Multi-Agent Distributed Graph Traversal

MULTI-AGENT DISTRIBUTED GRAPH TRAVERSAL

BY

MARKOV MIKHAIL, Engineer, B.A., C.E.Sc.

A Thesis

Submitted to the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

Master of Science (2016)	McMaster University
(Computing and Software)	Hamilton, Ontario, Canada

TITLE:	Multi-Agent Distributed Graph Traversal
AUTHOR:	Markov Mikhail Engineer, (Control and informatics in engineering systems), St Petersburg State Electrotechnical University, St Petersburg, Russia
SUPERVISOR:	Dr. Borzoo Bonakdarpour
NUMBER OF PAGES:	xiv, 128
LEGAL DISCLAIMER:	This is an academic research report. I, my supervisor, defense committee and McMaster University make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the usage of the algorithms, findings, code, or recommendations in this thesis.

To my family

Abstract

The industry of the civil Unmanned Aerial Vehicles (*UAVs*) has been growing rapidly in past few years. In many scenarios, accomplishing a task using a single *UAV* is either not cost-effective due to the size of the project or not even feasible due to the existence of unforeseen environment conditions and constraints (e.g., weather conditions and/or physical obstacles). This limitation motivates the need to move to solutions that incorporate a network of autonomous *UAVs* that carry out a joint and coordinated mission.

This thesis introduces a multi-agent system and related algorithms that solve the graph traversal problem in a distributed and decentralized manner while optimizing a set of costs. The environment is modeled as a graph where every node is the point for the agents to accomplish some task or to distinguish the point as an obstacle where traveling is not possible. The online distributed algorithms are implemented on a network of *UAVs* and we report the results of rigorous simulations and real experiments with a network of *UAVs*. The results clearly validate our claim that a network *UAVs* can be effectively employed to accomplish a given task.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Borzoo Bonakdarpour for giving me the opportunity to work on this challenging and exciting area of computer science, as well as, for his support and guidance throughout my degree.

I would like to thank my committee members, Dr. George Karakostas and Dr. Alan Wassyng, for their valuable feedback on my research work.

I would also like to thank my colleague Akhil Krishnan for his help with conducting experiments.

Finally, I would like to thank my parents, family, and friends for their support and encouragement.

Notation and Abbreviations

ANTS – Ants Nearby Treasure Search

CSP – Constraint Satisfaction Problem

DCSP – Distributed Constraint Satisfaction Problem

ILP – Integer Linear Programming

LP – Linear Programming

MADGT – Multi-Agent Distributed Graph Traversal

MADGTA – Multi-Agent Distributed Graph Traversal Algorithm

MRCP – Multi-Robot Coverage Problem

MRPP – Multi-Robot Patrol Problem

mTSP – Multiple Traveling Salesman Problem

OCTO – Octocopter

QUAD – Quadrocopter

RPi – Raspberry Pi

RTL – Return to the Launch

SI – Swarm Intelligence

SFOC – Special Flight Operations Certificate

TSP – Traveling Salesman Problem

UAV – Unmanned Aerial Vehicle

VRP – Vehicle Routing Problem

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Problem Statement

1.1 Informal Problem Description

The industry of the civil Unmanned Aerial Vehicles (*UAVs*) has been growing rapidly in the last few years. In the past, just some rare enthusiasts worked with flying prototypes and devices. Nowadays, almost everyone can afford to buy a small *UAV*. Meanwhile, the companies spend their resources on more complex professional *UAVs* that can operate in very hard conditions, can lift quite big cargo, and give the detailed information about the environment, using the cameras and sensors.

Besides the obvious military applications, *UAVs* can be used in practice in many industries. Examples include:

- mining (finding the potential areas for mining using the sensors);
- agriculture (aerial crop surveys);

- aerial photography (for cartographic or 3D mapping, landslide measurement);
- search and rescue (finding the missed people in the forest);
- patrolling (border patrol missions, convoy protection, forest fire detection, crowd monitoring);
- data collection (inspection of power lines and pipelines, counting wildlife, environment monitoring, large-accident investigation);
- logistics (delivering the cargo, for example, the medical supplies to inaccessible regions; or the new fancy Amazon Prime Delivery);
- motion pictures (fancy effects in the movies, clips, footages); etc.

A *UAV* can be operated manually by a ground pilot using a remote control device. It works this way in most of the above mentioned examples. However, automatic control over a *UAV* can make a task faster and can exclude the risk of a human error.

In practice, automatic control over *UAVs* is mostly centralized. A pilot or an engineer can assign waypoints to a *UAV*, for instance, with the 'ground station' or 'mission planner' software. Moreover, the majority of mission planning problems for *UAVs* are solved offline, before any particular operation. After that, a *UAV* arms its motors, takes off from the ground, flies through the required waypoints in the environment, accomplishes some tasks, returns to the initial position, lands, and disarms the motors.

This approach works only for small-scale territories where every *UAV* is in the range of communication and we know the precise structure of the environment initially. In the early stages of our research work, we have implemented this off-line centralized algorithm by solving integer linear programs for optimal flights planning for every UAV.

This offline centralized solution, however, comes short in larger territories with unforeseen obstacles. It also may not be possible to have centralized communication with every agent during an operation. There are many real situations where we do have such constraints, for instance, in search and rescue operation after a catastrophic event; delivery of a medication to not easily accessable areas, etc. In such cases, we cannot use the offline centralized algorithm and we cannot solve the problem from a starting point (*nest*) in an optimal manner. These constraints motivate a need to develop online and decentralized step-by-step algorithms. Decentralization inherently leads us to implementation of a distributed system, where agents should decide how to operate during a mission, according to real situation with the environment, their current positions and all known decisions of the other agents (*rival* agents).

With this motivation, in the thesis, we focus on design and implementation of online decentralized algorithms for suboptimal multi-agent graph traversal. The environment here is modeled as a graph. Every node in the graph is a point for an agent to accomplish some task or to distinguish a point as an obstacle, where traveling is not possible. There can be made any kind of tasks, for example, search of a target, delivery of a cargo, taking a photo, recording a video, inspection of something, etc. Therefore, in such settings the problem can be reduced to the multi-agent distributed graph traversal ($MADGT$).

1.2 Thesis Statement

Our research hypothesis is that it is more practical to design distributed algorithms for online multi-agent graph traversal and to implement them on *UAVs* to carry out real-world autonomous missions, rather than to use offline algorithms and centralized control over *UAVs*.

In the rest of the thesis, we defend this statement and propose solutions that realize our claims.

1.3 Research Goals

Our goals in this research are

- to design an online distributed algorithm for multi-agent graph traversal;
- to implement an algorithm for computer simulation and experimentation with real *UAVs*;
- to produce and record a series of simulations of algorithm implementation with a different number of distributed agents, different parameters of agents, positions of initial points, obstacles, etc;
- to conduct real experiments with the implemented algorithm on real *UAVs* (multicopters).

Our approach is distributed to achieve robustness and autonomy in our system. There is no risk of loosing contact with the central unit, even if all agents except the last one stop their working due to a hardware malfunction, this last agent will finish a common mission. We assume broadcast communication among all agents. In our implementation we use long-range *WiFi* modules that can cover up to 2 km: an acceptable range for autonomous *UAVs* in real-world scenarios. Of course, wider ranges of communication are possible using 2G or 3G cellular communication. In most cases, the communication range depends on the quality of the signal and on external interferences. But if the test area has good coverage by the mobile operator we can have almost unlimited broadcast range for our agents.

As mentioned earlier, the implementation of the algorithm should work on real *UAVs*. This involves the following challenges:

- addressing atmospheric problems such as those experienced by aircraft (e.g., weather, humidity, wind, temperature, etc.);
- energy limits: i.e. we can fly a *UAV*, on average, only up to 15 minutes with a fully charged battery;
- complex hardware and software system integration. Our experiments depend on all sensors, moving parts, and electronics working at all times. The loss of any one of these can lead to a fatal breakdown of a *UAV*.

The algorithm should work in all complex real-world situations, for instance in search and rescue operation after a catastrophic event. It should take into account the hardware and software possibilities of the modern *UAV*. That is why, in this thesis, we make the following assumptions:

- an environment is modelled as an undirected graph;
- there exist, at any time, at least 2 agents at any initial nodes of a graph. The algorithm should not depend on the number and the initial positions of any of

the agents. However, as the algorithm should work in a distributed manner, at least two agents are required for any particular mission;

- there is a limited knowledge of the environment (no initial information about obstacles). An obstacle can be detected by an agent only in nodes adjacent to its current location. This assumption is based on the real capabilities of sensors and camera systems which, on average, are effective in the range of 1 to 40 meters;

- there is neither a communication protocol nor a common agreement between agents. Therefore, only a synchronous broadcast is used. This assumption mimics the most complex situations in which *UAVs* should work in, such as large territories with a lot of interferences of communication signals, and more specifically: areas where we cannot wait until the agents establish communication and come to an agreement. Please note that in this research it is assumed that the broadcast is global and perfect. However, in future work we need to adapt this algorithm model to under suboptimal communication conditions;

- the total number of targets is unknown, which is why the agents should travel through every node of a graph and should continue along in their job (such as the search and rescue of survivors post-disaster);

- the decisions about the next position that an agent will fly to is made by the onboard Raspberry Pi of each *UAV* using the latest local knowledge of the environment and broadcasts from other *UAVs* coming in through long-range *WiFi*. We also assume that only a limited amount of computations is made by every agent (as the embedded onboard computers are nowhere near as fast as model workstations);

- when an agent starts the work in a common mission, it continues to work until a termination condition is reached or the agent encounters some form of hardware problem. An agent cannot become idle so long as there is at least one remaining vacant node and should always travel with the intent on doing as much work as possible.

Figure 1.1 shows the overall picture of our multi-*UAV* system.

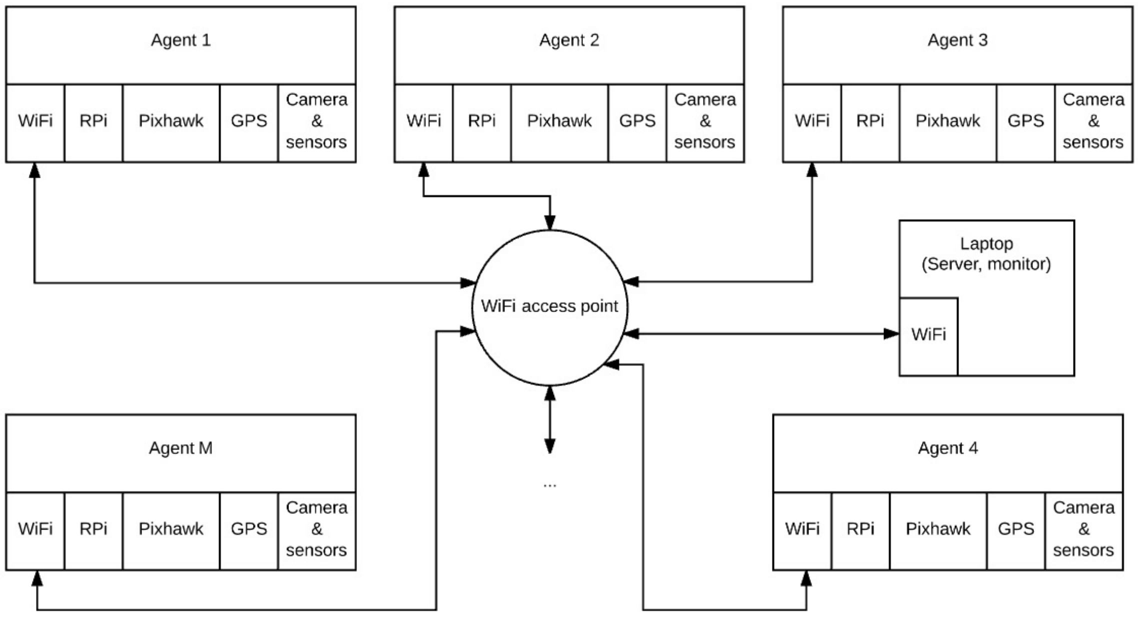

Figure 1.1: Global model of the system

Every *UAV* has a set of devices and sensors. The core computing hardware components of our *UAVs* are Pixhawk flight controllers and onboard Raspberry Pis (*RPi*). Flight controller takes the information from the *GPS* module, cameras, and sensors. This information helps to operate an agent in an environment, get the precise position, identify obstacles and avoid collisions with any rival agents. The onboard computer, in addition to the fact that it is running an implementation of the proposed algorithm, helps in communication through a long-range *WiFi* module, to which it is connected. The *WiFi* signal goes to an access point, which has even longer range *WiFi* antenna than any of the agents which is connected to a server. The server introduces the required mission parameters to the agents and, under the current implementation, helps with the broadcasts during the mission. Another feature of our implementation on the laptop server is that we are able to monitor the current situation of the agents, which we can see on the constantly updated visualization of the environment and in reported logs.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 presents our distributed algorithm. Chapter 4 describes an implementation of the algorithm. Chapter 5 discusses results of the simulations and real experiments. Chapter 6 concludes this thesis.

Chapter 2

Related work

In this chapter, we describe the research efforts that are closest to the problem under investigation in this thesis.

2.1 Distributed Constraint Satisfaction Problem

A distributed constraint satisfaction problem (*DCSP*) [Yokoo *et al.*, 1998] is a constraint satisfaction problem (*CSP*), in which variables and constraints are distributed among multiple agents. According to [Yokoo *et al.*, 1998; Yokoo and Hirayama, 2000; Modi *et al.*, 2005; Yokoo, 2012], a *CSP* consists of n variables $x_1, x_2, \cdots, x_n$ and a set of constraints at their values. Values of variables are taken from finite discrete domains $D_1, D_2, \cdots, D_n$. Each constraint is defined by a predicate $p_k(x_{k1}, \cdots, x_{kl})$. Predicate becomes true when values of variables satisfy constraints. Therefore, solution of a *CSP* is obtained when all assigned variables satisfy all constraints.

In an article [Yokoo *et al.*, 1998], authors solve a *DCSP* using a message passing communication model with a finite delay. An agent attempts to determine a value of its own variable according to inter-agent constraints. A relation $belongs(x_j, i)$ determines that a variable x_j belongs to an agent i and that only this agent can change a value of the variable. A relation $known(p_k, i)$ shows that an agent i knows a constraint p_k. Notice that agents can have partial knowledge of constraints.

A *DCSP* is solved when the next condition is satisfied: "$\forall i, \forall x_j$ where $belongs(x_j, i)$ the value of x_j is assigned to d_j and $\forall l, \forall p_k$ where $known(p_k, l)$, p_k is true under the assignment $x_j = d_j$"[Yokoo *et al.*, 1998] .

There are several assumptions, proposed by authors [Yokoo *et al.*, 1998]: (1) each agent has exactly one variable; (2) all constraints are binary; (3) each agent knows all constraint predicates relevant to its variable.

The authors [Yokoo *et al.*, 1998; Yokoo and Hirayama, 2000; Modi *et al.*, 2005; Yokoo, 2012] have proposed several algorithms to solve a DSCP:

- 'Centralized method', using a selection of a leader agent among all agents, which should gather all information about variables, domains, and constraints. Having such a global view, the leader solves a *DCSP* alone, using the ordinary centralized *CSP* algorithms, and translates its solution to all other agents. The main problem of such an approach is a huge communication overhead;

- 'Synchronous Backtracking', using an order, or priority of agents' moves. This order can be achieved via communication with a common agreement. Each agent instantiates its own variable based on constraints and sends this partial solution to the next agent according to the mentioned above order. The next agent appends its value to the partial solution and sends it further. If an agent cannot find a value that satisfies constraints, the agent sends a backtracking 'nogood' message to the previous agent. In such a case the previous agent should try to find another solution. Notice, that a partial solution is never modified, unless it is shown, that it cannot be a part of any complete solution;

- 'Asynchronous Backtracking', is an algorithm, which allows agents to run their computations concurrently and asynchronously. Each agent instantiates its variable concurrently. An agent sends its value to the lower priority agents and waits for their responses. The agent holds current value of the variable (on 'ok' messages) or makes a re-evaluation (on 'nogood' messages). For example, each agent has a set of values from connected agents (an agent's view), represented by a set of pairs $[(x_1, 1), (x_2, 2), (x_3, 3)]$. This means, that a value of a variable x_1 is 1, x_2 is 2, and so on. When an agent receives an 'ok' message, it adds a new pair to its agent's view and checks its consistency. Its own assignment of a value is consistent with its agent's view when all constraints are true, and all communicated 'nogood' messages are not compatible with the agent's view. If its own assignment is not consistent, the agent tries to change a current value, so that, it becomes consistent. A subset of an agent's view is called a 'nogood' when an agent is not able to find any consistent values of a subset. Agents try to avoid situations, previously found to be 'nogood'. However, due to a time delay in message delivery, an agent's view can occasionally be inconsistent, or identical to a previously found 'nogood';

- 'Asynchronous Weak-Commitment Search', is an algorithm, which aims to improve weaknesses of an asynchronous backtracking algorithm, where it is possible to have an exhaustive search after a bad choice of agents' priority. An asynchronous weak-commitment search algorithm: "introduces the min-conflict heuristic to reduce risk of making bad decisions. Furthermore, an agents' order is dynamically changed, so that, a bad decision can be revised, without performing an exhaustive search" [Yokoo and Hirayama, 2000]. According to this algorithm, each agent concurrently assigns a value to its variable, and sends this information to the other agents. This time, it sends it to all connected agents, i.e. to lower and higher priority agents. Notice, that a partial solution here is not modified. It is completely abandoned after only one failure or backtracking event.

The DCSP approach and algorithms do not work in our case, because:

- first of all, in the approach of authors [Yokoo *et al.*, 1998; Yokoo and Hirayama, 2000; Modi *et al.*, 2005; Yokoo, 2012], agents use communication with common agreement. Agents in such an approach do a 'move' only when everybody has agreed on a common solution, i.e. all changes are made to satisfy constraints. When any agent finds out a problem with satisfaction of constraints, it sends a 'nogood' message, and initiates re-computations. In our case, agents move asynchronously, without any common solution. They use an information from broadcasts to improve their own travel time and time of doing a job. However, agents do not especially care about other participants of a mission. We do have a rule, that who comes first to a precise position in a node takes it for a job. Therefore, there is even a contest between agents, rather than a common agreement;

- agents do not go back in our algorithm when some agent is 'not happy' about a current partial solution. Therefore, we cannot use any analog of a 'nogood' message. We cannot use a technique, like a synchronous or asynchronous backtracking, proposed by authors [Yokoo *et al.*, 1998] in their paper;

- authors of *DCSP* algorithms rely on a priority between agents, which helps them to avoid infinite loops of computation. We do not distinguish leaders between agents in our algorithm. All agents are equal in their decisions.

Therefore, we cannot use the *DCSP* approach and algorithms.

2.2 Traveling Salesman Problem

'Traveling Salesman Problem' (*TSP***)** is one of the most popular and intensively studied combinatorial optimization problems. It was formulated in the 1930s by Merrill Flood. Today, it is used as a benchmark for many optimization methods. The problem is computationally difficult. However, there exists a large number of heuristics and exact algorithms, using which, it is possible to solve even big instances with tens of thousands of nodes in a given graph.

The formal description of a *TSP*, given by a famous researcher [Laporte, 1992a]: "Let $G = (V, A)$ be a graph where V is a set of n vertices. A is a set of arcs or edges, and let $C : (C_{ij})$ be a distance (or cost) matrix associated with A. The *TSP* consists of determining a minimum distance circuit passing through each vertex once and only once. Such a circuit is known as a 'tour' or 'Hamiltonian circuit' (or cycle). In several applications, C can also be interpreted as a cost or travel time matrix. It will be useful to distinguish between the cases where C (or the problem) is 'symmetrical', i.e. when $c_{ij} = c_{ji}$, $\forall i, j \in V$, and the case where it is 'asymmetrical'. Also, C is said to 'satisfy the triangle inequality' if and only if $c_{ij} + c_{jk} \geq c_{ik}$, $\forall i, j, k \in V$".

Finding a Hamiltonian circuit is shown to be *NP*-complete [Garey and Johnson, 2002]. The *TSP* is an *NP*-hard problem, however, there are some special cases of

TSP, which it is possible to solve in polynomial time [Laporte, 1992a].

There are several variations of a classical *TSP*. One of them is called a '*TSP* with Time windows', where an agent needs to visit some certain nodes in a given amount of time. An 'Open *TSP*' is another variation of a *TSP*, where an agent does not need to come back to an initial node after it has visited all necessary nodes. There is such a comeback in the genuine *TSP*.

There are several ways to solve a *TSP* using

- 'Exact algorithms', which are based on integer linear programming (*ILP*) formulations, as given by [Dantzig *et al.*, 1954]:

  $Minimize\ \sum_{i \neq j} c_{ij} x_{ij}$ s.t.
  $\sum_{j=1 \cdots n} x_{ij} = 1,\ i = 1, \cdots, n$
  $\sum_{i=1 \cdots n} x_{ij} = 1,\ j = 1, \cdots, n$
  $\sum_{i,j \in S} x_{ij} \leq |S| - 1,\ S \subset V,\ 2 \leq |S| \leq n - 2$
  $x_{ij} \in 0, 1,\ i, j = 1, \cdots, n,\ i \neq j$

  In this formulation, a binary variable x_{ij} is equal to 1, if it is used in optimal solution, and $i \neq j$; otherwise, it is 0. First two constraints specify, that each node can be entered, or left only once. The third crucial constraint is a 'subtour elimination', which makes it impossible to have tours in subsets of nodes, less than total number n. The last constraint shows that it is an *ILP*. An objective function tries to minimize a total cost of an optimal tour. Overall, exact algorithms can give an optimal solution. However, they require very complex computations;

- 'Heuristics algorithms', which guarantee a worst-performance, or a good empirical performance. These algorithms require fewer computations, than exact algorithms, however, they give relatively good results in a reasonable amount of time. That is, why heuristics have such a big usage in practice.

The TSP approach and algorithms do not work in our problem, because:

A *TSP* is a way to find an optimal solution, however, it is only for one agent. We do have much more than one agent in our problem. That is the most obvious reason, with the other ones, why we cannot use a *TSP*. It is a different problem. However, there are generalizations of a *TSP* which can be closer to our theme. The first such a problem is a Multiple *TSP*.

'Multiple TSP' (*mTSP*) is a generalization of a *TSP*, such that, more than one agent is allowed to travel through a graph. In a general case, we are given n nodes (cities), m agents, one depot where agents are located initially, and costs of travel. A goal is to find a tour for each agent, such that, a total tour cost is minimized. Each node can be visited only by one agent, and only once. An ordinary *mTSP* formulation, given above, can be named a 'Single-depot *mTSP*', as it has only one depot. However, there can be multiple initial points. In such a case, it is named a 'Multi-Depots *mTSP*'. There are also several kinds of *mTSP*, according to a number of agents. This number can be given initially, or it can be bounded with some value. Thus, it can also be necessary to find out an optimal number of agents to make a work.

An *mTSP* can be solved with the exact algorithms or heuristics. Exact algorithms can contain a transformation of an *mTSP* to a *TSP*, as it was initially made by [Gorenstein, 1970; Svestka and Huckfeldt, 1973; Bellmore and Hong, 1974], or can be done without such a transformation, but, with a relaxation of some constraints, as it was proposed by [Laporte and Nobert, 1980; Gavish and Srikanth, 1986].

The mTSP does not work for us, because:

Our problem, in its formulation, seems to be similar to a Multi-Depot Open *mTSP*. However, a crucial difference is that an *mTSP* requires an offline optimal solution from a nest. It should be solved initially, and final paths are just translated to agents, as an exact map. This is not what we do in this thesis. Our goal is to create an online algorithm which should give to distributed agents only immediate solutions for their next short steps, i.e. which node to visit next, and in which node to do a job. We cannot use a linear programming or other similar instruments. That is why we cannot use the *mTSP* approach and algorithms to solve our problem.

2.3 Vehicle Routing Problem

Vehicle Routing Problem is a popular combinatorial optimization and integer programming problem.

'Vehicle Routing Problem' (*VRP*) is a generalization of a *TSP*. First, it was proposed by authors [Dantzig and Ramser, 1959]. A *VRP* solves a problem of how to find an optimal set of routes for a fleet of vehicles to serve a set of customers [Golden *et al.*, 2008]. A *VRP* problem is shown to be *NP*-hard [Lenstra and Kan, 1981], so, it is possible to solve it optimally only if it has a limited size graph.

A formal definition of a *VRP*, given by researcher [Laporte, 1992b]: "Let $G = (V, A)$ be a graph, where $V = \{1, \cdots, n\}$ is a set of vertices, representing cities with the depot, located at vertex 1, and A is the set of arcs. With every arc (i, j), such that $i \neq j$, is associated a non-negative distance matrix $C = (c_{ij})$. In some contexts, c_{ij} can be interpreted as a 'travel cost', or as a 'travel time'. When C is symmetrical, it is often convenient to replace A by a set E of undirected edges. In addition, assume, there are m available vehicles based at the depot, where $m_L \leq m \leq m_U$. When $m_L = m_U$, m is said to be 'fixed'. When $m_L = 1$ and $m_U = n-1$, m is said to be 'free'. When m is not fixed, it often makes sense to associate a fixed cost f for the usage of a vehicle. Each vehicle have the capacity D. The *VRP* consists of a design of a set of least-cost vehicle routes in such a way, that 1) each city in $V \backslash \{1\}$ is visited exactly once by exactly one vehicle; 2) all vehicle routes start and end at a depot; 3) some side constraints are satisfied".

There are a lot of types of a *VRP* [Caceres-Cruz *et al.*, 2014], that try to solve different practice problems:

- 'Asymmetric cost matrix *VRP* (*AVRP*)', where a cost of $(i, j) \neq (j, i)$. It was represented by [Laporte *et al.*, 1987];

- 'Distance-Constrained *VRP* (*DCVRP*)', outlined by [Laporte *et al.*, 1984, 1985; Li *et al.*, 1992]. A total length of arcs in a route of such problem cannot exceed some maximum route length;

- 'Heterogeneous fleet *VRP* (*HVRP*)', described by [Gendreau *et al.*, 1999; Li *et al.*, 2007b; Baldacci *et al.*, 2008], is closer to real-world situations. According to this problem, routes should be designed, taking into account different capacities of each vehicle. For an unlimited number of vehicles, a problem is called a 'Fleet Size and Mix *VRP* (*FSMVRP*)'. In a case, when some types of vehicles cannot reach some clients, a problem is named a 'Site-Dependent *VRP* (*SVRP*)'. If a vehicle is allowed to perform more than one trip, it is a '*HVRP* with Multiple usages of vehicles (*HVRPM*)';

- 'Multiple Depots *VRP* (*MDVRP*)' is a problem, in which there are several depots, from which a company can serve its customers. Thus, optimal routes, in such a case, can have the different starting and end points. A problem was researched by [Min, 1989; Chan *et al.*, 2001; Wu *et al.*, 2002; Nagy and Salhi, 2005; Ho *et al.*, 2008];

- 'Open *VRP* (*OVRP*)' is a problem, in which routes can be finished at nodes, different to initial depot locations. It was outlined, for instance, by [Brandão, 2004; Fu *et al.*, 2005; Li *et al.*, 2007a; Fleszar *et al.*, 2009];

- 'Periodic delivery *VRP* (*PVRP*)', where customers have different delivery frequencies. This problem was described, for example, by [Gaudioso and Paletta, 1992; Cordeau *et al.*, 1997; Alonso *et al.*, 2008; Hemmelmayr *et al.*, 2009];

- 'Pickup-and-delivery *VRP* (*PDVRP*)', researched, for instance, by [Savelsbergh and Sol, 1995; Dethloff, 2001; Berbeglia *et al.*, 2007; Bianchessi and Righini, 2007]. In this problem each customer has two parameters: a demand to be delivered to the customer, and a demand to be picked up and returned to a depot from the customer. This factor adds new constraints to a problem, such that, the total pickup and total delivery in a route cannot exceed a capacity of a vehicle at any point of time. There is another kind of such a problem, when a pickup demand should be delivered to another customer, instead of returning it to a depot;

- 'Split-delivery *VRP* (*SDVRP*)' is a problem, where the same customer can be served partially by different vehicles. For example, a customer can have a complex order, which is going to be delivered from different warehouses. A research on this theme was made, for example, by [Dror and Trudeau, 1990;

Archetti *et al.*, 2006a,b, 2008];

- ‘Green *VRP* (*GVRP*)’ is a problem, which includes as constraints different environmental issues, such as, the pollution, gas emission, waste, noise, etc. A research on this problem was done, for instance, by [Bektaş and Laporte, 2011; Erdoğan and Miller-Hooks, 2012];
- ‘*VRP* with Time Windows (*VRPTW*)’, where each customer can be served only in a given personal time interval. A research on this problem was done, for example, by [Solomon, 1987; Desrochers *et al.*, 1992; Potvin and Rousseau, 1993; Cordeau *et al.*, 2001; Bräysy and Gendreau, 2005];
- ‘Stochastic *VRP*’ is a more realistic problem, where we do have an uncertainty in a presence of a customer, in its demand, a service time we spend for a customer, a travel time between customers (because of a road traffic). This problem was outlined, for instance, by [Dror and Trudeau, 1986; Bertsimas and Van Ryzin, 1991; Gendreau *et al.*, 1995, 1996a,b].

As with an ordinary *TSP*, all algorithms to solve a *VRP* can be divided to the exact algorithms and heuristics.

The VRP does not work for us, because:

In a given classification of a *VRP*, we do have a problem similar to a Symmetric, Heterogeneous Fleet, Multiple Depots, Stochastic, Open *VRP*. However, we cannot use these approaches and algorithms, because they are offline centralized solutions from a nest. Also, there are no operations in depots, while in our case, we do have a job to do in each nest node. Therefore, as we need to create a distributed decentralized online step-by-step algorithm for multi-agent distributed graph traversal, we cannot use the *VRP* approach and algorithms.

2.4 Swarm Intelligence and Swarm Robotics

According to [Wikipedia, 2016b], "**Swarm intelligence** (*SI*) is the collective behavior of decentralized, self-organized systems, natural or artificial. *SI* systems consist typically of a population of simple agents, interacting locally with each other and with their environment. The agents follow very simple rules, and although there is no centralized control structure, dictating how individual agents should behave, l ocal, and to a certain degree random, interactions between such agents lead to the emergence of 'intelligent' global behavior, unknown to the individual agents. Examples in natural systems of *SI* include ant colonies, bird flocking, animal herding, bacterial growth, fish schooling and microbial intelligence. The application of swarm principles to robots is called **Swarm Robotics**".

The biggest inspiration for the *SI* algorithms comes from insects. According to [Prabhakar *et al.*, 2012], "Social insect colonies operate without any central control. Their collective behavior arises from local interactions among individuals". These model and behavior are similar to what we implement. We do have a distributed system of agents, which tries to solve a global problem in, some kind of, cooperation between each other.

From the most popular *SI* algorithms we can especially distinguish:

- 'Ant colony optimization', which was described by [Dorigo and Gambardella, 1997; Dorigo and Stützle, 2004; Dorigo *et al.*, 2008]. It is a class of optimization algorithms that model actions and decisions of a colony of ants. These algorithms aims to find a better path through a graph. Like real ants, that lay down pheromones to direct the other ants to a source of food, artificial agents keep the records, so that agents, which come after them, have a better knowledge of a graph;

- 'Bees algorithm', introduced by [Pham *et al.*, 2011; Pham and Ghanbarzadeh, 2007], mimics a food foraging behavior of swarms of honey bees. An algorithm starts with a random assignment of 'scout-bees' to some locations. After that, these scouts evaluate a fitness function of their areas. The area, that has the biggest fitness function, is selected for a more precise neighborhood search, which is made together with additional scouts;

- 'Particle swarm optimization', introduced by [Eberhart *et al.*, 1995; Kennedy and Eberhart, 1997; Shi and Eberhart, 1998; Eberhart and Shi, 2001],"is a computational method, that optimizes a problem, by iteratively trying to improve a candidate solution, with regard to a given measure of quality. It solves a problem, by having a population of candidate solutions (dubbed particles), and moving these particles around in the search-space, according to a simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best known position, but, is also guided toward the best known positions in the search-space, which are updated as better positions, are found by other particles. This is expected to move the swarm toward the best solutions" [Wikipedia, 2016a];

- 'Multi-swarm optimization', outlined, for instance, by [Liang and Suganthan, 2005; Blackwell *et al.*, 2004; Li and Yang, 2008; Marinakis and Marinaki, 2010], is an approach, that is close to a Particle swarm optimization, with a main difference, that there are a lot of sub-swarms, focused on the different regions of interest. It also has an additional problem of how to merge these sub-swarms;

- 'Ants Nearby Treasure Search' is one of the most modern approaches which we discuss here.

Ants Nearby Treasure Search ($ANTS$) problem was first introduced by [Feinerman *et al.*, 2012]. It became a generalization of a 'cow-path' problem, outlined by [Kao *et al.*, 1996] and other, which is relevant to collective foraging in groups of animals.

According to a model of [Feinerman *et al.*, 2012], there are k mobile agents, initially placed in a single cell of an infinite grid, which collaboratively do search for an adversarially hidden treasure. Agents are controlled by Turing machines. There is no communication. Authors [Feinerman and Korman, 2012] have proved, that it is necessary to have $O(D + D^2/k)$ time units to find a treasure, where D is a distance between a central location and a target. Such an approach was inspired by real creatures – Desert ants (Cataglyphys) and honeybees (Apis mellifera). This type of ants cannot rely on communication, due to a dispersedness of individuals, and their inability to leave chemical trails [Harkness and Maroudas, 1985]. However, they can successfully do the food sources search and they can perform: "the large-scale foraging excursions, and then, return to a nest by path integration" [Wohlgemuth *et al.*, 2001]. Desert ants use a similar pattern of search, as [Feinerman *et al.*, 2012] proposed in their work. Desert ants have a long, mostly straight, walk from a nest in some direction. After they reach a target area, they start a precise search, often with a trajectory, similar to a spiral [Harkness and Maroudas, 1985; Wehner *et al.*, 2004].

Another group of researchers [Emek *et al.*, 2014] have proposed an adaptation of this model, with the differences from [Feinerman *et al.*, 2012] in agents' computation and communication capabilities. Their agents are controlled by a weak control unit – asynchronous randomized finite state machine [Emek *et al.*, 2013], which does not allow these agents to store anything (like coordinates, or even a number of agents), or to reproduce any complex patterns of a search. However, agents may communicate with each other within the same cell. Authors [Langner *et al.*, 2014] provide a protocol, that allows agents to locate a treasure in time $O(D + D^2/n + Df)$, where D is a distance to a treasure, and $f \in O(n)$ is a maximum number of failures. Authors [Emek *et al.*, 2015] evaluate a number of agents necessary for a lucky search.

In contrast to this approach, our control system of each UAV gives agents enough computational power to do their jobs, even with more complex computations. For instance, we have a collision avoidance protocol, which requires a lot of computations of a complex data from the rangefinder sensors, cameras, etc. Our agents do have enough memory to store their own paths, as well as, the graph, travel and job speeds of each rival agent, current location of each agent, etc.

The ANTS approach is different from our in terms:

- our algorithm does not depend on a form of a graph, i.e. it works in any graph. The *ANTS* approach by [Emek *et al.*, 2014] is based only on a grid. Authors [Feinerman *et al.*, 2012] use an infinite grid, or a plane. Their agents can move only to the North, South, East, West, or can stay at the same node [Emek *et al.*, 2014]. However, our agent can choose any edge, connected to a node. There can be more than 4 adjacent nodes and, thus, more than 4 types of a movement.

Besides our agent does not stay in a node. An agent is always moving;

- all agents of the *ANTS* approaches by the mentioned authors are identical. In our case, each agent runs the same algorithm, however, their parameters can be very different. That is, each agents has a different job or travel speed, different target altitude, etc. Therefore, swarm control in our case becomes a more complex problem;
- there is only one starting point (a nest) in the mentioned *ANTS* approaches. We do not have this constraint in our algorithm. Each agent can start a job from any node, at any point in time. Therefore, we can have multiple nests;
- there is no communication in our algorithm – only a broadcast, or a global snapshot. In this meaning our approach is closer to [Feinerman *et al.*, 2012]. Authors [Emek *et al.*, 2014], in their research, explicitly rely on a communication within a cell;
- a form of a job accomplishing (for instance, a search) in our approach is also different from both of these. Our search can be made in different ways. It expands from an initial point in mostly random direction, which depends on local decisions of all agents. The idea is to move each agent as far, as possible, from each other in initial stage of a mission, to avoid possible collisions and multiple entrances to the same nodes. Later, agents come closer, as there are less not done nodes remaining. Meanwhile, authors [Emek *et al.*, 2014] in their paper use a 'diamond' search, which goes slowly around a nest, consequently expanding a radius of a search. Authors [Feinerman *et al.*, 2012] have a different solution. Their agents choose some direction from a nest, go at some distance

and start a spiral search from that place;

- there is not much treasures (targets) in the *ANTS* algorithms. In our case, all nodes can be imagined as targets. There is no chance to find these treasures somehow faster. We need to go through all nodes of a graph.

Therefore, we need to obtain a balance between the communication simplicity, minimalism of computations and a quality of doing jobs and travels. We need to get an information about a path and all metrics of each agent. That is, why in our case, having such complex hardware and software systems, as *UAVs* with the onboard computers, additional devices and sensors, it is necessary to have more complex agents, their control systems and behaviour. That is, why the *ANTS* algorithms and approaches do not work for us.

The Swarm Intelligence approach does not work for us, because:

As it was already mentioned about the *ANTS* problem, we do not have a better area, one or several treasures, etc. However, this factor prevails almost in all other *SI* approaches. We also do not have the inter-agent communication, which can be crucial. We keep a track of all accomplished nodes and can broadcast this information. This is similar to pheromones from the Ants Colony Optimization. However, we do not depend on a nest and we do not try to find a unique path from a nest to a 'food source'. We do not try to collect all agents at that place. We do not care about a formation of a group of our agents, and we even try to do everything, to move them further from each other, to avoid collisions. Because of these reasons, we cannot use the *SI* algorithms to solve our problem.

2.5 Multi-Robot Patrol Problem

'Multi-Robot Patrol Problem' (*MRPP*) is a problem, where: "agents must coordinate their actions, while continuously deciding, which place to move next, after clearing their locations. This problem is commonly addressed, using centralized planners with global knowledge, and/or calculating priori routes for all robots before the beginning of the mission" [Portugal and Rocha, 2013].

One of the first fundamental works on this problem was introduced by [Machado *et al.*, 2003]. In their paper authors have given several architectures and a bunch of evaluation criteria. There is a review of a *MRPP*, presented by [Portugal and Rocha, 2011], where they have given information about a condition of a research in this field for the last decade.

Most of researches on a Multi-Robot Patrol Problem can be called 'Adversarial Patrol', as they are facing with a presence of some adversary, or an opponent, an intruder. This opponent can be weak or strong in its knowledge of an algorithm, that is used by each agent of a patrolling team. In most cases, a question is how to coordinate agents, so they can catch an intruder as fast, as possible. We can especially distinguish a research on an Adversarial Patrol by [Agmon *et al.*, 2011, 2009].

There are two main fields of a research on a *MRPP*:

- 'Perimeter Patrol', outlined by [Agmon *et al.*, 2008b, 2009, 2008a; Marino *et al.*, 2009] and other, is a problem of how to guarantee the frequent visits of agents to borders of a given area, to prevent an intruder from an entrance to the area. There are two main approaches to solve this problem. It can be done using the deterministic, or non-deterministic patrol algorithms. When a strong opponent (intruder) is presented, i.e. an opponent that perfectly knows a deterministic algorithm, that is used by all agents, there is a probability close to 1, that the intruder can successfully enter the area. That is, why many researchers propose the non-deterministic algorithms, in which agents make their decisions with some probability. In such a case, even a strong opponent has less chances to enter the area;

- 'Area Patrol', presented, for instance, by [Elmaliach *et al.*, 2010; Fazli *et al.*, 2013], is a problem of how to guarantee frequent visits of agents to every part of an area, i.e. how to generate concrete patrol paths, such that every point in these paths is repeatedly covered. Every point of a given area can be repeatedly visited by one or more agents. Authors [Elmaliach *et al.*, 2010] have presented an algorithm: "that guarantees maximal uniform frequency, i.e., each point in the target area is covered at the same optimal frequency".

The Multi-Robot Patrol does not work for us, because:

- first of all, a patrolling itself has goals different to ours. We do not work currently on a finding of some moving object. Our goal is to accomplish all tasks in every node of a graph by distributed agents in the best possible time;
- we do not need to visit nodes multiple times with some frequency. We even do have a constraint of only one job in each node. And our objective function also minimizes a number of additional travels through nodes. The only, somehow close problem, can be a Non-Adversarial Area Patrol. However, it is similarly about multiple visits, so, it is different.

Therefore, the algorithms and approaches of a *MRPP* do not work in our case.

2.6 Multi-Robot Coverage Problem

'Multi-Robot Coverage Problem' (*MRCP*) is a modern problem of how to make a coverage of some territory in the best way, using multiple robots.

A *MRCP* has found an application in many fields, as de-mining, lawn mowing, harvesting, mapping, etc. For instance, there is a big usage of it in a navigation of cleaning robots. There is a bunch of papers on this theme, for example, by [De Carvalho *et al.*, 1997; Lawitzky, 2000; Jäger and Nebel, 2002; Luo and Yang, 2002; Luo *et al.*, 2003; Oh *et al.*, 2004; Liu *et al.*, 2008].

The *MRCP* can be divided to a coverage and a path planning. A path planning *MRCP* is similar to that, what we have described before in a *MRPP*. A coverage problem is closer to our theme.

The first solutions of a *MRCP* were made with heuristics, that can work good enough, but do not guarantee a success of a coverage, i.e. that agents will cover an entire environment space. For instance, there can be used some simple behavior, or a randomized search, which can work on simple robots, without expensive sensors. Such randomized approach was proposed by [Gage, 1994] and has achieved good results in a problem of de-mining where a probability of a mine finding by a robot in one search is less than one.

Later, there came algorithms, based on a cellular decomposition, which can guarantee a total coverage. There are three main types of a cellular decomposition:

- 'Approximate', where a space is represented by a grid with the same size cells. A union of such cells can only approximate an environment space. However, it is assumed, that if a robot enters a cell, it covers it entirely. This approach was proposed by [Moravec and Elfes, 1985; Elfes, 1987];

- 'Semi-Approximate' cellular decomposition – is a partitioning of a space by cells, which can be fixed in a size of one of its parameters, as a hight or a width, but otherwise, can be of any form. The are papers on this theme by [Lumelsky *et al.*, 1990; Hert *et al.*, 1996];

- 'Exact' cellular decomposition – is a set: "of non-intersecting regions, each termed a cell, whose union fills the target environment" [Choset, 2001]. These cells are made that way, that a robot can cover each cell with simple back-and-forth movements. That is, a coverage problem is reduced in such a case to a planning of a movement between cells. There is a 'trapezoidal decomposition' [Seidel, 1991], where each cell is represented by a trapezoid, i.e. a convex quadrilateral with at least one pair of parallel sides. Another popular type of such an exact decomposition is called a 'Boustrophedon decomposition'. It was developed by [Choset and Pignon, 1998; Choset *et al.*, 2000]. Later, it was used in many *MRCP* papers, for instance, by [Rekleitis *et al.*, 2004; Kong *et al.*, 2006]. In this decomposition, "a line segment (slice) is swept through the environment. Whenever there is a change in connectivity of the slice, a new cell is formed. When the connectivity increases, two new cells are spawned". When decreases - the cells are merged into one [Choset, 2001].

After defining a type of a decomposition to take in a *MRCP*, the next decision should be made on an inter-agent communication. There are a lot of papers, that deal with an unlimited communication range, for instance, by [Kong *et al.*, 2006; Rekleitis *et al.*, 2008], or a limited communication, for example, by [Rekleitis *et al.*, 2004; Sheng *et al.*, 2006].

If agents use a communication, in most cases, these *MRPP* algorithms work with a common agreement between agents. For example, authors [Sheng *et al.*, 2006] in their research use a 'bidding protocol', according to which every agent makes its bid about the next cell, and the best agent takes it, according to a common agreement. Authors [Kong *et al.*, 2006] use a similar 'common task selection protocol'.

The difference of our approach and the algorithm from the current solutions in a MRCP:

- we do solve a more general problem. Our agents can do any job during a mission. It can be search, coverage, cleaning, delivery, etc. There is an assumption, that each node should be visited, and in each node, an agent should do some amount of work. However, it can be a different job.

  In a case of a search problem, it can be video recording, or taking a series of photographies of a territory. This can significantly depend on the cameras and computer vision algorithms, that are used by each agent. Therefore, a search of a same territory will take a different search time for each agent. For any our problem, there should be assigned some measure for each node. We can take as such a measure, fro example, a radius or a precision of a search. Each agent can evaluate a search time for a concrete node.

  In case of a delivery, in every target node, an agent should do a landing, put a package, and make a takeoff to a target altitude to continue a mission. Again, it can be translated to a time of job in every node, depending on a location of a place, challenges of making a landing, and so on.

  Our algorithm can successfully work in a coverage and a cleaning. For instance, as a graph we can take a grid with the same distances between each adjacent node. All cells, in such a case, will be of the same size. Therefore, an amount of a work for each node becomes the same. Agents need to travel to each node and do a cleaning, or a coverage of each cell, i.e. some radius of a node. We do have used such an approach in our simulations and the *UAV*'s implementation;

- we do not use a rendezvous communication between agents, and do not run biddings, or some similar procedures, to find out the best agent to go 'to that node'. Our agents are mostly greedy, and they are always in a movement. They fight for each next node up to the end. Our aim is to show, that even if we do not have an inter-agent agreement, we still can achieve suboptimal results, with just a broadcast and movements step-by-step. Besides, we save some time, when exclude a stage of a bidding from our algorithm. A solution about the next node is achieved almost immediately, and agents do not stop. Every agent gets an information about movements of rivals, so, they can evaluate their chances on every next step. And they can implicitly 'guess' all destinations of rivals;
- we do not go with an obvious approach for agents from a *MRCP*, to stay closer together, if they have a limited communication range. In opposite, we propose and prove an idea, that it is better to disperse agents further from each other in initial stage of a mission, and to come in the same area, only at the end of the mission. Even in our future algorithms with a limited-range broadcast, we explicitly force agents to go closer to borders of a graph initially, when they do not 'see' rivals. From borders, they should consequently move closer to a middle of a graph. If they have received a broadcast from some agent, they start to move, as in a case of an unlimited range broadcast.

Therefore, we use some elements of a *MRCP* in our solutions. However, we cannot use the entire algorithms of *MRCP* in our problem.

Chapter 3

Multi-Agent Distributed Graph Traversal Algorithm

3.1 Formal Problem Description

Let $A = \{1, 2, \ldots, m\}$ be a set of agents and $G = \langle V, E \rangle$ be an undirected graph where $V = \{1, 2, \ldots, n\}$. We consider two cost functions $J : A \times V \to \mathbb{Z}_{\geq 0}$ (the cost of accomplishing a job at a vertex of G) and $T : A \times E \to \mathbb{Z}_{\geq 0}$ (the cost of traveling from one vertex to another). A walk of G is a sequence $w = v_1 \cdots v_n$, where for all $i \in [1, n)$, we have $(v_i, v_{i+1}) \in E$. Let $V'_a = \{v \mid v \in V \wedge \text{an agent } a \text{ does a job in } v\}$ and

$$V' = \bigcup_{a \in A} V'_a$$

i.e., V' contains all nodes where a job was done by all agents.

Thus, the cost of a walk w, taken by an agent $a \in A$ (denoted $C(a, w)$) is

$$C(a, w) = \sum_{v \in V'_a} J(a, v) + \sum_{i \in [1,n)} T(a, (v_i, v_{i+1}))$$

Our problem is to find a set W of walks and a plan function $P : W \to A$, such that

$$\bigcup_{v \in V'} = V$$

i.e., the walks cover all vertices of G except obstacles, and P achieves the following optimization objective

$$\min \sum_{w \in W} C(P(w), w)$$

3.2 Algorithm Description

3.2.1 Overall Idea of the Algorithm

One of the most popular solutions for distributed multi-agent coordination without communication and agreement between agents is that they use some formations (swarm) and go through an environment close to each other doing their jobs. An idea of our Multi-Agent Distributed Graph Traversal Algorithm (*MADGTA*) is to obtain suboptimal solutions without common agreement and without using formations. Furthermore, we have shown in our simulations and real experiments with *UAVs*, that it is more beneficial to disperse agents through a graph in initial stage of a mission. Such disperse helps agents to avoid collisions, when they go to the same node to do a job. If they work in different subareas of a graph an amount of collisions is reduced significantly.

Every iteration of the *MADGTA* goes through the popular in a 'control world' procedures LOOK, COMPUTE and MOVE, as it is used, for instance, by authors [Cieliebak *et al.*, 2003; Gervasi and Prencipe, 2004; Klasing *et al.*, 2008; Flocchini *et al.*, 2008; Klasing *et al.*, 2010]. The first procedure LOOK updates agent's knowledge of a graph. The second procedure COMPUTE finds out the best node for a travel and a job. The last procedure MOVE does these travel and job, as well as updates agent's knowledge with a node done by itself.

In our algorithm, every agent starts a mission in its initial *nest* node. Notice, that an initial node can be different for every agent. That is, a number of nests in a mission is from 1 to m, where m is a total number of agents. Agents do the jobs in the nest(s). After that, they update their knowledge of a graph using incoming broadcasts from rival agents. In the next step, each agent computes its chances (*utility*) of going to one of its adjacent nodes. A node has the best utility for an agent if it is harder reachable for all rival agents. This leads to a situation, that in every next step of the *MADGTA*, an agent tries to move further from its rivals. That is, all agents disperse in a graph.

After agents have done their jobs mostly in the border areas of a graph, they come closer to a middle part of the graph. Therefore, they come closer to each other and finish their work in the entire graph. A graph (grid) in Figure 3.1 clearly demonstrates this approach. Agents of three different colors start a mission from the initial node 1 (left bottom corner). After that they clearly disperse in the graph. Notice, that in the given graph a node has a color of an agent that have done a job in it, while colored edges represent a travel of that agent. Black squares show obstacles.

Our algorithm is based on a greedy nature of agents. In our current implementation of the *MADGTA* without communication and common agreement, agents are rivals, rather than collaborators. However, such competition between agents, with the proposed restrictions for each agent, leads to a suboptimal solution of a given problem.

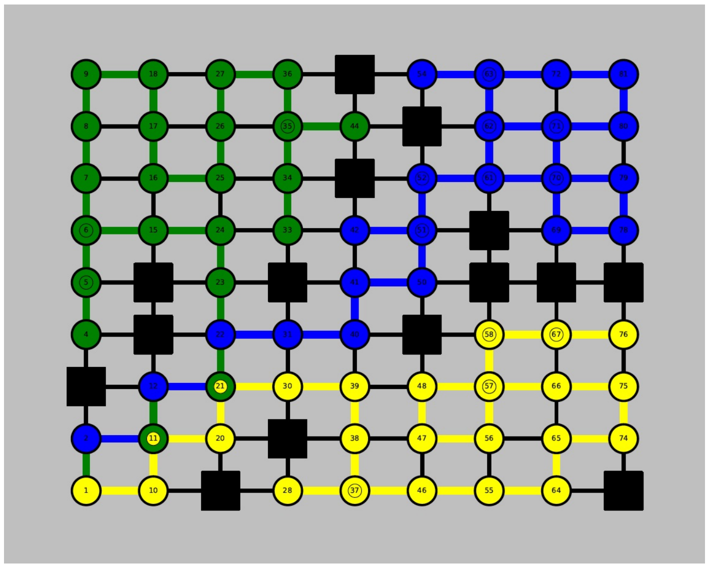

Figure 3.1: Dispersion of agents

Notice, that the *MADGTA* works in any undirected graph. For example, there can be some nodes in a graph of a big degree (10, 20, etc). The algorithm gives the best adjacent node in such a case, the same way, as it makes it in simpler graphs, like grids. Therefore, for simplicity of explanation we use grids in all our examples, simulations and experiments in this thesis. For a future work, especially on a problem of energy efficiency and in an implementation of the algorithm with *UAVs* in a complex environment with a lot of heels and obstacles of different height, we want to use a three dimensional graph representation of an environment, rather than a grid.

3.2.2 Notation and Assumptions

The input to our algorithm is an undirected graph $G = \langle V, E\rangle$, a set of agents A, and their associated cost functions. The output of the *MADGTA* is a walk of all nodes traveled and accomplished by an agent. The algorithm terminates if all nodes of a graph are visited by agents. It means, that agents accomplished jobs in nodes or nodes were identified as obstacles.

Assumptions: Each agent $a \in A$ is initially located in some arbitrary node of a graph G. Each agent has two primitives for synchronous broadcast and computing a global snapshot of a graph. An agent has enough memory to store nodes covered by itself ($PATH$) and a current location of all agents, as well as a list of all nodes accomplished by any other agent. Thus, an agent can determine a status of each rival agent at each point of time. An agent a has no battery limit, i.e. it can do all work in the entire graph. All decisions about the next moves are taken by each agent locally, without any centralized control. There is no inter-agent common agreement. All agents are equal in their status (there is no hierarchy or leadership). Agents can enter the same node in a graph, however, there is no physical collisions. That is, because agents run a collision avoidance protocol. An agent knows all cost functions J and T. An agent can distinguish obstacles in adjacent nodes.

Notation: Let function $loc : A \to V$ return a current location of an agent, according to the last global snapshot. Set *KM* defines current agent's knowledge of a graph. It is a set of all nodes accomplished by any agent or nodes revealed as obstacles. Set *UN* denotes a set of all not accomplished nodes respectively. Set O denotes nodes known as obstacles. Variable *SP* denotes a minimum time that is required for an agent to travel through a shortest path distance between two given points, according to its unique travel speed. Sets $util$, $util_v$, $utilmax$, $utilmaxmin$ and UN are sets of temporary local variables. Function $adj()$ gets all adjacent nodes of a node given as an input parameter. Sequence $PATH$ stores a sequence of nodes traveled and accomplished by a current agent.

Initially, all nodes in a graph are marked as 0 (there was done no job in these nodes). An example is shown in Figure 3.2. These marks are depicted in top right corner of every node. If an agent encounters an obstacle it changes a value of a node to -1. If a node is accomplished by an agent a value is changed to an agent's ID number, which is some number, more or equal 1. This is shown in Figure 3.3, where all nodes have values equal to 1, 2 or -1. Values 1 and 2 are specific IDs of agents. A value -1 demonstrates an obstacle.

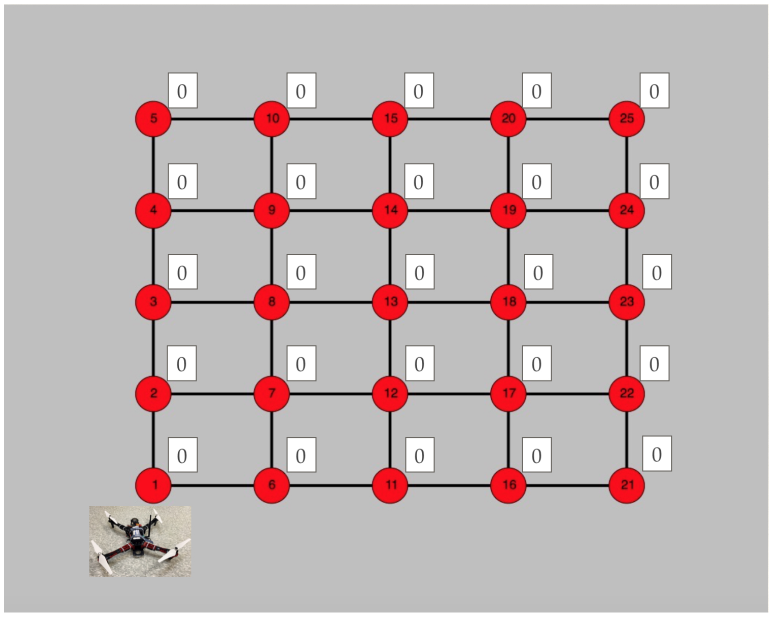

Figure 3.2: Initial graph

However, to make it more clear in most examples given further in this work we represent obstacles with black squares. Nodes, where agents accomplished jobs, are depicted with circles of some color (assigned to each agent). In the end of a mission there should be no red nodes (which depicts not discovered nodes). Such a graph is given in Figure 3.4, which represents the same example. In some cases, we also add edges of a color of an agent to depict its travel.

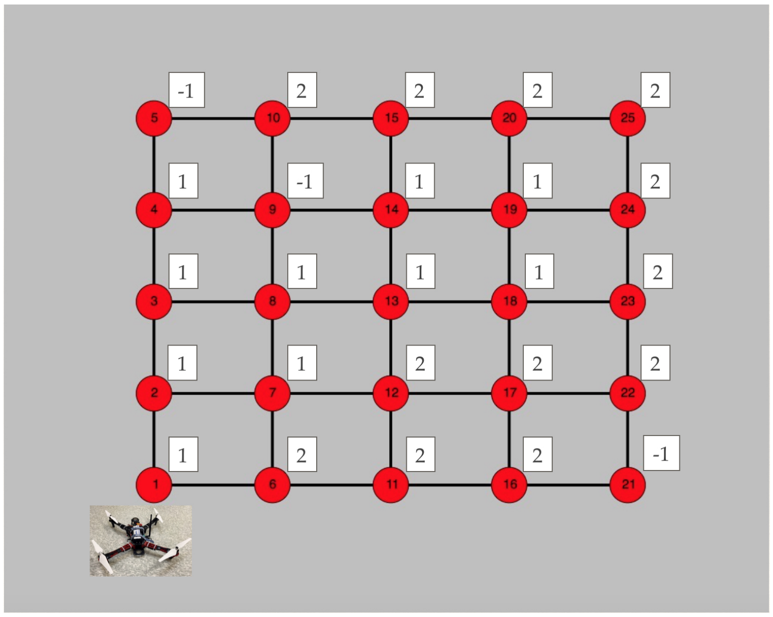

Figure 3.3: Final graph

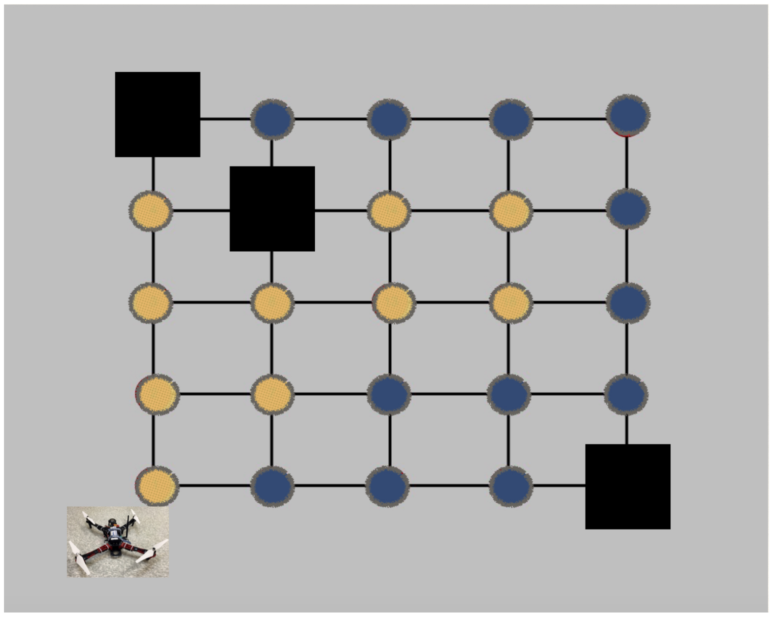
Figure 3.4: Final graph representation using colors

3.2.3 Detailed Description

In the *MADGTA*, the first line initializes a set KM and a sequence $PATH$ as empty set and sequence. Set UN is equal to a set V of all vertices in a graph.

A while loop (lines 2 to 39) runs until a set KM is equal to a set V, i.e. until a knowledge of vertices of a graph is not becoming full. This condition is a termination condition of the *MADGTA*. The while loop executes three consecutive procedures, namely the Look (lines 3 to 6), Compute (lines 7 to 29), and Move (lines 30 to 38). In the procedure look (line 4), we define a set of obstacles O. As it is given here, an obstacle can be found only in an adjacent node to a current node. In practice, a UAV has the limited range sensors to find out obstacles. That is why we use such assumption.

Algorithm 1 Agent a

Input: An undirected graph $G = \langle V, E \rangle$

Output: $PATH$

```
1: $KM = \emptyset$, $PATH = \langle\rangle$; $UN = V$;
2: while $|KM| \neq |V|$ do
3:     procedure LOOK
4:         Let $O = \{v \mid (loc(a), v) \in E \wedge \text{there is an obstacle in } v\}$;
5:         $KM \leftarrow KM \cup \{loc(a')\}_{a' \in A} \cup O$;
6:     end procedure
7:     procedure COMPUTE
8:         if $(loc(a)) \notin KM$ ) then
9:             $v \leftarrow loc(a)$;
10:            return $v$;
11:        end if
12:        if $(\exists u.u \in adj(loc(a)) \wedge u \notin KM$ ) then
13:            $W \leftarrow \emptyset$;
14:            for each $u \in adj(loc(a)) \wedge u \notin KM$ do
15:                $W \leftarrow W \cup \{u\}$;
16:            end for
17:            $v \leftarrow GetBestNode(W)$;
18:            return $v$;
19:        end if
20:        if $(\nexists u \mid u \in adj(loc(a)) \wedge u \notin KM$ ) then
21:            $W \leftarrow \emptyset$; $UN \leftarrow UN \setminus KM$;
22:            for each $u \in UN$ do
23:                $W \leftarrow W \cup \{u\}$;
24:            end for
25:            $f \leftarrow GetBestNode(W)$;
26:            Let $v$ be an arbitrary node in $\{v \mid (loc(a), v) \in E \wedge v \text{ is on the } SP(f, loc(a))\}$;

27:            return $v$;
28:        end if
29:    end procedure
30:    procedure MOVE($loc(a), v$)
31:        $loc(a) \leftarrow v$;
32:        $PATH = PATH.\langle v \rangle$;
33:        if $(v \notin KM)$ then
34:            Job $(v)$;
35:            $PATH = PATH.\langle v* \rangle$;
36:        end if
37:        $KM \leftarrow KM \cup \{v\}$;
38:    end procedure
39: end while
```

The major part of the procedure LOOK is given in line 5. In every iteration of an outer while loop, an agent updates its knowledge of a graph (KM) with a set of nodes traversed by all rival agents ($\{loc(a')\}$) and a set of all obstacles known to date (O). With an updated knowledge of a graph, we can do an essential part of the *MADGTA* defined in the next procedure COMPUTE.

We now describe the procedure COMPUTE. This procedure has three main parts (if conditions) in lines 8 – 11, 12 – 19, and 20 – 28. The first if condition (lines 8 – 11) checks if a current node of an agent (i.e, $loc(a)$) is not in a set KM, i.e. not done yet. When a current node is vacant to do a job, $v = loc(a)$, i.e. the same node is returned to the procedure MOVE. Notice, that this if condition works only for a nest node.

Second if condition (lines 12 – 19) checks if there are adjacent nodes u which are not done yet and are not obstacles (not in a set KM). When there are such vacant nodes, an agent should evaluate a set of utilities ($util$) of going to every such a node, using a function $GetBestNode(W)$. This function is described later. For now, assume that a result of the $GetBestNode(W)$ is the best next node which should be given as an input to the last procedure MOVE.

As a parameter of the function $GetBestNode(W)$ we need to form a set of goal nodes. We create an empty set W (line 13) and assign to it all vacant adjacent nodes that are not obstacles (lines 14 – 16). In line 17 we execute the function $GetBestNode(W)$ and get the best next node v for a travel and a job.

The third if condition (lines 20 – 28) describes a case when there is no vacant adjacent node to a current. This time an amount of computations is much bigger, as an agent should find out the best further node to travel and do a job. There can be multiple vacant nodes.

In line 21, the algorithm gets a set of all nodes which are known to be vacant. In the first run of this if condition a set UN is equal to a set of all nodes V (line 1). Later, the *MADGTA* works just with a reduced set UN. A set W is initialized as an empty set.

In lines 22 – 24 the algorithm assigns to a set W all nodes that are vacant. In line 25 the *MADGTA* finds the best further node f for a travel and a job. However, the algorithm works step-by-step and cannot assign an entire path to an agent. That is why we need to choose only the next intermediate adjacent node. This is done in line 26. According to this condition, an agent should take as a node v, a node which is adjacent to a current location, and which is in a shortest path to the best node f.

The procedure Move initiates a travel of an agent from a current node ($loc(a)$) to an adjacent node v which is given by the procedure Compute. This step is shown in line 31 where a node v is assigned to a current location of an agent. An agent updates a path (line 32). In lines 33 – 36 (if condition), when a new node of an agent is known to be vacant, i.e. it is not in a set KM (line 9), an agent should do a job in this node. After a job is done, an agent updates a sequence $PATH$ (line 35) and its set KM with a node done by itself (line 37), so we can find out the entire path of this agent from a nest.

An agent does a job when it travels to a new node, if possible. Notice that, agents cannot jump to some profitable areas of a graph. Therefore, having such a requirement we reduce a greedy nature of agents. There are some cases, when such a solution is not the best. However, as it is shown by our simulations and real experiments, this solution is suboptimal in most cases. The *MADGTA* goes again through the same procedures until a termination condition is achieved.

The function *GetBestNode*(W) is shown in an algorithm 2. It starts with an initialization of a set *util* as an empty set (line 2).

Algorithm 2 Function GetBestNode from the set W

```
function GETBESTNODE($W$)
2:      $util \leftarrow \emptyset$;
        for each $v \in W$ do
4:          $util_v \leftarrow \emptyset$;
            for each $a' \in A \setminus \{a\}$ do
6:              $util_v \leftarrow util_v \cup$
                        $\{(SP(v, loc(a') + J(a', v)) - SP(v, loc(a))) - J(a, v)\}$;
8:          end for
            $util \leftarrow util \cup util_v$;
10:     end for
        $utilmaxmin \leftarrow \max_{v \in adj(loc(a))}(\min_{a' \in A \setminus \{a\}}(util))$;
12:     if $|utilmaxmin| = 1$ then
            Let $v$ be a node that resulted in $utilmaxmin$;
14:         return $v$;
        end if
16:     if $|utilmaxmin| \neq 1$ then
            $utilmax \leftarrow \max_{v \in utilmaxmin} \sum_{a' \in A \setminus \{a\}} util$;
18:         if $|utilmax| = 1$ then
                Let $v$ be a node that resulted in $utilmax$;
20:             return $v$;
            end if
22:         if $|utilmax| \neq 1$ then
                Let $v$ be an arbitrary node from $utilmax$;
24:             return $v$;
            end if
26:     end if
    end function
```

There is a nested for loop over all nodes in a given as a parameter set W and all rival agents a' (lines 3 – 10). The aim of these construction is to create a set *util*, which holds tuples of utilities values for all necessary nodes and rival agents. A temporary set $util_v$, initiated in line 4, and defined in line 7, is a difference between a time of a travel through a shortest path to a node of interest v and a time cost of doing a job in that node $J(a', v)$ by a rival agent, and the same time, of a travel and a job of a current agent a. At the end of these for loops (line 10), an agent should have a set $util$, which is a set of sets of utilities $util_v$ per each node $v \in W$. For example, it can be $\{-1, 8\}, \{3, 3\}, \{3, 4\}$ in a case, if an agent looks at three potential adjacent nodes, and has only two rival agents.

Notice, that a set $util_v$ mimics a cost of a walk w, given earlier in a formal description of a problem. This way we minimize a sum of costs for each agent, and achieve an optimization objective. There were tried different approaches to evaluate utilities, including slicing, utilities arrangement for each node from a nest, etc. However, a formula for a utility defined in line 7 gives the best results.

In line 11, an agent creates a new set *utilmaxmin*, which holds the maximum values from all minimums in each tuple of a set $util$. In a previous example, minimums form a set of $-1, 3, 3$. The maximum of this temporary set is a value 3. Notice that in a given example, the first node has the biggest utility 8 for one rival, and the same time the smallest utility -1 for another. We cannot take just a node with the biggest utility. In this case, an agent can go to a node which is very far from one rival, but the same time, it is closer to another. There is a bigger risk that such travel will be useless, i.e. an agent can become not the first agent to do a job in that node. This is a reason why in the first stage of evaluations of the function $GetBestNode(W)$ we use a maximum over minimums among utilities.

If there is only one resulting in a set *utilmaxmin* node, a function executes an if condition (lines 12 – 15), according to which this single resulting node v is returned by the function.

However, there can be more than one resulting node (see an example). In this case, the function executes an if condition in lines 16 – 26. It evaluates a set *utilmax* (lines 17), which combines a maximum of summations of values in tuples of a set *util* for all promising nodes, that resulted in a previous step in *utilmaxmin*. In our example, a *utilmax* should find a maximum over summations between 6 and 7. The maximum here is 7, i.e. the third node in a set of tuples becomes the best node for a travel and a job. This is defined in the if condition (lines 18 – 21), according to which the function returns this best node. Notice, that we cannot evaluate just a set *utilmax* without preliminary evaluation of a set *utilmaxmin*. If we try to make it, a *utilmax* for a given example could give us the first node either, as it has the same summation value 7, however, we already have shown, that the first node is the worst as the next node for a travel and a job.

The last possible outcome is that, even after this stage, an agent can have few best options, which becomes equal in a previous stage in *utilmax*. In such a rare case, according to lines 22 – 25, the function should return only one random node v, that was resulted in a set *utilmax*.

3.2.4 An Example of Execution of the Algorithm

An example of execution of the *MADGTA* is given here.

An initial graph is given in Figure 3.5. It is a 5×5 nodes grid. There are obstacles in nodes 3, 13, 14 and 19, not known by agents initially. There are 3 agents in the nest-nodes 1 (yellow agent), 11 (blue agent) and 23 (green agent). All nodes are initially red (not done). An obstacle is depicted with a black square. A node done by an agent takes a color of that agent.

Agents are identical. They have the same time of a travel equal to 2 sec per one edge. A time of a job in a node is equal to 9 sec. As a time of doing a job, for simplicity, is taken the same between all agents, it gives us easier computations in a function *GetBestNode*(W), as all job cost values J compensate each other. Agents start their work almost synchronously.

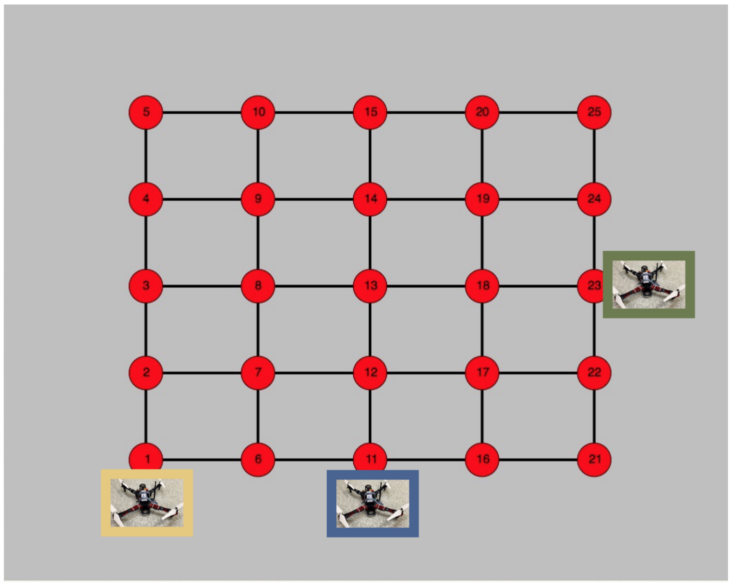

Figure 3.5: Initial graph for an example of execution of the algorithm

Now, we look at the *MADGTA* execution only for an agent, which starts from node 11 (blue agent). The execution is given consequently for each iteration of a while loop of the algorithm:

1. In the first iteration of a while loop, in procedure Look an agent finds out that there is nothing to add to a KM. It goes to the first if condition of the procedure Compute. Current node 11 is returned as a node v. In procedure

Move a location remains to be the same, however, a job is done (see in Figure 3.6). An accomplished node is added to a set KM. So at the end of the first iteration of a while loop, a set KM is equal to $V \setminus loc(a)$. An agent updates a path with a starting node and a sign, that it was done. We use a number of a node with a star to show an accomplished job in a node. The final path of each agent is demonstrated later. Notice, that a path is updated in every iteration;

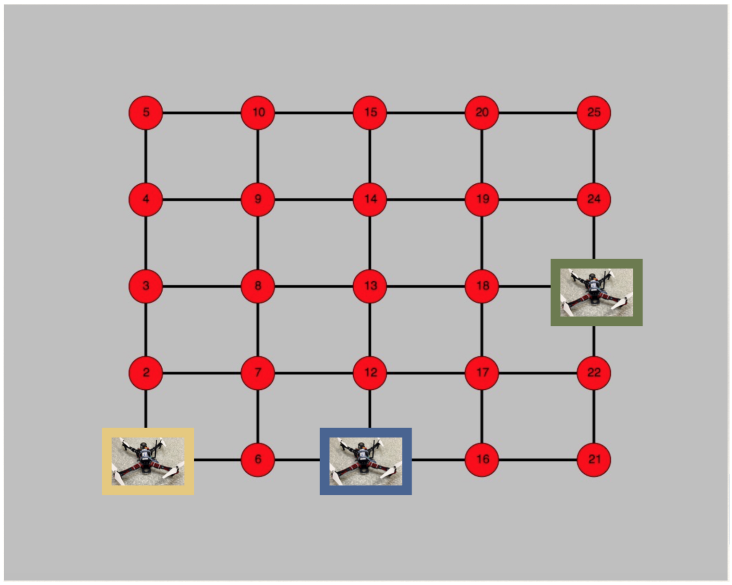

Figure 3.6: Step 1 of the example

2. In the procedure Look an agent updates its KM with nodes done by the rival agents (nodes 1 and 23). There were no obstacles found yet. It goes to the procedure Compute. An agent in node 11 has 3 vacant adjacent nodes 6, 12, and 16. A set *util* is equal to $\{\{0, 8\}, \{4, 4\}, \{4, 4\}\}$. Maximum value from minimums is 4, which gives it two nodes in a set *utilmaxmin*, namely 12 and 16. Asathe cardinality of a set *utilmaxmin* is not equal to one, the function goes to the next if condition and evaluates a maximum summation of utilities per node

for remaining nodes. So the function looks only at values of nodes 12 and 16. A maximum summation is equal to 8. Both nodes 12 and 16 has such a value. This moves the function to the last if condition, according to which, it should choose a random node from all equally best. In this run of the *MADGTA*, a node number 12 was chosen as the best next node. In procedure Move a travel and a job were done in node 12 and a set *KM* is updated (see in Figure 3.7);

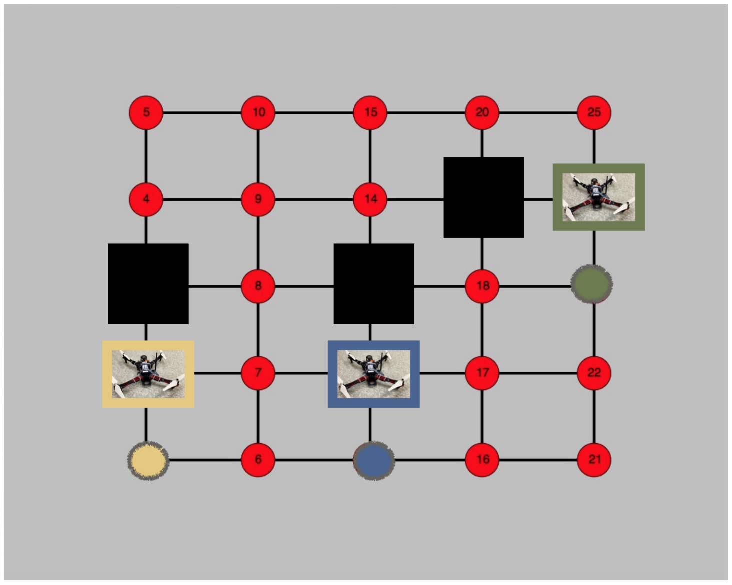

Figure 3.7: Step 2 of the example

3. The procedure Look updates a set *KM* with new nodes of rivals (2 and 24), as well as, with three obstacles found in nodes 3, 13 and 19. These obstacles from now change a graph structure and take a role in evaluation of a shortest path. The procedure Compute for a blue agent is simple at this step as it has just one vacant adjacent node number 17. In the procedure Move node 17 is done (see in Figure 3.8);

4. The procedure LOOK updates a set KM with new nodes of rivals (7 and 25). The procedure COMPUTE for a blue agent gives a set *util* equal to $\{\{4,8\},\{4,4\},\{4,4\}\}$ for adjacent nodes 16, 18 and 22. A maximum value from minimums is 4. A set *utilmaxmin* is even bigger now, and holds all 3 nodes. However, this time, a set *utilmax* gives only one best node 16 (the summation values are 12, 8 and 8). In the procedure MOVE node 16 is done (see in Figure 3.9);

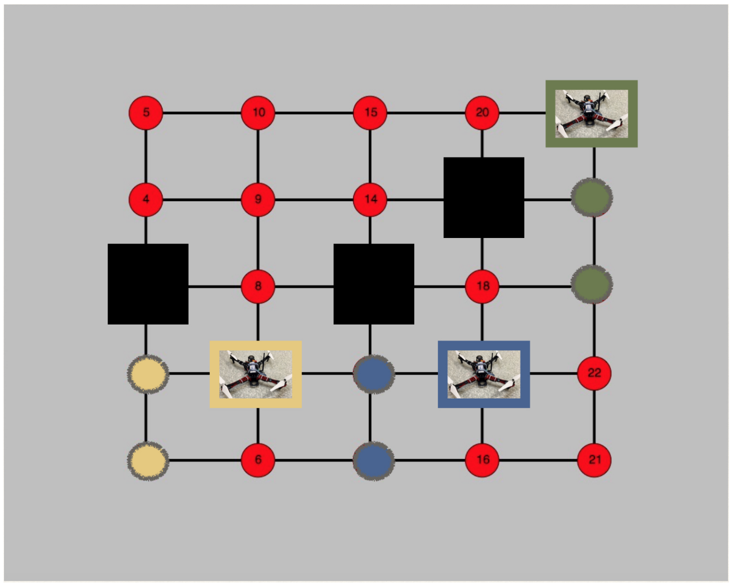

Figure 3.8: Step 3 of the example

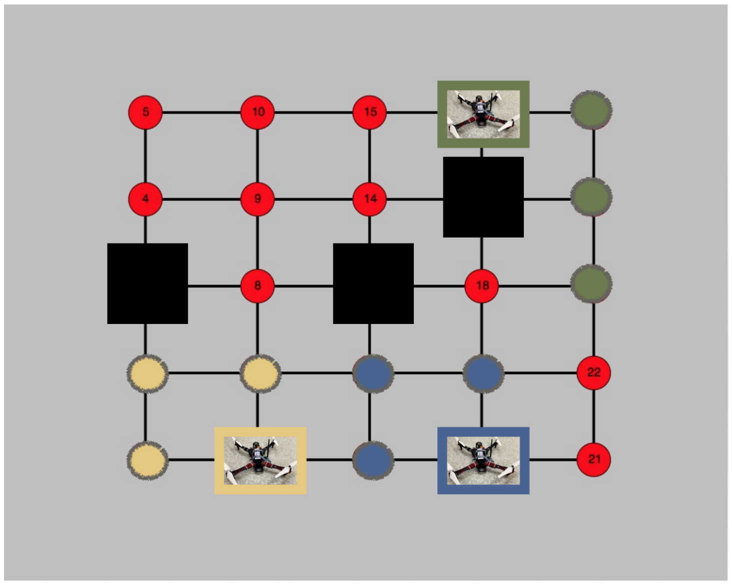

Figure 3.9: Step 4 of the example

5. The procedure LOOK updates a set KM with new nodes of rivals (6 and 20). The procedure COMPUTE for a blue agent gives only one possible next node 21, which is done in the procedure MOVE (see in Figure 3.10);

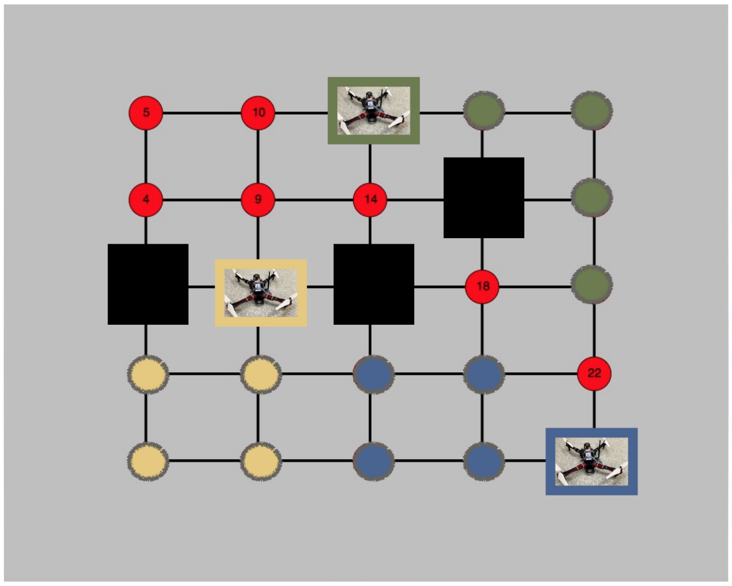

Figure 3.10: Step 5 of the example

6. The procedure Look updates a set KM with new nodes of rivals (8 and 15), as well as an obstacle in 14. The procedure Compute for a blue agent gives one possible next node 22, which is done in the procedure Move (see in Figure 3.11);

7. The procedure Look updates a set KM with new nodes of rivals (9 and 10). The procedure Compute for a blue agent gives the best further node for a travel and a job number 18. Node number 23 is found to be the next node in a shortest path to node 18. In the procedure Move only a travel to node 23 is executed, as node 23 was done initially by a green agent (see in Figure 3.12);

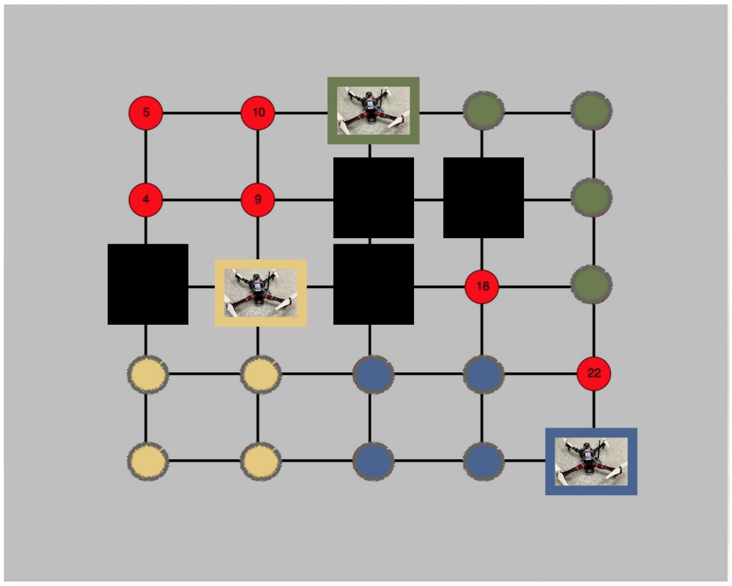

Figure 3.11: Step 6 of the example

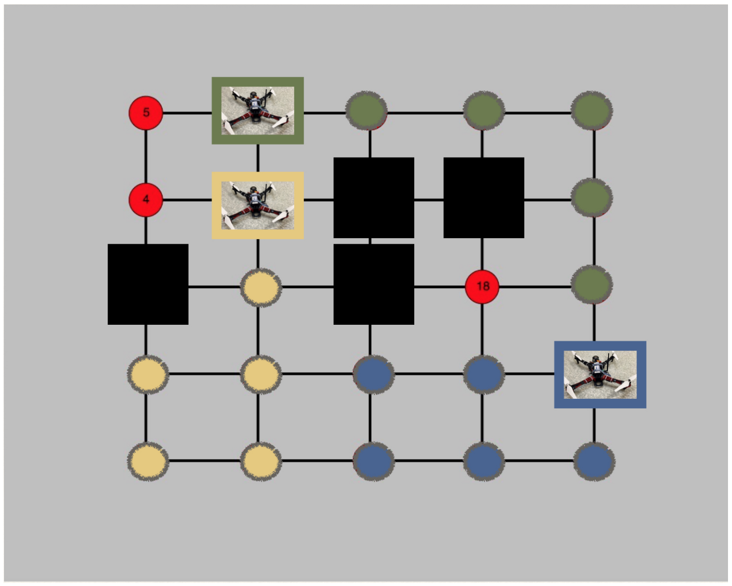

Figure 3.12: Step 7 of the example

8. In the procedure Look, there is nothing to update as the rival agents still

do their jobs. The procedure COMPUTE for a blue agent gives the best node number 18, which becomes to be an adjacent in this step of an execution. In the procedure MOVE node 18 is done (see in Figure 3.13);

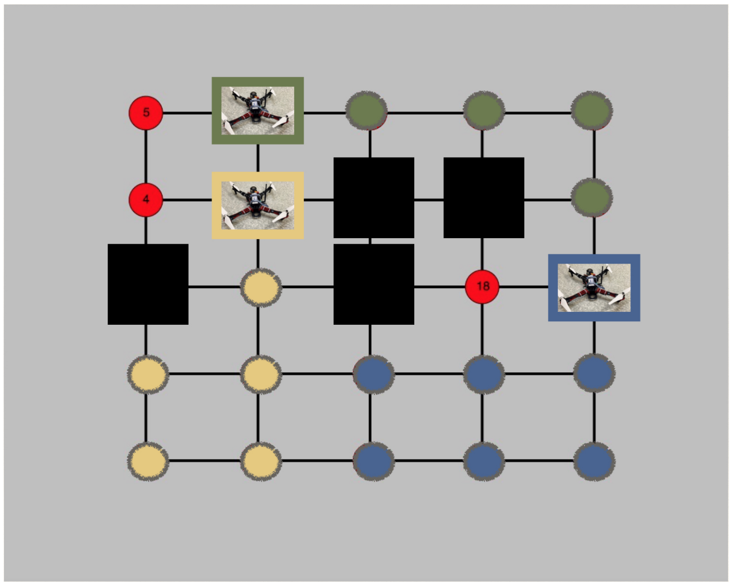

Figure 3.13: Step 8 of the example

9. In the procedure LOOK, the rival agents come to the last vacant nodes 4 and 5. These nodes are done by rivals, which leads to a termination condition, as a set KM becomes equal to V. All nodes were done by agents or found to be obstacles (see in Figure 3.14).

   The final paths of agents in the experiment are:

   - a yellow agent: 1 - 1* - 2 - 2* - 7 - 7* - 6 - 6* - 7 - 8 - 8* - 9 - 9* - 4 - 4* (termination condition);
   - a blue agent: 11 - 11* - 12 - 12* - 17 - 17* - 16 - 16* - 21 - 21* - 22 - 22* - 23 - 18 - 18* (termination condition);

- a green agent: 23 - 23* - 24 - 24* - 25 - 25* - 20 - 20* - 15 - 15* - 10 - 10* - 5 - 5* (termination condition);

As we have already mentioned, if a node was traveled by an agent, it appears in its path. If a job was done in a node, a node number with a star sign appears in a path.

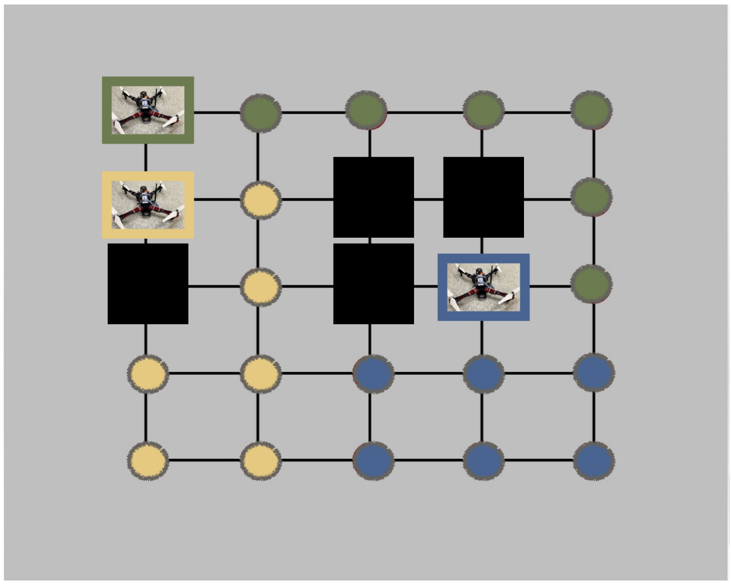

Figure 3.14: Step 9 of the example

Figure 3.15 shows agents returned to their initial positions after all nodes were done.

Clearly, this result is suboptimal, because the agents performed some extra travel. A yellow agent has an extra travel at node 7, a blue agent - at node 23. Notice, that the result is close to optimal, because there was only a small number of additional path segments. Moreover, the chosen flight paths did not result in any potential collisions.

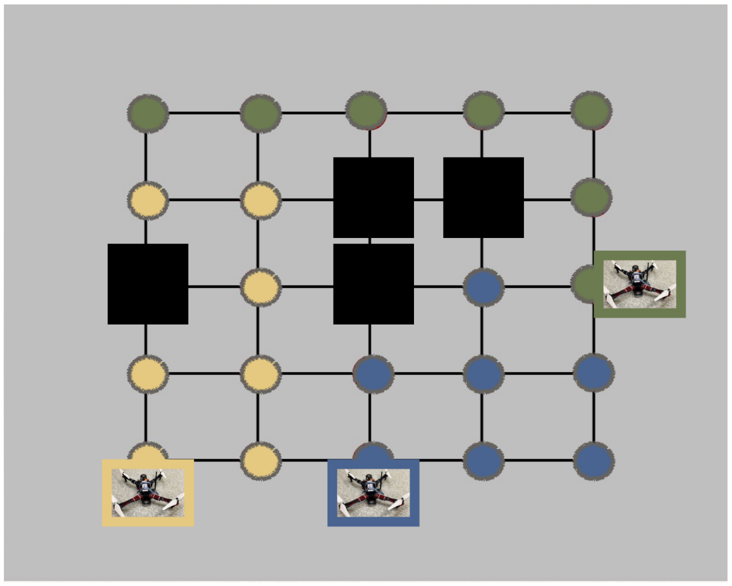

Figure 3.15: Final graph for the example of execution of the algorithm

Chapter 4

Implementation of the Algorithm

4.1 Communication Model of the System

As it was mentioned earlier, there is no inter-agent communication in our algorithm. That is, agents do not attempt to agree on what to do, for instance, who goes to the next node, who does a job in that node. We claim that it is possible to obtain suboptimal results even without such an agreement between agents, using just a global snapshot, or in our case, using a global synchronous broadcast. In this thesis, we assume that a broadcast is perfect, i.e. all messages are delivered to all clients in a reasonable time and properly, as well as the broadcast range can cover an entire graph. In our future work, we want to relax these conditions to the limited range imperfect broadcast.

In our system implementation in Python, a communication model is based on a *TCP/IP* protocol. We have implemented a *TCP* server (in a laptop) and *TCP* clients for each agent. We use multicopter *UAVs* or simulated agents to increase a number of agents in real experiments. A communication model is centralized (see in Figure

4.16). Every broadcast goes through a server, which in our implementation takes a role of an access point for agents and acts as a monitor for a researcher or an engineer. It transmits all received messages to all agents. It does not give any commands of what to do to agents, besides that are given in the initial stage of computations.

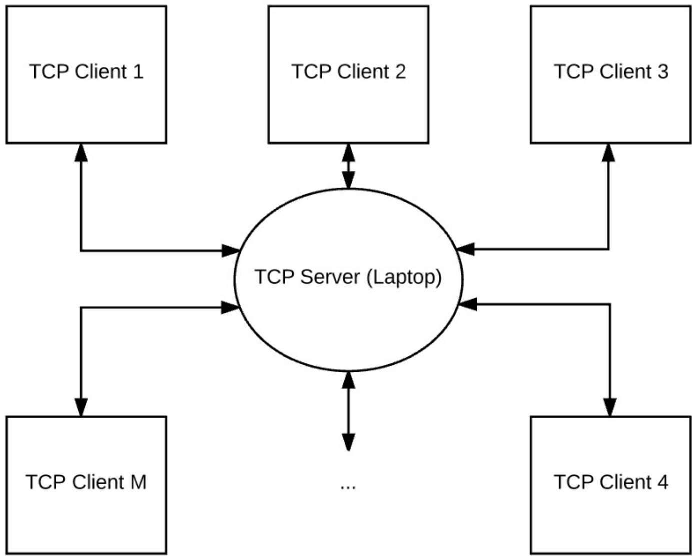

Figure 4.16: Communication model of the system

A structure of a unified broadcast message is shown in Figure 4.17. Each agent should include in a message its up-to-date information.

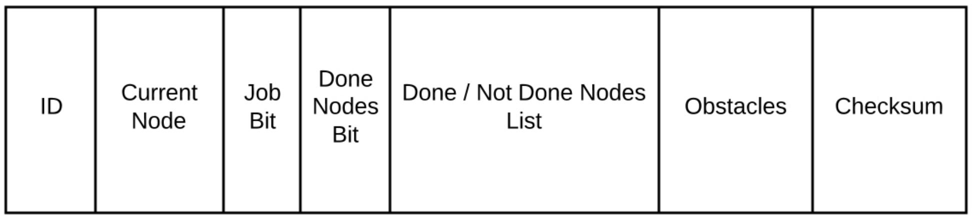

Figure 4.17: Structure of a broadcasted message

Every message contains an *ID* number of a sender.

The second part is a current location of an agent. Depending on an implementation, it can be just a number of a current node, or a tuple, containing agent's latitude, longitude, and altitude.

A 'job bit' takes a value 1 when an agent does a job in a current node, and a value 0 when an agent only travels through a node.

The fourth part contains a list of nodes, that were done by an agent or by its rivals, according to its knowledge (a set KM from the *MADGTA*). To reduce a size of this list we introduce a 'done nodes bit' in every message right before a list. This bit takes a value 1 if a number of done nodes is less than a half of a total amount of nodes in a graph. The bit takes a value 0 when a number of done nodes is more or equal to a half of a total. When this bit changes its value to 0, a list of done nodes transforms to a list of not done nodes. That is, an agent from this point in time uses numbers of all remaining nodes in its broadcast. Therefore, such an approach helps to reduce a message complexity at least twice in comparison to a broadcast of done nodes only. To obtain even better message complexity, we encode and compress this part of a message, according to a specific encoding protocol.

The fifth part of a message is a list of obstacles (a set O from the *MADGTA*). This part can be eliminated if a total number of nodes in a graph is very big.

The final part of a broadcasted message is a checksum. An agent updates its set KM and uses a location of the rival agent in its computations only if a checksum of the incoming broadcast is correct. If it is not correct, a message is skipped.

In a case of an unlimited range broadcast with a perfect quality, i.e. with a guaranteed delivery, we can eliminate a done nodes list, as each agent can parse all necessary information just from previous parts of a message. However, if a broadcast is imperfect, or it has a limited range, a done nodes list is a crucial part for a quality of a solution with the algorithm.

4.2 Computation Model of the System

A computation model of the system consist of three consequent main parts (given in Figure 4.18):

- 'initialization', which is responsible for an initial setup of an agent, safety checks and a move of the agent at its initial position, i.e. at its target altitude in a nest node;
- 'mission', which is the most important computation part, where the *MADGTA* makes it possible to do all travels and jobs in suboptimal time;
- 'termination', which returns an agent at a starting home position, and sends all necessary information and statistics to a server.

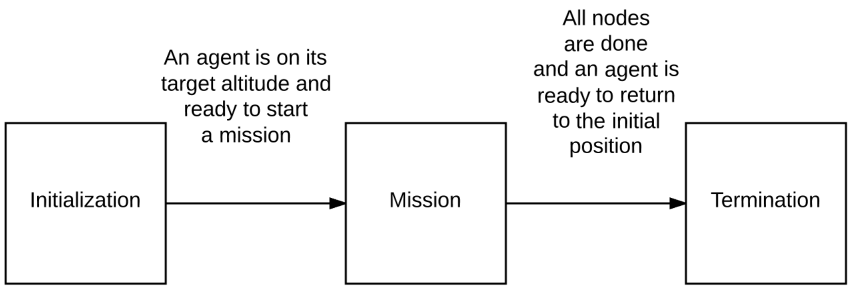
Initialization
An agent is on its target altitude and ready to start a mission
Mission
All nodes are done and an agent is ready to return to the initial position
Termination

Figure 4.18: Computation model of the system

First, we describe an 'initialization' stage (see in Figure 4.19).

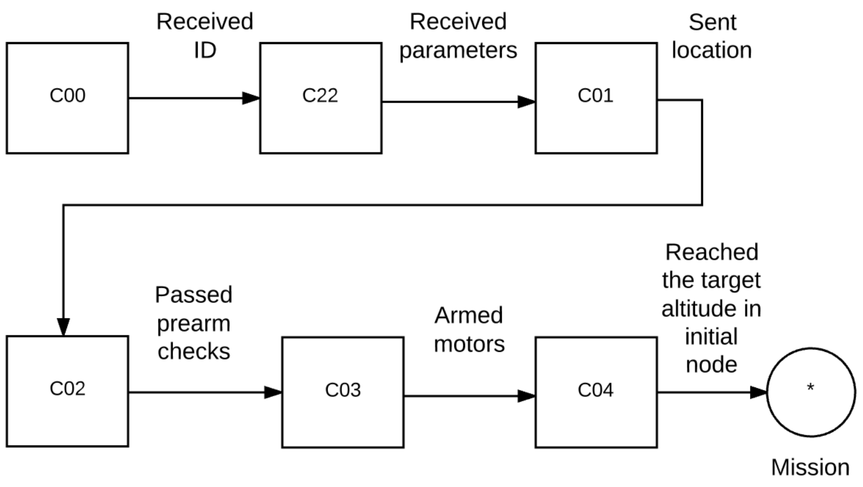

Figure 4.19: Initialization stage of the computation

As the most part of this stage goes on a ground in a close distance to a server (laptop), it can be made using a communication with acknowledgments. It is important to assure, that every message from a server comes correctly. We change parameters of a real *UAV* here, as well as, give a description of a mission. Wrong changes in parameters can lead to a crash of a *UAV*.

In Figures 4.19 – 4.21 each square represents some part of computations. Messages, that an agent sends currently for testing purposes, hold the same names, as commands, i.e. '*C01*', '*C02*', etc. These messages help us to determine a crucial information to broadcast. Also, it guarantees a perfect qualities of a current implementation of a broadcast.

Each client (virtual device from a simulation or a real *UAV*) starts its work from a command '*C00*' with an empty *ID* '*D00*'. The goal of this command is to inform a server that there is a new agent, ready to start its work. When a server receives a message '*D00 C00*' from a client, it chooses an *ID* for that client from a list of possible *IDs* and sends it back. It is also possible to assign concrete *IDs* to all real *UAVs*.

The next message from a client should contain a command '*C22*' with an *ID*, that it has received as a result of a previous message. If an *ID* is incorrect, a server stops a connection with such a client, as it possibly can be some intruder. If an *ID* is correct, a server in response on a message '*C22*', sends back to a client all required information, including a configuration of a graph, all *IDs*, job and travel speeds of each possible in a mission rival agent. A client receives a list of possible commands. The most important information for a client in the message is its personal parameters for a mission. It includes an initial node number (only in case of simulations); target altitude (in meters); travel speed (in cm/sec); job time (in sec, a pattern of a job is defined by the *MADGTA*); move up speed (in cm/sec); move down speed (in cm/sec).

In our test system there is a check if a client has sent a message with '*C22*' and received a response before it sends any other message. If it tries to send another message without '*C22*', it means that a client has not updated its parameters, which can be unsafe to people and the other agents. In such a case, a server raises an error for this client. If everything was good up to this stage, and all parameters were successfully updated, according to requirements of a server, a client sends a message '*C01*'. It contains a current location (node) of an agent. It tells other agents, that they have a new rival, and they get a location of it to use in the *MADGTA*. This way an agent enables a start of a mission, as currently, an agent should wait unless it has at least one rival before it can start any work. Thus, if there is only two agents in a mission, such an information can be crucial. A message '*C01*' is just an additional information, so a server does not need to send any acknowledgements. However, in a current implementation, for a convenience of testing of a system and a work of the algorithm, we use acknowledgments on any message.

The next message from a client is '*C02*'. This message is sent only if all safety pre-arm checks were passed successfully. If not passed, a *UAV* does not take a part in a mission, as it can be dangerous. Safety checks include tests of electronics (gyroscopes, barometer, accelerometers, etc.) and a quality of a *GPS* lock, i.e. a precision of positioning, using *GPS* satellites. If all safety checks are passed, in a small interval of time a *UAV* tries to arm its motors. If motors were successfully armed, a client sends a message '*C03*'.

The final part of the initialization stage forces a *UAV* to move to a starting position, i.e. it assures that a *UAV* has done a take off to a target altitude and a travel to the closest node of a graph. These operations can take some time, depending on a move up speed and a target altitude, as well as a distance to a real point of an environment. When an agent reaches a necessary position and altitude, it sends a message '*C04*' and goes to a mission stage.

The mission stage (see in Figure 4.20) starts with a command '*C05*' and the same name message, containing the next node of an agent. Our algorithm makes its computations in this stage, and a result of it goes to a message.

The next message is '*C06*'. An agent sends it on a half way to the next node. An agent just informs everyone that it enters a radius of a node. Having such an information, rival agents can operate in neighboring areas at higher speeds, without a risk of collisions.

When an agent finishes its travel to an exact position of the next node, it sends a message '*C07*' to inform about it, and that it initiates a job in that node. Agents should have this information to avoid double jobs in the same nodes. A node which is started by any agent is marked as 'conditionally accomplished'. If some serious error happens during a job, and an agent does not finish that job, a status of a node comes back to 'not done'. If a job is done successfully, an agent informs about it with a message '*C08*'. If an agent gets a termination condition at this time, it goes to a termination stage of computations. If there are any not done nodes in a graph, an agent repeats a cycle. It goes to a computation of the next node with the *MADGTA*, sends a message '*C05*', etc.

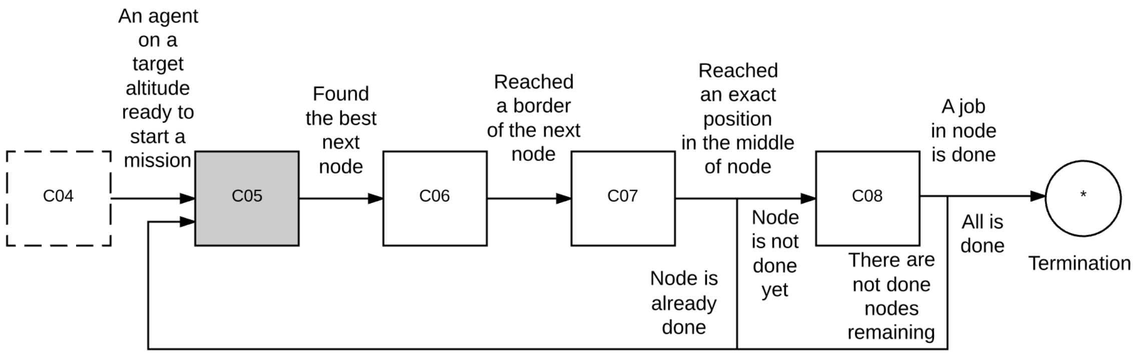
An agent on a target altitude ready to start a mission
C04
C05
Found the best next node
C06
Reached a border of the next node
C07
Reached an exact position in the middle of node
C08
A job in node is done
*
Termination
Node is not done yet
Node is already done
There are not done nodes remaining
All is done

Figure 4.20: Mission stage of the computation

Transfer to a termination stage of a computation can happen in any point of time, as soon as an agent has received an information, that all nodes of a graph are accomplished or found as obstacles. Therefore, it can be before or after messages '*C05*', '*C06*', '*C07*' or '*C08*'. It is not shown in Figure 4.20 to keep it clear.

The termination stage of a computation, shown in Figure 4.21, can be started in two cases. In the first case, a correct termination condition is achieved, i.e. all nodes were done or distinguished as obstacles. The second case is when some serious error occurs with an agent during a mission stage. It can be a low battery level or any serious hardware problem.

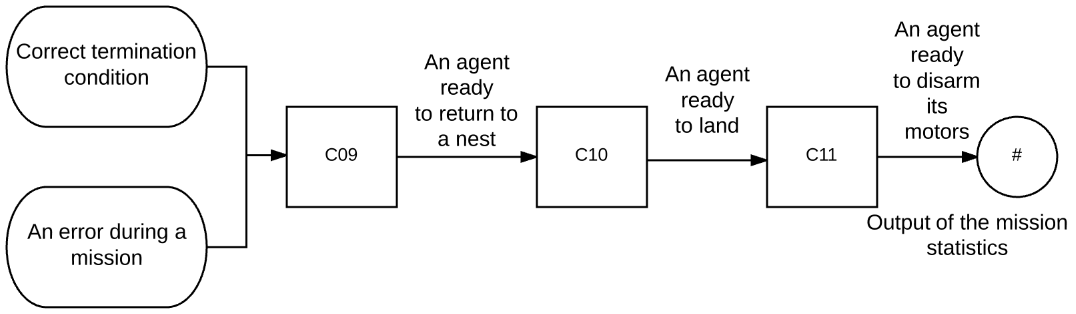
Correct termination condition
An error during a mission
C09
An agent ready to return to a nest
C10
An agent ready to land
C11
An agent ready to disarm its motors
#
Output of the mission statistics

Figure 4.21: Termination stage of the computation

The termination stage starts with the finding a way to a nest, or some other position, that should be a final destination of an agent. In an ordinary case, it is the same position as a starting point. An agent sends a message ‘*C09*’, initiates a command ‘return to the launch’ (*RTL*), and goes towards that node. When an agent has reached an exact position in a final destination node, and it is ready to land, it informs about it with a message ‘*C10*’. After a message, an agent lands to a ground. A *UAV* disarms its motors and informs about it with a message ‘*C11*’. The final part of a termination stage is a transfer of statistics and information, that an agent gets during a mission.

4.3 Implementation of the Algorithm in Python

An implementation of the *MADGTA* was made, using Python language. A choice of this programming language was made, mostly because of real *UAVs*, that we use in our experiments. Our *UAVs* - multicopters (octocopters, hexacopters and quadrocopters) are controlled by a flight controller Pixhawk, which has an *API* 'DroneKit-Python-*API*'. Therefore, it is more convenient to use the same programming language, as the *API* uses.

The programming implementation of the *MADGTA* and its environment consists of three Python files, of more than 6000 lines of code overall:

- 'server.py' - a file, that is executed in a laptop. This file creates a *TCP* server, which provides agents with an initial information and all requirements for a mission, as well as, gives a response to every client's command. A server does not control agents. It only works in a monitor mode, which is very convenient for a testing purposes. In the future work, we want to transform an implementation of a server, to be able to work in an ad-hoc mode, without usage of an access point;

- 'client.py' - the main file of our implementation, which creates a *TCP* client, and holds all logic of the *MADGTA* and its environment. We run instances of this file in every agent's onboard computer in a real experiment mode, or we can create multiple virtual agents in a simulation mode in a computer. As every agent makes its personal decisions, using a logic from this file, we obtain a distributed autonomous system of *UAVs*;

- 'simulations.py' - the file that gives us an opportunity to run a required number of simulation experiments with the different number of agents, specific configurations of a graph and different parameters.

We describe some major modules and functions of our implementation of the *MADGTA*.

The first file to describe is a '**server.py**'. We create a socket, bind hosts and port numbers and start listening, having in mind all possible exceptions. When a server receives a connection request from a new client, it opens a personal thread for this client. After that, a server implements all logic of a computation model, described by us earlier.

We do have multiple options for a giving of an input. We have implemented several functions to get an input from a file, from arguments, entered through a command line, and an input through a python file itself. We have implemented a function '*writegraphToArray*(*radius*, *pointsOnEdge*, *centerLat*, *centerLon*, **obstacles*)' which creates a grid just with a given central point's *GPS* coordinates, i.e. latitude and longitude of a point, with a required radius of work from this point, number of nodes in an edge of a grid and a number of nodes, containing obstacles. For instance, a command '*python server.py* –*portNum* 1000 –*radius* 50 –*nodesOnEdge* 9 –*lat* 43.268304 –*lon* –79.911135 –*obstacles* 2 10 23 35 46 48 59 64 76' gives a grid of size 100 by 100 meters, with 81 nodes in total, with a central point (not a node) in a given latitude and longitude, with obstacles in nodes 2, 10, 23. As a result, we have a graph table in a format '[1, 43.267849, -79.911591, 0] [2, 0, 0, 0] [3, 43.268051, -79.911591, 0][4, 43.268152, -79.911591, 0] [5, 43.268253, -79.911591, 0] $\cdots$'. Each of nodes has a precise latitude and longitude. For example, a string '1, 43.267849, -79.911591, 0' tells us, that node number 1 has such a latitude and longitude, and it is not done yet. If a node was done by an agent, a server changes the last '0' value to an *ID* number that agent. The second node was defined as an obstacle. In our graph table, it is shown as '2, 0, 0, 0'. Therefore, an implemented function creates any configurations of a grid with almost the same distances between adjacent nodes and with real coordinates for field experiments with *UAVs*. A server holds a list of all commands, that can be used in the mission by clients, a list of vacant *IDs* for clients, and all personal parameters for each agent, according to its *ID*. For instance, it can look as '*D01*','2', '5', '300', '10', '150', '30', '*D02*','12', '15', '200', '12', '250', '50', '*D03*', $\cdots$, which shows, that an agent with an *ID* '*D01*' has a nest in node number 2, a target altitude equal to 5 meters, a travel speed 300 cm/sec, it needs to spend 10 sec to do a job in each node, a speed of moving up is 150 cm/sec and moving down 30 cm/sec. An agent with an *ID* '*D02*' has a different starting node 12. It works at a higher altitude of 15 meters. The second agent has a slower travel speed of 200 cm/sec, and worse abilities to do a job, as it needs to spend 12 sec per node. For instance, in a search problem, it can be, because of a worse quality of a computer vision and sensors. However, an agent moves faster up and down with the speeds of 250 cm/sec and 50 cm/sec respectively. A list of personal parameters continues in the same way.

As it was already mentioned, a server in our implementation works as a monitor, so we can see a progress of an experiment in all modes: simulations, real experiment or hybrid. A server also combines all statistics from clients and outputs it to files. We also have implemented several functions to achieve a maximum robustness of the system. For instance, if an agent stops its work preliminary, because of some error, including a low battery level, its *ID* is entered back to a list of vacant *IDs*. So if we do have some spare agent, we can arm it in a process of a mission and continue the work of that broken agent. A list of *IDs* is also used by a server, to determine when a test mission is over, i.e. all agents came back to their initial positions, have successfully landed and disarmed their motors, transmitted all statistics of the mission. An *ID* of each such an agent returns back to a list. So if the list is the same, as it was initially, a mission is done and a server can output all combined statistics.

The most important file of our implementation is a '**client.py**'.

First of all, this file works both at real *UAVs*, in field experiments, and at virtual *UAVs*, in a simulations mode. The type of a device for a current experiment can be switched very simple, just with a change of a value of one variable. The file has specific implementations for both cases.

The program creates a *TCP* client. Therefore, it creates a socket, tries to connect to a server, and starts its work after a successful connection. If a client was not able to connect properly to a server, the program prints out a code of an error and continues to try to connect within a given time interval.

When a connection with a server (through an access point) is established, a client starts its work, according to a computation model, that we have described earlier. An agent first tries to get its *ID* for a current mission. It sends a message, containing a command '*C00*' and receives an *ID* from a list of vacant *IDs*. After that a client requests from a server its personal parameters and all initial information about a mission.

Here comes the first difference between a virtual agent and a real *UAV*. For a virtual agent, it is enough just to assign its parameters values to some variables. In a case of a real *UAV*, we need to flash every single parameter and check that it was changed correctly.

After that, a real *UAV* should go through a stage of a safety pre-arm checks. It can take quite a big amount of time and it strongly depends on an environment. For instance, if a sky is very cloudy, there is a big risk that a *GPS* lock is not enough to make a real experiment. For instance, a *GPS* module finds just 5–6 satellites, which gives a bad precision of the *GPS* positioning with several meters radius. A *GPS* lock also depends on many other factors. Having in mind real *UAVs*, our virtual agents goes through the same stages. The only thing, that with virtual agents, we assume that every such a stage goes successfully every time within a given time interval. However, with real *UAVs* it can be different in every experiment, depending on the result of a special function '*uavPreARMcheck*()'.

An agent also needs to check, that it has at least one rival, according to our assumption. When an agent finds a rival or rivals and their location, it arms its motors with a function '*uavARM*()', goes to a target altitude in a takeoff mode, using a function '*uavTakeOff*(*uavTakeOffAltitude*)', and starts its mission.

A *UAV* does its travel in two stages, using the next functions. First function '*uavGoToCell*(*pointLat*, *pointLon*, *pointAlt*, *pointNum*)' just leads a *UAV* to a given radius of a node. Therefore, it can be taken, as entering borders of a cell. After a *UAV* has reached such a radius, it continues its movement toward a precise position in a middle of a cell, i.e. to a given nodes coordinates, using a different function '*uavGoToPoint*(*pointLat*, *pointLon*, *pointAlt*)'.

A *UAV* does a job using a function '*uavDoJob*()'. For example, in a case of a coverage, we can give as parameters a radius of a circle in centimeters and a number of circles to do. A *UAV* makes a given number of circles of such a radius around a central point of a node. A pattern of a job can be made differently, according to a problem we solve. However, as it is not a main purpose of our research, in real-world experiments we use just an altitude change movements.

We have implemented dozens of useful functions to make a parsing of incoming messages, to form our messages, to make a rounding, to get a relation between a node number and its coordinates, to get adjacent nodes to a current, to evaluate a time necessary to travel to a point, to find a shortest path between nodes of a graph, to write an information in a proper data structures, to count a time of each step, to write an information to a log file, etc.

We have implemented commands, according to a given before computation model. Every such command contains its different logic. The most important for us is a command '*C05*' where our algorithm is used. The *MADGTA* is written as the combination of functions '$doAlgorithm()$' and '$GetBestNode(W)$'. It implements a logic of our algorithm and a mentioned function.

In a final termination stage a UAV consequently uses the next functions '$uavRTL()$', '$uavLAND()$' and '$uavDISARM()$'. After a UAV has returned to an initial point, it sends to a server all statistics of its work during a mission, including a path, that it has traveled, nodes in which a job was done by an agent, the total travel, job and additional time, etc.

The last file of our implementation is the **'simulations.py'**.

We have implemented different options of an input for simulations. One of such options is to use command line arguments. For example, the next line '*python simulations.py –radius* 100 *–nodesOnEdge* 10 *–portNum* 9087 *–agentsNum* 4 *–obstacles* 10 23 35 46 48 59 64 76' gives us as a result an experiment in a simulation mode with four agents with parameters from a file 'server.py', in a grid of 200 by 200 meters with

100 nodes and given obstacles. The nest for each agent is also defined in a 'server.py'. Simulations are made, using a 'subprocess' library. First, we start a server, to which we consequently connect new clients in some time interval, like 0.5 sec from each other. We can assign a number of experiments to do. A result of all experiments is written into the log files. After that, we can use an additional program, written by us to get statistics in a format that is given in tables in the next chapter of this thesis.

Chapter 5

Simulations and Experiments on UAVs

In this chapter, we present our results of simulations and experiments in Sections 5.1 and 5.2.

5.1 Simulation Results

5.1.1 Parameters and Metrics of Simulations

Parameters that have been changed in simulations:

- a number of nests (from 1 nest to m nests, where m is a total number of agents in a current experiment) and their location in a graph;
- a number of obstacles;
- a type of agents, i.e agents with the same, slightly different, or very different

costs.

Notice, that costs of a job C_j in given simulations are the same for all agents.

We evaluate the following **metrics** for each agent i in every experiment:

- $numNodesDone_i$, a total number of nodes where a job was done by an agent i;
- $numNodesTraveled_i$, a total number of nodes traveled by an agent i;
- $numNodesWoJ_i = numNodesTraveled_i - numNodesDone_i$;
- $travelTime_i$ a total time spent on travel by an agent i;
- $jobTime_i$, a total time spent on accomplishing of jobs by an agent i;
- $totalTime_i = travelTime_i + jobTime_i$;
- $additionalTime_i$, a time parameter equal to a summation of the takeoff, return to a launch (RTL), and land time spent by an agent i.

The following global metrics are additionally measured for all agents in each experiment:

- $totalTimeAll = \sum_{i \in [1,m]} (travelTime_i + jobTime_i)$, a total time spent on a travel and accomplishing of a job by all agents, where m is a number of agents in the current experiment;
- $numNodesWoJAll = \sum_{i \in [1,m]} (numNodesTraveled_i - numNodesDone_i)$, a number of nodes travelled without doing a job by all agents, where m is a number of agents in the current experiment.

5.1.2 Results of the Simulations

Impact of a Location of a Nest

The first series of a simulations was done with a 6×6 grid, without obstacles, with a total size of 40×40 meters. Each adjacent node is located at almost equal distance from each other (about 6.67 meters).

In the first experiment we use 3 identical agents with the following parameters:

- a travel time between every two nodes is around 2.2 sec (a travel speed = 300 cm/sec);
- a job time is 10 sec per node for all agents;
- parameters, that are used in evaluation of an additional time: a target altitude = 5 meters; a move up speed = 150 cm/sec; a move down speed = 30 cm/sec.

Each agent starts its work in a mission consequently with a time interval equal to 0.5 sec. It goes through a stage of pre-arm safety checks, which we do not count in our time metrics as this time depends on a specific hardware and safety settings in each concrete case. Important time metrics that we need to count are

- a takeoff time, i.e. a time taken by an agent to reach a target altitude;
- a time of traveling back to a nest from a final agent's node of a graph when a termination condition was achieved (*RTL*);
- a time of landing, according to a move down speed and some other parameters of a flight controller.

All these time values are included in a metric $additionalTime_i$.

The first simulation is done with the aforementioned parameters of agents and a graph settings. Only a location of a nest was changed. The experimental results are given below in Tables 5.1 and 5.2.

In Figures 5.1 and 5.2 the given graphs are based on Tables 5.1 and 5.2, respectively. Notice, that for all remaining simulations only their resulting graphs are shown. The graphs in Figures 5.1 and 5.2 demonstrate a relation between a location of a single nest for 3 agents, and an accumulated time ($totalTimeAll$). Figure 5.1 shows such a relation when a nest is moved at the border of a grid in nodes from 1 to 3. Each bar of a graph shows four values. Two whiskers give a minimum and a maximum values of the $totalTimeAll$ among all experiments. In most cases, there are done 100 or more experiments for each metric, which gives us a 95% confidence interval. A body of a candlestick shows a confidence interval. Notice, that in the remaining graphs with candlesticks they show the same values of a maximum, minimum, and a confidence interval. Figure 5.2 shows a similar relation between a location of a nest in the middle of a grid in nodes in a physical diagonal, at a distance from 1 to 3 from a corner node number 1.

The results in Tables 5.1 and 5.2 show:

- There is a relation between a location of a nest and a $totalTimeAll$ on such a small grid, as 6×6 nodes. If a nest is located in one of the corners of a given grid, we see the best confidence interval of $[450.45, 453.07]$ seconds of an accumulated time with a probability 95%.

  The next ‘good’ stable location for a nest is on the border of a grid, closer to a corner, with a confidence interval of $[452.75, 455.4]$ seconds.

Parameter	A nest in node 1, 6, 31 or 36					
	Min	Max	Median	Mean	STD	CI
$totalTimeAll$	439.89	475.32	451.08	451.76	6.94	[450.45,453.07]
$numNodesWoJAll$	1.0	17.0	6.0	6.33	3.12	[5.73,6.92]
$totalTime_i$	85.55	209.97	151.15	150.59	20.25	[146.76,154.42]
$travelTime_i$	15.55	44.37	31.02	30.59	4.95	[29.66,31.52]
$jobTime_i$	70.0	170.0	120.0	120.0	16.88	[116.81,123.19]
$additionalTime_i$	24.47	42.22	35.58	34.74	4.33	[33.93,35.56]
$numNodesDone_i$	7.0	17.0	12.0	12.0	1.69	[11.68,12.32]
$numNodesTraveled_i$	7.0	21.0	14.0	14.11	2.34	[13.67,14.55]
$numNodesWoJ_i$	0.0	9.0	2.0	2.11	1.77	[1.77,2.45]
Parameter	A nest in node 2, 7, 5, 12, 25, 32, 30 or 35					
	Min	Max	Median	Mean	STD	CI
$totalTimeAll$	439.86	473.3	453.285	454.07	7.0	[452.75,455.4]
$numNodesWoJAll$	1.0	16.0	7.0	7.36	3.15	[6.77,7.96]
$totalTime_i$	77.75	195.49	151.09	151.36	18.87	[147.79,154.93]
$travelTime_i$	17.75	51.11	31.09	31.36	5.14	[30.39,32.33]
$jobTime_i$	60.0	160.0	120.0	120.0	15.86	[117.0,123.0]
$additionalTime_i$	22.2	39.98	33.35	31.98	4.41	[31.15,32.81]
$numNodesDone_i$	6.0	16.0	12.0	12.0	1.59	[11.7,12.3]
$numNodesTraveled_i$	8.0	24.0	14.0	14.45	2.44	[14.0,14.91]
$numNodesWoJ_i$	0.0	13.0	2.0	2.45	2.1	[2.06,2.85]
Parameter	A nest in node 3, 4, 13, 18, 19, 24, 33 or 34					
	Min	Max	Median	Mean	STD	CI
$totalTimeAll$	437.66	495.44	454.4	455.22	8.54	[453.61,456.83]
$numNodesWoJAll$	0.0	26.0	7.5	7.88	3.84	[7.15,8.62]
$totalTime_i$	57.78	214.4	151.09	151.74	18.26	[148.29,155.19]
$travelTime_i$	15.58	51.04	31.09	31.74	5.83	[30.63,32.85]
$jobTime_i$	40.0	170.0	120.0	120.0	14.91	[117.18,122.82]
$additionalTime_i$	22.2	37.78	31.15	30.4	4.21	[29.61,31.2]
$numNodesDone_i$	4.0	17.0	12.0	12.0	1.49	[11.72,12.28]
$numNodesTraveled_i$	7.0	23.0	14.0	14.63	2.82	[14.09,15.16]
$numNodesWoJ_i$	0.0	12.0	2.0	2.63	2.53	[2.15,3.1]

Table 5.1: Simulation results in a grid of 6×6 nodes, 40×40 meters, with no obstacles, 1 nest, 3 same agents, part 1

Parameter	A nest in node 8, 11, 26 or 29					
	Min	Max	Median	Mean	STD	CI
totalTimeAll	444.34	491.11	457.75	459.32	9.19	[457.58,461.07]
numNodesWoJAll	3.0	24.0	9.0	9.7	4.14	[8.93,10.47]
$totalTime_i$	87.78	208.84	155.585	153.11	21.01	[149.14,157.08]
$travelTime_i$	17.75	51.11	33.29	33.11	6.21	[31.94,34.28]
$jobTime_i$	70.0	160.0	120.0	120.0	17.14	[116.77,123.23]
$additionalTime_i$	20.0	37.75	28.87	29.33	4.3	[28.52,30.14]
$numNodesDone_i$	7.0	16.0	12.0	12.0	1.71	[11.68,12.32]
$numNodesTraveled_i$	8.0	24.0	15.0	15.23	2.95	[14.68,15.79]
$numNodesWoJ_i$	0.0	12.0	3.0	3.23	2.46	[2.78,3.69]
Parameter	A nest in node 9, 10, 14, 17, 20, 23, 27 or 28					
	Min	Max	Median	Mean	STD	CI
totalTimeAll	440.01	486.49	457.75	459.83	9.97	[457.94,461.71]
numNodesWoJAll	1.0	22.0	9.0	9.94	4.49	[9.08,10.79]
$totalTime_i$	72.26	220.04	154.985	153.28	21.39	[149.23,157.32]
$travelTime_i$	15.55	53.2	33.29	33.28	6.49	[32.05,34.51]
$jobTime_i$	50.0	180.0	120.0	120.0	17.07	[116.77,123.23]
$additionalTime_i$	20.0	35.55	28.87	28.23	3.95	[27.48,28.99]
$numNodesDone_i$	5.0	18.0	12.0	12.0	1.71	[11.68,12.32]
$numNodesTraveled_i$	8.0	24.0	15.0	15.31	3.03	[14.74,15.89]
$numNodesWoJ_i$	0.0	12.0	3.0	3.31	2.52	[2.84,3.79]
Parameter	A nest in node 15, 16, 21 or 22					
	Min	Max	Median	Mean	STD	CI
totalTimeAll	444.41	488.84	460.045	462.02	9.04	[460.31,463.73]
numNodesWoJAll	3.0	23.0	10.0	10.92	4.07	[10.14,11.69]
$totalTime_i$	92.26	207.77	155.53	154.01	20.73	[150.08,157.94]
$travelTime_i$	20.02	51.15	33.33	34.01	6.3	[32.82,35.2]
$jobTime_i$	70.0	170.0	120.0	120.0	16.77	[116.83,123.17]
$additionalTime_i$	20.0	33.35	26.67	26.92	2.9	[26.37,27.48]
$numNodesDone_i$	7.0	17.0	12.0	12.0	1.68	[11.68,12.32]
$numNodesTraveled_i$	9.0	24.0	15.0	15.64	3.01	[15.06,16.21]
$numNodesWoJ_i$	0.0	11.0	3.0	3.64	2.55	[3.16,4.12]

Table 5.2: Simulation results in a grid of 6×6 nodes, 40×40 meters, with no obstacles, 1 nest, 3 same agents, part 2

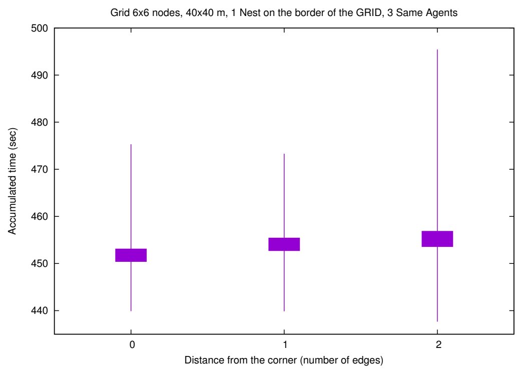

Figure 5.1: Simulation results in a grid of 6 × 6 nodes, 40 × 40 meters, with no obstacles, 3 same agents, 1 nest on the border of the grid

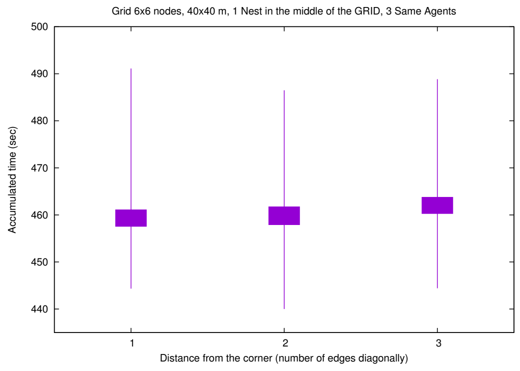

Figure 5.2: Simulation results in a grid of 6 × 6 nodes, 40 × 40 meters, with no obstacles, 3 same agents, 1 nest in the middle of the grid

The position of a nest on the border, somewhere in the middle of the grid (lower Table 5.1), also gives pretty good results of the $[453.61, 456.83]$ seconds.

As a location of a nest is moved out from a border, closer to an exact central point of a grid (Table 5.2), we get worse results of a confidence interval and a bigger standard deviation. Such a relation can appear, because of a 'bigger freedom', that agents achieve, when a nest is somewhere in the middle. That is because there is a lack of a space. Agents have bigger options for beginning of a mission. It means, that they have a bigger amount of possible starting points, which can make a difference in a direction of their job and travel. As agents start their work almost synchronously, they are not able to distinguish a first movement of rivals. So, they can choose the same starting direction, which leads to a bigger amount of useless travels (collisions). A situation can become even worse, if they choose a direction to a small subarea of a graph, closer to the borders, and they will continue to work in such an area for some time;

- As can be seen in Tables 5.1 and 5.2, individual agent's results are also better, when a nest is located in corners and in a border of a grid, rather than somewhere in the middle. Agents have the least number of traveled nodes, with a confidence interval from $[13.67, 14.55]$, while a middle position of a nest gives $[15.06, 16.21]$. So, each agent in a 'middle' nest has about 2 extra travels. As a number of nodes done by different agents is, more or less, the same in all these cases, an additional useless travel shows up in a metric *numNodesWoJAll*. Again, when agents start from a corner or a border, they have cumulatively from 1 to 17 nodes without a job for all of them. A 'middle' nest gives from 3 to 23 nodes without a job.

Therefore, when we need to get more stable results in our mission, it is better to position the only nest closer to the borders and the corners of a small grid, as 6×6 nodes. Meanwhile, a location of a nest even in the middle of such a grid does not mean, that we are not able to receive any good results. Actually, according to our simulations, it is possible to get even an optimal result in 2-5% of experiments.

The second part of our experiments is done with a bigger grid of 10×10 nodes, but, of the same physical size 40×40 meters. All parameters of a *UAV* remains the same, i.e. a travel speed = 300 cm/sec; a job time = 10 sec per node; a target altitude = 5 meters; a move up speed = 150 cm/sec; a move down speed = 30 cm/sec. However, this time, we use from a minimum of 2 agents to a maximum of 10 agents in the same experiment.

All simulations are done in the same manner, as given before. However, this time, we omit all tables, and just show their resulting graphs on a base of an information in a similar format. Two graphs 5.3 and 5.4 demonstrate a result of a relocation of a single nest for 5 agents at a border and in a middle part of a grid respectively.

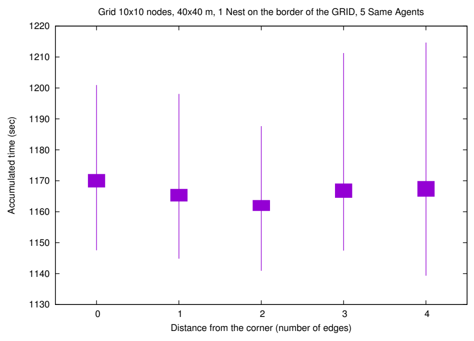

Figure 5.3: Simulation results in a grid of 10 × 10 nodes, 40 × 40 meters, with no obstacles, 5 same agents, 1 nest on the border of the grid

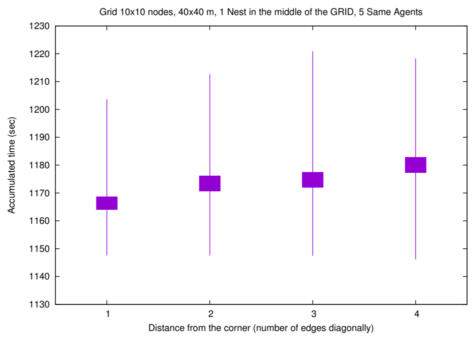

Figure 5.4: Simulation results in a grid of 10 × 10 nodes, 40 × 40 meters, with no obstacles, 5 same agents, 1 nest in the middle of the grid

Now a relation, between a position of a nest and a *totalTimeAll*, does not seem to be that clear. As it is shown in our next graphs, this correlation disappears in bigger grids.

The last part of experiments is done with a big grid of 20×20 nodes, of a physical size 80×80 meters.

The first two graphs (in Figures 5.5 and 5.6) demonstrate a relation between a location of a single nest for 10 agents and an accumulated time (*totalTimeAll*).

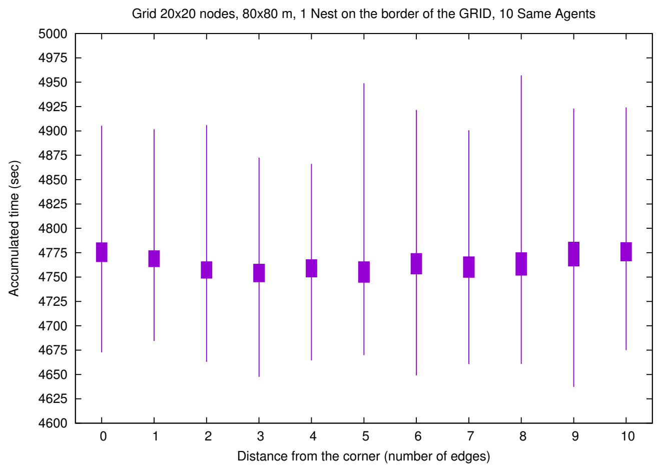

Figure 5.5: Simulation results in a grid of 20×20 nodes, 80×80 meters, with no obstacles, 1 nest in the given node number, 10 same agents

Figure 5.5 shows a relation between a location of a nest on a border of a grid in nodes from 1 to 10, i.e. to a middle part of the grid, as it is symmetric. Such a relation to a middle location of a nest is demonstrated in Figure 5.6.

It is clear, that in a big grid of 20×20 nodes, of a size 80×80 meters, there is no more correlation between a location of a single nest and a *totalTimeAll*. All values are close to each other – about 4750 seconds. However, we should mention that a number of agents in that experiment was raised to 10. Nevertheless, this leads us to a conclusion, that in a case of a big grid and a bigger number of agents, the *MADGTA* works stable, with the similar results, wherever we introduce a nest node. However, for a smaller grid and smaller number of agents, there can be a correlation, and it is recommended to position a nest closer to a border and a corner of a grid.

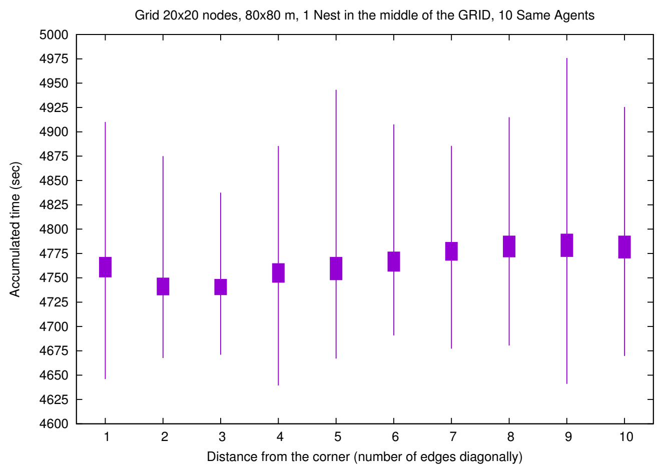

Figure 5.6: Simulation results in a grid of 20×20 nodes, 80×80 meters, with no obstacles, 1 nest in the node in the middle of the grid, 10 same agents

Impact of a Number of Active Agents

The next graph (Figure 5.7), shows a relation between a number of agents in an experiment and a resulting accumulated time in seconds. As we can see, the accumulated time goes up rapidly, as we raise the number of agents. It is, because of a lack of a common decision between agents and a raised number of collisions. It is much easier to avoid such collisions if there are just few agents.

We also have evaluated an accumulated time for a similar settings, but, for a bigger grid of 15×15 nodes of a size 40×40 meters (Figure 5.8), and 15×15 nodes of a twice bigger size 80×80 meters (Figure 5.9).

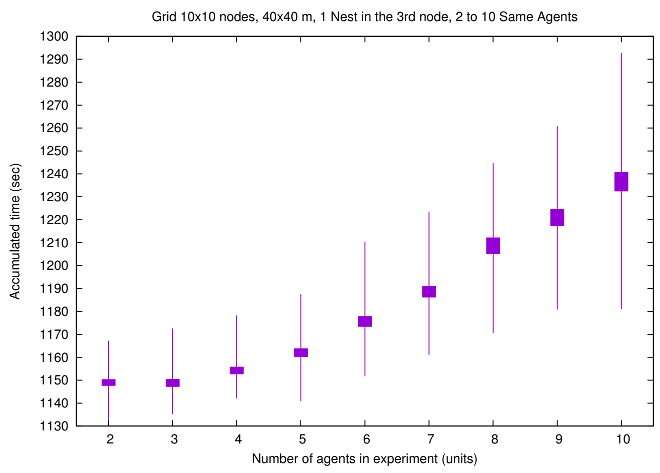

Figure 5.7: Simulation results in a grid of 10 × 10 nodes, 40 × 40 meters, with no obstacles, 1 nest, 2 to 10 same agents

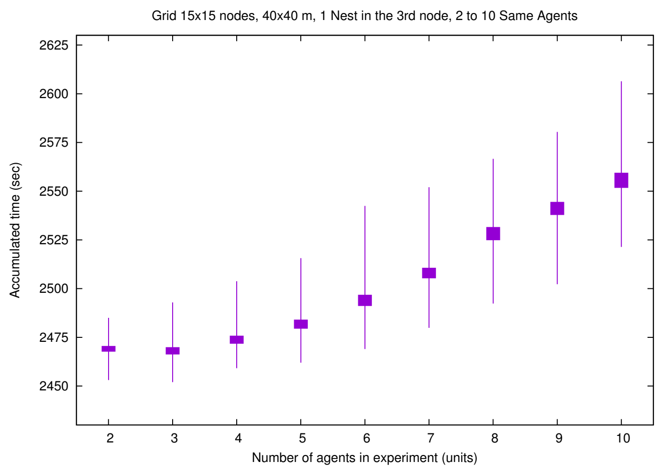

Figure 5.8: Simulation results in a grid of 15 × 15 nodes, 40 × 40 meters, with no obstacles, 1 nest, 2 to 10 same agents

From two graphs (Figures 5.8, 5.9) we can see the same relation between a number of agents and an accumulated time. It seems, that the best number of agents in terms of the accumulated time is less than 5 (Figure 5.9).

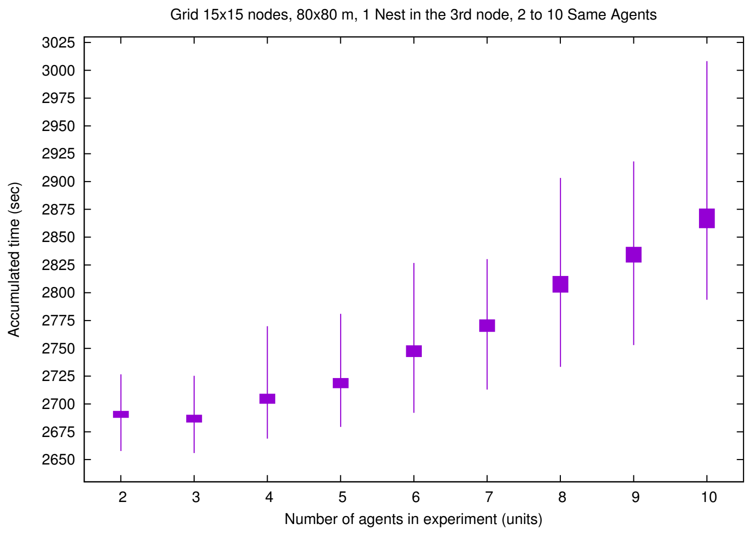

Figure 5.9: Simulation results in a grid of 15 × 15 nodes, 80 × 80 meters, with no obstacles, 1 nest, 2 to 10 same agents

Impact of Obstacles

The last part of experiments is done in a big grid of 20 × 20 nodes, of a physical size 80 × 80 meters. Figure 5.10 shows a relation between a number of obstacles and an accumulated time. Notice, that all bars in a given graph have different bodies length because of a different number of experiments made.

In Figure 5.10 we can see, that with an increase of an amount of nodes-obstacles, an accumulated time decreases. It happens because agents need to make fewer jobs in a grid, which is the most time-consuming operation. In opposite, the obstacles detection time for an agent is almost 0. A travel time in our settings here is less than a job time. Therefore, it can be even faster to go around a node-obstacle or even several obstacles, rather than to enter a node and to make a job. The bars in the graph has long whiskers, i.e. the difference between the minimum and maximum is big. It mostly depends on a dispersion of obstacles and their density in some areas of a grid.

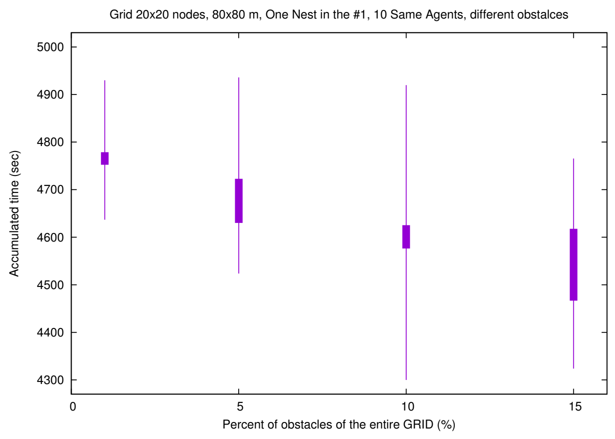

Figure 5.10: Simulation results in a grid of 20×20 nodes, 80×80 meters, with different obstacles, 1 nest, 10 same agents

Impact of Preparation Time

The next graph (see Figure 5.11) demonstrates a relation between a number of the same agents in an experiment (starting from a nest in node 3) and a $totalTimeAll$. As we can see from this graph, a smaller number of agents gives us the best $totalTimeAll$ about 4580 seconds, while each new agent introduced in the experiment raises this time up to 4760 seconds.

It could be an easy solution, to use a minimum number of agents in every experiment. However, in a real-world situation, we are interested not in an accumulated time $totalTimeAll$. An operational time $totalTime_i$ is a better metric in such a case. This personal parameter gives us more precise information about an experiment. The maximum value $totalTime_i$ of all agents from an experiment, in a case when they start to work almost synchronously, shows us how much time should we spend 'in a field' to make a necessary amount of a total work. And so, this parameter leads us to a different conclusion.

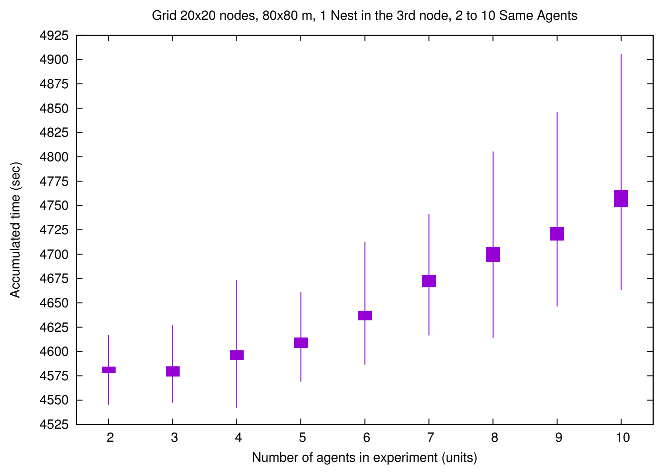

Figure 5.11: Simulation results in a grid of 20 × 20 nodes, 80 × 80 meters, with no obstacles, 1 nest, 2 to 10 same agents

A graph in Figure 5.12 demonstrates, how bad it can be if we use just 2 agents in a mission. In such a case, we need to wait (work) at least about 2300 seconds in total. Introducing the third agent decreases this time to 1500 seconds. And every new agent gives us much better time, up to 450 seconds, if we use all 10 agents in a mission. Thus, this graph shows that a bigger number of agents is better.

However, if we look at a situation realistically, it is not easy and not economically efficient to use a total number of *UAVs*, that we have to do the jobs. A preparation time of a *UAV* is an easiest to measure and compare parameter. In practice, it is unavoidable to spend a lot of time on preparation of a *UAV* before an experiment. It is necessary to assemble propellers, to check the sensors, accelerometers, *GPS*, compass, *WiFi*, cameras, gimbals, etc. And thus, every new *UAV* in an experiment leads to these preparation time expenses.

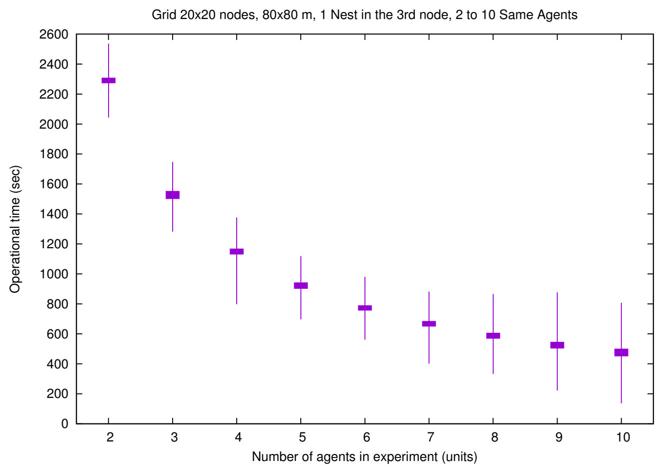

Figure 5.12: Simulation results of $totalTime_i$ in a grid of 20×20 nodes, 80×80 meters, with no obstacles, 1 nest, 2 to 10 same agents

In this thesis we simplify a situation. We assume that every agent has an unlimited battery charge, and $UAVs$ do not have an amortization and are risk-free. But, at least, we can evaluate a relation between a preparation time of each UAV and the best number of agents to use in an experiment.

Suppose, that on a preparation of $UAVs$, we have only one person (engineer, pilot). This person should prepare every UAV consequently. We can tell, that a preparation time for each similar UAV is on average equal. Thus, we can take it as a constant. The results of such an evaluation are given in Figures 5.13 and 5.14.

If a preparation time is close to 0, or, at least, less than 48 seconds, it is recommended to use all 10 $UAVs$. If the preparation time, for instance, is between 81 and 104 seconds, we should use 7 $UAVs$, 227 to 377 – 4 agents, more than 764 seconds – the minimum number of 2 $UAVs$. These values are very realistic. For example, a preparation time of an $OCTO$, used in our real experiments, can easily be about 5–7 minutes.

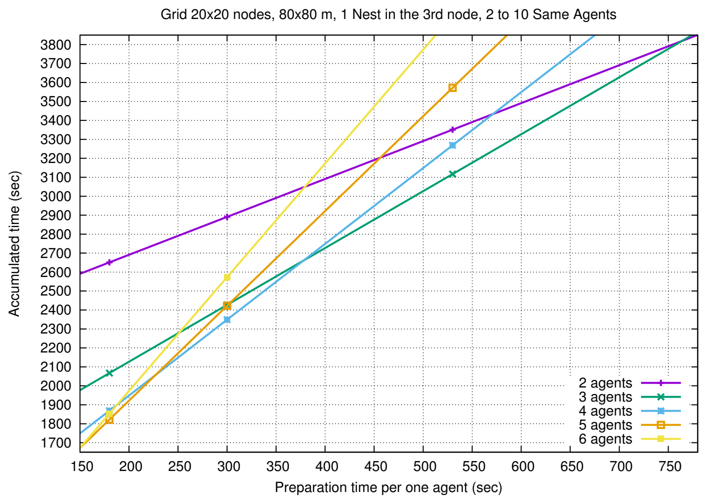

Figure 5.13: Simulation results of a preparation time and the best number of agents in a grid of 20 × 20 nodes, 80 × 80 meters, no obstacles, 1 nest, 2 to 6 same agents

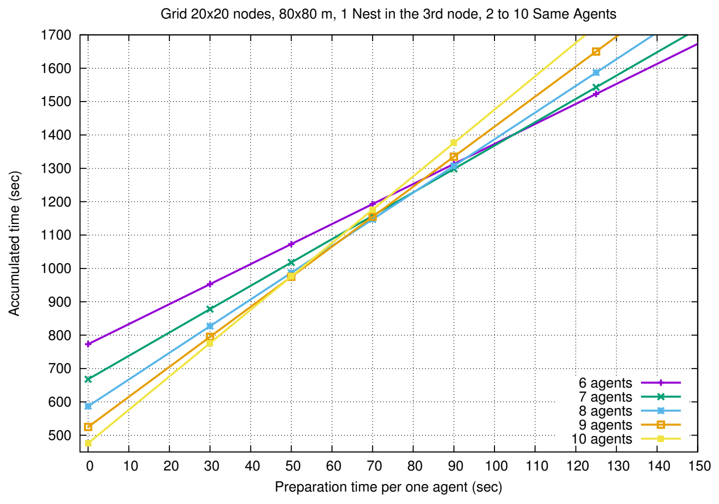

Figure 5.14: Simulation results of a preparation time and the best number of agents in a grid of 20 × 20 nodes, 80 × 80 meters, no obstacles, 1 nest, 6 to 10 same agents

Of course, these values are for a given grid and specific parameters of agents.

In a different situation, these values also should be different. Nevertheless, it is necessary to remember, that we need to find the best number of agents in a mission. It should give us a balance between the costs and outcome. In most cases, it should be somewhere in between of a minimum and a maximum number of agents, that we have can use in a mission.

Simulation Runs in a Big Graph

To finish with our simulations, we attach Figure 5.15, which shows two consecutive runs of the *MADGTA* in a big grid of 20×20 nodes, 80×80 meters, with multiple obstacles, one nest in node 1, and 3 same agents.

In Figure 5.15, it is clearly shown, that final paths of agents are different from one run to another in such a big graph, where agents have bigger options to move differently. However, results of such missions are very close to each other (up to a second, while overall each mission has taken about 23 minutes in our settings). In these graphs, it is shown, how three agents disperse from a nest, which is located in a bottom left corner of a grid. The green agent has mostly done a left sub area of a grid. The yellow agent has a complex path, and mostly did a right part of the graph. While the blue agent did its work in a bottom part of the graph. In the end of a mission, all agents came close to each other in a middle part of the graph.

Therefore, the simulations, described in this thesis, clearly show the abilities of the *MADGTA* to give stable suboptimal results for most cases, graphs of a different size, a different number of agents, obstacles, nests, etc.

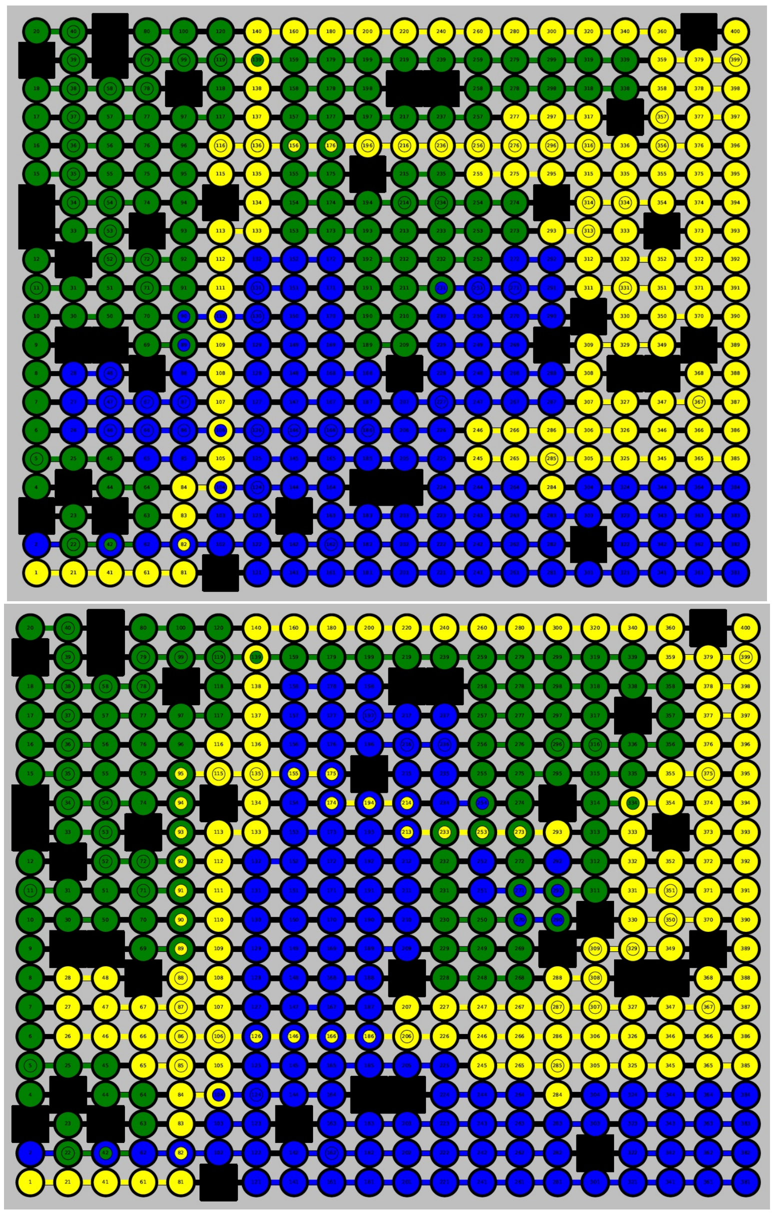

Figure 5.15: The resulting grid of 20 × 20 nodes, 80 × 80 meters, with multiple obstacles, 1 nest, 3 same agents

5.2 Experimental Results

5.2.1 Description of UAVs Used by us in Real Experiments

Real experiments were made with *UAVs*, more precisely, with multicopters. The conducted experiments were done with a satisfaction of all legal requirements of Transport Canada (see in Appendix B for a detailed description).

In experiments of our laboratory we use two types of multicopters: (1) an octocopter 3DR X8+ (in this thesis we call it shortly the *OCTO*), and (2) a modification of a quadrocopter RCT Spider (*QUAD*). These multicopters have a big size, powerful motors and can lift all necessary additional cargo. It is important, as our experiments are made outdoors. Therefore, there is a strong dependence on the environmental conditions, i.e. a speed of the wind, levels of the humidity and temperature, an amount of clouds, etc. Nevertheless, these *UAVs* have a better stability to do their work in such complex conditions.

Our multicopters combine two principal systems, which we describe here.

First, we show an energy system of our typical *UAV*. A scheme of a connection of all major parts is demonstrated in Figure 5.16.

The moving parts (motors and propellers) are connected to the speed controllers. Wires of the speed controllers goes to a module '*XT60* to the bullet connectors'. Instead of a module '*XT60* to the bullet connectors', in our *OCTO* there is a factory made 'power board' module. *UBECs* 5V are important modules, necessary to power all additional devices. A *UBEC* gives a power to our onboard computer Raspberry Pi. The *XT60* module (or the power board) is connected through a Pixhawk (or its analog Fixhawk) 'power module' to a battery. Also, we recommend using the small devices – battery voltage testers, to control a level of a charge of each cell of a battery. Another connector from a power module goes to the flight controller Pixhawk, which is shown in Figure 5.17.

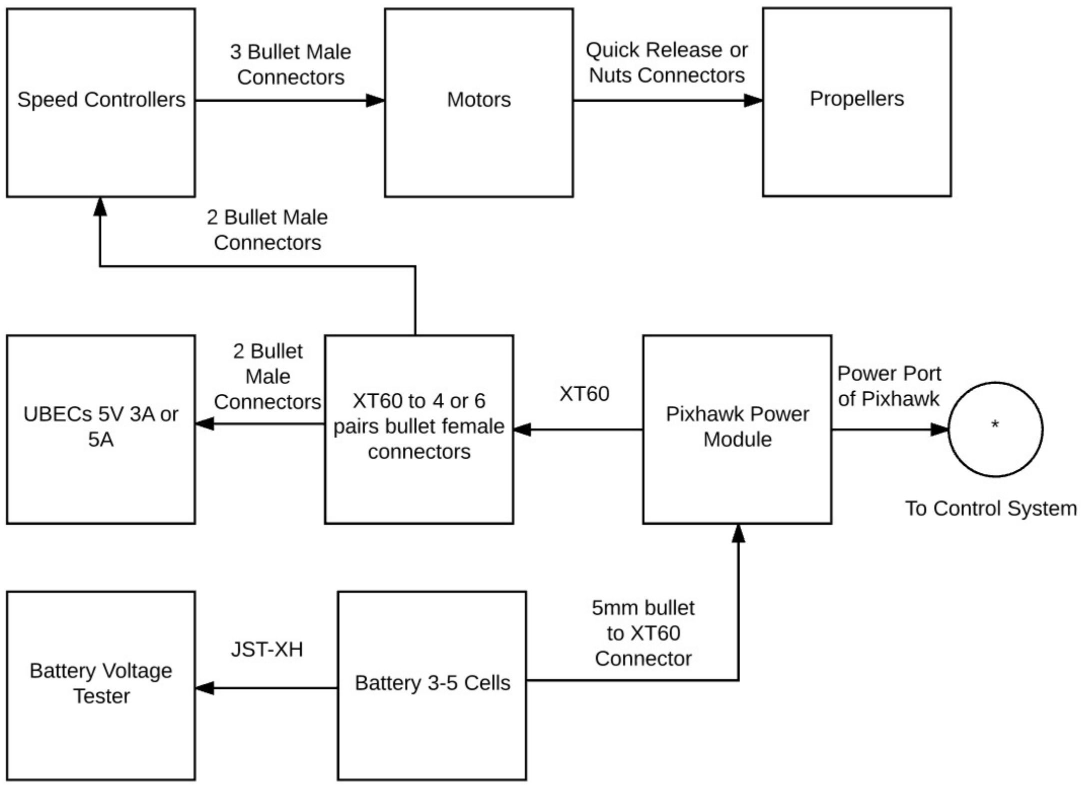

Figure 5.16: Energy system of our UAV

The module '*XT60* to the bullet connectors' makes an energy system more agile, as we can easily change a speed controller, or a motor, in a case of malfunctioning. We can make an additional power supply to all new devices and sensors with the s*UBEC*. Otherwise, we need to solder everything, as it is made with a factory power board of our *OCTO*.

A control system of our *UAV*, which is shown in Figure 5.17, is the most important for our algorithm.

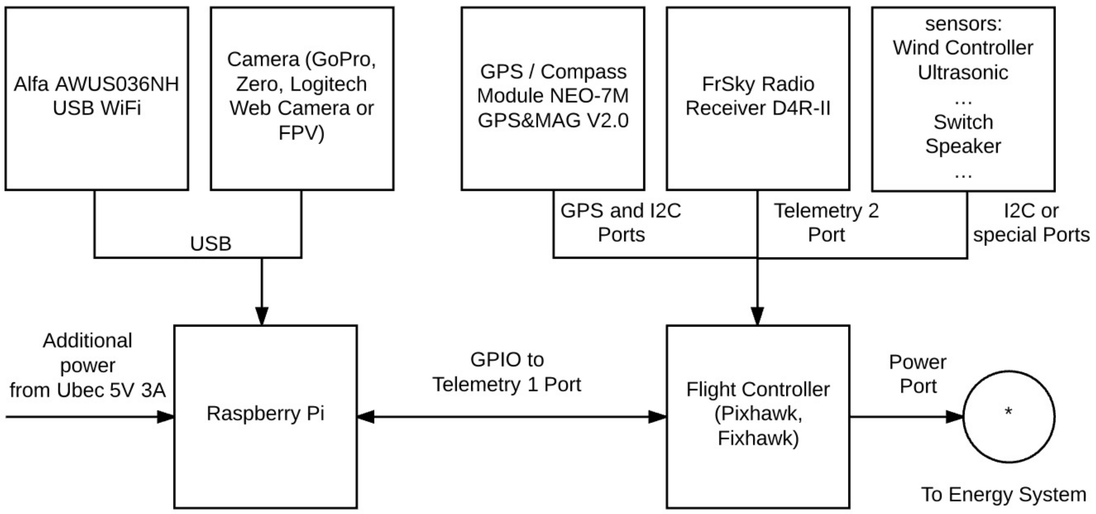
Alfa AWUS036NH USB WiFi
Camera (GoPro, Zero, Logitech Web Camera or FPV)
GPS / Compass Module NEO-7M GPS&MAG V2.0
FrSky Radio Receiver D4R-II
sensors: Wind Controller Ultrasonic … Switch Speaker …
USB
GPS and I2C Ports
Telemetry 2 Port
I2C or special Ports
Additional power from Ubec 5V 3A
Raspberry Pi
GPIO to Telemetry 1 Port
Flight Controller (Pixhawk, Fixhawk)
Power Port
*
To Energy System

Figure 5.17: Control system of our UAV

There are two main modules in the control system: a flight controller and an onboard computer. A flight controller Pixhawk that we use in our *UAV* is a high-performance autopilot-on-module. We also use an exact analog of it by the name Fixhawk. As an onboard computer, we have chosen a Raspberry Pi 2 model B.

All additional devices and sensors are connected to either a Pixhawk or to an *RPi*. A Pixhawk has a bunch of modules, as the *GPS* and compass module, radio receiver, switch, speaker, etc.

In some of our *UAVs* we have attached cameras, to get the best views of an experiment. We use different cameras: from a small *FPV* to a *GoPro* camera.

The most important part for a communication is a *WiFi* module. We have chosen a long-range *WiFi* module Alfa, as we need to obtain the biggest possible communication coverage with the *WiFi*. These modules are lightweight, do not consume much power, and can work up to two kilometers from each other.

Figure 5.18 demonstrates one of our *QUADs*, on a base of the RCT Spider model, which was assembled from scratch by us for the experiments. Our *QUAD* has principally new design, introduced by us. A step-by-step manual on how to assemble such a multicopter is prepared and should be available to download soon.

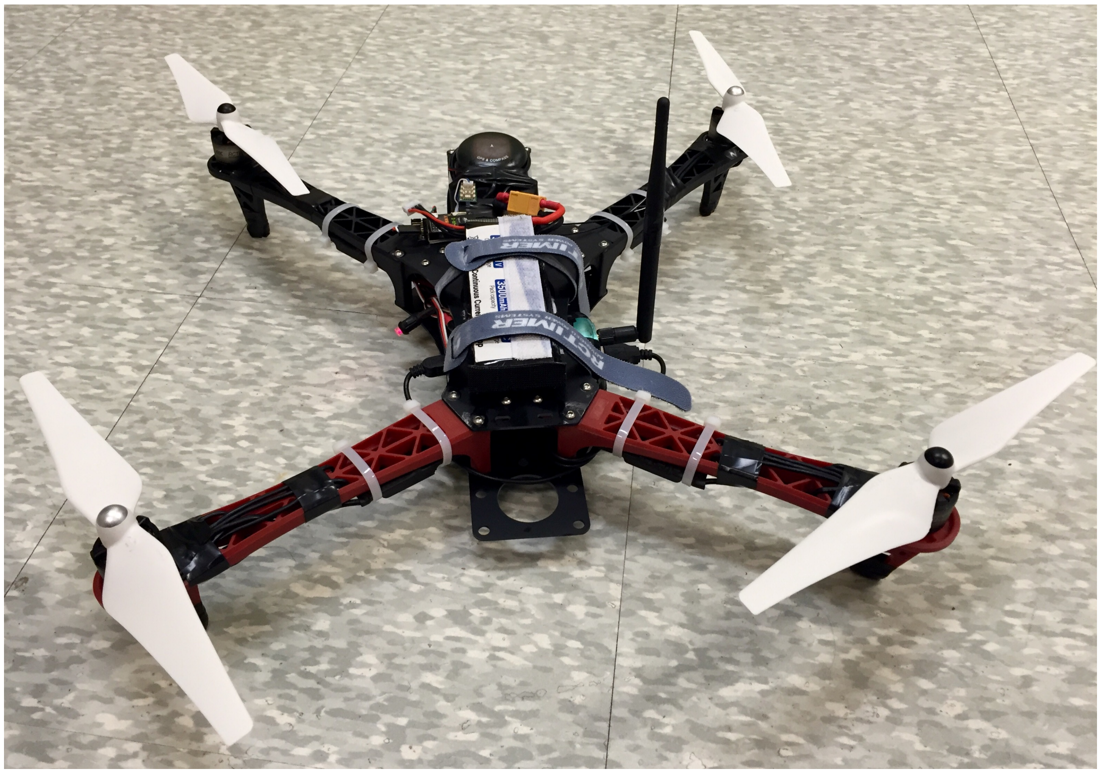

Figure 5.18: Our quadrocopter UAV

Some of the mentioned above major parts of the energy and control systems of a *UAV* can be seen in Figure 5.18. An onboard computer and a long-range *WiFi* module are positioned inside a body of a *QUAD*. The *WiFi* module is a green box with an antenna in the middle of a picture. The onboard computer sits right below this module. A *GPS* and compass module, which is crucial for all outdoors autonomous experiments, is located on the top part of the *UAV* (the black circle box).

5.2.2 Model and Analysis of Real Experiments

Real experiments were done in a field. A snapshot of a map of the field is given in Figure 5.19. This figure demonstrates 9 nodes, which gives us a small grid of a size 3×3.

UAVs can be located in any place of a map. If we position a *UAV* outside a graph, it finds the closest node of a graph and initially goes to that node. A *UAV* starts a mission only when it reaches a target initial node. When a *UAV* is located initially within a given radius of some node of a graph, it just goes to a precise *GPS* position of that node and starts a mission.

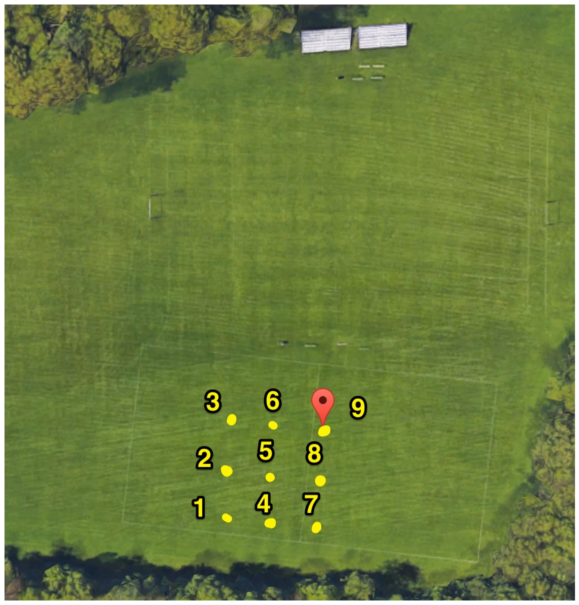

Figure 5.19: A snapshot of a map of an experiment

A snapshot of an experiment from the onboard *GoPro* camera of our *OCTO* is shown in Figure 5.20. The snapshot has additional marks of nodes of a grid. A rival agent can be seen in the top right corner of the picture. In this case, it is a *QUAD*.

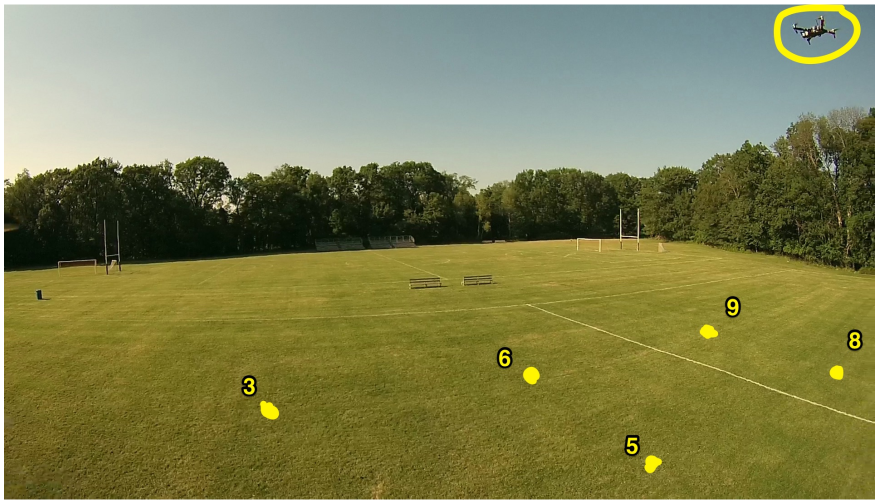

Figure 5.20: A snapshot of an experiment from an onboard camera of our octocopter

A photo of an experiment from the ground is demonstrated in Figure 5.21. The photo shows a moment of a synchronous takeoff from the ground of an *OCTO* with an onboard camera and a *QUAD*.

We have attached a video of several experiments from the ground and from the onboard *GoPro* camera (see a link in Appendix A).

In the first experiment, we use two agents: an *OCTO* and a *QUAD*. The *OCTO* has a travel speed = 250 cm/sec and a target working altitude = 4 meters. The *QUAD* has a slightly slower travel speed = 200 cm/sec and a higher target working altitude = 9 meters. Both of them has the same speed up = 150 cm/sec and speed down = 30 cm/sec. Two meters altitude change signals that a job in a node is done.

Figure 5.21: A photo of our multicopters doing a takeoff from the ground

The *QUAD* starts a mission first. In a small interval of time, the *OCTO* joins the mission. Both agents start their work outside a grid. The *OCTO* finds that the closest node of the grid is node 1. Similarly, the *QUAD* goes to node 4. As they reach the exact positions in the target initial nodes, they start the mission.

A trace of an experiment is:

- The *OCTO*: 0, 1, 1*, 2, 2*, 3, 3*, 6, 9, 8, 8*, 7, 7* – done (*RTL*);
- The *QUAD*: 0, 4, 4*, 5, 5*, 6, 6*, 9, 9*, 8 – not done (a *WiFi* connection error).

The numbers with a star in a given trace denotes a job done by an agent in a node, while only the numbers denotes a travel made by an agent.

This experiment has shown, how agents can finish all work in the entire grid, even in a case of malfunctioning of the other agents. Thus, our code gives us a good reliability of the system.

Agents have started a mission from the initial nodes 1 and 4 respectively. Initial nodes were vacant, therefore agents accomplished these nodes first. After that, agents have initiated our algorithm and have found the best next nodes to travel, as 2 and 5. The *QUAD* could also choose node 7, but, it has chosen one of possible nodes, i.e. node 5. For the *OCTO* a choice was obvious and it has chosen node 2. The next, agents moved to nodes 3 and 6 respectively, doing their jobs. At a moment of finishing a job in node 3, the *OCTO* has found that there are no vacant adjacent nodes to do a job, and it has chosen a further node 7, as a potential target. While travelling to that node, it has taken an adjacent node number 6 in the shortest path to a further node 7. Then it has taken node 9, and later 8. Meanwhile, the *QUAD* was able to finish a job in node 9. It has moved to an exact position of node 8. And at this point in time, its *WiFi* has given a malfunctioning response, which lead the *QUAD* to be stopped at node number 8 at 9 meters altitude. That implementation of our code, in a case of an error, stopped a *UAV* in a current position in the air. Later, we have introduced an *RTL* for such cases. As we have mentioned before, this moment shows a reliability of the entire system. As one of two agents has stopped its work, because of some malfunctioning, another agent finished a mission by itself. On its way to target node 7, the *OCTO* did a job in node 8 either. After that, it has traveled to the last node number 7, did a job, and has finished the entire mission in the grid.

In the second experiment, we use the same two agents: an *OCTO* and a *QUAD*. However, the parameters of *UAVs* were changed. The *OCTO* now has a faster travel speed = 400 cm/sec and a target working altitude = 9 meters. The *QUAD* has a travel speed = 250 cm/sec and a target working altitude = 4 meters. Both of them, again, has the same speed up = 150 cm/sec and speed down = 30 cm/sec. Two meters altitude change signals that a job in a node is done.

This time, an *OCTO* starts at the same position outside a grid, while a *QUAD* was moved to the *GPS* coordinates of node 4.

A trace of an experiment is:

- The *OCTO*: 0, 1, 1*, 2, 2*, 3, 3*, 6, 6*, 9, 9* – done (*RTL*);
- The *QUAD*: 4, 4*, 5, 5*, 8, 8*, 7, 7* – done (*RTL*).

In the second experiment, the mission was done, without any malfunctioning of *UAVs*. Therefore, we have a trace, demonstrating an optimal solution without any additional travel. As the *OCTO* has much higher travel speed, it was able to do a job in 5 out of 9 nodes of a grid. The *QUAD* did the remaining 4 nodes.

The third experiment was done in a much bigger and more complex grid. It was done in the same field, as shown in Figure 5.19, however, now a grid has 5 by 5 nodes and a size of 40 by 40 meters. In this experiment we have introduced 3 nodes, containing obstacles, which create a corridor near two borders of a grid. An initial graph is shown in Figure 5.22.

In this experiment, we use two agents: an *OCTO* (blue agent) and a *QUAD* (yellow agent) with mostly the same parameters. Both of them has the same travel speed = 400 cm/sec, speed up = 150 cm/sec, and speed down = 30 cm/sec. The difference is in a target working altitude: 4 meters for the *QUAD* and 9 meters for the *OCTO*. Two meters altitude change signals that a job in a node is done.

The *OCTO* starts from node 6 of a grid, the *QUAD* starts from node 22.

A trace of an experiment is:

- The *OCTO* (blue agent): 6, 6*, 1, 1*, 2, 2*, 7, 7*, 8, 8*, 3, 3*, 4, 4*, 5, 5*, 10, 10*, 9, 9*, 10, 15, 15*, 20, 20*, 25, 25* – done (*RTL*);

- The *QUAD* (yellow agent): 22, 22*, 21, 21*, 16, 16*, 11, 11*, 12, 12*, 17, 17*, 18, 18*, 23, 23*, 24, 24* – done (*RTL*).

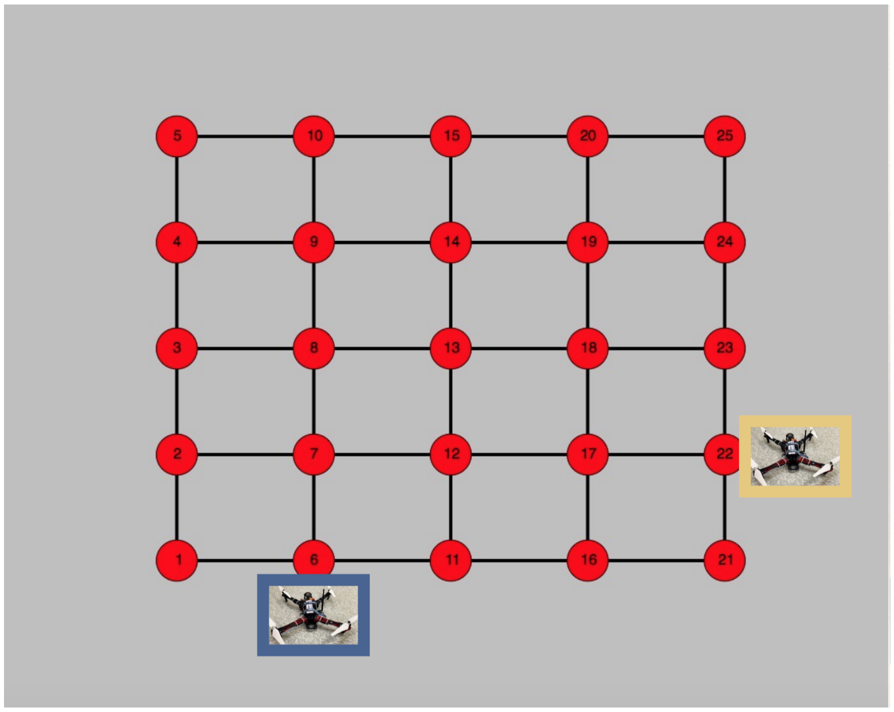

Figure 5.22: Initial graph of the third real experiment

This time, a mission was done with some hardware errors of the *QUAD* (yellow agent), which can be seen in the attached video (in Appendix A) in not precise evaluation of a current altitude during a mission. The *OCTO* (blue agent) worked perfectly. That is, why the *OCTO* was able to do the work in a bigger number of nodes. However, other than that, the *MADGTA* itself was reproduced correctly. As a result, we can see only one additional travel back by the *OCTO* in node 10. After the work in the entire graph was done (see Figure 5.23), *UAVs* made the return to the initial positions and landed almost synchronously.

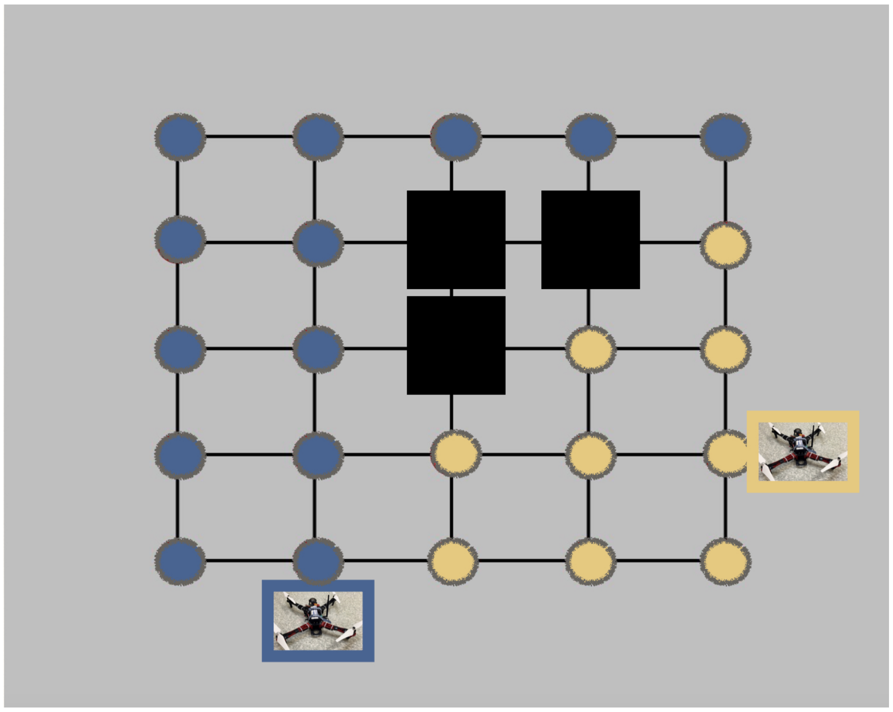

Figure 5.23: The result of the third real experiment

In the fourth experiment, we use three agents: one *OCTO* and two *QUADs*, which are shown in Figure 5.24. The *OCTO* (green agent) has a travel speed = 250 cm/sec, a target working altitude = 13 meters. Both *QUADs* has a travel speed = 400 cm/sec, one of them has a target working altitude = 8 meters (blue agent), while the second – 3 meters (yellow agent). All three *UAVs* have the same speed up = 150 cm/sec and speed down = 30 cm/sec. Two meters altitude change signals that a job in a node is done.

A grid has the same structure and parameters, as from the third experiment, with only one difference in an additional obstacle in node 3 (see Figure 5.25). Starting nodes of *UAVs* in a mission are 1, 16 and 22 (which was the closest to the initial location of that *QUAD*).

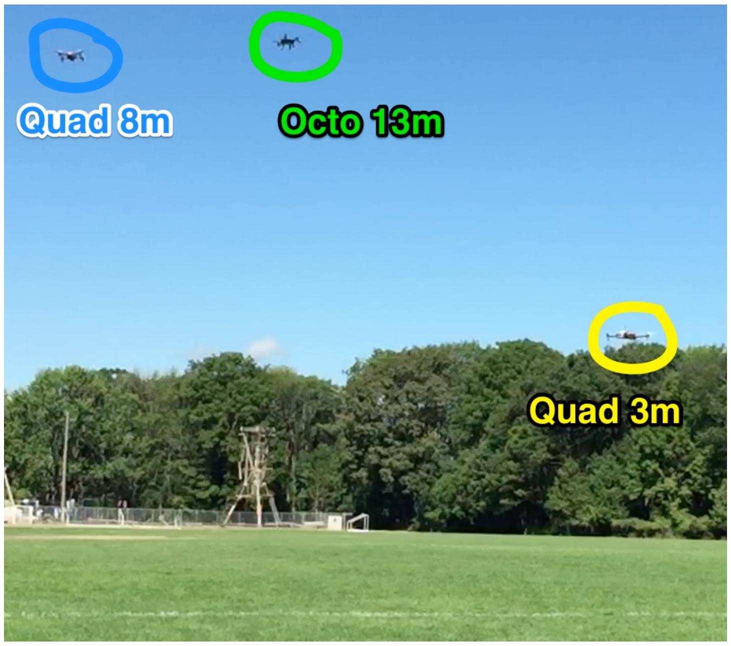

Figure 5.24: A photo of three UAVs doing their work in the fourth real experiment

A trace of an experiment is:

- The *OCTO* (green agent): 1, 1*, 2, 2*, 7, 7*, 8, 8*, 9, 9*, 4, 4*, 5, 5*, 10, 10* – done (*RTL*);
- The *QUAD* (blue agent): 16, 16*, 11, 11*, 6, 6*, 11, 12, 12*, 17, 17*, 18, 18*, 23, 22, 21, 21* – done (*RTL*);
- The *QUAD* (yellow agent): 0, 22, 22*, 23, 23*, 24, 24*, 25, 25*, 20, 20*, 15, 15* – done (*RTL*).

The mission was done according to the *MADGTA* and without any errors. The final grid is shown in Figure 5.26. As expected, three agents did a mission faster in comparison to the results of the previous experiments.

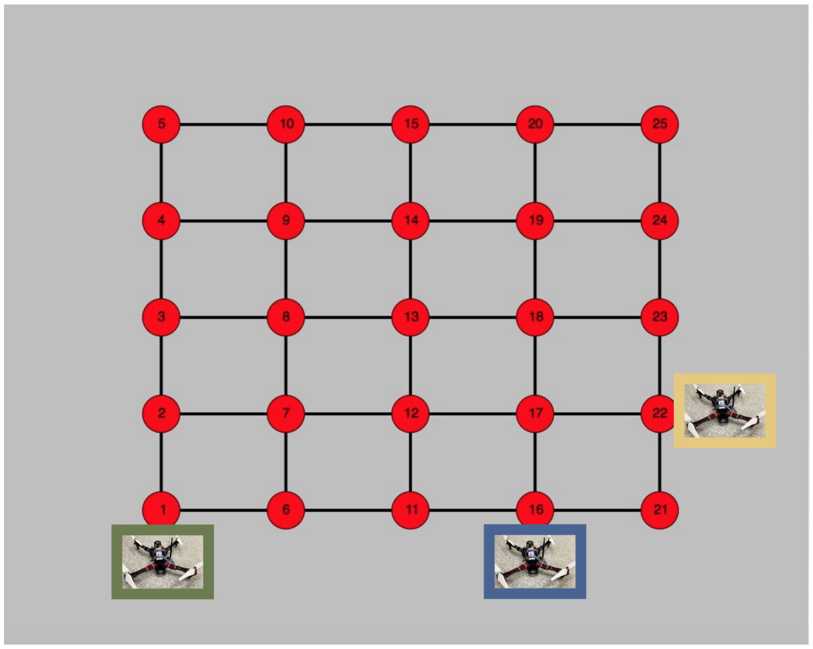

Figure 5.25: The initial graph of the fourth real experiment

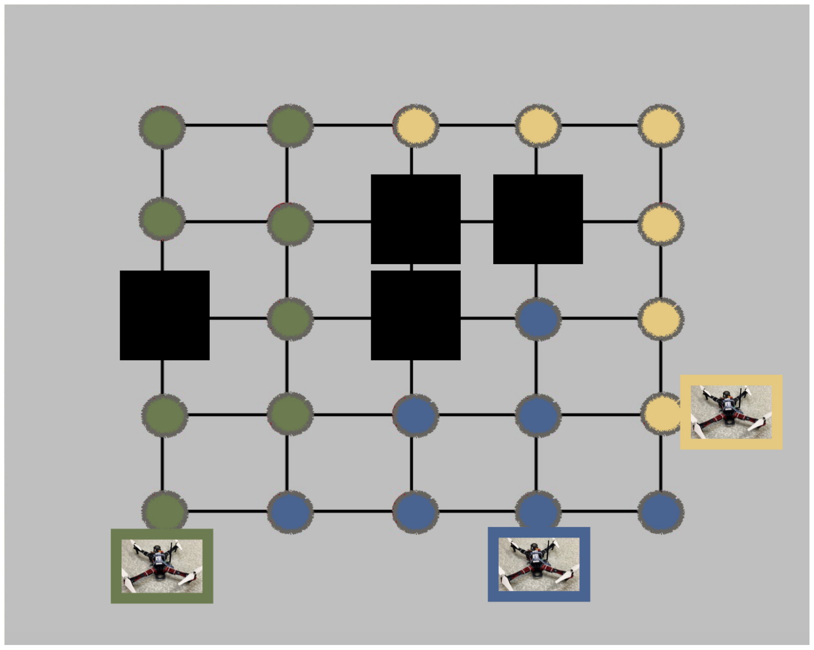

Figure 5.26: The result of the fourth real experiment

Thus, the provided experiments demonstrates a real implementation of the *MADGTA*. The programming implementation of the *MADGTA* and *UAVs* created by us during the research work proves a high reliability of the entire network of a distributed *UAVs*. Our agents are able to do their work in a fast manner with the least number of additional travels. And, even when some agents crashes, the remaining agents are able to successfully finish a mission.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this thesis, we studied the problem of decentralized and multi-agent graph traversal with application in the *UAV* technology. In particular, we

- proposed the online distributed algorithm for multi-agent graph traversal. The *MADGTA* works for any number of agents, their speed parameters, initial locations, position of obstacles, initially unknown by agents;
- implemented the *MADGTA* in Python. The programming implementation of the *MADGTA* works in computer simulations and experimentation with real *UAVs*, where each vehicle is controlled by an onboard Raspberry Pi 2 and a flight controller Pixhawk. In addition, our implementation allows a hybrid mode of real and simulated *UAVs*;
- evaluated our algorithm by a series of simulations on the *MADGTA* with different number of distributed agents, different parameters of agents, positions

of initial points, obstacles, etc. Our algorithm obviously exhibits suboptimal results in most cases;

- conducted experiments on the *MADGTA* implementation on a real network of *UAVs* (multicopters), which in practice confirms all results of the simulations.

Our simulations and experiments clearly validate our claim that multi-agent graph traversal to accomplish a joint mission is indeed feasible and significantly improves the operation time.

6.2 Future Work

There are still numerous open research and practical problems:

- **Energy.** An important challenge in the *UAV* technology is energy efficiency. Thus, one has to consider the energy limits of agents during a joint mission. The most obvious solution for such a case is to measure the energy necessary to make a come back to a nest at each point in time. If an agent reaches the limit, it initializes a return to the launch command;

- **Fault-tolerance.** One can design algorithms that can tolerate the different type of faults. This includes crash faults (that stop an operation of an agent), message loss, and Byzantine faults (that misrepresent a location or a job accomplishment of an agent). Our current algorithm implementation in the simulation mode, with minor improvements, can give us the tools, to stop any agent at any point in time, invoking a crash fault. If a crash fault is detectable, other agents should immediately correct their knowledge, and continue a mission with an addition of a node where the agent crashed;

- **Environment.** The environment can contribute to the overall operation of *UAVs*. For instance, computer vision can give us tools for finding and distinguishing targets and obstacles, to introduce a feedback from *UAVs* to do a better implementation of planning, collision avoidance, etc;

- **Number of agents.** An interesting problem for the future research, is the evaluation of an optimal number of agents, depending on a goal we choose. For instance, we need to find what is a minimum number of agents to make the tasks in the entire environment in a guaranteed suboptimal time with the least energy waste. We already have mentioned a relation between a preparation time and the best number of agents in an experiment, however, there can be done much more;

- **Scale of experiments.** We are planning to scale up our experiments to the range of tens of kilometers size of the environment, represented by a graph. This means, that a perfect *WiFi* broadcast, that we use currently, will be translated to an 'any-cast', as some agents will be out of a coverage range and will not get messages. However, with a reasonably big amount of a battery charge, our approach should work in this case. We also are planning to introduce 'dummy agents' in nodes, closer to the borders of the next area of a global environment. That is, this will be made in the places, from which rival agents have bigger chances to appear in the subarea of a graph, where a current agent is located. This way, a 'real' agent, using our algorithm, will make a job in its part of a graph from the further area to the borders with dummy agents. Later, an agent can switch to another area of the environment, when it is done with its part. Therefore, the algorithm should work, even in a case of a presence of only one agent, while currently, we require having initially at least two working agents;

- **Improvement of message complexity.** We are going to try different approaches to the creation of the optimal checksums with a reduced size, which should guarantee the same quality of checks of the broadcasted messages.

Appendix A

Link to Video of the Experiments

https://drive.google.com/open?id=0B61bDhJZgNFKOUQwVlB3eWxGdVE

Appendix B

Legal Requirements to Conduct Experiments with UAVs

According to the Transport Canada, “most *UAV* operators must get Transport Canada’s permission to use a *UAV* for any kind of work or research. However, under very specific, lower-risk circumstances, you may qualify for an exemption if you meet all the safety conditions” [A1].

The experiments conducted with the Quadrocopter *UAVs* of a maximum take-off weight not exceeding 2 kilograms and with the Octocopter *UAVs* of weight equal, or slightly bigger than 2 kg. For both types of *UAVs* the conducted experiments are under the exemptions and do not require a Special Flight Operations Certificate (*SFOC*).

First exemption is for *UAVs* that weigh two kg or less. According to [A2], “This exemption relieves persons conducting non-recreational *UAV* system operations utilizing a *UAV* with a maximum take-off weight not exceeding 2 kilograms, operated within visual line-of-sight from the requirement to obtain a Special Flight Operations

Certificate (*SFOC*) as required by sections 602.41 and the requirement to comply with the conditions of an *SFOC* as required by section 603.66 of the CARs. The exemption will permit non-recreational *UAVs* with a maximum take-off weight not exceeding 2 kilograms to be operated away from built-up areas, controlled airspace, aerodromes, forest fire areas and other restricted locations. The exemption includes conditions which address the need for the safe and responsible use of certain *UAV* systems".

Second exemption is for *UAVs* above two kg up to and including 25 kg. According to [A3], "This exemption relieves persons conducting non-recreational *UAV* system operations utilizing a *UAV* with a maximum take-off weight exceeding 2 kgs but not exceeding 25 kgs, operated within visual line-of-sight from the requirement to obtain a Special Flight Operations Certificate (*SFOC*) as required by sections 602.41 and the requirement to comply with the conditions of an *SFOC* as required by section 603.66 of the CARs. The exemption will permit non-recreational *UAVs* with a maximum take-off weight exceeding 2 kgs but not exceeding 25 kgs and with maximum calibrated airspeed of 87 knots or less to be operated away from built-up areas, airspace, controlled aerodromes, forest fire areas and other restricted locations. The exemption includes conditions which address the need for the safe and responsible use of certain *UAV* systems".

Both documents contain similar general and flight conditions. The document [A3] has more additional conditions, as the *UAVs* with a bigger take-off weight are obviously more dangerous in terms of safety.

However, for both types of *UAVs* used in the experiments all general, flight, and other conditions were satisfied. We mention here the most important requirements.

The *UAVs* were operated away from built-up areas, controlled airspace, aerodromes, forest fire areas and other restricted locations. The *UAVs* had the maximum speed of 4 meters per second. A requirement of a visual contact with the *UAVs* was satisfied, as the experiments were conducted in a field with the straight unaided view of the environment. It was done in the territory about 40×40 meters and at the altitude, not exceeded 16 meters. The *UAVs* were operated with a backup of a single control station and with the ability to take immediate active control of a *UAV* at all times.

Therefore, all legal requirements to conduct the experiments with the *UAVs* were satisfied.

[A1] http://www.tc.gc.ca/eng/civilaviation/opssvs/getting-permission-fly-drone.html
[A2] http://www.tc.gc.ca/civilaviation/regserv/affairs/exemptions/docs/en/2880.htm
[A3] http://www.tc.gc.ca/civilaviation/regserv/affairs/exemptions/docs/en/2879.htm

Bibliography

Agmon, N., Sadov, V., Kaminka, G. A., and Kraus, S. (2008a). The impact of adversarial knowledge on adversarial planning in perimeter patrol. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '08, pages 55–62, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Agmon, N., Kraus, S., and Kaminka, G. A. (2008b). Multi-robot perimeter patrol in adversarial settings. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 2339–2345.

Agmon, N., Kraus, S., Kaminka, G. A., and Sadov, V. (2009). Adversarial uncertainty in multi-robot patrol. In *IJCAI*, pages 1811–1817.

Agmon, N., Kaminka, G. A., and Kraus, S. (2011). Multi-robot adversarial patrolling: facing a full-knowledge opponent. *Journal of Artificial Intelligence Research*, pages 887–916.

Alonso, F., Alvarez, M., and Beasley, J. E. (2008). A tabu search algorithm for the periodic vehicle routing problem with multiple vehicle trips and accessibility restrictions. *Journal of the Operational Research Society*, pages 963–976.

Archetti, C., Speranza, M. G., and Hertz, A. (2006a). A tabu search algorithm for the split delivery vehicle routing problem. *Transportation Science*, **40**(1), 64–73.

Archetti, C., Savelsbergh, M. W., and Speranza, M. G. (2006b). Worst-case analysis for split delivery vehicle routing problems. *Transportation science*, **40**(2), 226–234.

Archetti, C., Speranza, M. G., and Savelsbergh, M. W. (2008). An optimization-based heuristic for the split delivery vehicle routing problem. *Transportation Science*, **42**(1), 22–31.

Baldacci, R., Battarra, M., and Vigo, D. (2008). Routing a heterogeneous fleet of vehicles. In *The vehicle routing problem: latest advances and new challenges*, pages 3–27. Springer.

Bektaş, T. and Laporte, G. (2011). The pollution-routing problem. *Transportation Research Part B: Methodological*, **45**(8), 1232–1250.

Bellmore, M. and Hong, S. (1974). Transformation of multisalesman problem to the standard traveling salesman problem. *Journal of the ACM (JACM)*, **21**(3), 500–504.

Berbeglia, G., Cordeau, J.-F., Gribkovskaia, I., and Laporte, G. (2007). Static pickup and delivery problems: a classification scheme and survey. *Top*, **15**(1), 1–31.

Bertsimas, D. J. and Van Ryzin, G. (1991). A stochastic and dynamic vehicle routing problem in the euclidean plane. *Operations Research*, **39**(4), 601–615.

Bianchessi, N. and Righini, G. (2007). Heuristic algorithms for the vehicle routing problem with simultaneous pick-up and delivery. *Computers & Operations Research*, **34**(2), 578–594.

Blackwell, T., Branke, J., *et al.* (2004). Multi-swarm optimization in dynamic environments. In *EvoWorkshops*, volume 3005, pages 489–500. Springer.

Brandão, J. (2004). A tabu search algorithm for the open vehicle routing problem. *European Journal of Operational Research*, **157**(3), 552–564.

Bräysy, O. and Gendreau, M. (2005). Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation science*, **39**(1), 104–118.

Caceres-Cruz, J., Arias, P., Guimarans, D., Riera, D., and Juan, A. A. (2014). Rich vehicle routing problem: Survey. *ACM Comput. Surv.*, **47**(2), 32:1–32:28.

Chan, Y., Carter, W. B., and Burnes, M. D. (2001). A multiple-depot, multiple-vehicle, location-routing problem with stochastically processed demands. *Computers & Operations Research*, **28**(8), 803–826.

Choset, H. (2001). Coverage for robotics–a survey of recent results. *Annals of mathematics and artificial intelligence*, **31**(1-4), 113–126.

Choset, H. and Pignon, P. (1998). Coverage path planning: The boustrophedon cellular decomposition. In *Field and Service Robotics*, pages 203–209. Springer.

Choset, H., Acar, E., Rizzi, A. A., and Luntz, J. (2000). Exact cellular decompositions in terms of critical points of morse functions. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 3, pages 2270–2277. IEEE.

Cieliebak, M., Flocchini, P., Prencipe, G., and Santoro, N. (2003). Solving the robots gathering problem. In *International Colloquium on Automata, Languages, and Programming*, pages 1181–1196. Springer.

Cordeau, J.-F., Gendreau, M., and Laporte, G. (1997). A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks*, **30**(2), 105–119.

Cordeau, J.-F., Laporte, G., Mercier, A., *et al.* (2001). A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational research society*, **52**(8), 928–936.

Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, **2**(4), 393–410.

Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, **6**(1), 80–91.

De Carvalho, R. N., Vidal, H., Vieira, P., and Ribeiro, M. (1997). Complete coverage path planning and guidance for cleaning robots. In *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE International Symposium on*, volume 2, pages 677–682. IEEE.

Desrochers, M., Desrosiers, J., and Solomon, M. (1992). A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, **40**(2), 342–354.

Dethloff, J. (2001). Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up. *OR-Spektrum*, **23**(1), 79–96.

Dorigo, M. and Gambardella, L. M. (1997). Ant colonies for the travelling salesman problem. *BioSystems*, **43**(2), 73–81.

Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA.

Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stützle, T., and Winfield, A. F. T., editors (2008). *Ant Colony Optimization and Swarm Intelligence, 6th International Conference, ANTS 2008, Brussels, Belgium, September 22-24, 2008. Proceedings*, volume 5217 of *Lecture Notes in Computer Science*. Springer.

Dror, M. and Trudeau, P. (1986). Stochastic vehicle routing with modified savings algorithm. *European Journal of Operational Research*, **23**(2), 228–235.

Dror, M. and Trudeau, P. (1990). Split delivery routing. *Naval Research Logistics (NRL)*, **37**(3), 383–402.

Eberhart and Shi, Y. (2001). Particle swarm optimization: developments, applications and resources. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 81–86 vol. 1.

Eberhart, R. C., Kennedy, J., *et al.* (1995). A new optimizer using particle swarm theory. In *Proceedings of the sixth international symposium on micro machine and human science*, volume 1, pages 39–43. New York, NY.

Elfes, A. (1987). Sonar-based real-world mapping and navigation. *Robotics and Automation, IEEE Journal of*, **3**(3), 249–265.

Elmaliach, Y., Agmon, N., and Kaminka, G. A. (2010). Multi-robot area patrol under frequency constraints. *Annals of Mathematics and Artificial Intelligence*, **57**(3), 293–320.

Emek, Y., Langner, T., Uitto, J., and Wattenhofer, R. (2013). Ants: Mobile finite state machines. *CoRR*, **abs/1311.3062**.

Emek, Y., Langner, T., Uitto, J., and Wattenhofer, R. (2014). Solving the ANTS Problem with Asynchronous Finite State Machines. In *Automata, Languages, and Programming*, pages 471–482. Springer Berlin Heidelberg, Berlin, Heidelberg.

Emek, Y., Langner, T., Stolz, D., Uitto, J., and Wattenhofer, R. (2015). How many ants does it take to find the food? *Theoretical Computer Science*, **608, Part 3**, 255 – 267. Structural Information and Communication Complexity.

Erdoğan, S. and Miller-Hooks, E. (2012). A green vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, **48**(1), 100–114.

Fazli, P., Davoodi, A., and Mackworth, A. K. (2013). Multi-robot repeated area coverage. *Autonomous Robots*, **34**(4), 251–276.

Feinerman, O. and Korman, A. (2012). *Distributed Computing: 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, chapter Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology, pages 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg.

Feinerman, O., Korman, A., Lotker, Z., and Sereni, J.-S. (2012). Collaborative search on the plane without communication. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 77–86, New York, NY, USA. ACM.

Fleszar, K., Osman, I. H., and Hindi, K. S. (2009). A variable neighbourhood search algorithm for the open vehicle routing problem. *European Journal of Operational Research*, **195**(3), 803–809.

Flocchini, P., Ilcinkas, D., Pelc, A., and Santoro, N. (2008). Remembering without memory: Tree exploration by asynchronous oblivious robots. In *International Colloquium on Structural Information and Communication Complexity*, pages 33–47. Springer.

Fu, Z., Eglese, R., and Li, L. Y. (2005). A new tabu search heuristic for the open vehicle routing problem. *Journal of the operational Research Society*, **56**(3), 267–274.

Gage, D. W. (1994). Randomized search strategies with imperfect sensors. In *Optical Tools for Manufacturing and Advanced Automation*, pages 270–279. International Society for Optics and Photonics.

Garey, M. R. and Johnson, D. S. (2002). *Computers and intractability*, volume 29. wh freeman New York.

Gaudioso, M. and Paletta, G. (1992). A heuristic for the periodic vehicle routing problem. *Transportation Science*, **26**(2), 86–92.

Gavish, B. and Srikanth, K. (1986). An optimal solution method for large-scale multiple traveling salesmen problems. *Operations Research*, **34**(5), 698–717.

Gendreau, M., Laporte, G., and Séguin, R. (1995). An exact algorithm for the vehicle routing problem with stochastic demands and customers. *Transportation science*, **29**(2), 143–155.

Gendreau, M., Laporte, G., and Séguin, R. (1996a). Stochastic vehicle routing. *European Journal of Operational Research*, **88**(1), 3–12.

Gendreau, M., Laporte, G., and Séguin, R. (1996b). A tabu search heuristic for the vehicle routing problem with stochastic demands and customers. *Operations Research*, **44**(3), 469–477.

Gendreau, M., Laporte, G., Musaraganyi, C., and Taillard, É. D. (1999). A tabu search heuristic for the heterogeneous fleet vehicle routing problem. *Computers & Operations Research*, **26**(12), 1153–1173.

Gervasi, V. and Prencipe, G. (2004). Coordination without communication: the case of the flocking problem. *Discrete Applied Mathematics*, **144**(3), 324–344.

Golden, B. L., Raghavan, S., and Wasil, E. A. (2008). *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media.

Gorenstein, S. (1970). Printing press scheduling for multi-edition periodicals. *Management Science*, **16**(6), B–373.

Harkness, R. and Maroudas, N. (1985). Central place foraging by an ant (cataglyphis bicolor fab.): a model of searching. *Animal Behaviour*, **33**(3), 916 – 928.

Hemmelmayr, V. C., Doerner, K. F., and Hartl, R. F. (2009). A variable neighborhood search heuristic for periodic routing problems. *European Journal of Operational Research*, **195**(3), 791–802.

Hert, S., Tiwari, S., and Lumelsky, V. (1996). A terrain-covering algorithm for an auv. In *Underwater Robots*, pages 17–45. Springer.

Ho, W., Ho, G. T., Ji, P., and Lau, H. C. (2008). A hybrid genetic algorithm for the multi-depot vehicle routing problem. *Engineering Applications of Artificial Intelligence*, **21**(4), 548–557.

Jäger, M. and Nebel, B. (2002). Dynamic decentralized area partitioning for cooperating cleaning robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 3577–3582. IEEE.

Kao, M.-Y., Reif, J. H., and Tate, S. R. (1996). Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. *Information and Computation*, **131**(1), 63 – 79.

Kennedy, J. and Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 5, pages 4104–4108. IEEE.

Klasing, R., Markou, E., and Pelc, A. (2008). Gathering asynchronous oblivious mobile robots in a ring. *Theoretical Computer Science*, **390**(1), 27 – 39.

Klasing, R., Kosowski, A., and Navarra, A. (2010). Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. *Theoretical Computer Science*, **411**(34), 3235–3246.

Kong, C. S., Peng, N. A., and Rekleitis, I. (2006). Distributed coverage with multi-robot system. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2423–2429. IEEE.

Langner, T., Uitto, J., Stolz, D., and Wattenhofer, R. (2014). *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, chapter Fault-Tolerant ANTS, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg.

Laporte, G. (1992a). The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, **59**(2), 231 – 247.

Laporte, G. (1992b). The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, **59**(3), 345–358.

Laporte, G. and Nobert, Y. (1980). A cutting planes algorithm for the m-salesmen problem. *Journal of the Operational Research Society*, pages 1017–1023.

Laporte, G., Desrochers, M., and Nobert, Y. (1984). Two exact algorithms for the distance-constrained vehicle routing problem. *Networks*, **14**(1), 161–172.

Laporte, G., Nobert, Y., and Desrochers, M. (1985). Optimal routing under capacity and distance restrictions. *Operations research*, **33**(5), 1050–1073.

Laporte, G., Nobert, Y., and Taillefer, S. (1987). A branch-and-bound algorithm for the asymmetrical distance-constrained vehicle routing problem. *Mathematical Modelling*, **9**(12), 857–868.

Lawitzky, G. (2000). A navigation system for cleaning robots. *Autonomous Robots*, **9**(3), 255–260.

Lenstra, J. K. and Kan, A. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, **11**(2), 221–227.

Li, C. and Yang, S. (2008). Fast multi-swarm optimization for dynamic optimization problems. In *Natural Computation, 2008. ICNC'08. Fourth International Conference on*, volume 7, pages 624–628. IEEE.

Li, C.-L., Simchi-Levi, D., and Desrochers, M. (1992). On the distance constrained vehicle routing problem. *Operations research*, **40**(4), 790–799.

Li, F., Golden, B., and Wasil, E. (2007a). The open vehicle routing problem: Algorithms, large-scale test problems, and computational results. *Computers & operations research*, **34**(10), 2918–2930.

Li, F., Golden, B., and Wasil, E. (2007b). A record-to-record travel algorithm for solving the heterogeneous fleet vehicle routing problem. *Computers & Operations Research*, **34**(9), 2734–2742.

Liang, J.-J. and Suganthan, P. N. (2005). Dynamic multi-swarm particle swarm optimizer with local search. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 522–528. Ieee.

Liu, Y., Lin, X., and Zhu, S. (2008). Combined coverage path planning for autonomous cleaning robots in unstructured environments. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, pages 8271–8276. IEEE.

Lumelsky, V. J., Mukhopadhyay, S., and Kang, S. (1990). Dynamic path planning in sensor-based terrain acquisition. *IEEE Transactions on Robotics and Automation*, **6**(4), 462–472.

Luo, C. and Yang, S. X. (2002). A real-time cooperative sweeping strategy for multiple cleaning robots. In *Intelligent Control, 2002. Proceedings of the 2002 IEEE International Symposium on*, pages 660–665. IEEE.

Luo, C., Yang, S. X., and Stacey, D. A. (2003). Real-time path planning with deadlock avoidance of multiple cleaning robots. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 3, pages 4080–4085. IEEE.

Machado, A., Ramalho, G., Zucker, J.-D., and Drogoul, A. (2003). *Multi-Agent-Based Simulation II: Third International Workshop, MABS 2002 Bologna, Italy, July 15–16, 2002 Revised Papers*, chapter Multi-agent Patrolling: An Empirical Analysis of Alternative Architectures, pages 155–170. Springer Berlin Heidelberg, Berlin, Heidelberg.

Marinakis, Y. and Marinaki, M. (2010). A hybrid multi-swarm particle swarm optimization algorithm for the probabilistic traveling salesman problem. *Computers & Operations Research*, **37**(3), 432–442.

Marino, A., Parker, L., Antonelli, G., and Caccavale, F. (2009). Behavioral control for multi-robot perimeter patrol: A finite state automata approach. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 831–836.

Min, H. (1989). The multiple vehicle routing problem with simultaneous delivery and pick-up points. *Transportation Research Part A: General*, **23**(5), 377–386.

Modi, P. J., Shen, W.-M., Tambe, M., and Yokoo, M. (2005). Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, **161**(12), 149 – 180. Distributed Constraint Satisfaction.

Moravec, H. P. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 116–121. IEEE.

Nagy, G. and Salhi, S. (2005). Heuristic algorithms for single and multiple depot vehicle routing problems with pickups and deliveries. *European journal of operational research*, **162**(1), 126–141.

Oh, J. S., Choi, Y. H., Park, J. B., and Zheng, Y. F. (2004). Complete coverage navigation of cleaning robots using triangular-cell-based map. *Industrial Electronics, IEEE Transactions on*, **51**(3), 718–726.

Pham, D. and Ghanbarzadeh, A. (2007). Multi-objective optimisation using the bees algorithm. In *Proceedings of IPROMS 2007 Conference*.

Pham, D., Ghanbarzadeh, A., Koc, E., Otri, S., Rahim, S., and Zaidi, M. (2011). The bees algorithm–a novel tool for complex optimisation. In *Intelligent Production Machines and Systems-2nd I* PROMS Virtual International Conference 3-14 July 2006*, page 454. Elsevier.

Portugal, D. and Rocha, R. (2011). *Technological Innovation for Sustainability: Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 21-23, 2011. Proceedings*, chapter A Survey on Multi-robot Patrolling Algorithms, pages 139–146. Springer Berlin Heidelberg, Berlin, Heidelberg.

Portugal, D. and Rocha, R. P. (2013). Distributed multi-robot patrol: A scalable and fault-tolerant framework. *Robotics and Autonomous Systems*, **61**(12), 1572 – 1587.

Potvin, J.-Y. and Rousseau, J.-M. (1993). A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, **66**(3), 331–340.

Prabhakar, B., Dektar, K. N., and Gordon, D. M. (2012). The regulation of ant colony foraging activity without spatial information. *PLoS Comput Biol*, **8**(8), 1–7.

Rekleitis, I., Lee-Shue, V., New, A. P., and Choset, H. (2004). Limited communication, multi-robot team based coverage. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3462–3468. IEEE.

Rekleitis, I., New, A. P., Rankin, E. S., and Choset, H. (2008). Efficient boustrophedon multi-robot coverage: an algorithmic approach. *Annals of Mathematics and Artificial Intelligence*, **52**(2-4), 109–142.

Savelsbergh, M. W. and Sol, M. (1995). The general pickup and delivery problem. *Transportation science*, **29**(1), 17–29.

Seidel, R. (1991). A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, **1**(1), 51–64.

Sheng, W., Yang, Q., Tan, J., and Xi, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, **54**(12), 945–955.

Shi, Y. and Eberhart, R. C. (1998). Parameter selection in particle swarm optimization. In *Evolutionary programming VII*, pages 591–600. Springer.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, **35**(2), 254–265.

Svestka, J. A. and Huckfeldt, V. E. (1973). Computational experience with an m-salesman traveling salesman algorithm. *Management Science*, **19**(7), 790–799.

Wehner, R., Meier, C., and Zollikofer, C. (2004). The ontogeny of foragwehaviour in desert ants, cataglyphis bicolor. *Ecological Entomology*, **29**(2), 240–250.

Wikipedia (2016a). Particle swarm optimization — wikipedia, the free encyclopedia. [Online; accessed 20-March-2016].

Wikipedia (2016b). Swarm intelligence - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Swarm_intelligence`. Online; accessed 14-March-2016.

Wohlgemuth, S., Ronacher, B., and Wehner, R. (2001). Ant odometry in the third dimension. *Nature*, **411**(6839), 795–798.

Wu, T.-H., Low, C., and Bai, J.-W. (2002). Heuristic solutions to multi-depot location-routing problems. *Computers & Operations Research*, **29**(10), 1393–1415.

Yokoo, M. (2012). *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer Science & Business Media.

Yokoo, M. and Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, **3**(2), 185–207.

Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, **10**(5), 673–685.