Individual Thesis

IMPROVEMENTS AND EXPERIMENTAL EVALUATION OF TASK ALLOCATION FOR MULTI-AGENT SYSTEMS

Alexis Andreou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

June 2025

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Improvements and experimental evaluation of task allocation for multi-agent systems

Alexis Andreou

Supervising Professor:

Dr Yiannis Dimopoulos

The Individual Thesis was submitted in partial fulfillment of the requirements for the degree of Computer Science of the Department of Computer Science of the University of Cyprus.

Abstract

This thesis focuses on the centralized approach to interdependent task allocation in cooperative multi-agent systems, particularly in scenarios such as wildfire detection using autonomous drones. While much prior research has emphasized decentralized coalition formation—the framework proposed by Douae Ahmadoun (2022) on the doctoral research of at the University of Paris, titled: ***Ahmadoun, D. (2022). Interdependent Task Allocation via Coalition Formation for Cooperative Multi-Agent Systems. PhD Thesis, University of Paris*** [3]—this work pivots to the design, implementation, and experimental evaluation of centralized solutions using Answer Set Programming (ASP) via Clingo.

By leveragingg the expressive power and computational effectiveness of Clingo, this thesis presents an ASP formulation of the centralised variation of the coalition formation model by Ahmadoun (2022), that explicitly accounts for task interdependencied, resource constraints, and environmental uncertainties. Extensive experiments are conducted to evaluate performance in terms of solution quality, computational efficiency, and scalability.

In addition to implementing the centralized models, this thesis also explores and attempts targeted improvements to the previously proposed decentralized approach [3], assessing their relative strengths and limitations.

The results demonstrate the efficacy of centralized ASP-based methods in producing high-quality task allocations under complex constraints, offering a strong baseline for comparison with decentralized solutions. Future directions include the integration of learning-based adaptations, real-world validation, and further refinement of both centralized and decentralized frameworks.

Table of Contents

CHAPTER 1: Introduction

1.1 Background and Motivation

Multi-agent systems have attracted a of lot of interest in many areas such as disaster management, defence, autonomous robotics and enviromental monitoring. Particularly in real-time, high-risk situations, the growing dependence on autonomous agents, such as drones, has created new difficulties in organising and maximising their activities. Task distribution is one of the main difficulties in these systems since it guarantees that several agents mutually achieve activities in the most efficient and effective way.

Traditional task allocation methods often assume task independence, where each task can be assigned to an agent without considering dependencies on other tasks. However, real-world scenarios often involve interdependent tasks, meaning that the execution of one task can be influenced by another. In applications such as wildfire detection and monitoring, drones must coordinate their actions, share data, and optimize coverage based on environmental conditions, fuel constraints, resources and position of agents and tasks.

While current centralised task allocation methods including Answer Set Programming (Clingo) and constraint programming (MiniZinc) have great optimisation potential, they have limited scatability capacities. Therefore, distributed tasks is essential to allow drones working in dynamic surroundings to enable adaptive, fault-tolerant, and real-time decision-making.

Emphasising autonomous drone cooperation in fire detection and monitoring missions, this dissertation experiment the scalability of centralized methods and the distributed coalition building method for multi-agent task allocation proposed by Douae Ahmadoun [3]. The suggested approach guarantees that drones dynamically create coalitions, adapt to environmental constraints, and maximise objective in a dispersed way.

1.2 Research Problem

The challenge of interdependent task allocation in cooperative multi-agent systems requires an efficient method for allocating tasks among autonomous agents while taking into consideration a variety of dependencies including:

- **Resource limitation:** Agents may not have the required resources to execute tasks.
- **Dynamic coalition formation:** Agents have to self-organise into the best possible groups according to task priorities and available resources.
- **Spatial Constraints:** The allocation must consider coverage optimization to avoid unnecessary searching.

Due to high computational complexity and requirment for a global knowledge of the problem, traditional centralized task allocation approaches have difficulty meeting these limitations. While centralized methods such as Answer Set Programming and Constraint programming can provide optimal solutions, they are not the best approach for dynamic problems where real-time adaptability and efficiency is crucial. This thesis contrasts the decentralized coalition formation method proposed by Ahmadoun [3]—where agents autonomously communicate via message passing to allocate tasks—with a centralized approach implemented using Answer Set Programming. While Ahmadoun's method enables agents to dynamically form coalitions based on their capabilities, location, and resource availability—offering scalability, fault tolerance, and adaptability to real-time conditions—it does not guarantee optimal task allocation. In contrast, the centralized approach developed in this thesis aims to achieve higher-quality solutions through global optimization, enabling a direct comparison of performance, scalability, and allocation quality between the two paradigms.

1.3 Objectives

The main objective of this thesis is to develop and evaluate a centralized approach for interdependent task allocation in multi-agent systems using Answer Set Programming (ASP) with Clingo. The focus is on implementing scalable ASP models capable of handling large-scale problem instances while maintaining solution quality. Additionally, the thesis aims to contrast this centralized approach with the decentralized method (FICSAM) proposed by Ahmadoun [3], and to explore potential improvements to the decentralized method in terms

of allocation quality. To validate the proposed work, a series of experiments are conducted to assess scalability, efficiency, and the utility of solutions in realistic task allocation scenarios.

1.4 Research Contributions

This research contributes to the field of multi-agent task allocation by implementing a centralized coalition formation model using Answer Set Programming with Clingo, aimed at efficiently handling interdependent tasks. The study emphasizes the advantages of Clingo over MiniZinc [3] for solving large-scale, complex allocation problems due to its enhanced computational performance and scalability. Furthermore, it provides a comparative analysis between the centralized Clingo-based method and the decentralized approach proposed by Ahmadoun, highlighting the strengths and limitations of each. Through extensive simulations in drone-based task allocation scenarios, the thesis demonstrates how the centralized method improves solution quality and responsiveness under real-world constraints.

1.5 Source Attribution and Original Contributions

The decentralized task allocation methods implemented in this thesis—specifically the FICSAM (Feasible Interdependent Coalition Structure Anytime Method) and IFICSAM (Improved FICSAM) algorithms—are based on the doctoral research of Douae Ahmadoun at the University of Paris, titled:

Ahmadoun, D. (2022). Interdependent Task Allocation via Coalition Formation for Cooperative Multi-Agent Systems. PhD Thesis, University of Paris.

The architecture, class structure, token-based communication protocol, and core decision-making logic for agent coordination were adapted from this source. The class diagram used in this thesis was also derived directly from Ahmadoun's original framework.

However, significant components of this work are original and were developed from scratch. This includes the centralized model implemented in Clingo, which were designed independently to benchmark the performance of the decentralized approaches. Furthermore, the utility function, data structures for agent-task modeling, and all experimental scenarios, automation scripts, and result visualizations were designed specifically for this thesis.

Additionally, enhancements and new prototypes were introduced in Section 3.3.5 reflecting contributions that tried to improve the functionality and performance of the original decentralized algorithmic base.

Where appropriate, in-text citations and footnotes reference the original source. This separation ensures transparency in methodology and guards against unintentional attribution errors.

CHAPTER 2: Literature Review

2.1 Agents

Agent is an autonomous entity that recognize its environment, makes decisions, and executes actions to achieve specific goals. Artificial intelligence, robotic, and distrinuted systems use agents, which can be either physical (e.g. robot, drone) or virtual (e.g. software bot, simulated entities [16].

In multi-agent systems, multiple agents interact with each other, either cooperatively or competitively, to complete complex tasks. Unlike traditional computational models where a central entity governs decision-making, agents in MAS operate independently, communicate via message-passing, and dynamically adapt to their surroundings [3, 22].

An agent perceives its environment through sensors, collecting relevant data to complete its tasks. For drones involved in forest-fire detection, perception typically involves gathering real-time information through sensors, such as thermal cameras, optical cameras, and environmental sensors. Through this perception, drones collect the necessary data for detecting a fire. After perceiving its enviroment, agents process this infromations to make decisions autonomously. This decision-making capability distinguishes agents from tradional remote-controlled systems. Autonomy is when the agent analyze sensor data, internal states (e.g. battery status, computing reousrces), and task requirments. The agent then calculates how to allocate its resources optimally, whether to continue its current activity, form or join a coalition, or to alter its actions.

Agents perform certain actions based on thesis decision. For drones detecting a forest-fire, these may be flying to specific coordinates, searching areas for thermal signatures, indicating fires discovered, joining or leaving coalitions, or transmitting information ot other drones or a ground station. The success of a drone agent is its ability to respond fast and react to dynamic circumstances, most significant in tracking forest fires where swift raction time could be a major factor in determining the effectiveness and efficiency of efforts to fight fire.

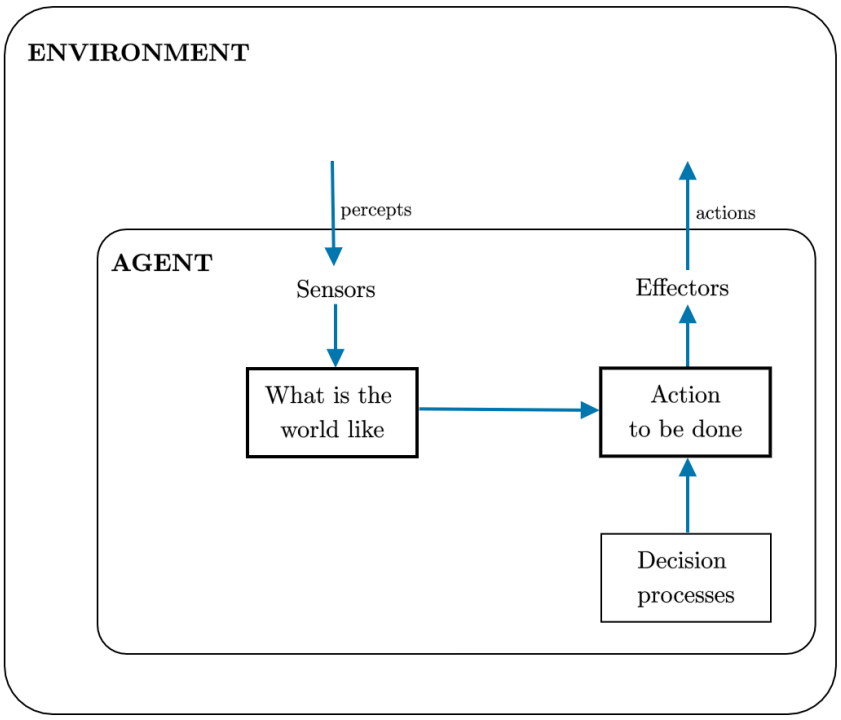

Figure 2.1: Simple representation of an agent

Within the larger structure of multi-agent systems, agents rely on continuous interaction and collaboration to optimize performance. By exchanging messages and dynamically adjusting their coalitions, drones can rapidly adapt to environmental changes such as new fire detections, shifting wind patterns, and agent failures or limitations [8]. Such adaptability makes multi-agent systems particularly suitable for complex and unpredictable environments, including wildfire detection and management, disaster response, and emergency services.

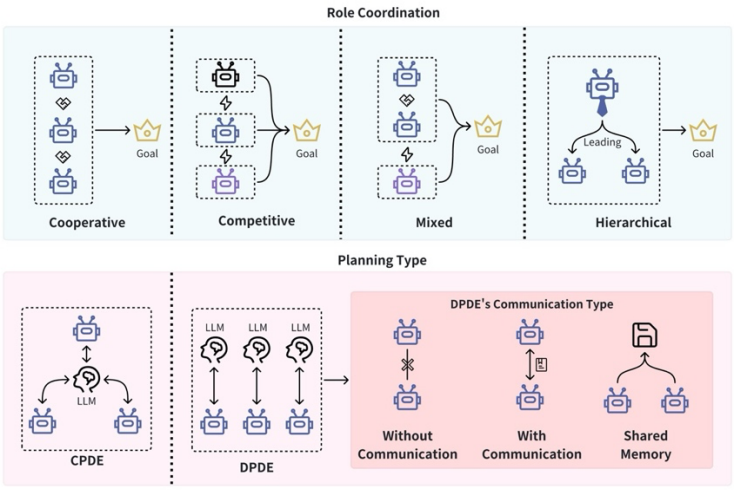

Figure 2.2: Communication flow between agents

In this dissertation, agents are modeled as autonomous drones that perform wildfire detection and monitoring. Each drone functions as an intelligent agent, equipped with sensors, communication modules, and decision-making algorithms to scan forests, detect fire outbreaks, and report their findings to emergency response systems.

2.2 Multi-Agent Systems

A multi-agent system consists of a group of autonomous agents interacting and collaborating within a shared environment to achieve specific objectives or solve problems [22]. Unlike traditional computational systems where a single centralized entity controls and directs task execution, MAS distributes decision-making capabilities across multiple independent entities, allowing each agent to operate autonomously yet in coordination with others [8].

MAS are used in application such as disaster management, surveillance, environmental monitoring, and defence that call for a high coordination and flexibility. Decentralisation, flexibility, fault tolerance and scalability are the characteristics of MAS that makes it so popular. When there is no single centralized controller, each agent acts locally and collaborates with others to achieve global efficiency. This is known as decentralisation. When MAS is implemented in real-world settings with multiple agents and tasks, its decentralised structure effectively scales by lowering the computational complexity and communication problems that are typical of centralised approaches [22]. The felxibility of multi-agent systems also lies to the fact that agents can adjust dynamically to the changes in enviroments, resource limitations, and unexpected events.

Multi-agents systems' most important feature is the ability of agents to communicate and interact with each other. Agents communicate with other agents using messages, exchanging information such as status of ongoing tasks. Agent communication follows a message-passing protocol with the goal of maintaining low communication overhead so that agents are able to coordinate their activities effectively while retaining autonomy. Typical message types in drone-based systems include task allocation updates, coalition formation proposals, resource availability notifications, and real-time environmental updates [19].

This distributed communication enables drones to form coalitions dynamically. A coalition is a temporary formation of agents that cooperate to obtain a specific collective objective.

2.3 Tasks in Multi-Agent Systems

Tasks in MAS represent the specific actions agents must complete to accomplish their overall objectives [7]. In any MAS cooperative framework, tasks define agents behaviour, and the objectives that the system must complete. In cooperative environments, agents share a common global goal, which is divided into separate tasks. Each task usually has particular requirements, and constraints that affect how and by which agent it can be completed [9].

Depending on the application context, tasks may differ based on the environment. For instance, tasks might be just collecting environmental data or scanning an area, or they could be as complex as cooperatively moving objects or jointly monitoring a dynamic area. All these tasks require capabilities in the executing agents, which include sensory accuracy, processing power, mechanical attributes, or communication capabilities. For instance, imagine surveillance missions requiring drones with high-resolution thermal cameras and enough battery life to monitor a large area for extended periods of time. On the other hand, cooperative missions may require agents with mechanical arms and enough strength, requiring specific coordination between multiple agents to successfully accomplish the task [7].

To ensure efficient coordination among agents, efficient task allocation is crucial in ensuring the capability of agents is matched with the task's requirements. Task allocation addresses this challenge by assigning agents to appropriate tasks according to their abilities, availability, resources, and the current context of operation [19]. Proper task allocation significantly improves system efficiency, enhances resource utilization, and maximizes overall mission performance.

However, there are scenarios where the available agents may not meet the existing tasks' requirements, resulting in infeasible assignments. For example, when there are numerous surveillance tasks but few drones with suitable sensing capabilities and sufficient battery life, some tasks will remain unassigned. This infeasibility emphasises the importance of balancing agent availability with task requirements. When feasible, agents dynamically reassign their assignments to identify task distributions that maximize an objective [9].

Performance in multi-agent task allocation is typically quantified using a measure known as utility. Utility is a numerical representation of the effectiveness or efficiency of a particular

assignment of agents to tasks [20]. Each potential agent-task allocation is mapped to a numerical score by utility functions, which show how well that arrangement achieves system objectives. This quantitative measure allows the system to compare and evaluate task allocations alternately in a structured way, guiding agents to optimal or near-optimal solutions.

Overall, tasks are core components of MAS, specifying the actions agents have to perform collectively. Proper task allocation that cautiously balances agent capability with task requirements, supported by properly defines utility metrics, goes a long way towards the performance and reliability of MAS, especially in dynamic, time-constrained applications.

2.4 Multi-Agent Task Allocation

Multi-agent task allocation (MATA) is a fundamental problem in artificial intelligence and robotics, where a group of autonomous agents collaboratively assign themselves to a set of tasks to optimize overall system performance. The problem is common in domains such as disaster response, surveillance, logistics, and industrial automation [7]. The challenge in multi-agent task allocation arises from various constraints, including task dependencies, resource limitations, and communication overhead among agents [9].

Early approaches to task allocation were centralized, using optimization techniques such as mixed-integer programming and constraint satisfaction to achieve globally optimal solutions [17]. While these centralized approaches provide strong guarantees on solution quality, they often suffer from scalability issues and communication bottlenecks in large-scale multi-agent systems. To address this, decentralized and distributed methods have gained popularity, enabling agents to make local decisions while coordinating their actions through negotiation or auction-based mechanisms [11].

Auction-based allocation has been a widely studied approach, where agents bid for tasks based on their capabilities and availability, leading to efficient task distribution with minimal communication overhead [11]. However, these methods often assume independent tasks, whereas real-world scenarios frequently involve interdependencies that require cooperative execution strategies [19]. To handle such dependencies, coalition formation techniques have been explored, where agents dynamically form groups to execute complex, interdependent tasks [15].

Coalition-based task allocation enables multiple agents to work together on a given task by using their collective capabilities. This is particularly useful in heterogeneous agent teams where different agents may possess unique skills or resources required for task completion [21]. Various models for coalition formation have been proposed, including negotiation-based and distributed constraint satisfaction approaches, which allow agents to iteratively adjust their task commitments based on utility maximization principles [12].

Another key aspect of multi-agent task allocation is **robustness and fault tolerance**. In dynamic environments, failures are inevitable, and an efficient allocation mechanism should be resilient to agent dropouts or task changes. To ensure continued task execution, modern approaches often incorporate self-adaptive behaviors that allow agents to autonomously reallocate tasks when disruptions occur. These capabilities are particularly important in real-world scenarios such as search-and-rescue operations, where environmental conditions and agent availability can change unexpectedly.

Recent advances have also explored Answer Set Programming (ASP) and constraint programming for solving MATA problems. Tools such as Clingo and MiniZinc provide powerful optimization capabilities, allowing for the formal modeling of task dependencies and constraints [7]. However, these centralized approaches struggle with scalability in large multi-agent environments. Decentralized methods that allow agents to locally compute task allocations while maintaining global coordination are becoming more viable, offering a balance between computational efficiency and solution quality [15].

Given the growing complexity of multi-agent systems, ongoing research continues to explore hybrid approaches that combine optimization techniques with decentralized coordination strategies. By using advances in artificial intelligence and distributed computing, multi-agent task allocation is evolving toward more adaptive, scalable, and autonomous frameworks capable of handling real-world challenges.

2.5 Coalition Formation

Coalition formation is defined as the temporary grouping of agents into coalitions to accomplish complex tasks, which would otherwise be unfeasible for a single agent due to their resource requirements or the necessity for diverse capabilities [18]. This concept is

essential in multi-agent systems where tasks often demand multiple agents with complementary skills to collaborate effectively.

Initially developed within the field of game theory, coalition formation has traditionally focused on rational agents seeking to maximize their own individual utilities through strategic coalitions [18,19]. Early solutions to coalition formation were primarily centralized and computationally intensive, often exhibiting exponential complexity with respect to the number of agents involved, thus limiting their applicability in real-world scenarios [19].

However, recent advancements in Distributed Artificial Intelligence (DAI) have significantly improved coalition formation approaches by emphasizing decentralized algorithms that distribute computational loads among agents, thus enhancing efficiency and scalability [1,19]. Such decentralized solutions are especially beneficial in cooperative multi-agent environments where agents collaboratively pursue common global goals rather than focusing solely on individual benefits.

In fully cooperative multi-agent systems—such as the drone-based wildfire detection scenario addressed in this dissertation—agents seek to form coalitions to maximize overall system utility rather than their individual utilities [2]. Here, utility represents the global effectiveness and efficiency of task execution achieved by the coalition structure. The coalition formation process involves dynamically assigning agents to coalitions, each responsible for a specific task. Consequently, each task is managed by its respective coalition, consisting of agents whose combined capabilities and resources satisfy the task's specific requirements [19].

Formally, coalition formation in cooperative multi-agent systems typically involves two main phases:

The first phase, known as coalition structure generation, determines the distribution of agents into coalitions. This step aims to identify optimal or near-optimal agent groupings that can collectively fulfill task requirements. The goal here is to form coalitions that maximize the global utility, considering constraints like agent capabilities, resources, communication overhead, and task complexity [15]. The second phase, known as coalition execution, involves coalitions carrying out their allocated tasks. Agents within coalitions coordinate closely, exchanging information, reallocating subtasks as necessary, and adapting dynamically to real-time conditions to optimize their collective performance. The

performance of each coalition directly influences the utility of the overall coalition structure [1].

Coalition formation is particularly suitable for modeling complex task allocation problems, especially when individual tasks require a precise combination of diverse capabilities and coordinated resources. In drone-based wildfire monitoring scenarios, coalition formation facilitates effective collaboration among drones with varied capabilities such as thermal imaging, extended flight endurance, communication relay functions, and accurate localization. Drones dynamically form coalitions based on real-time data to optimize coverage, responsiveness, and overall operational efficiency.

Therefore, the coalition formation paradigm provides a robust and flexible framework for addressing task allocation challenges in cooperative multi-agent environments. By leveraging decentralized coalition formation methods, multi-agent systems such as drone fleets deployed for wildfire monitoring can dynamically adapt and maintain effective cooperation even in highly dynamic and unpredictable operational contexts.

CHAPTER 3: Constraint Satisfaction and Answer Set Programming

3.1 Introduction

In this chapter, we introduce two declarative programming paradigms used extensively in this thesis to model and solve coalition formation problems: Constraint Satisfaction Problems (CSPs) and Answer Set Programming (ASP). These approaches are used to implement and evaluate centralized versions of the coalition assignment problem, where the objective is to find optimal or feasible task-agent assignments under a set of constraints.

The tools used for these models are MiniZinc, a high-level modeling language for CSPs, and Clingo, a solver for ASP. This chapter explains the core concepts behind these paradigms, highlights their differences, and illustrates how they were applied in the context of multi-agent task allocation.

3.2 Constraint Satisfaction Problems (CSP)

3.2.1 Definition

Constraint Satisfaction Problems (CSPs) represent a powerful class of combinatorial problems characterized by a set of variables, each of which must be assigned a value from a finite domain, subject to a collection of constraints. A formal CSP is defined as a triple (X, D, C) where:

- $X = \{x_1, x_2, \dots, x_n\}$ is a set of variables.
- $D = \{D_1, D_2, \dots, D_n\}$assigns each variable x_i a finite domain D_i of values it can take.
- $C = \{c_1, c_2, \dots, c_m\}$ is a set of constraints that define allowable combinations of values for subsets of variables.

A solution to a CSP is an assignment of values to all variables such that all constraints are simultaneously satisfied. In many applications, including the one addressed in this thesis, CSPs are extended with an objective function to form a Constraint Optimization Problem

(COP), where the goal is not only to find a feasible solution, but one that maximizes (or minimizes) a given utility measure.

CSPs are well-suited for problems where explicit constraints govern feasibility, and the objective can be described arithmetically or logically. They are widely used in scheduling, planning, resource allocation, and configuration problems.

3.2.2 Introduction to MiniZinc

MiniZinc is a high-level, solver-agnostic modeling language used to define constraint satisfaction and optimization problems. In the context of this thesis, it was employed to model the centralized version of the coalition formation problem for drone-based task allocation, allowing precise encoding of constraints and the definition of objective functions over agent-task assignments.

Below is a representative example directly inspired by the real implementation.

Example 1: Resource Constraint for Task Feasibility

```
% Resource A for single tasks
constraint forall(j in tasks) (
    sum(i in agents where agents_resA[i])(assignment[i] == j) >= tasks_resA[j]
);
```

This constraint ensures that for each task j, the total number of agents assigned who possess resource A is at least the required amount tasks_resA[j].

Example 2: Logical Constraint Involving Neighboring Tasks

```
(nb_tasks < 5 /\ exists(k in tasks)(
    (tasks_nearest[j] == k /\
        (sum(i in agents where agents_resA[i]) (assignment[i] == j) +
         sum(i in agents where agents_resA[i]) (assignment[i] == k)) >=
         tasks_resA[j] + tasks_resA[k] + 1)
))
\/
(nb_tasks >= 5 /\ exists(k, l in tasks)(
    (tasks_nearest[j] == k /\ tasks_nearest_sd[j] == l /\
        (sum(i in agents where agents_resA[i]) (assignment[i] == j) +
         sum(i in agents where agents_resA[i]) (assignment[i] == k) +
```

```
        sum(i in agents where agents_resA[i]) (assignment[i] == l)) >=
        tasks_resA[j] + tasks_resA[k] + tasks_resA[l] + 1)
))
```

This complex logical constraint activates different conditions depending on the number of tasks in the instance. For smaller task sets (less than 5), it ensures that a task and its nearest neighbor collectively receive enough resource A. For larger configurations, it ensures that a task and its two closest neighboring tasks together meet the cumulative resource requirement.

Objective Function: Maximizing Autonomy Utility

```
var float: obj_value =
    sum(i in agents)(autonomy[i]) * (1 - abs(assignment[i] - i) / num_tasks);

solve maximize obj_value;
```

This objective function aims to maximize the overall utility of the structure. It rewards agent-task assignments where agents are allocated tasks with spatial or index proximity (simulating reduced travel or logical alignment), scaled by their autonomy level. This encourages strategic grouping of more capable agents near demanding tasks.

3.3 Answer Set Programming (ASP)

Answer Set Programming (ASP) is a form of declarative programming used to solve complex search and reasoning problems, especially those involving combinatorics, constraints, and logic-based conditions. ASP differs from traditional imperative or procedural programming by allowing the programmer to describe the problem using logic-based rules, and letting the solver compute the solutions (known as "answer sets").

An ASP program defines:

- Facts: Ground truths, e.g., `agent(a1).`
- Rules: Conditions that generate conclusions from facts, e.g., `can_assign(A,T) :- agent(A), task(T), suitable(A,T).`
- Constraints: Rules that eliminate undesired solutions, e.g., `:- assigned(A,T1), assigned(A,T2), T1 != T2.`

- Optimization Directives: Instructions to maximize or minimize a function , e.g., `#maximize { U,A,T : utility(A,T,U) }.`

The most commonly used ASP solver today is Clingo, which was used in this thesis for centralized coalition structure generation.

In the context of this thesis, Answer Set Programming (ASP) was employed as a centralized approach to model and solve the coalition formation problem. Specifically, ASP was used to encode the assignment of agents to tasks under a rich set of domain constraints, such as autonomy levels, resource availability, and task requirements. This modeling allowed the problem to be expressed declaratively using logic-based rules, while the Clingo solver computed valid coalitions that maximized overall utility.

To represent the problem in ASP, a series of logical facts were used to define the basic elements of the system: agents, tasks, their positions, and their capabilities. For instance, each agent was described with facts indicating its autonomy, its available resources A and B, and its position in a 2D space. Similarly, each task was described by its location and a set of requirements, including minimum and maximum autonomy thresholds, minimum agent count, and required resources.

Building on this foundation, a set of rules was defined to describe valid assignments. A key rule ensured that each agent could be assigned to at most one task, thus maintaining exclusivity in agent-task relationships. Additional rules captured relationships such as distance between agents and tasks, enabling reasoning over spatial feasibility.

Crucially, constraints were introduced to enforce task feasibility. For example, one constraint ensured that a task could not be assigned unless it received a sufficient number of agents. Others required that the sum of the assigned agents' resources met the task's demands, and that at least one agent in a coalition possessed the autonomy necessary to complete the task safely. These constraints were written in ASP's expressive syntax and eliminated invalid configurations from the solution space.

To guide the solver toward high-quality solutions, a utility function was encoded using ASP's `#maximize` directive. This function typically rewarded coalitions composed of highly

autonomous agents with sufficient resources, thus aligning solution quality with the real-world goal of maximizing operational effectiveness.

Overall, ASP provided a flexible and powerful framework for representing coalition formation as a logical problem. Its ability to declaratively express constraints and compute optimal solutions made it ideal for benchmarking the performance of the decentralized methods developed later in the thesis. Although ASP does not scale as well as heuristic approaches for large problem instances, its strength lies in producing exact, interpretable results for moderate configurations—offering a valuable baseline for evaluation.

3.3.1 Introduction to Clingo

To solve the answer set programs defined in this thesis, we employed **Clingo**, a state-of-the-art solver for Answer Set Programming. Developed by the Potassco group at the University of Potsdam, Clingo integrates both a grounder (Gringo) and a solver (based on the Conflict-Driven Clause Learning paradigm) into a single tool. It enables users to define logic-based models using a declarative syntax and computes one or more answer sets that represent solutions satisfying all program rules and constraints.

Here is a simple Clingo program:

```
agent(a1; a2).
task(t1; t2).

% Only one assignment per agent
{ assigned(A,T) : task(T) } = 1 :- agent(A).

% Constraint: avoid same task for both agents
:- assigned(a1,T), assigned(a2,T).

#show assigned/2.
```

This program models two agents and two tasks. Each agent is assigned to exactly one task, and the second constraint ensures that a_1 and a_2 are assigned to different tasks. This results in one valid assignment: for example, `assigned(a1,t1)` and `assigned(a2,t2)`.

Clingo operates in two main stages. First, it grounds the logic program by expanding variables into concrete instances based on the input facts. This step transforms abstract rules

into a propositional form that can be processed by the solver. Then, in the second stage, the solver searches for stable models that satisfy all constraints and rules. These answer sets represent complete and consistent sets of literals that fulfill the program logic.

One of Clingo's key features is its support for optimization statements. These allow developers to not only find feasible solutions, but also to guide the solver to maximize or minimize an objective function. In this thesis, Clingo was used to maximize the total utility of agent-task assignments, ensuring that solutions were not only valid but also desirable in terms of performance.

3.3.2 Application in Thesis

The Clingo models used in this project were written as .lp files and executed through Python scripts that parsed the output. The models included sections to define agent and task data, assignment rules, resource and autonomy constraints, and optimization criteria. Once executed, Clingo returned the highest-utility coalition structure among all feasible configurations. This centralized formulation was important to establish a performance baseline against which the decentralized algorithms, FICSAM and IFICSAM [3], could be rigorously compared.

Clingo was chosen for its expressiveness, solver performance, and its ability to handle logical dependencies that would be cumbersome in constraint-based systems like CSPs. However, as the number of agents and tasks increased, the grounding size and search space also expanded rapidly. This limited the practical scalability of Clingo in very large problem instances. Nevertheless, for medium-sized configurations, Clingo produced globally optimal coalition structures that served as a benchmark for evaluating the performance of decentralized methods such as FICSAM and IFICSAM.

In summary, Clingo proved to be a powerful tool for modeling and solving the centralized coalition formation problem. Its logical syntax, combined with optimization capabilities, allowed for concise and readable problem encodings and enabled the generation of high-quality solutions for evaluation and comparison purposes.

CHAPTER 4: Problem Definition and Methodology

4.1 Problem Definition

4.1.1 Research Context and Motivation

Wildfires pose significant environmental, economic, and social threats, causing widespread damage, loss of natural resources, and endangerment to human life. Early and time effiecent detection of wildfires is essential, but traditional detection methods frequently have issues with coverage, responsivness, and adaptability to quickly shifting conditions.

Unmanned aerial vehicles (UAVs), commonly known as drones, have emerged as promising solution for wildfire monitoring due to their flexibility, rapid deployment, capabilities, and suitability for operation in inaccessible areas. Despite these advantages, effective wildfire detection and monitoring using drones require careful coordination among multiple agents, especially when covering large areas and dynamically changing environmental conditions. Efficient management and allocation of tasks among agents, ensuring optimal utilization of resources, coverage, and response times, present significant research challenges.

4.1.2 Formal Problem Statement

This dissertation explores the challenge of task allocation and coalition formation within cooperative drone-based wildfire detection systems, building upon the framework introduced by Ahmadoun in [3]. The problem can be formally defined as follows:

Consider a set of autonomous agents $A = \{a_1, a_2, \dots, a_n\}$, representing drones equipped with various sensors, communication capabilities, and computational resources. Each agent a_i has a set of specific capabilities defined by:

- Resources (e.g. battery life)
- Sensor types (e.g. thermal imaging, optical sensors).
- Communication range.
- Computational resources for decision-making.

These drones are deployed within a spatial region R partitioned into distinct areas requiring continuous monitoring for wildfire detection. A set of monitoring tasks $T = \{t_1, t_2, \dots, t_n\}$ are identified, each task t_j characterized by specific requirements such as:

- Area to be covered (geographic coordinates).
- Required resources.
- Number and types of drones required for successful execution.

Given these definitions, the coalition formation problem involves dynamically grouping subsets of agents into coalitions $C_k \subseteq A$ to perform specific tasks t_j. Each coalition C_k is characterized by the combined capabilities of its member drones, which must satisfy the task requirements. A valid coalition structure S is thus defined as:

$$S = \{(C_k, t_j) | C_k \subseteq A, t_j \in T, C_k \; satisfies \; requirments \; of \; t_j$$

The primary objective is to generate and continuously refine coalition structures S to optimize a predefined global utility function $U(S)$, representing the overall system performance. Formally, the goal is to find the coalition structure S^* that maximizes global utility:

$$S^* = arg \; max \; U(S)$$

subject to constraints:

- Capability Constraints: Each coalition must collectively fulfill all capability requirements of its assigned task.
- Resource Constraints: Agents have limited resources such as battery and computational power.
- Communication Constraints: Coalition formations and adjustments must minimize communication overhead.
- Real-time Constraints: Task allocation and coalition formation must adapt dynamically to real-time changes (e.g., evolving fire locations, agent failures).

4.1.3 Challenges Addressed

This research specifically addresses several critical challenges inherent to the described scenario:

- **Scalability:** As the number of agents and tasks grows, computational complexity increases exponentially. Traditional centralized coalition formation approaches struggle with real-time task allocation in such cases.
- **Communication Overhead:** High-frequency information exchange among agents can lead to significant delays and inefficient resource usage, negatively impacting real-time decision-making.
- **Dynamic Adaptability:** The system must respond swiftly and effectively to unexpected events such as the sudden spread of wildfires, drone failures, or environmental changes.
- **Efficiency:** In real-life scenarios the system must provide with time efficient solution

4.1.4 Research Objectives

The primary objective of this research is to design, implement, and evaluate a centralized coalition formation model using Answer Set Programming (ASP) with Clingo, specifically aimed at solving the interdependent task allocation problem in cooperative multi-agent systems. The focus is on modeling complex constraints and optimizing utility for agent-task assignments in scenarios such as wildfire detection and monitoring.

A key aspect of this study is the contrast between the newly implemented ASP-based Clingo models and existing centralized models developed using MiniZinc [3], a constraint satisfaction problem (CSP) modeling language. This comparison highlights the trade-offs between logic-based and constraint-based approaches in terms of expressiveness, performance, and scalability.

Additionally, the thesis conducts a comparative evaluation against the decentralized coalition formation method known as FICSAM, originally proposed by Ahmadoun (2022) [3]. While the decentralized method itself was not developed in this thesis, targeted improvements are explored to enhance its solution quality. The specific research objectives are to:

- Implement centralized coalition formation models using ASP (Clingo) and evaluate their effectiveness.
- Compare the ASP (Clingo) implementation with MiniZinc-based models to assess differences in performance, scalability, and solution quality.
- Investigate the scalability of the Clingo-based approach on large-scale, interdependent task allocation scenarios.
- Evaluate and contrast centralized and decentralized methods in terms of utility, efficiency, and adaptability.
- Explore improvements to the decentralized FICSAM algorithm to enhance its practical utility in dynamic environments.

4.2 System Model

4.2.1 Agent

In the drone-based wildfire detection scenario considered in this study [3], each agent represents an autonomous drone capable of performing pecific tasks based on its reousrces and capabilities. Formally, we define our agent set as:

$$A = \{a_1, a_2, \dots, a_n\}$$

Each agent a_i is uniquely characterized by a set of attributes extracted directly from the Python implementation [3].

$$a_i =< id, position, autonomy, resources\_a, resources\_b >$$

These attributes explicitly describe drone capabilities and resource constrains as follows:

- **id:** A unique identifier for each drone agent
- **position:** The current geographical location of the drone, representanted by coordinates (x, y) in the operational area
- **autonomy:** Represents the drone's battery life or operational endurance, crucial for task feasibility
- **resources_a & resources_b:** Specific resource capabilities that each drone carries, which are required for task execution

These attributes are instantiated in the Agent class contructor as follows:

```
class Agent(AgentBase):
    def __init__(self, id, position, autonomy, resources_a, resources_b):
        super().__init__(id)
        self.position = position
        self.autonomy = autonomy
        self.resources_a = resources_a
        self.resources_b = resources_b
```

Example 4.1 (Drone Agent Characteristics)

Consider a set of five drone agents:

$$A = \{a_1, a_2, a_3, a_4, a_5\}$$

These drones are scattered over a square grid of 10 km side length within a forest area. Each drone is characterized by its autonomy, position, and resources type. Table 4.1 summarizes their respective characteristics.

Table 4.1: Drone Agent Characteristics

Attribute	**Domain (D_k)**	a_1	a_2	a_3	a_4	a_5
Autonomy	[0,10]	8	4	6	4	7
Position	$[0,10]^2$	(0,0)	(5, 4)	(2, 8)	(9, 1)	(3, 5)
Resource	$\{A, B, AB\}$	A	AB	B	AB	A

This agent model was adopted from prior work [3] and integrated into the experimental model without modification, serving as the foundation for the centralized and decentralized coalition formation evaluations presented in this thesis.

4.2.2 Task

Tasks in the context of this study represent distinct wildfire detection and monitoring responsibilities that must be executed cooperatively by drone coalitions. These tasks are designed to simulate real-world operational demands in which multiple drones must

coordinate based on location, resource availability, and endurance constraints. The set of tasks is formally defined as:

$$T = \{t_1, t_2, \dots, t_m\}$$

Each task t_j is characterized by a set of attributes determining the requirments and constrains agents must satisfy to execute the task. Based on Python implementation (`task.py`), each task t_j is represented as:

$$t_j =< id, position, \text{min\_autonomy}, \text{max\_autonomy}, \text{max\_distance},$$

$$\text{resources\_a}, \text{resources\_b}, \text{min\_agents} >$$

Where the task attributes are defined as follows:

- **id:** Unique identifer of the task.
- **position:** Geographic coordinates where the task is located.
- **min_autonomy:** Minimum battery autonomy required for an agent to participate in the task.
- **max_autonomy:** Maximum battery autonomy to perform the task.
- **max_distance:** Maximum allowable distance from an agent's position to the task location.
- **resources_a:** Quantity of resource type A required.
- **resources_b:** Quantity of resource type B required.
- **min_agents:** Minimum number of agents required to execute the task.

These attributes are instantiated in the Task class constructor:

```
class Task(TaskBase):
def __init__(self, id, position, min_autonomy=0, max_autonomy=0,
max_distance=float("inf"), resources_a=0,resources_b=0, min_agents=0):
        super().__init__(id)
        self.position = position
        self.min_autonomy = min_autonomy
        self.max_autonomy = max_autonomy
        self.max_distance = max_distance
        self.resources_a = resources_a
        self.resources_b = resources_b
```

```
        self.min_agents = min_agents
```

Example 4.2 (Task Characteristics)

Consider a set of four tasks:

$$T = \{t_1, t_2, t_3, t_4\}$$

These tasks are scattered over a square grid of 10 km side length within a forest area. Each task is characterized by its attributes. Table 3.2 summarizes their respective characteristics.

Table 4.2: Tasks Characteristics

Attribute	**Domain (D_k)**	t_1	t_2	t_3	t_4
Position	$[0,10]^2$	(2,3)	(5.2, 4.7)	(1.5, 8)	(0, 2.7)
Min. autonomy	[0,100]	50	70	60	40
Max. autonomy	[0,100]	90	100	85	70
Max. distance	$[0, \infty)$	5	3	7	10
Resource A	$\mathbb{N}$	2	1	4	1
Resource B	$\mathbb{N}$	0	2	1	1
Min. agents required	$\mathbb{N}$	2	3	4	1

Like before, the task model was adopted from prior research [3] and incorporated into this thesis without modification. It provided the necessary structure for defining task requirements such as autonomy, resources, spatial constraints, and agent participation thresholds—forming the basis for both the centralized and decentralized coalition formation experiments conducted herein.

4.2.3 Mission and Scenario

In this system, a mission defined as a set of tasks that need to be performed in an environment, while a scenario joins the mission together with a group of agents that are

spread in a spatial grid. The scenario simulates simulates actual scenarios in which a group of drones need to self-organize and form coalition to complete the defined mission.

4.2.3.1 Mission Definition

A mission M is formally defines as a set of tasks $T = \{t_1, t_2, \dots, t_m\}$, generated procedurally with randomized but realistic constraints. Each task represents a discrete wildfire assignment, localized withing a 2D grid area of size $grid \times grid$.

In the implementation (`mission.py`), the Mission class is respnsible for initializing these tasks via the method:

```
def create_tasks(self, nb_tasks, nb_agents, grid):
```

This method:

- Randomly places each task in a unique position on the grid.
- Assigns autonomy requirements (min_autonomy, max_autonomy) within reasonable bounds.
- Randomizes resource requirements (resources_a, resources_b) based on the number of agents and tasks.
- Sets the minimum number of agents required to complete each task.

Formally, each mission M is defined as:

$$M = \{t_j = < pos_j, \text{min}a_j, maxa_j, resA_j, resB_j, \text{min\_}agents_j > | \text{j} = 1 \dots \text{m}\}$$

Where:

- pos_j: spatial location of the task t_j in the grid
- $mina_j, maxa_j$: required autonomy range
- $resA_j, reaB_j$: required amounts of reasource A and B
- $\text{min\_}agents_j$: minimum number of agents needed

This task generation method is bounded by grid size and number of agents, ensuring that the generated mission is both feasible and diverse.

4.2.3.2 Scenario Structure

A scenario S brings together:

1. A mission $M$with a defined number of tasks,
2. A team of agents $A = \{a_1, \ldots, a_n\}$ deployed on the grid.

The scenario simulates the environment in which the agents must coordinate to complete all tasks in the mission, foarming coalition dynamically.

This is in the Scenario class, which creates agents and tasks:

```
def create_scenario(self):
        tasks = self.mission.create_tasks(self.nb_tasks, self.nb_agents, self.grid)
        agents = self.team.create_agents(self.nb_agents, self.grid, tasks)
        agents = self.team.initialize_agents(self.mission.tasks)
        return agents, tasks
```

Example 4.3 (Scenario)

Lets us consider an example scenario with data from Table 3.1 and 3.2:

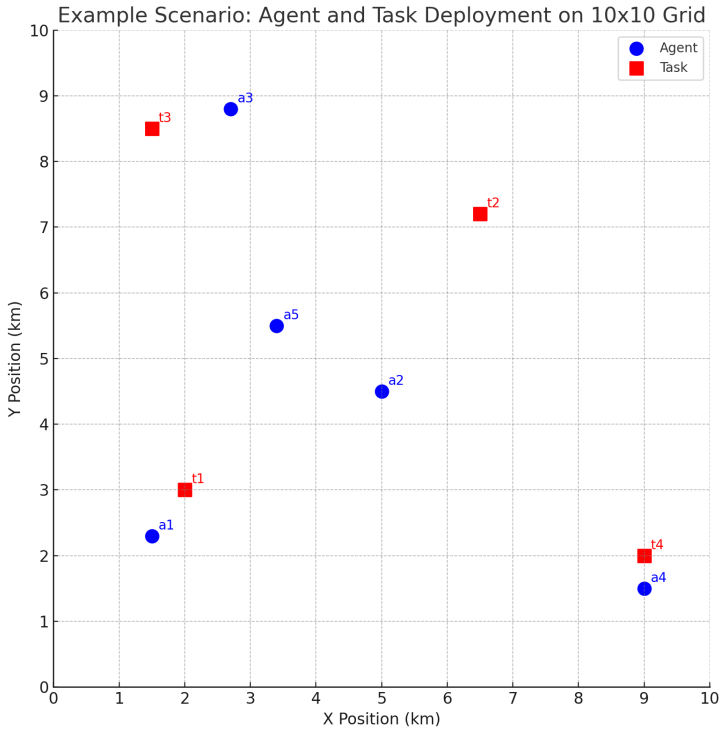

Figure 3.3: Spatial Scenario Deployment of Agents and Tasks

4.2.4 Team and Coalition

A team refers to the entire set of drone agents initialized for a mission, while a coalition refers to a temporary, task-specific subgroup of these agents formed to collaborativelly complete a given task.

4.2.4.1 Team Definition

Let the full team of agents denoted as:

$$A = \{a_1, a_2, \dots, a_n\}$$

Agents are distributed randomly in the operational area and made aware of the existing mission tasks. This prepares them to participate in coalition formation, where each agent decided locally whether it can contribute to a task.

4.2.4.2 Coalition Definition

A coalition $C_k \subseteq A$ is a dynamic, task-specific group of agents assigned to work collaboratively on a single task $t_j \in T$. Each coalition is constructes based on constraints defined in the task model (Section 3.2.2).

Formally, for a task t_j, the coalition C_j must satisfy:

- $\sum_{a_i \in C_j} resources\_a(a_i) \geq resources\_a(t_j)$
- $\sum_{a_i \in C_j} resources\_b(a_i) \geq resources\_b(t_j)$
- $\exists a_i \in C_j \; such \; that \; autonomy(a_i) \geq \max\_autonomy(t_j)$
- $\forall a_i \in C_j, autonomy(a_i) \geq \min\_autonomy(t_j)$
- $|C_j| \geq \min\_agents(t_j)$

Coalitions are dynamic and may be modified over time as agents reassess the utility of the global coalition structure and react to environmental changes. This is core to the anytime decentralized formation strategy described in later sections.

4.3 Methodological Approach

To address the task allocation problem under interdependent constraints, this thesis explores both centralized and decentralized computational paradigms. The centralized methodologies involve formal modeling approaches based on Constraint Satisfaction Problems (CSP) and Answer Set Programming (ASP), providing structured and globally optimized solutions. In contrast, the decentralized methodology builds on the previously developed agent-based anytime system known as FICSAM [3], in which autonomous agents iteratively form feasible coalitions based on local information and dynamic interactions. This dual approach enables a comprehensive evaluation of the trade-offs between optimality, scalability, and responsiveness in interdependent multi-agent task allocation.

4.3.1 Constraint Satisfaction Problem (CSP)

The initial formulation of the coalition formation problem was approached as a Constraint Satisfaction Problem (CSP), modeled using the MiniZinc language. The baseline model—originally developed in prior work [3]—was extended and refined as part of this thesis. Modifications included the addition of new constraints, refinement of variable definitions, and integration of additional solver back-ends to evaluate performance across a broader spectrum.

The objective was to assign each agent to at most one task while satisfying key feasibility constraints such as autonomy limits, resource availability, and spatial proximity to tasks. This formulation reflects a centralized optimization strategy, where the entire problem is solved globally using constraint reasoning.

Solvers used in this study included CBC, Gecode, Chuffed, and others, each offering unique performance characteristics that enabled a more comprehensive evaluation of the model's scalability and efficiency. The MiniZinc-based CSP implementation provided a valuable foundation and comparison point for the more expressive ASP-based models and the decentralized FICSAM method discussed in subsequent sections.

4.3.1.1 MiniZinc Model Description

The CSP model used in this study is defined in `model_centralized.mzn`, which consists of:

Agents and Tasks Parameters:

- **nb_agents, nb_tasks:** The number of agents and tasks.
- Arrays like **agents_autonomy, agents_resA, agents_resB,** and **distances** encode agent capabilities and spatial constraints.
- Task-specific requirments are captured via **tasks_autonomy_min, tasks_resA, tasks_reasB, tasks_agents_min,** etc.

```
array[agents] of int : agents_autonomy;
array[agents] of bool : agents_resA;
array[tasks] of int : tasks_autonomy_min;
array[tasks, agents] of int: distances;
```

Decision Variable: Each agent is assigned to a task or none (0 means no assignment).

```
array[agents] of var tasks0: assignment;
```

Constraints:

- **C0:** Ensure the required agents to execute the task
- **C1:** Each agent is mono-task (assigned to at most one task).
- **C2–C4:** Enforce minimal autonomy, at least one highly autonomous agent per task, and maximum distance limits.
- **C5–C7:** Ensure agents with resources A/B meet task requirements.
- **C8:** Each task must receive the minimum required number of agents.

```
% C0
constraint nb_agents >= sum(j in tasks)(tasks_agents_min[j]);

% C2: Minimal autonomy
constraint forall (j in tasks) (
   forall (i in agents)
      (assignment[i] == j -> agents_autonomy[i] >= tasks_autonomy_min[j]));

% C3: Autonomy threshold for one agent at least
constraint forall (j in tasks) (
   tasks_agents_min[j] > 0 -> (exists(i in agents)(assignment[i] == j  /\
agents_autonomy[i] >= tasks_autonomy_one[j]))
);
```

```
% C4: Maximal distance
constraint forall (j in tasks) (
   forall (i in agents)
      (assignment[i] == j -> distances[j, i] <= tasks_distance_max[j])
);

% C5: Resources A for single tasks
constraint forall (j in tasks) (
    sum(i in agents where agents_resA[i])(assignment[i] == j) >= tasks_resA[j]
);

% C5': Resources A for single tasks
constraint forall (j in tasks where j mod 2 == 1) (
    (sum(i in agents where agents_resA[i])(assignment[i] == j) mod 2 ) == 1
);

% C6: Resources B for single tasks
constraint forall (j in tasks) (
    sum(i in agents where agents_resB[i])(assignment[i] == j) >= tasks_resB[j]
);

% C7: Resources A for couples of nearest tasks
constraint forall (j in tasks where tasks_resA[j] > 0) (
    (nb_tasks < 5 /\ exists(k in tasks)
      (tasks_nearest[j] == k /\
        (sum(i in agents where agents_resA[i]) (assignment[i] == j) +
         sum(i in agents where agents_resA[i]) (assignment[i] == k)) >=
            tasks_resA[j] + tasks_resA[k] + 1))
  \/ (nb_tasks >= 5 /\ exists(k, l in tasks)
      (tasks_nearest[j] == k /\ tasks_nearest_sd[j] == l /\
        (sum(i in agents where agents_resA[i]) (assignment[i] == j) +
         sum(i in agents where agents_resA[i]) (assignment[i] == k) +
         sum(i in agents where agents_resA[i]) (assignment[i] == l)) >=
            tasks_resA[j] + tasks_resA[k] + tasks_resA[l] + 1))
);

% C8: Minimal number of agents
constraint forall (j in tasks) (
    sum(i in agents)(assignment[i] == j) >= tasks_agents_min[j]
);
```

Objective Function: The obj variable computes a weighted utility based on autonomy margins, distance constraints, resource coverage, and agent participation.

```
obj = (count([exists(j in tasks)(assignment[i] == j /\ agents_autonomy[i] >=
tasks_autonomy_min[j] + margin_autonomy_min) | i in agents]) +
       count([exists(j in tasks)(assignment[i] == j /\ distances[j, i] <=
tasks_distance_max[j] - margin_distance_max) | i in agents]) +
       count([agents_resA[i] /\ assignment[i] != 0 | i in agents]) +
       count([agents_resB[i] /\ assignment[i] != 0 | i in agents]) +
       2*count([assignment[i] != 0 | i in agents])
      ) * nb_tasks +
      (count([count([assignment[i] == j /\ agents_autonomy[i] >=
tasks_autonomy_one[j] | i in agents]) >= tasks_agents_min[j] | j in tasks]) +
       count([count([assignment[i] == j | i in agents]) >= tasks_agents_min[j] +
margin_nb_agents | j in tasks])
      ) * nb_agents;
```

Solver mode: The problem is solved to find the maximum utility using:

```
solve maximize obj;
```

4.3.1.2 Python Integration & Data Feeding

The model is fed through a JSON data file, `data_centralized.json`, which maps directly to MiniZinc's parameters. An example is:

```
{
  "nb_agents": 5,
  "nb_tasks": 2,
  "agents_autonomy": [1, 10, 9, 4, 8],
  "agents_resA": [true, true, true, true, true],
  "agents_resB": [true, false, false, false, true],
  ...
}
```

Python uses the `minizinc-python` API to load the model and inject data:

```
import minizinc

    solver = minizinc.Solver.lookup(MYSOLVER)
    model = minizinc.Model("model_centralized.mzn")
    instance = minizinc.Instance(solver, model)
    instance.add_file("centralized.json")
```

The result includes agent-task assignments, objective utility, and feasibility.

4.3.1.3 Advantages of the CSP Approach

The MiniZinc-based modeling approach adopted in this thesis offers several distinct advantages, particularly in the context of centralized coalition formation. Its declarative syntax and solver independence make it well-suited for experimenting with complex constraint formulations.

- **Declarative and Structured**: MiniZinc allows the problem to be expressed in a high-level, human-readable format, separating the logic of the problem from the solving strategy. This clarity is especially useful for representing coalition feasibility constraints and inter-agent dependencies in a structured and maintainable way.
- **Modular Objective Design**: The modularity of MiniZinc models enables rapid experimentation with different objective functions. Utility definitions and coalition preferences can be adjusted independently of the constraint logic, facilitating the exploration of alternative optimization policies without restructuring the model.
- **Solver Agnostic**: One of MiniZinc's major strengths is its compatibility with a variety of constraint solvers, including Gecode, Chuffed, and CBC. This flexibility allows researchers to switch solvers depending on performance needs or to validate solution robustness across different solving paradigms.

Together, these features make MiniZinc a powerful and flexible tool for prototyping and evaluating centralized coalition formation strategies in multi-agent systems.

4.3.1.4 Limitations Identified

Despite its strengths, the use of MiniZinc in coalition formation also presents several limitations, particularly when transitioning to large-scale or decentralized environments.

- **Scalability**: One of the primary limitations of MiniZinc lies in its computational scalability. As the number of agents and tasks increases, the complexity of the constraint satisfaction problem grows exponentially. This results in significantly longer solve times, especially in configurations with dense interdependencies or numerous feasible coalitions. For real-time or large-scale applications, this becomes a critical bottleneck.

- **Rigid Logic for Dynamic Reasoning**: Although MiniZinc supports rich constraint modeling, it lacks native support for dynamic or stateful reasoning. Modeling behaviors such as iterative improvement, agent negotiation, or sequential decision-making—which are common in multi-agent systems—can be cumbersome and unintuitive. For example, expressing token-passing protocols or adaptive coalition adjustments requires artificial encodings or external control logic, which are more naturally expressed in logic programming frameworks like ASP or procedural systems.
- **Global Solving Paradigm**: MiniZinc is inherently centralized. It requires complete knowledge of all agents, tasks, and constraints at the outset and solves the entire problem in one global optimization step. While this is effective for finding optimal solutions, it is unsuitable for settings where decentralized agents must make decisions based on partial knowledge or under real-time constraints. This limitation makes MiniZinc less applicable in scenarios demanding anytime behavior, adaptivity, or distributed coordination.

In summary, while MiniZinc excels at clearly modeling and solving centralized coalition problems, its limitations in scalability, dynamic reasoning, and decentralized execution highlight the need for complementary approaches in broader multi-agent system contexts.

4.3.2 Answer Set Programming (ASP)

To address the limitations of Constraint Satisfaction Problems (CSP) in scaling to large instances, this thesis explored an alternative centralized methodology using Answer Set Programming (ASP). ASP is a form of declarative programming oriented toward difficult combinatorial search problems, well-suited for knowledge representation and reasoning tasks. The ASP-based method was implemented using the Clingo solver, which combines a high-performance grounder and solver for logic programs under stable model semantics.

4.3.2.1 Overview of ASP Implementation

The ASP approach uses logic rules and constraints to describe the valid configurations of agent-task assignments. Each agent is assigned to at most one task, and multiple global and local feasibility conditions must be satisfied to form a valid coalition structure.

The ASP model is divided into three main components:

1. **Logic rules and constraints** (`model_centralized.lp`)
2. **Problem instance data** (`centralized_data.lp`, generated by Python)
3. **ASP solving interface** (`myclingo.py`)

ASP Logic Model (`model_centralized.lp`)

The ASP model defines the search space and constraints as follows:

Assignment Rule:

```
{ assignment(A,T) : task(T) } :- agent(A).
:- assignment(A,T1), assignment(A,T2), T1 != T2.
```

Each agent may be assigned to one task (or none, if allowed). The second line ensures uniqueness by disallowing multiple task assignments for a single agent.

Constrains:

C0: Global minimum agents requirement

```
total_min_agents(Sum) :- Sum = #sum { M, T : tasks_agents_min(T, M) }.
:- nb_agents(N), total_min_agents(Sum), Sum > N.
```

This constraint checks whether the total number of available agents is enough to cover the minimum required agents for all tasks. If the required sum is greater than the available agents, the answer set is invalid.

C2: Autonomy and Distance Constraints

```
:- assignment(A,T), agents_autonomy(A,X), tasks_autonomy_min(T,Y), X < Y.
:- assignment(A,T), distance(T,A,D), tasks_distance_max(T,MaxD), D > MaxD.
```

- **Line 1**: An agent cannot be assigned to a task if its autonomy is less than the task's minimum autonomy requirement.
- **Line 2**: An agent cannot be assigned to a task if it is too far from the task's location (exceeds MaxD).

C3: At least one highly autonomous agent per task.

```
:- tasks_agents_min(T, N), N > 0, #count { A : assignment(A,T),
agents_autonomy(A,X), tasks_autonomy_one(T,Y), X >= Y } < 1.
```

If a task requires at least one highly autonomous agent (above a secondary threshold), and the task is assigned any agents, then at least one must satisfy that higher autonomy level.

C5 & C6: Resource Constraints

```
:- tasks_resA(T,R), R > #count { A : assignment(A,T), agents_resA(A) }.
:- tasks_resB(T,R), R > #count { A : assignment(A,T), agents_resB(A) }.
:- tasks_agents_min(T,N), N > #count { A : assignment(A,T) }.
```

- Each task has a resource requirement.
- If fewer agents with resource A or B are assigned than required, the solution is invalid.

C5': For every odd task, the number of Resource A agents must be odd.

```
count_resA(J, Count) :- task(J), Count = #count { I : assignment(I,J),
agents_resA(I) }.
:- oddTask(J), count_resA(J, Count), not odd(Count).
```

For tasks with an odd index, the number of agents assigned to that task with resource A must also be odd.

C7: Resources A for couples of nearest tasks

```
% Case 1: For nb_tasks < 5.
valid_nearest_pair(T) :-
    tasks_nearest(T, K),
    count_resA(T, S1), count_resA(K, S2),
    tasks_resA(T, R1), tasks_resA(K, R2),
    S1 + S2 >= R1 + R2 + 1.
:- nb_tasks(NT), NT < 5, tasks_resA(T, R), R > 0, not valid_nearest_pair(T).

% Case 2: For nb_tasks >= 5.
valid_nearest_triple(T) :-
    tasks_nearest(T, K), tasks_nearest_sd(T, L),
    count_resA(T, S1), count_resA(K, S2), count_resA(L, S3),
    tasks_resA(T, R1), tasks_resA(K, R2), tasks_resA(L, R3),
```

```
    S1 + S2 + S3 >= R1 + R2 + R3 + 1.
:- nb_tasks(NT), NT >= 5, tasks_resA(T, R), R > 0, not valid_nearest_triple(T).
```

If the number of tasks is less than 5, a task and its closest neighbor must collectively have enough resource A agents to exceed the combined requirement by at least one unit. If there are five or more tasks, the resource requirement is extended to include two nearest neighbors. Their combined assigned resource A agents must exceed the total requirement by at least one.

Objective Function: The goal is to maximize a utility score composed of two parts: agent-level and task-level satisfaction.

Agent-Level Compenents:

- High autonomy
- Distance compliance
- Possession of required resources
- Total assignments

Task-Level Components:

- Tasks having at least one high-autonomy agent
- Tasks that exceed the minimum agent requirment

Then objective value is computed and the program is solved:

```
% --- Objective Function ---

%%%%% Agent-Level Counts %%%%%

obj_autonomy(S1) :-
    S1 = #count { A : agent(A), task(T), assignment(A,T),
                  agents_autonomy(A,X), tasks_autonomy_min(T,Y), X >= Y + 2 }.

obj_distance(S2) :-
    S2 = #count { A : agent(A), task(T), assignment(A,T),
                  distance(T,A,D), tasks_distance_max(T,MaxD), D <= MaxD - 20 }.

obj_resA(S3) :-
    S3 = #count { A : agent(A), task(T), assignment(A,T), agents_resA(A) }.
```

```
obj_resB(S4) :-
    S4 = #count { A : agent(A), task(T), assignment(A,T), agents_resB(A) }.

obj_assignment(S5) :-
    S5 = #count { A : agent(A), task(T), assignment(A,T) }.

obj_part1(P1) :-
    obj_autonomy(S1), obj_distance(S2), obj_resA(S3), obj_resB(S4),
obj_assignment(S5),
    P1 = S1 + S2 + S3 + S4 + (2 * S5).

%%%%% Task-Level Counts %%%%%

% Helper: high autonomy condition per task.
high_autonomy_ok(T) :-
    task(T),
    tasks_agents_min(T,R), R > 0,
    #count { A : agent(A), assignment(A,T), agents_autonomy(A,X),
tasks_autonomy_one(T,O), X >= O } >= R.

obj_task1(S6) :-
    S6 = #count { T : task(T), high_autonomy_ok(T) }.

% Helper: total agents condition per task.
enough_agents(T) :-
    task(T),
    tasks_agents_min(T,R),
    margin_nb_agents(MNB),
    #count { A : agent(A), assignment(A,T) } >= R + MNB.

obj_task2(S7) :-
    S7 = #count { T : task(T), enough_agents(T) }.

obj_tasks(P2) :-
    obj_task1(S6), obj_task2(S7),
    P2 = S6 + S7.

%%%%% Final Objective %%%%%

obj_value(N) :-
    obj_part1(P1), obj_tasks(P2),
    nb_tasks(NT), nb_agents(NA),
    N = (P1 * NT) + (P2 * NA).
```

```
#maximize { N : obj_value(N) }.
```

Data Generation (`centralized_data.lp`)

The problem-specific data for each scenario is encoded dynamically in the `centralized_data.lp` file. This includes all agent and task attributes, such as this example for 5 agents and 2 tasks:

```
agent(1..5).
task(1..2).
nb_agents(5).
nb_tasks(2).
margin_autonomy_min(2).
margin_distance_max(20).
margin_nb_agents(1).
agents_autonomy(1, 1).
agents_resA(1).
agents_resB(1).
agents_autonomy(2, 10).
agents_resA(2).
agents_autonomy(3, 9).
agents_resA(3).
agents_autonomy(4, 4).
agents_resA(4).
agents_autonomy(5, 8).
agents_resA(5).
agents_resB(5).
distance(1,1,20).
distance(1,2,12).
distance(1,3,73).
distance(1,4,83).
distance(1,5,70).
distance(2,1,15).
distance(2,2,23).
distance(2,3,77).
distance(2,4,103).
distance(2,5,75).
tasks_autonomy_min(1,2).
tasks_autonomy_one(1,5).
tasks_distance_max(1,140).
tasks_agents_min(1,1).
tasks_resA(1,0).
tasks_resB(1,0).
```

```
tasks_nearest(1,2).
tasks_nearest_sd(1,1).
tasks_autonomy_min(2,1).
tasks_autonomy_one(2,2).
tasks_distance_max(2,132).
tasks_agents_min(2,1).
tasks_resA(2,0).
tasks_resB(2,0).
tasks_nearest(2,1).
tasks_nearest_sd(2,2).
oddTask(1).
odd(1).
odd(3).
```

Python Integration (`myclingo.py`)

To implement the centralized coalition formation approach, a full pipeline was developed using Answer Set Programming (ASP) with Clingo. The process involves three major components:

1. Data Generation: A Python script was created to dynamically convert agent and task instances into ASP facts. These facts capture agent attributes (e.g., autonomy, resources), task requirements (e.g., autonomy bounds, resource needs), and relational data such as distances between agents and tasks. The script also precomputes the nearest and second-nearest task for each task, which can assist the ASP model with neighborhood reasoning. The resulting facts are saved into a `.lp` file used as input for Clingo.
2. ASP Solver Integration: The Clingo solver is invoked using a Python wrapper. It loads both the static ASP logic (`model_centralized.lp`) and the problem-specific data file. The solver is configured with a parallel optimization strategy and a custom `on_model` callback, which extracts each computed answer set during solving. A timeout is used to ensure the solver does not run indefinitely on complex instances.
3. ASP Output Parsing: Once a solution is found, a custom parser processes the output atoms (symbols). It reconstructs the assignment of agents to tasks and extracts the optimized objective value. The output is normalized to a utility score between 0 and 1, facilitating comparison across problem sizes and methods.

This modular setup allows flexible experimentation with various problem instances and ASP encodings. The full implementation of these components is provided in the Appendix (see Appendix A).

4.3.2.2 Evaluation of ASP with Clingo

Answer Set Programming (ASP), particularly through the Clingo solver, offers a powerful and expressive framework for solving combinatorial and logic-based problems. One of its primary advantages is its high expressiveness: it allows complex constraints and interdependencies to be encoded in a concise and readable manner. This expressiveness is especially beneficial in coalition formation scenarios, where rules involving resources, autonomy, and task feasibility must be tightly enforced. Additionally, Clingo supports built-in optimization capabilities through directives such as #maximize, which allow the model to search directly for utility-optimal solutions without the need for external scoring logic.

Another key benefit of Clingo is its relative scalability compared to traditional Constraint Satisfaction Problem (CSP) approaches. While CSP solvers often suffer from performance degradation as the number of agents and tasks increases, Clingo manages larger problem instances more efficiently by leveraging advanced techniques such as constraint propagation and the Vmtf (Variable Move-To-Front) heuristic for decision-making during the solving phase.

However, the use of Clingo also comes with limitations. One notable drawback is the grounding bottleneck: as problem size increases, the time required to ground the abstract ASP rules into concrete instances can grow significantly, sometimes becoming a limiting factor before the solving phase even begins. Moreover, while Clingo provides support for various language extensions, it is not inherently designed to handle real-time dynamics, asynchronous agent behavior, or probabilistic conditions. Incorporating such features often requires non-trivial workarounds or external integration with other systems, limiting its applicability in dynamic and real-world environments where decisions must adapt over time.

In summary, Clingo is a highly expressive and efficient tool for modeling centralized coalition formation problems, especially when optimality and clear logical structure are priorities. Nevertheless, its limitations in handling scalability and real-time adaptability must be considered when choosing it for complex multi-agent applications.

4.3.2.3 Summary

In summary, ASP with Clingo serves as a powerful centralized solution for coalition formation problems, particularly when the scenario involves strict combinatorial constraints and the solution space can be exhaustively explored. Its integration into this study demonstrated strong performance in moderate-sized problem instances, often outperforming classical CSP in both runtime and utility optimization.

4.3.3 Feasible Interdependent Coalition Structure Anytime Method (FICSAM)

To address the challenges of dynamic, decentralized, and efficient task allocation among drone agents, this dissertation employs the Feasible Interdependent Coalition Structure Anytime Method (FICSAM), originally proposed in the doctoral [3]. FICSAM enables agents to collaboratively and iteratively construct feasible coalition structures through a distributed message-passing protocol, without relying on centralized control.

To further improve coalition quality, this study also applies the enhanced method known as IFICSAM (Improved FICSAM), originally introduced in prior work [3], which incorporates dynamic local optimization strategies based on utility-driven decision-making. Together, FICSAM and IFICSAM represent the core decentralized methodologies leveraged in this research for coalition formation under interdependent constraints.

4.3.3.1 FICSAM Overview

FICSAM is a decentralized, anytime method where each agent maintains its view of the current coalition structure, evaluates feasibility, and collaborativelly updates the structure by passing a token among agents.

Each agent takes turns modifying the structure based on the feasibility validation.

This process continues until convergence is reached or until no further improvments are possible.

FICSAM Algorithms Steps:

1. Initialization:
Each agent initializes its local state, including:

- A list of available tasks.
- Knowledge of potential teammates.
- An initial coalition structure (typically empty or randomly seeded).

2. First Round – Feasibility Search:
Each agent performs a feasibility check:

- If the current coalition structure is invalid, the agent invokes a centralized CSP solver (global_csp) to construct a new feasible structure.
- If successful, the agent broadcasts this structure to others using an AnytimeMessage.

3. Token Passing:
After evaluating or updating the structure, the agent passes control to the next agent via a TokenMessage. The selection of the next agent can follow either a predefined order (e.g., nearest-neighbor) or a randomized scheme.

5. Convergence Check:
Agents monitor how long the coalition structure has remained unchanged (using an internal counter). If no changes occur across all agents for a certain number of rounds, they broadcast a StopMessage, which halts the process and finalizes the structure.

6. Finalization:
All agents adopt the final coalition structure and return a success or failure status.

Example 4.4 FICSAM execution example

To better understand the operation of the system by Ahmadoun [3] , we present a step-by-step execution of the FICSAM algorithm on a small but representative scenario.

We consider a multi-agent setting involving four agents: $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_3}$, and $\mathbf{a_4}$, and two tasks: $\mathbf{t_1}$ and $\mathbf{t_2}$. The agents and tasks are configured with different properties to reflect realistic constraints. The positions, autonomy levels, and resource capabilities of the agents are as follows: agent $\mathbf{a_1}$ is located at (0,0), with autonomy 10, possessing one unit of resource A; $\mathbf{a_2}$ is located at (3,3), autonomy 5, and holds one unit of resource B; $\mathbf{a_3}$ is positioned at (1,2), autonomy 7, with a unit of resource A available; and $\mathbf{a_4}$ is placed at (5,3), autonomy 8, with one unit of resource A.

On the task side, $\mathbf{t_1}$ is situated at (3,0) and requires a minimum of one agent, a minimum autonomy of 3, a maximum distance of 4, and at least one unit of resource A. $\mathbf{t_2}$, positioned at (4,4), requires two agents, with minimum autonomy of 7 and maximum distance from the agent 10, and at least one unit of resource A and B.

Initially, the system enters the first stage governed by the **FICSAM** [3] procedure. In this phase, agents sequentially receive the token and attempt to build a feasible coalition structure without aiming to optimize utility. The process halts once a feasible structure is found.

Agent $\mathbf{a_1}$ receives the token first and attempts to join task $\mathbf{t_1}$, creating a preliminary structure $S = \{t_1: [a_1]\}$ with u_global(S) = 0.45. Although the agent meets the task requirments the coalition is infeasible. The token is then passed to $\mathbf{a_2}$, who joins $\mathbf{t_1}$, forming $S = \{t_1: [a_1, a_2]\}$. The structure's global utility is evaluated as u_global(S) = 0.55. This becomes the current best solution (S_max), and the token is passed to $\mathbf{a_3}$.

Agent $\mathbf{a_3}$ attempts to contribute by selecting task $\mathbf{t_2}$. $\mathbf{a_3}$ joins $\mathbf{t_2}$, forming $S = \{t_1: [a_1, a_2], t_2: [a_3,]\}$. The structure is infeasible but meets all the requirements of $\mathbf{t_1}$. The utility improves slightly to u_global(S) = 0.65, so **S_max** is updated accordingly.

Finally, agent $\mathbf{a_4}$ receives the token and looks for a feasible coalition structure. As it knows all the information from all the agents, it finds one, for example $S = \{t_1: [a_1], t_2: [a_3, a_4]\}$ with global utility of u_global(S) = 0.7. Both coalitions now satisfy their corresponding task constraints, including autonomy levels, resource requirements, and the number of agents. FICSAM terminates at this point, and the structure is shared among the agents using StopMessage.

4.3.3.2 Message Protocol

Three types of structured messages coordinate agent communication:

Message Type	Purpose	Triggered When
TokenMessage	Passes control to another agent to modify the structure	After decision/update

AnytimeMessage	Shares an improved feasible found during an agent's decision process	After successful local improvement
StopMessage	Signals termination when convergence is detected (success or failure)	When no further improvments are possible

All messages are derived from the base `Message` class, containg sender and receiver fields.

4.3.4 Improved Feasible Interdependent Coalition Structure Anytime Method (IFICSAM)

The Improved Feasible Interdependent Coalition Structure Anytime Method **(**IFICSAM**)** extends the FICSAM procedure by incorporating a second phase aimed at improving the utility of an already feasible coalition structure. Once a valid structure has been established in round 1 (via FICSAM), agents enter an anytime improvement loop in subsequent rounds, attempting to locally adjust the coalition composition to maximize the global utility u_global.

Unlike the first phase, which prioritizes feasibility, this phase focuses on incremental utility optimization while maintaining feasibility. The IFICSAM model supports various improvement strategies through replacements or swaps between single or groups of agents. Improvements are broadcast to other agents using AnytimeMessages. If no further improvement is detected after all agents have taken their turn, a StopMessage is issued to conclude the process.

Example 4.5 IFICSAM execution example

Following the successful identification of a feasible coalition structure in the first phase (example 4.4), the system transitions into the second stage, governed by the Improved Feasible Interdependent Coalition Structure Anytime Method (IFICSAM). This stage aims to enhance the quality of the solution by incrementally improving the global utility of the coalition structure found in FICSAM.

Unlike the feasibility-focused first round, IFICSAM assumes that a feasible solution has already been reached and initiates a utility-driven optimization process. Each agent receives the token again and, in turn, evaluates potential improvements to the current structure. Agents

explore local modifications such as moving to another task, swapping with another agent, or reassigning coalitions, provided that the resulting structure remains feasible. If a local change increases the overall utility, the agent updates the structure and informs the other agents through an `AnytimeMessage`.

We continue with the same scenario involving agents $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_3}$, and $\mathbf{a_4}$, and tasks $\mathbf{t_1}$ and $\mathbf{t_2}$, as defined in the previous section. The structure produced by FICSAM was: S = {t_1: [a_1], t_2: [a_3, a_4]}. This configuration was feasible and yielded a global utility of u_global(S) = 0.7. It now serves as the baseline for the improvement phase.

The token first returns to agent $\mathbf{a_1}$, who is currently part of $\mathbf{t_1}$. Agent $\mathbf{a_1}$ analyzes whether relocating to another task or initiating a swap could lead to an increase in global utility. However, moving to $\mathbf{t_2}$ would decrease the utility to 0.55. Then it tries swapping with $\mathbf{a_3}$, the new structure is S = {t_1: [a_3], t_2: [a_1, a_4]} with u_global(S) = 0.75 which is greater than the previous one. Return from the improvement step $\mathbf{a_1}$ checks if the the structure is still feasible. As it is, and informs $\mathbf{a_3}$ and $\mathbf{a_4}$.

Agent $\mathbf{a_3}$ receives the token and begins its decision-making process based on the current coalition structure. Since a feasible structure has already been identified and shared, $\mathbf{a_3}$ evaluates whether any further improvement can be made. However, after attempting local optimization using the selected improvement strategy, no better configuration is found. As a result, $\mathbf{a_3}$ retains the existing coalition structure.

Recognizing that no changes have occurred during its turn, $\mathbf{a_3}$ passes the token to agent $\mathbf{a_4}$. At this stage, all agents are informed of the current best-known configuration. With no further improvements detected, the process concludes, and the system finalizes an improved and feasible coalition structure: S = {t_1: [a_3], t_2: [a_1, a_4]}

4.3.5 Additional Experimental Evaluation

To further investigate the performance limits of decentralized coalition improvement in IFICSAM, several experimental variations of the original improvement functions were developed and tested. These prototypes aimed to address observed utility plateaus and decision latency that emerged in larger or more interdependent task allocation scenarios. While none of the experimental methods outperformed the original approaches in terms of

utility or stability, they offered valuable insights into the complexity and scalability trade-offs in decentralized coalition formation.

Baseline Methods:

The baseline implementations included improve_csp() and improve_greedy(). These functions respectively applied a local CSP-based search over agent subsets and a greedy heuristic that evaluated task switching and one-to-one swaps to improve global utility. These methods served as the reference for evaluating all subsequent experimental enhancements.

Greedy Extension: improve_greedy2

To make the greedy search more robust, a revised method was introduced under the name improve_greedy2. This version implemented an iterative improvement loop where agents explored task switches and agent swaps until no further utility gain was detected. Despite its exhaustive behavior, the method often stagnated due to local optima and yielded no significant improvement over the original greedy baseline.

CSP Extensions

Three alternative CSP-based improvement strategies were developed:

- improve_csp_new: This variant introduced a utility caching system and an adaptive iteration cap per task, aiming to avoid redundant evaluations and fine-tune coalition construction dynamically. Although computationally efficient, the utility gains remained comparable to the baseline.
- improve_csp2: This approach applied a beam search mechanism to restrict the search to the top-k scoring agent subsets for each task. It improved performance speed but did not yield better coalition quality.
- improve_csp3: Based on the best subset beam strategy, this method explored near-optimal subsets for each task independently using a beam width of 10. It aimed to balance exploration depth and breadth but showed no measurable advantage over the original CSP implementation in practice.

Conclusion

Although these experimental prototypes did not outperform the baseline methods in practice, they highlighted key bottlenecks and trade-offs in decentralized utility-driven coalition formation. The findings suggest that local search enhancements, even with added complexity or parallelization, may quickly plateau in utility due to the decentralized and interdependent nature of the problem. Future directions may benefit from hybridizing these techniques or integrating learning-based heuristics to guide coalition restructuring more effectively.

4.3.6 Summary

The Feasible Interdependent Coalition Structure Anytime Method (FICSAM) is a decentralized protocol designed [3] to find a feasible task allocation among multiple autonomous agents. In its basic form, FICSAM focuses solely on feasibility, ensuring that all task requirements are met by forming appropriate coalitions without pursuing further optimization. Agents collaborate minimally, exchanging only essential information through token passing, anytime updates if a feasible structure is found, and termination signals once convergence is detected. This approach offers very high scalability with low communication overhead.

The improved version, IFICSAM, extends the basic method by introducing local optimization strategies. In IFICSAM, agents not only aim for feasibility but also iteratively seek to improve the global coalition structure. They apply methods such as greedy reassignment, coalition reformation through combinations, and constraint satisfaction programming (CSP) techniques. Whenever an agent finds a structure that improves the overall utility, it broadcasts the new structure to its teammates. While IFICSAM introduces slightly more communication compared to the basic FICSAM, it significantly enhances the quality of the final coalition structures. Both methods are suitable for large-scale, dynamic environments, with FICSAM prioritizing speed and minimal overhead, and IFICSAM balancing scalability with coalition quality through adaptive improvements.

4.4 Implementation Details

While the foundational FICSAM and IFICSAM algorithms were adopted from prior work [3], this thesis integrated them into a broader evaluation framework and extended their logic with experimental modifications. This section describes the main technologies, frameworks, and design choices used to implement and extend the system.

4.4.1 Programming Language and Libraries

The core implementation was developed in Python 3, leveraging its rich ecosystem for multi-agent system development, constraint solving, and rapid prototyping. Key Python libraries used include:

- **NumPy:** for numerical operations, matrix handling, and efficient calculations (used for agent positioning and disntance computations).
- **Matplotlib:** for visualizing agent-task scenarios and results during simulation analysis
- **MiniZinc/Clingo Python API**: to interface Python code with the MiniZinc/Clingo solver for constraint satisfaction problem (CSP) modeling

4.4.2 Optimization and Solving Tools

Two external optimization tools were integrated to model and solve parts of the problem:

- **MiniZinc Solver**: Used during feasibility checking, where agents use CSP models (`model_globalcsp.mzn`, `model_cspnode.mzn`) to validate coalition structures.
    - MiniZinc models express task requirements (resources, autonomy) as logical constraints.
    - Agents invoke MiniZinc via Python during improvement steps if CSP-based refinement is needed.
- **Clingo (ASP Solver)**: Used for experimental centralized comparisons. The centralized models (`model_centralized.lp`, `centralized_data.lp`) were solved using Clingo to produce optimal coalition structures against which decentralized methods (FICSAM/IFICSAM) were evaluated.

Both solvers were configured to run locally with standard default settings, ensuring reproducibility.

4.4.3 Software Architecture

The system was organized into modular components following object-oriented principles. The architectural foundation, including the relationships between core classes such as Agent,

Task, Structure, and message types (TokenMessage, AnytimeMessage, StopMessage), was directly adopted from the original implementation developed by Ahmadoun [3].

This architecture defines a clear separation between algorithm-related and application-related components, as illustrated in Figure 3.4. Agents inherit behavior from AgentBase, which handles token passing, structure evaluation, and improvement methods. Tasks extend TaskBase, encapsulating requirements such as autonomy, resource needs, and agent count. Communication is achieved through structured message passing among agents.

Although the class structure remained unchanged, this thesis integrated the architecture into a new evaluation framework, modifying internal logic—especially in improvement functions—to experiment with alternative coalition formation strategies. The original design served as a stable and extensible base for both centralized and decentralized evaluations performed in this work.

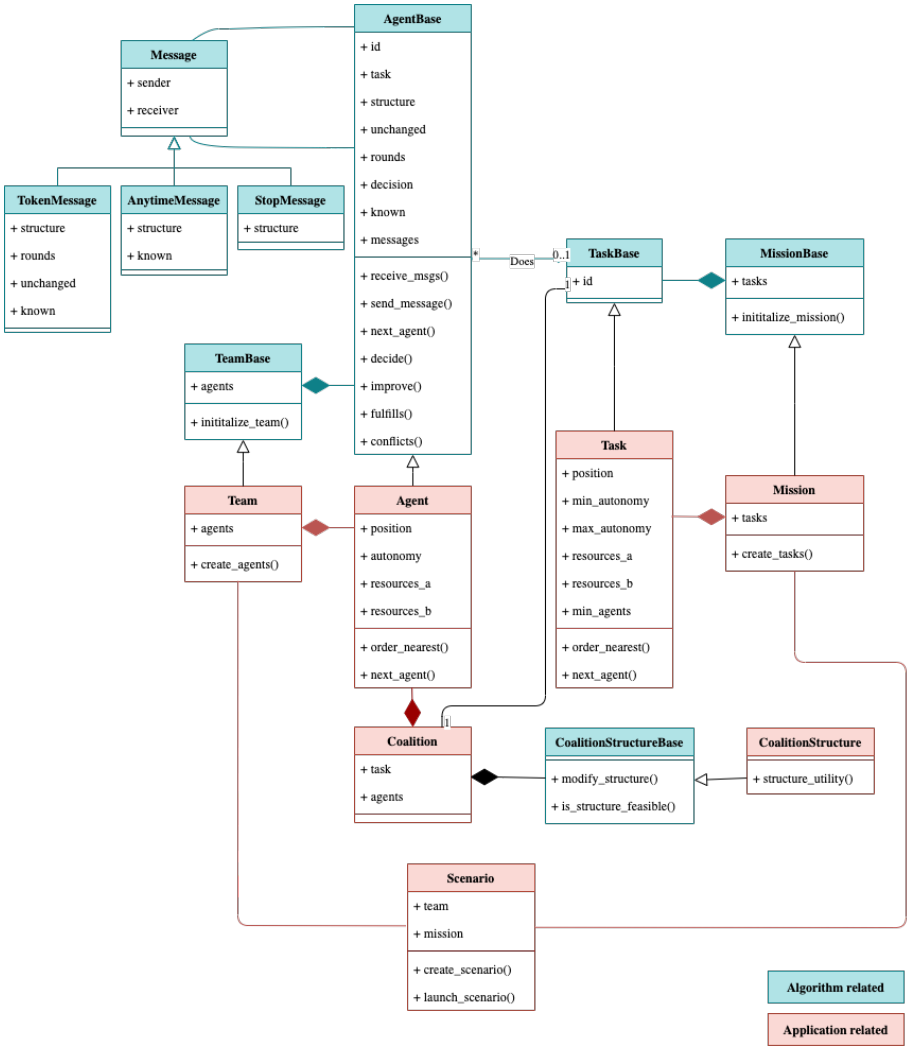

Figure 3.4: Object relation scheme (figure from [3])

4.5 Summary

This chapter presented the formal definition of the decentralized task allocation and coalition formation problem in the context of wildfire detection using drone agents. It described the core components of the adopted system [3], including agents, tasks, missions, teams, and coalition structures, offering a clear understanding of the environment and the operational logic governing agent behavior.

The chapter also introduced the methodological foundations upon which this work is built. Specifically, it discussed the use of Constraint Satisfaction Problems (CSP) and Answer Set Programming (ASP) as centralized modeling techniques for optimal coalition generation. In

the decentralized domain, the Feasible Interdependent Coalition Structure Anytime Method (FICSAM) and its improved version, IFICSAM, were adopted from the original system [3] and used as the foundation for experimentation and evaluation.

Implementation details were then outlined, highlighting the integration of Python, MiniZinc, and Clingo. These technologies collectively supported the development of a robust, extensible, experimental platform for coalition formation.

The next chapter presents the experimental evaluation, comparing the performance, efficiency, and scalability of the centralized and decentralized approaches under a variety of simulated problem configurations.

CHAPTER 5: Experimental Results and Evaluation

5.1 Objectives of the Evaluation

The primary objective of this experimental evaluation is to assess the performance, scalability, and effectiveness of the proposed decentralized coalition formation methods, FICSAM and IFCSAM, in the context of wilflife monitoring using drone agents.

The evaluation aims to systematically investigate the following aspects:

- **Feasibility Achievement:** To determine the ability of centralized methods, FICSAM and IFCSAM to find feasible coalition strctures that satisfy task requirments under different agent and task configurations.
- **Coalition Structure Quality:** To measure the final global achieved by different methods, reflecting the efficiency and optimality of the task allocation.
- **Scalability with Problem size:** To evaluate how the computational time evolve as the number of agents and tasks increases.
- **Communication Overhead:** To quantify the number of messages exchanged during coalition formation, highligthing the communication efficiency of the decentralized approach compared to centralized baselines.
- **Comparison Against Centralized Solvers:** To constrast the decentralized methos with conventional centralized optimization techniques, such as those implemented with MiniZinc (Gecode, Coin-BC, Chuffed solvers) and Answer Set Programming (Clingo), particularly in terms of solutions quality and execution time.

Through these objectives, the evaluation verifies the key advantages of the suggested methodologies – scalability, flexibility, and real-time responsiveness – while identifying points where trade-offs occur, such as between solution quality and communication effort.

5.2 Experimental Setup

5.2.1 Environment and Infustructure

All experiments were executed on a server equipped with the following specifications:

- **Operating System:** Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-131-generic x86_64)
- **CPU:** Multi-core Intel Xeon Processor
- **RAM:** 15GB
- **Python Version:** 3.10.12
- **MiniZinc Solver Version:** 2.6.4
- **Clingo ASP Solver Version:** 5.5

The server environment provided sufficient computational power to conduct large-scale experiments involving up to 100 agents and 20 tasks.

5.2.2 Scenario Configuration

The operational environment was modeled in a grid, where both agents and tasks were randomly placed.

Each agent was initialized with randomly generated attributes, including:

- Remaining battery autonomy
- Resources of type A and/or B
- Communication capabilities

Each task was randomly generated with:

- A specific location on the grid
- Minimum and maximum autonomy requirements
- Resource requirements for A and B
- Minimum number of agents required

The mission generator (`mission.py`) ensured diversity among task difficulty levels, simulating a range of simple to complex wildfire monitoring missions.

5.2.3 Experimental Variables

To comprehensively evaluate the system's performance, the following parameters were varied:

- **Number of Agents**: 5, 10, 20, 50, 100

- **Number of Tasks**: 2, 5, 10, 20
- **Test Repetitions**: Each configuration was tested 100 times (controlled by the parameter `d` in `launch.sh` and `main.py`) to ensure statistical reliability.

A constraint was enforced such that the number of agents was always at least twice the number of tasks to maintain feasible scenarios.

5.2.4 Evaluated Algorithms

The following methods were tested across all experimental configurations:

- **FICSAM** (noimprove): Basic decentralized feasible coalition formation
- **IFICSAM (**greedy/csp**)**: Decentralized coalition formation with one-to-one and best subset improvements
- **Centralized Approaches**:
    - MiniZinc Gecode Solver
    - MiniZinc Chuffed Solver
    - MiniZinc Coin-BC Solver
    - Clingo ASP Solver
    - ORTools CSP Solver

These algorithms were automatically executed using the provided `launch.sh` script and managed via the `main.py` controller.

5.3 Evaluation Metrics

To evaluate the performance and efficiency of the proposed decentralized coalition formation methods, a set of key evaluation metrics was defined. These metrics allow a detailed comparison between decentralized methods (FICSAM, IFICSAM) and centralized solvers across various problem sizes and scenarios.

The following metrics were systematically measured and recorded during the experiments:

5.3.1 Feasibility Rate

The **feasibility rate** measures the proportion of runs in which a feasible coalition structure satisfying all task requirements was successfully found. A run was considered successful if no task was left unassigned or infeasible due to resource, autonomy, or agent number constraints.

This metric reflects the fundamental ability of each method to solve the coalition formation problem under different levels of complexity.

5.3.2 Global Utility

The **global utility** quantifies the quality of the final coalition structure.
Utility is computed as a function of:

- How well the agents collectively satisfy the task requirements.
- How efficiently resources (battery, sensors) are utilized.
- How balanced and compact the coalitions are.

Higher utility values indicate better task fulfillment, more efficient use of agent capabilities, and higher cooperation effectiveness.

5.3.3 Execution Time

Execution time measures the total time taken by each method to reach a final coalition structure, starting from the initial agent-task setup.
This includes:

- Time spent on feasibility checking.
- Time spent on improvements
- Communication and decision-making delays.

Execution time was measured in **seconds** using system clocks (`time.process_time()` and `time.time()` functions).

This metric highlights the scalability and real-time suitability of the algorithms, particularly when comparing decentralized vs centralized approaches.

5.3.4 Number of Messages Exchanged

Since communication overhead is a critical factor in decentralized systems, the total number of messages exchanged among agents was recorded.

Each sent message was counted, allowing analysis of:

- The communication cost required for convergence.
- The efficiency of FICSAM (minimal communication) vs IFICSAM (slightly higher due to improvements).

Lower message counts indicate higher communication efficiency, which is crucial for large multi-agent deployments.

5.3.5 Number of Iterations (Rounds)

In addition to message count, the number of rounds—representing complete iterations of decision-making across the agent network—was measured. Each round captures the full cycle in which agents sequentially process incoming information, evaluate the current coalition structure, and perform potential updates based on feasibility or improvement strategies.

A lower number of rounds generally indicates faster convergence and lower decision latency in the decentralized process. This metric helps evaluate the responsiveness of each method, with fewer iterations implying quicker agreement on a feasible or optimal structure.

The measured values reflect the behavior of both baseline and improved methods across problem sizes, offering insight into the efficiency and reactivity of the decentralized coordination mechanism.

5.4 Experimental Results

This section presents and analyzes the resukts of the experimental evaluation conducted to assess the feasibility, utility, scalability, and communication overhead of the proposed coalition formation methods.

The evaluation focuses not only on raw performance figures byt also on understanding why performance varies across methods as the problem size increases.

5.4.1 Feasibility Analysis

The first metric evaluated was the ability of each method to find a feasible solution -i.e., a valid coalition structure where all tasks are assigned to agent groups satisfying their resource and autonomy requirements. For all settings with 5-20 agents, all methods were able to find feasible solutions with a 100% success rate. With more agents and tasks, however, there were subtle differences.

FICSAM, which prioritizes feasibility but does not include structural optimization, began to show small declines in performance. In larger configurations, particularly with 50 agents and 10 or more tasks, the feasibility success rate for FICSAM dropped. This drop can be attributed to its reliance on random initialization and limited structural exploration during the token-passing process. If agents converge prematurely or form coalitions that block each other, the system may settle in a dead-end configuration without discovering a feasible one.

In contrast, IFICSAM maintained a high feasibility success rate across the majority of tested configurations. This is due to its improvement strategies—such as greedy task reassignment and CSP-based coalition refinement—which allow agents to escape suboptimal configurations by attempting to reallocate themselves when better options are detected.

All centralized solvers—including MiniZinc (Gecode, Chuffed, Coin-BC), Clingo, and ORTools—maintained a perfect feasibility rate throughout, as expected. These solvers operate globally on the full problem specification and explore the complete search space, which guarantees that if a feasible solution exists, it will be found.

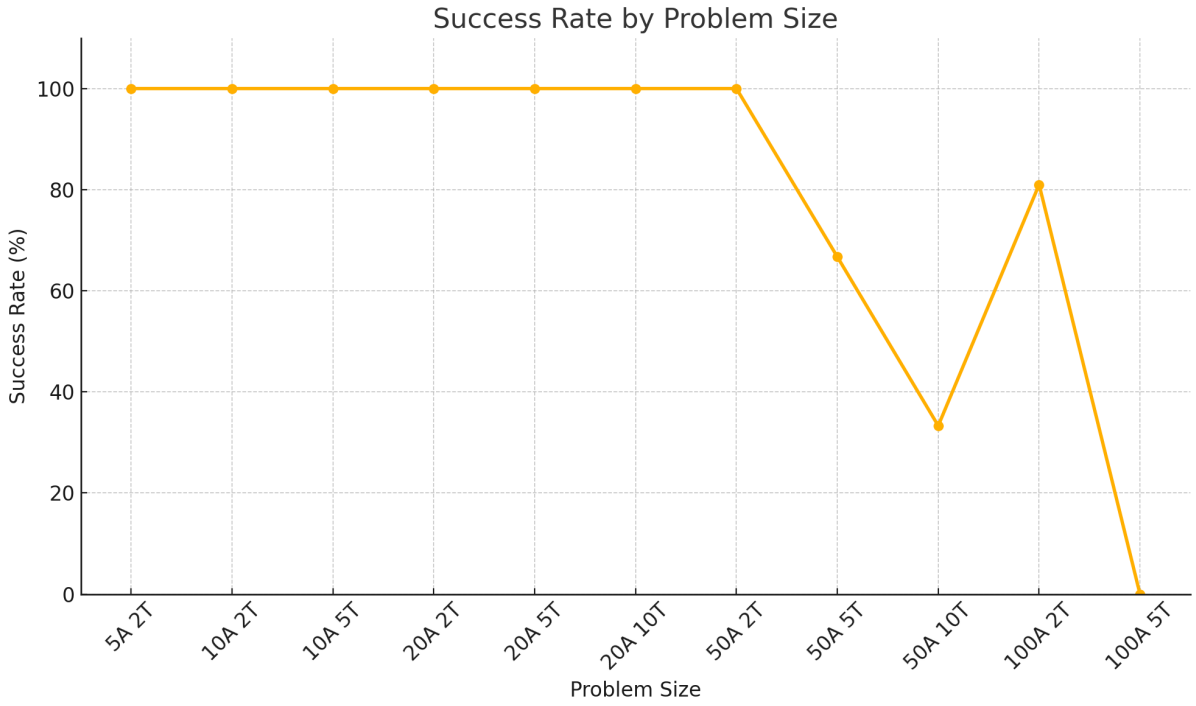

Figure 5.1: Feasibility rate by problem size across methods.

5.4.2 Coalition Utility and Solution Quality

In addition to feasibility, the quality of the final coalition structure is a critical performance indicator. This was evaluated using the system's global utility function, which accounts for how well each task's requirements are met and how efficiently agent capabilities are used.

For smaller problem instances, both FICSAM and IFICSAM achieved good utility scores, often close to those of centralized solvers. However, as the number of agents and tasks increased, the utility of decentralized methods—particularly FICSAM—began to decline.

This decline is explained by the limited search behavior of FICSAM. While it ensures feasibility, it does not attempt to optimize the coalition configuration once a feasible solution is found. As the search space grows, the probability of landing on a suboptimal but feasible configuration increases. Since FICSAM agents do not explore alternatives unless explicitly guided to do so, they miss potential improvements that a centralized solver would find.

In contrast, IFICSAM addresses this limitation through embedded improvement strategies. When an agent holds the token, it evaluates its coalition assignments and attempts to locally optimize them using procedures like `improve_greedy()` or `improve_csp()`. This results in higher utility values even in large-scale problems. Nonetheless, due to the decentralized and

local nature of these improvements, IFICSAM may still fall short of the global optima achieved by centralized solvers, especially in complex configurations with high inter-task dependencies.

Centralized solvers, especially Clingo, Chuffed, and Coin-BC, consistently produced the highest utility values. They operate with full knowledge of the problem and perform complete search and optimization over all agent-task assignments. However, this performance comes at the cost of significantly higher computation time and resource usage, as discussed in the next subsection.

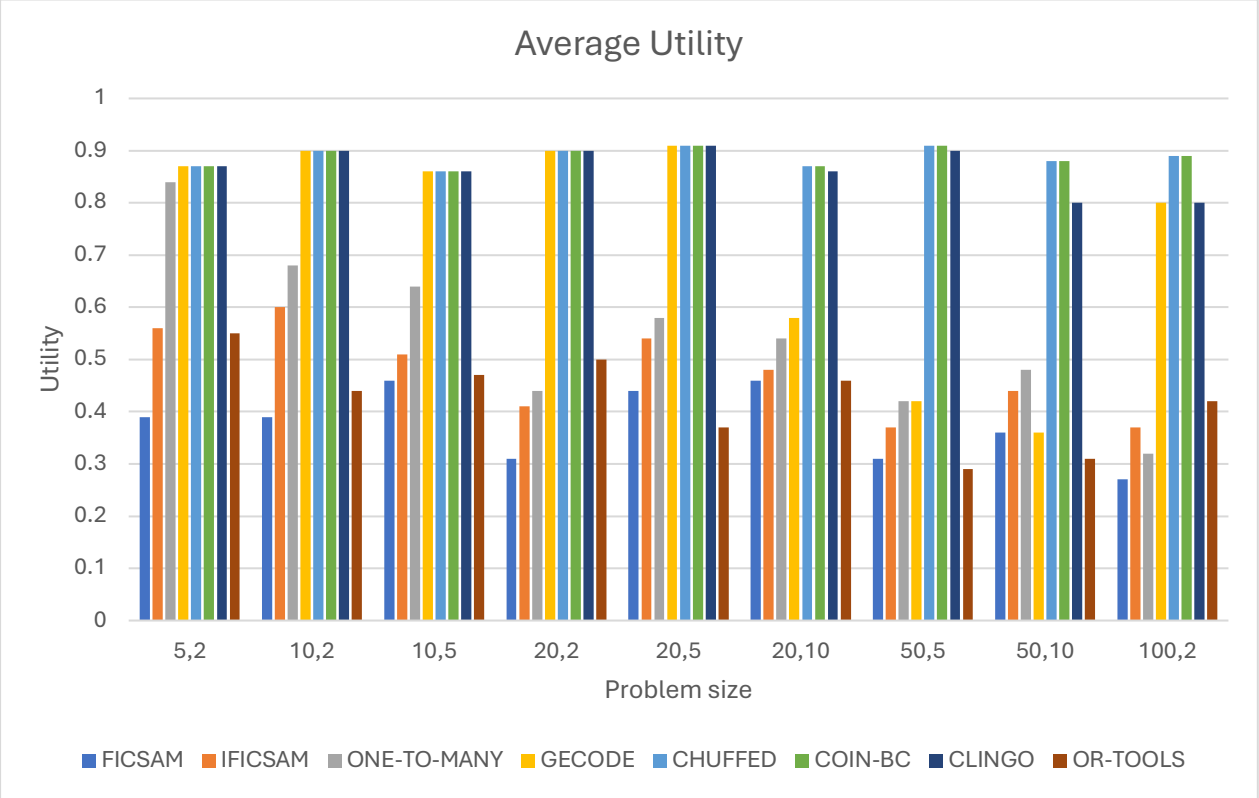

Figure 5.2: Average utility of each method per scenario.

5.4.3 Scalability and Execution Time

Execution time was another key metric, particularly relevant for real-time deployment scenarios. As expected, decentralized methods scaled much more gracefully than centralized ones. FICSAM and IFICSAM completed in under one minute for configurations below 20 agents and 5 tasks, but for larger problem the execution time increased dramatically.

In contrast, the centralized solvers began to experience severe slowdowns as the number of agents and tasks increased. Gecode, Chuffed, and Clingo took several minutes to solve large problem instances, and in some cases, failed to return results within practical time limits. This dramatic growth in execution time stems from the exponential increase in the number of agent-task assignments and constraints that centralized solvers must process. Unlike decentralized agents, which only reason about their local state and a subset of tasks, centralized methods must encode and search the entire problem space in one monolithic operation.

Therefore, while centralized methods may achieve marginally higher utility, they become impractical for large-scale or time-sensitive missions.

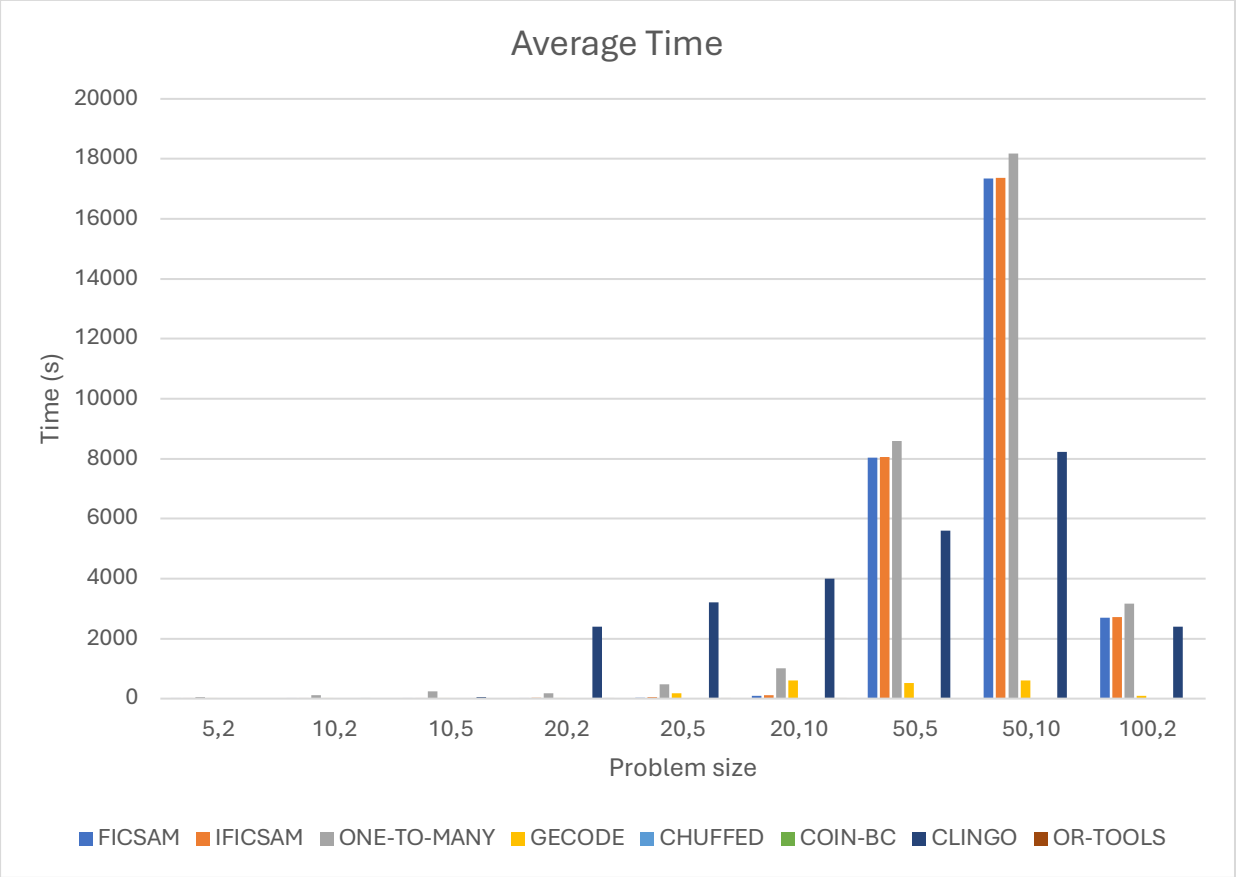

Figure 5.3: Average execution time across problem sizes.

5.4.4 Communication Overhead

In decentralized coalition formation, communication cost plays a critical role in evaluating the practicality of the method—especially in real-world systems where bandwidth, latency, or

energy consumption may be constrained. This subsection analyzes the number of messages exchanged between agents for each method across different problem sizes.

The experimental results, visualized in **Figure 4.4**, show that FICSAM consistently exhibits the lowest communication overhead across all scenarios. This is expected, as FICSAM uses a simple token-passing mechanism without any improvement-based messaging. Each agent receives the token once per cycle and performs only a feasibility check before passing the token to the next agent. As a result, the number of messages increases linearly with the number of agents and tasks, but at a relatively low slope.

In contrast, IFICSAM introduces significantly higher communication costs, particularly in larger scenarios. This increase is due to the use of `AnytimeMessages`, which are broadcast to all known agents every time an agent discovers an improved feasible structure. These broadcasts are critical for enabling utility optimization but result in a larger number of total messages—especially when agent configurations are highly dynamic or task dependencies are complex. The chart shows that in scenarios like 100 agents and 2 tasks, IFICSAM can generate over 1,200 messages, more than five times the message count of FICSAM for the same setting.

The CSP-based method, which incorporates constraint-satisfaction improvements, also shows elevated message counts—similar to IFICSAM. While it can improve solution quality by suggesting better coalitions, this comes at the cost of additional synchronization and messaging steps between agents.

This trade-off between utility optimization and communication is crucial. IFICSAM and CSP variants achieve higher utility (as shown in Figure 4.2), but they require substantially more inter-agent communication to do so. In real-world multi-agent drone systems, this could translate to increased battery consumption, potential delays due to network congestion, or synchronization issues if communication is not fully reliable.

The results clearly demonstrate that:

- FICSAM is best suited for **low-bandwidth** or **energy-limited environments** where feasibility is the priority.

- IFICSAM and CSP (one-to-many) variants are better suited for **high-performance networks** where solution quality is more critical than message minimization.

Overall, the communication overhead analysis confirms that while message volume increases with optimization complexity, the decentralized protocols still remain scalable and manageable under most deployment conditions.

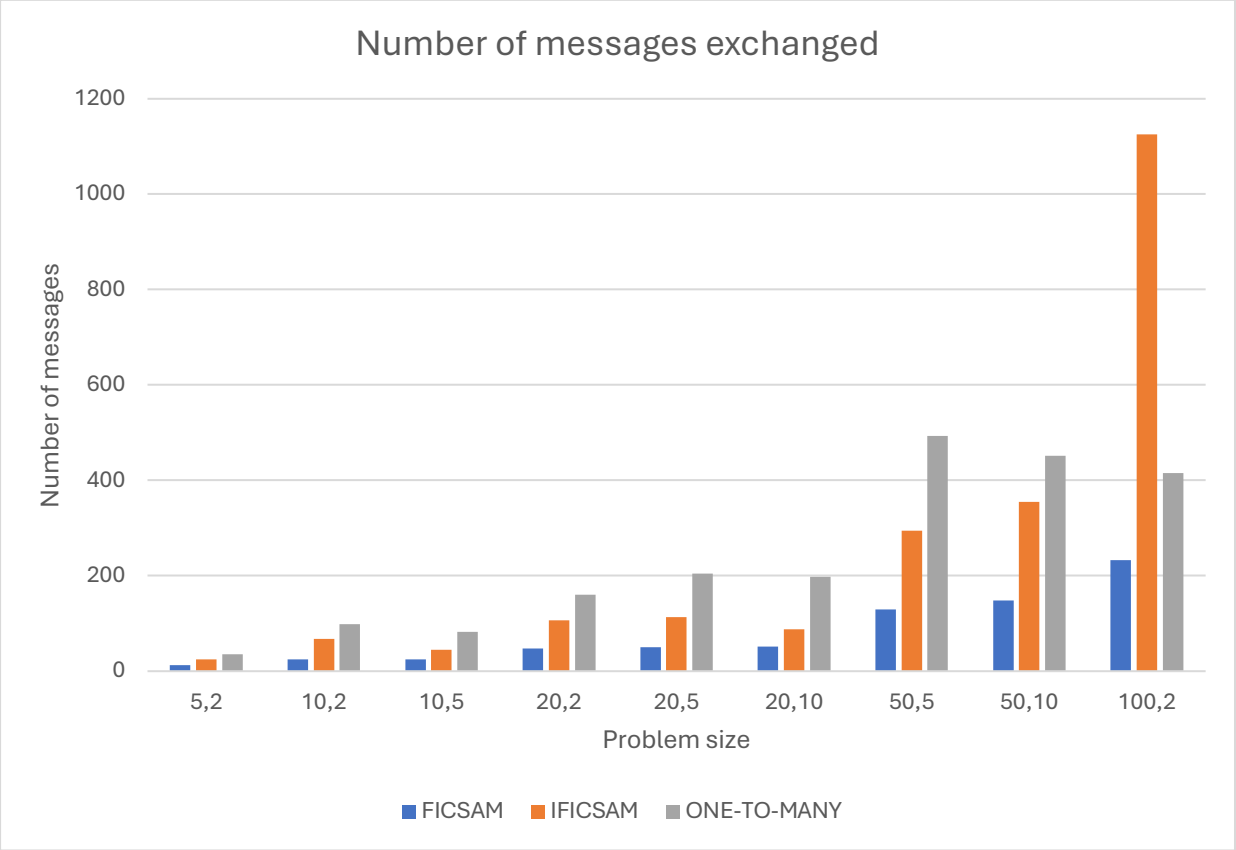

Figure 5.4: Number of messages exchanged by method and problem size.

5.4.5 Number of Iterations

An important performance metric in evaluating decentralized and centralized coalition formation methods is the **number of iterations** required to reach a final solution. In decentralized methods like FICSAM and IFICSAM, an iteration corresponds to a full round of agent decision-making, while for the centralized CSP-based solver, each invocation counts as one iteration. However, due to the significantly higher computational effort required per CSP iteration, the CSP results in the chart are scaled down by a factor of 10 (denoted as CSP/10) to enable visual comparison.

The results reveal several key patterns:

- **Low Complexity Settings (5–20 agents):**
  For smaller problem sizes (e.g., 5,2 and 10,2), all methods—FICSAM, IFICSAM, and CSP—require a minimal number of iterations, typically fewer than 10. The marginal differences in iteration count at this scale suggest that task dependencies and resource constraints are easily satisfied without extensive coordination or search.
- **Moderate Configurations (20,5 and 20,10):**
  As the number of agents and tasks increases, FICSAM begins to show a clear advantage over IFICSAM. While IFICSAM requires more iterations due to its improvement strategy. In contrast, the CSP solver begins to show a steeper increase in iteration count (even after scaling), indicating the growing cost of searching the full configuration space.
- **High Complexity Settings (50–100 agents):**
  In larger scenarios (50,5; 50,10; and 100,2), the difference becomes more pronounced. The CSP solver, despite being globally optimal, demonstrates a steep rise in iteration count, exceeding 18000 in the case of the (50,10) configuration (scaled to ~1800 actual invocations). This highlights the scalability limitations of centralized approaches when applied to large, interdependent problems.

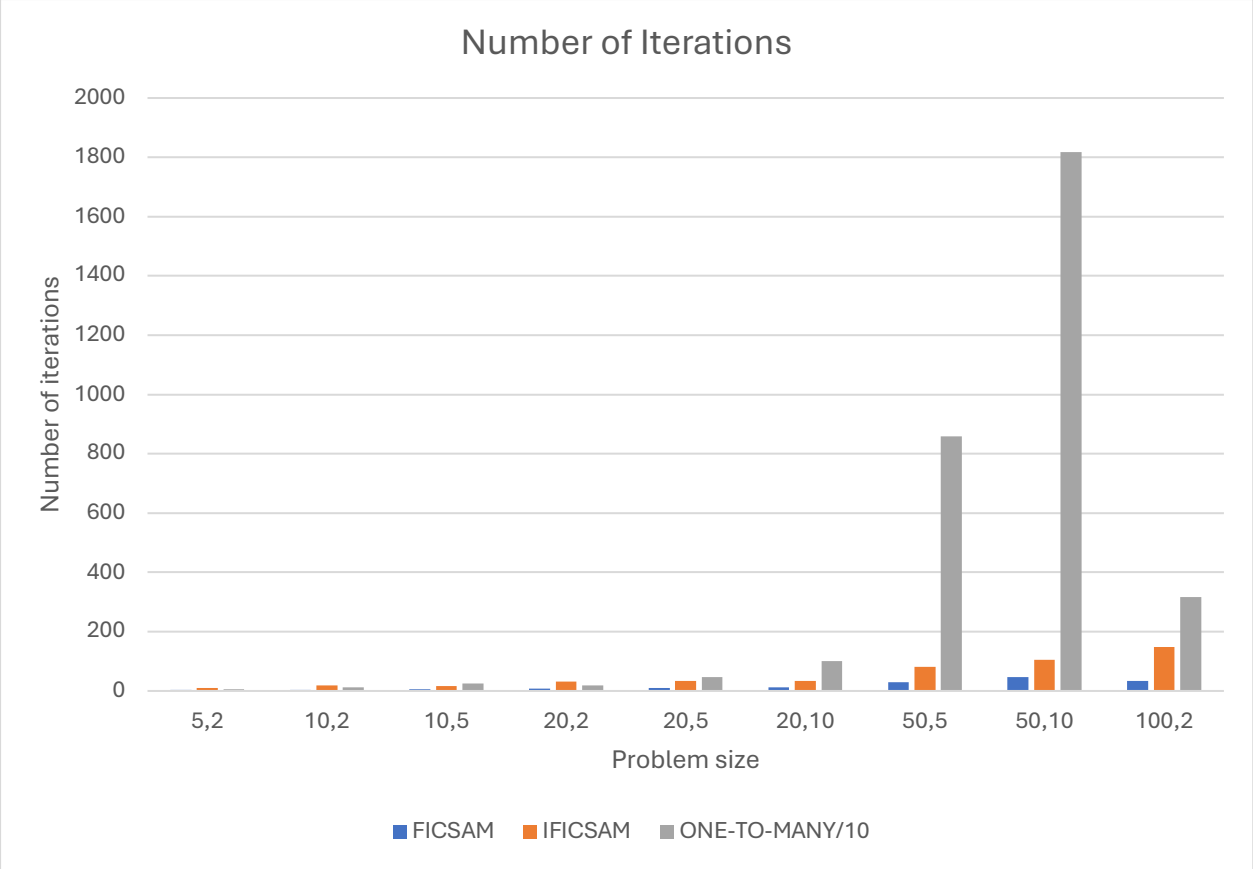

Figure 5.4: Number of iterations by method and problem size.

5.5 Discussion

The experimental results demonstrate clear differences between decentralized and centralized approaches to coalition formation, both in terms of behavior and performance. While centralized solvers generally achieve higher-quality solutions and faster execution for small problems, their scalability quickly deteriorates. Conversely, decentralized methods like FICSAM and IFICSAM are built for distributed deployment, offering greater flexibility and system-wide resilience—but not without trade-offs as problem size increases.

Decentralized Methods: Strong Scalability with Practical Limits

FICSAM and IFICSAM show strong conceptual scalability since they avoid the global optimization bottleneck of centralized solvers. However, as the number of agents and tasks grows, especially beyond 20 agents and 10 tasks, performance degrades both in terms of execution time and utility. This degradation is rooted in several factors:

1. **Local View and Decentralized Decisions:** Each agent only operates based on its own knowledge and the information shared through token-passing and message exchanges. As more agents and tasks are added, it becomes harder for agents to collectively converge to a globally efficient coalition structure, especially without centralized oversight.
2. **Limited Structure Exploration (FICSAM):** FICSAM prioritizes feasibility over utility and does not optimize coalition quality once a structure is found. This leads to early convergence on feasible but suboptimal structures, especially in dense problem spaces with many overlapping agent capabilities and complex task dependencies.
3. **Improvement Trade-offs (IFICSAM):** IFICSAM attempts to overcome FICSAM's limitations by enabling agents to improve their assignments using greedy or CSP-based refinements. While this results in better utility, it comes at a cost: more computational work per agent, longer convergence times, and more message traffic. In large configurations, the gains in utility are often offset by longer runtimes, and the local improvements may still fall short of finding global optima.
4. **Message Processing and Token Circulation:** As the number of agents increases, the number of token passes required for full structure updates grows. Even though the message count remains reasonable, the decision time per agent increases, especially when running local CSPs in improvement steps. This results in total execution times reaching or exceeding one minute for large cases.

Centralized Methods: High Utility, Low Scalability

Centralized solvers like Gecode, Clingo, and ORTools excel at finding high-utility, globally optimal solutions—but only for small to medium problem sizes. Their performance rapidly collapses in larger settings due to:

1. **Global Constraint Explosion:** The number of combinations of agents, tasks, and constraints grows exponentially, causing solvers to either slow down dramatically or run out of resources.
2. **Monolithic Processing Bottleneck:** Since all reasoning occurs at a single point (the solver), centralized methods do not benefit from parallel agent-based

distribution, making them impractical for real-time scenarios or large, dynamic networks.

Nevertheless, these solvers serve as useful benchmarks: they show what the best-case utility looks like in theory and allow us to evaluate how close decentralized methods can get under realistic constraints.

Practical Implications

From a deployment standpoint, the trade-off is clear:

If the application involves a small number of agents and strict requirements for optimal coalition formation, centralized methods may be preferable—assuming the problem can be solved fast enough.

If the application involves real-time responsiveness, distributed agents (e.g., autonomous drones), and scalability to dozens of agents and tasks, decentralized methods like FICSAM and IFICSAM are far more practical—even if they sometimes sacrifice a few percentage points of utility.

In real-world wildfire detection, the system's ability to quickly adapt, maintain autonomy, and distribute computation is far more important than absolute optimality. This makes decentralized approaches, especially those like IFICSAM that balance feasibility and quality, highly suitable for such critical, time-sensitive missions.

5.6 Summary

This chapter provided a comprehensive experimental comparison of decentralized and centralized coalition formation algorithms for multi-agent task allocation in wildfire detection scenarios. The comparison aimed at four most significant factors: feasibility success, solution quality, scalability, and communication overhead.

The results show that even though centralized methods always give the best solution, their scalability in terms of computation effort is very bad. In contrast, decentralized methods such as FICSAM and IFICSAM give an effective practical alternative, maintaining high feasibility rates alongside acceptable utility even when dealing with large instances. Utility is sacrificed

to some extent by epanding the number og agents as well as tasks but is largely countered by local improvement techniques employed by IFCSAM.

In summary, the experiments support the argument that at any time, decentralized coalition formation strategies reach a practical compromise between solution quality and system scalability—optimally making them well-fitted for distributed, dynamic applications like real-time wildfire monitoring using autonomous drones.

CHAPTER 6: Conclusion and Future Work

6.1 Summary and Key findings

This thesis extended prior work [3] on the problem of decentralized task assignment and coalition formation among autonomous drone agents in wildfire detection. The main objective was to assess and improve the scalability and effectiveness of multi-agent coordination under constraints related to resources, autonomy, and agent-task dependencies.

To support this, the Feasible Interdependent Coalition Structure Anytime Method (FICSAM), originally proposed in [3], was adopted and integrated into the system. FICSAM is a decentralized algorithm that enables agents to collaboratively form feasible coalition structures through local decision-making and asynchronous message passing. It ensures feasibility by design and is suitable for dynamic environments due to its anytime nature.

To enhance solution quality beyond feasibility, an improved variant named IFICSAM was implemented. This version includes local optimization mechanisms such as greedy reassignment and CSP-based refinement. IFICSAM agents evaluate and potentially refine their coalition assignments when they hold the token, aiming to increase global utility while maintaining feasibility.

The thesis included extensive empirical testing across a range of problem sizes and agent-task configurations. These experiments demonstrated the behavior and performance of both FICSAM and IFICSAM, particularly in terms of feasibility, utility, and convergence. Comparisons with centralized solvers such as MiniZinc, Clingo, and ORTools highlighted the scalability benefits and practical robustness of the decentralized methods.

Overall, the contributions of this thesis provide further insight into scalable, decentralized coalition formation and support the development of cooperative drone systems capable of operating effectively under real-world constraints.

6.2 Key Findings

The experimental evaluation of FICSAM and IFICSAM, compared against established centralized solvers, revealed several important insights into the behavior, strengths, and limitations of decentralized coalition formation in multi-agent systems.

First, both FICSAM and IFICSAM consistently achieved high feasibility success rates, even as the number of agents and tasks increased. While centralized methods always produced feasible solutions—as expected due to their complete global search—FICSAM and IFICSAM also demonstrated reliability, with IFICSAM achieving 100% feasibility in all tested scenarios.

Second, solution quality, as measured by the global utility of the final coalition structure, remained competitive in smaller problem instances. As the problem scale increased, the utility achieved by decentralized methods declined modestly compared to centralized solvers. However, IFICSAM's improvement strategies significantly narrowed the gap, often achieving utility within 45-50% of centralized optimal values while avoiding their computational burden.

Third, scalability emerged as a central advantage of the proposed approach. FICSAM and IFICSAM maintained acceptable performance in scenarios involving up to 100 agents and 20 tasks—where centralized methods often failed to return results within reasonable time limits. This confirms that decentralized, agent-driven algorithms are better suited for real-time applications, such as distributed drone coordination in wildfire monitoring.

Finally, the system demonstrated strong communication efficiency. Although IFICSAM introduced additional messages due to its improvement mechanism, the overall message and token counts remained scalable, and the fully distributed nature of decision-making avoided bottlenecks inherent in centralized approaches.

6.3 Limitations

While the proposed system demonstrated promising results in terms of feasibility, scalability, and communication efficiency, several limitations were identified during the course of development and experimentation.

The first and most prominent limitation lies in the **quality of the final solution** produced by the decentralized methods, particularly FICSAM. Since agents make decisions based on limited local knowledge and without global optimization, the resulting coalition structures are often feasible but not globally optimal. Although IFICSAM improves upon this by introducing local refinement techniques, these strategies are still constrained by the agent's partial view of the overall system and lack of coordination beyond pairwise coalition changes.

Second, **execution time scalability**, while significantly better than centralized solvers, was not linear. In larger scenarios—such as those involving 50 or more agents and highly interdependent tasks—both FICSAM and IFICSAM showed a noticeable increase in convergence time. This is due in part to the growing size of coalition structures, which increases the number of feasibility checks and local improvement iterations required for each agent.

A third limitation involves **communication overhead**. Although decentralized communication avoids a central bottleneck, the propagation of AnytimeMessages in IFICSAM can lead to message floods in dense agent-task environments. While this is mitigated by convergence rules and message pruning, it remains a factor to consider in real-time systems where bandwidth is limited.

Additionally, the system currently assumes **perfect communication and synchronization** among agents, which may not hold in real-world deployments involving physical drones. Network latency, dropped messages, and inconsistent state updates could impact convergence and correctness. Integrating asynchronous fault-tolerant mechanisms is a necessary step for deployment in practical settings.

Despite these limitations, the system provides a strong foundation for practical deployment and future research, as discussed in the next section.

6.4 Future Work

Building on the foundation established in this thesis, several avenues for future work can be pursued to enhance the system’s performance, robustness, and applicability in real-world deployments.

A primary direction involves the development of smarter agent decision strategies. Currently, IFICSAM relies on greedy or CSP-based local improvements, which are effective but still limited in scope. Future versions could integrate learning-based approaches, such as reinforcement learning, allowing agents to adapt their behavior over time based on past coalition outcomes. This could enable agents to prioritize certain coalition configurations or avoid repeated failure patterns in highly constrained environments.

Another promising enhancement would be the implementation of adaptive token-passing strategies. In the current design, the token is passed either randomly or based on distance. Incorporating heuristics that prioritize agents with high-impact tasks or critical resources could accelerate convergence and improve structure quality. Similarly, introducing priority queues or leader-based decision cycles might reduce redundant computation and improve overall efficiency.

In terms of communication, the system would benefit from message compression, filtering, and prioritization. Particularly in IFICSAM, where AnytimeMessages are frequently broadcast, mechanisms for limiting unnecessary updates and avoiding message duplication could further reduce bandwidth usage and speed up decision propagation.

Another significant direction for future development is real-world deployment and testing. While the current system was evaluated in simulation, integrating it with a physical drone platform—such as ROS-enabled UAVs—would introduce real-world complexities like network latency, partial observability, and asynchronous failures. Implementing the system in a distributed drone testbed would provide valuable insights into its robustness and operational limits.

Finally, exploring hybrid architectures that combine the scalability of decentralized methods with periodic centralized optimization could offer the best of both worlds. For example, agents could operate autonomously during most of the mission, but occasionally synchronize with a base station for global re-optimization when bandwidth or conditions allow.

Together, these extensions would significantly enhance the practical value of the system and push the boundaries of what decentralized coalition formation can achieve in real-time, distributed environments.

Bibliography

1. Abdallah, S., Lesser, V., & Jennings, N. R. (2004). Coalition formation in cooperative multi-agent systems. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 1416–1417.
2. Aumann, R. J., & Dreze, J. H. (1974). Cooperative games with coalition structures. *International Journal of Game Theory*, 3(4), 217–237.
3. Ahmadoun, D. (2022). Interdependent Task Allocation via Coalition Formation for Cooperative Multi-Agent Systems. PhD Thesis, University of Paris.
4. Casbeer, D. W., Beard, R. W., McLain, T. W., Li, S. M., & Mehra, R. K. (2006). Forest fire monitoring with multiple small UAVs. *Proceedings of the American Control Conference*, Minneapolis, Minnesota, USA, 3530–3535.
5. Casbeer, D. W., Kingston, D. B., Beard, R. W., & McLain, T. W. (2006). Cooperative forest fire surveillance using a team of small unmanned air vehicles. *International Journal of Systems Science*, 37(6), 351–360.
6. Ferber, J. (1999). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Professional.
7. Ferber, J. (1999). *Multi-agent systems: An introduction to distributed artificial intelligence*. Addison-Wesley.
8. Gerkey, B. P., & Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9), 939–954.
9. Jennings, N. R., Sycara, K., & Wooldridge, M. (1998). A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1), 7–38.
10. Korsah, G. A., Stentz, A., & Dias, M. B. (2013). A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12), 1495–1512.
11. Kumar, V., & Michael, N. (2012). Opportunities and challenges with autonomous micro aerial vehicles. *The International Journal of Robotics Research*, 31(11), 1279–1291.
12. Lagoudakis, M. G., Spirakis, P. G., & Stamatopoulos, T. (2005). Auction-based multi-robot routing. *Robotics and Autonomous Systems*, 52(1), 1–14.
13. Macarthur, H., Crane, C., & Armstrong, A. (2011). A decentralized approach to multi-agent task assignment using distributed constraint satisfaction. *Autonomous Agents and Multi-Agent Systems*, 22(2), 225–256.
14. Nunes, E., & Gini, M. (2017). Multi-robot auctions for allocation of tasks with temporal constraints. *Autonomous Robots*, 41(3), 539–562.
15. Parker, L. E. (1998). ALLIANCE: An architecture for fault-tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2), 220–240.
16. Rahwan, T., & Jennings, N. R. (2008). Coalition structure generation: A survey. *Artificial Intelligence*, 32(4), 297–328.
17. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
18. Sandholm, T., & Lesser, V. (1997). Coalitions among computationally bounded agents. *Artificial Intelligence*, 94(1–2), 99–137.

19. Sandholm, T., Larson, K., Andersson, M., Shehory, O., & Tohmé, F. (1999). Coalition structure generation with worst-case guarantees. *Artificial Intelligence*, 111(1–2), 209–238.
20. Shehory, O., & Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1–2), 165–200.
21. Shoham, Y., & Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.
22. Vig, L., & Adams, J. A. (2006). Multi-robot coalition formation. *IEEE Transactions on Robotics*, 22(4), 637–649.
23. Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. John Wiley & Sons.

APPENDIX

Appendix A: Python Code for Clingo Data Generation.

```python
def generate_clingo_data(tasks, agents,
filename="application/dir_clingo/centralized_data.lp"):
    data = []

    # Define agents and tasks
    data.append(f"agent(1..{len(agents)}).")
    data.append(f"task(1..{len(tasks)}).")

    # Problem Parameters
    data.append(f"nb_agents({len(agents)}).")
    data.append(f"nb_tasks({len(tasks)}).")
    data.append("margin_autonomy_min(2).")
    data.append("margin_distance_max(20).")
    data.append("margin_nb_agents(1).")

    # Agent Attributes
    for i, agent in enumerate(agents, start=1):
        data.append(f"agents_autonomy({i}, {agent.autonomy}).")
        if agent.resources_a:
            data.append(f"agents_resA({i}).")
        if agent.resources_b:
            data.append(f"agents_resB({i}).")

    # Compute distances between tasks and agents
    for t_idx, task in enumerate(tasks, start=1):
        for a_idx, agent in enumerate(agents, start=1):
            dist = calculate_distance(task.position, agent.position)
            data.append(f"distance({t_idx},{a_idx},{dist}).")

    # Compute nearest and second-nearest tasks
    task_distances = {i+1: [] for i in range(len(tasks))}
    for t1 in range(len(tasks)):
        for t2 in range(len(tasks)):
            if t1 != t2:
                dist = calculate_distance(tasks[t1].position, tasks[t2].position)
                task_distances[t1+1].append((t2+1, dist))

    tasks_nearest = {}
    tasks_nearest_sd = {}
```

```
    for t, dists in task_distances.items():
        dists.sort(key=lambda x: x[1])  # Sort by distance
        tasks_nearest[t] = dists[0][0] if len(dists) > 0 else t  # Nearest task
        tasks_nearest_sd[t] = dists[1][0] if len(dists) > 1 else t  # Second-
nearest task

    # Task Attributes
    for i, task in enumerate(tasks, start=1):
        data.append(f"tasks_autonomy_min({i},{task.min_autonomy}).")
        data.append(f"tasks_autonomy_one({i},{task.max_autonomy}).")
        data.append(f"tasks_distance_max({i},{task.max_distance}).")
        data.append(f"tasks_agents_min({i},{task.min_agents}).")
        data.append(f"tasks_resA({i},{task.resources_a}).")
        data.append(f"tasks_resB({i},{task.resources_b}).")
        data.append(f"tasks_nearest({i},{tasks_nearest[i]}).")
        data.append(f"tasks_nearest_sd({i},{tasks_nearest_sd[i]}).")

    # Dynamically add oddTask/1 for tasks with an odd index
    for j in range(1, len(tasks)+1):
        if j % 2 == 1:
            data.append(f"oddTask({j}).")

    # Dynamically add odd/1 facts for odd numbers up to a bound (e.g., the number
of agents)
    max_possible_count = len(agents)
    for num in range(1, max_possible_count):
        if num % 2 == 1:
            data.append(f"odd({num}).")

    # Write to file
    with open(filename, "w") as file:
        file.write("\n".join(data))
def clingo_centralized_cop(tasks, agents, all_solutions=False):
    print("\n\n############## CENTRALIZED START\n")

    # Initialize the Clingo control object with desired options.
    ctl = clingo.Control(["--parallel=4", "--opt-mode=opt"])
    ctl.configuration.solver.heuristic = "Vmtf"

    # Load your ASP files.
    ctl.load("model_centralized.lp")
    ctl.load("centralized_data.lp")

    # Ground the base part.
```

```
    ctl.ground([("base", [])])

    solution_structure = None
    solution_obj_value = None
    models = []

    # Custom on_model callback to process models.
    def on_model(model):
        nonlocal solution_structure, solution_obj_value
        atoms = model.symbols(shown=True)
        sol, obj = parse_atoms(atoms, tasks, agents)
        models.append((sol, obj))
        solution_structure = sol
        solution_obj_value = obj

    # Start solving asynchronously.
    handle = ctl.solve(async_=True, on_model=on_model)

    # Wait for up to 600 seconds (10 minutes).
    if not handle.wait(10):
        handle.cancel()

    print("############## CENTRALIZED END\n")
    return solution_structure, solution_obj_value
def parse_atoms(atoms, tasks, agents):
    """
    Parses Clingo symbols from the solution and builds the structure.
    Returns a tuple (structure, obj_value) similar to your original parser.
    """
    assignment_by_agent = {}
    obj_value = None

    # atoms is a list of clingo.Symbol objects.
    for sym in atoms:
        if sym.name == "assignment" and len(sym.arguments) >= 2:
            a_index = sym.arguments[0].number
            t_index = sym.arguments[1].number
            assignment_by_agent[a_index] = t_index
        elif sym.name == "obj_value" and sym.arguments:
            try:
                obj_value = float(sym.arguments[0].number)
            except Exception:
                obj_value = None
```

```
    # Build the structure mapping each task to a list of agents.
    structure = {t: [] for t in tasks}
    for a_idx in range(1, len(agents) + 1):
        if a_idx in assignment_by_agent:
            t_index = assignment_by_agent[a_idx]
            if t_index == 0:
                continue  # No assignment
            task_obj = tasks[t_index - 1]
            structure[task_obj].append(agents[a_idx - 1])

    if all(len(v) == 0 for v in structure.values()):
        structure = False

    # Normalize objective value similarly to your original function.
    if obj_value is not None:
        obj_value = round(obj_value / (8 * len(agents) * len(tasks)), 4)
    return structure, obj_value
```