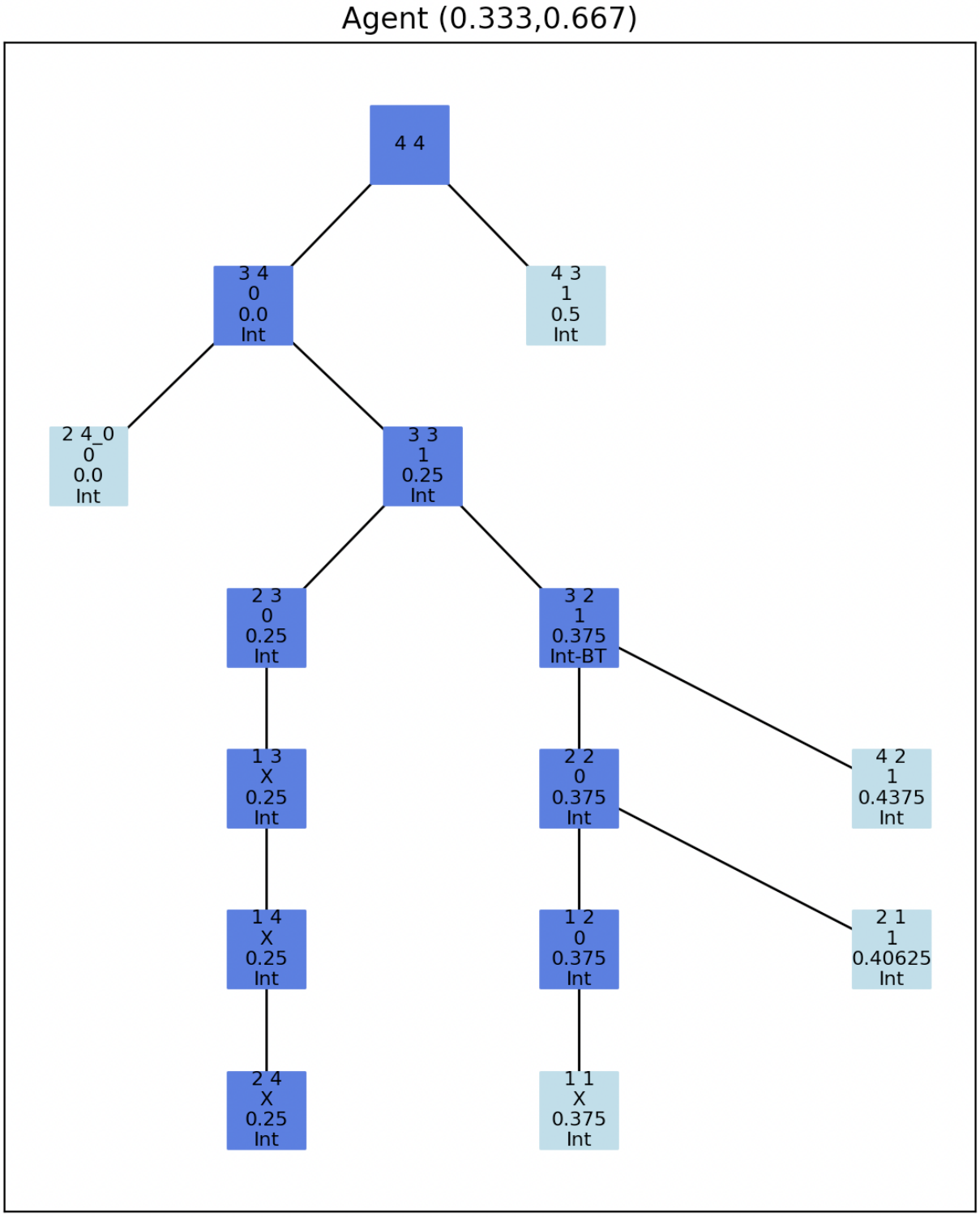

FIGURE 3.5 – Trees constructed by agent post-exploration: **Agent 2**.

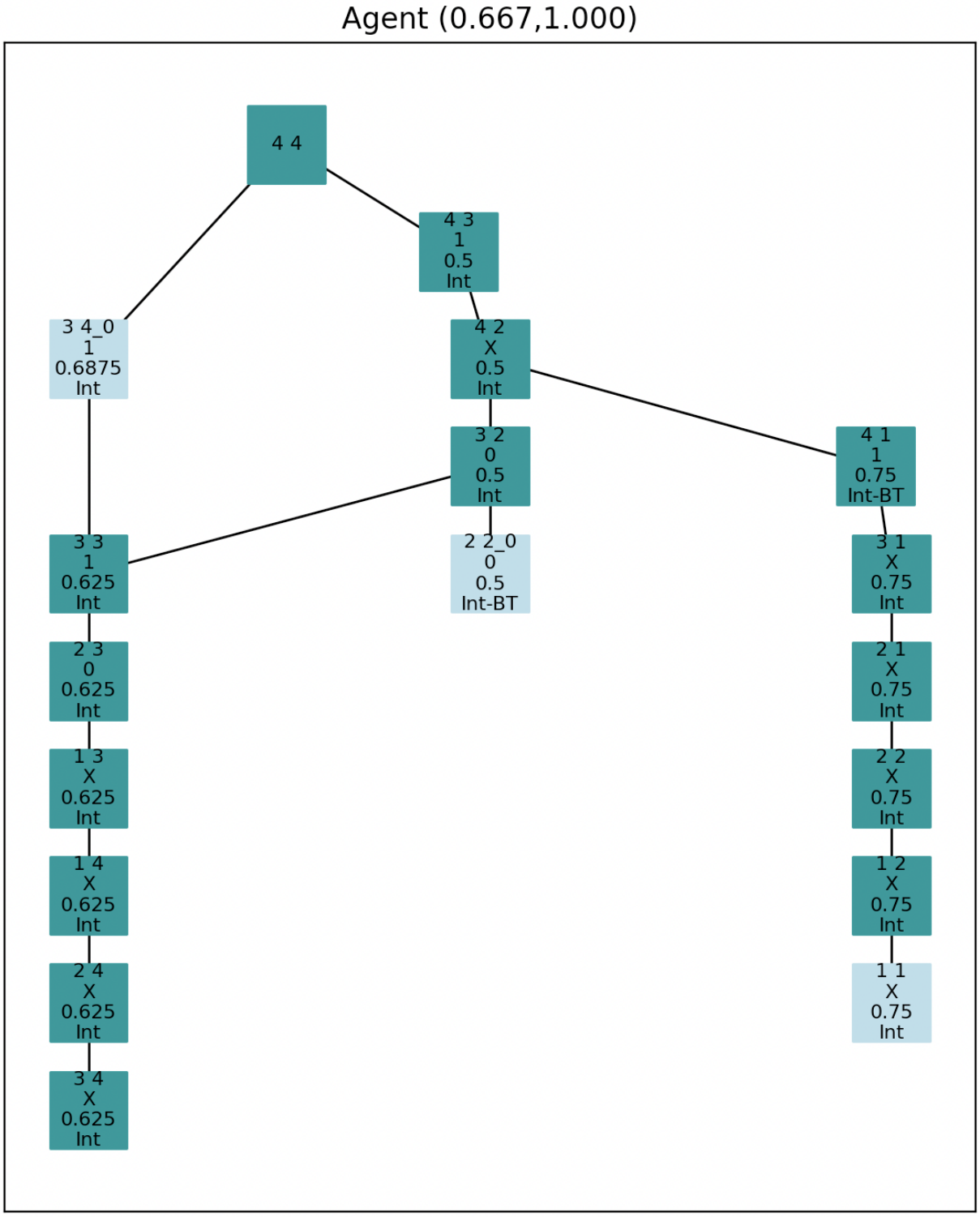

FIGURE 3.6 – Trees constructed by agent post-exploration: **Agent 3**.

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

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.

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	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	20 de Junho de 2024	DCTA/ITA/?/2024	39

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:

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. 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, paving the way for further advancements in autonomous graph exploration.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**