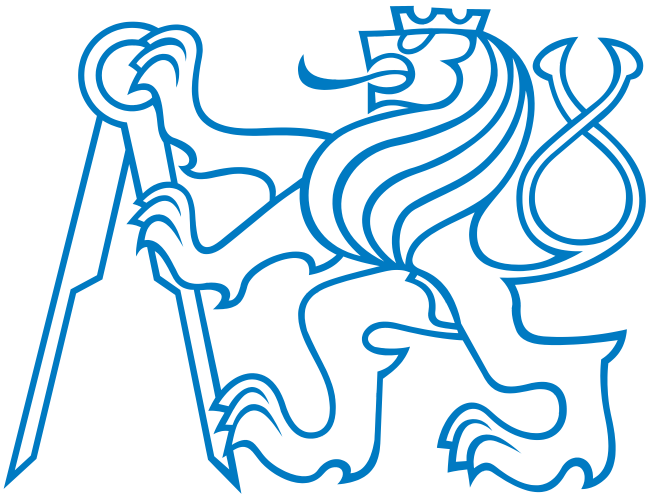
Czech Technical University in Prague
Faculty of Electrical Engineering

BACHELOR THESIS



Jiří Neliba

Forming of modular robot organisms from a swarm of robotic modules

Department of Control Engineering
Supervisor: **Ing. Vojtěch Vonásek**

Prague, 2014

Poděkování

Na tomto místě bych rád poděkoval především vedoucímu bakalářské práce Ing. Vojtěchu Vonáskovi, za jeho podnětné připomínky a náměty, které mi velmi pomohly při zpracování jednotlivých částí práce a díky nimž jsem vytvořil ucelenější dokument a vyvaroval se mnoha chyb při jeho tvorbě.

Dále děkuji všem kamarádům a rodině, kteří mě při práci podporovali.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostaně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Abstract

Tato práci se zabývá tématem formování modulárního organismu z hejna samostatných robotů. V této úloze je cílem pospojovat modulární roboty do předem známého robotického organismu. Cíl práce je implementovat state-of-the-art algoritmus, založený na náhodnosti, a A star algoritmus, který je založený na informovaném prohledávaní prostoru, pro formování organismu v našem naprogramovánem simulatoru v jazyku Java. Dále je cíelm práce tyto algoritmy porovnat.

In this work we're focusing on forming of modular robot organisms from a swarm of robotic modules. In this task is the goal to connect modular robots into pre-define robotics organism. The goal of this work is implement state-of-the-art algorithm, based on randomness, called Distributed Autonomous Morphogenesis, and A star algorithm, which is based on informed search of the space, for forming of organism in our programmed simulator with help of language Java. Then the goal of this work is to compare these both algorithms.

Contents

Chapter 1

Introduction

1.1 Motivation

Forming of a robotic organism from a swarm of autonomous robots is field of robotics and cybernetics, which can inspire from behaviour of e.g. insect, fish, birds. For instance, when ants search for food, there is an analogy in robotics when robots want to find sources of energy [18]. Sources of energy are randomly placed in the environment. It's used pheromone-like mechanism, due the other robots know, how much amount of energy is in the map. Another example is cooperative stick pulling by insects (Martinoli, 1999). However, not all collective behaviours have parallels in nature, such as coherent wireless networking(Nembrini et al., 2002). The advantage of this behaviour is clear, to make faster some process and saves the energy of an individual or to access thing, that one robot can't achieve. By organism, we mean a certain group of robot, which are connected to a common body and can communicate with each other.

It's the difference from swarm of robots, where they also can communicate with each other but they aren't physically connected, so they can move autonomously in the environment. It exists a transition from a swarm to an organism by which we will deal with in this work. Our algorithms for that use are A star (A*), with heuristic modifications, and Distributed Autonomous Morphogenesis (DAM) in a self-assembling robotic system [17], which is based on certain finite states. And more specifically, we want to compare the two algorithms by how effective they can achieve an given organism. We will describe more these two algorithms in the next section. We use the term as Forming algorithms which contains all forming algorithms, in our case A* and DAM. We can see example in Fig. 1.1. We use the term Cluster in the case of informed algorithms, which means robots with we're

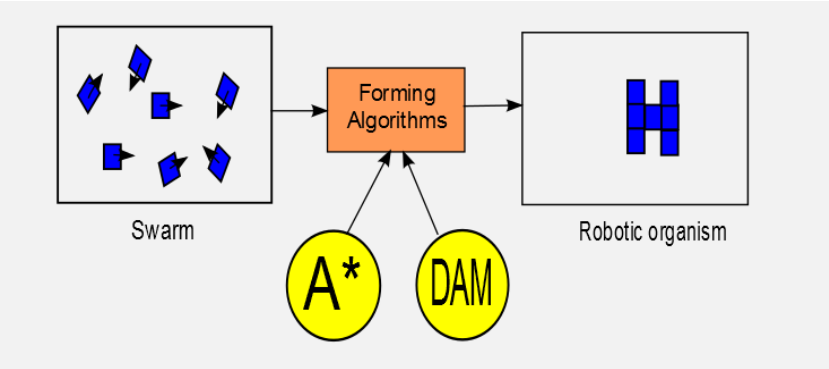


Figure 1.1: Process of forming an organism.

assembling the organism.

There are many different kinds of robots. We could divide them as aero, aquatic or ground robots. As aero robots we can use for example quadrocopters. As aquatic robots are using Tactically Expandable Maritime Platform (TEMP). As ground robots there are for example SYMBRION robots, rectangle robots. Here can be a combination of these robots in certain tasks as for example quadrocopter and ground robots as SYMBRION, which communicate with each other to fulfil certain task, as changing shape of the ground organism along the terrain. And there is the same analogy for combination aquatic and aero robots.

There are other very interesting possibilities of robots, as for example M-Blocks, which was developed in MIT and are work of student John Romanishin [8]. We can see M-Blocks in Fig. 1.2. They have special abilities as they can move in horizontal direction around organism above ground, thanks to a magnet, which is placed in each module. The modules can even jump, due to flywheel that spins at a maximum of 20,000 revolutions each minute. Then they suddenly released this energy at one time a transfer this specific kinetic energy to move or jump. The electro magnet can be on or off. This all are robots that can be in some swarm and form an organism or just make formation (in the air).

Swarm of robots can be used in various applications. For example, in rescue operations, the task is to find victims after an earthquake or in other similar situations. Although mobile robots are already used in these tasks, they cannot visit all places due to their size. A swarm of robots can be more flexible, as it can change its shape. For example, a modular robotic organism in form of a lizard is useful for walking on a terrain, and it can reconfigure to a snake to get into a tube or hole.

In military services for gathering information. They basically spread out,

Figure 1.2: Mblocks forming an organism. Courtesy of www.engadget.com.

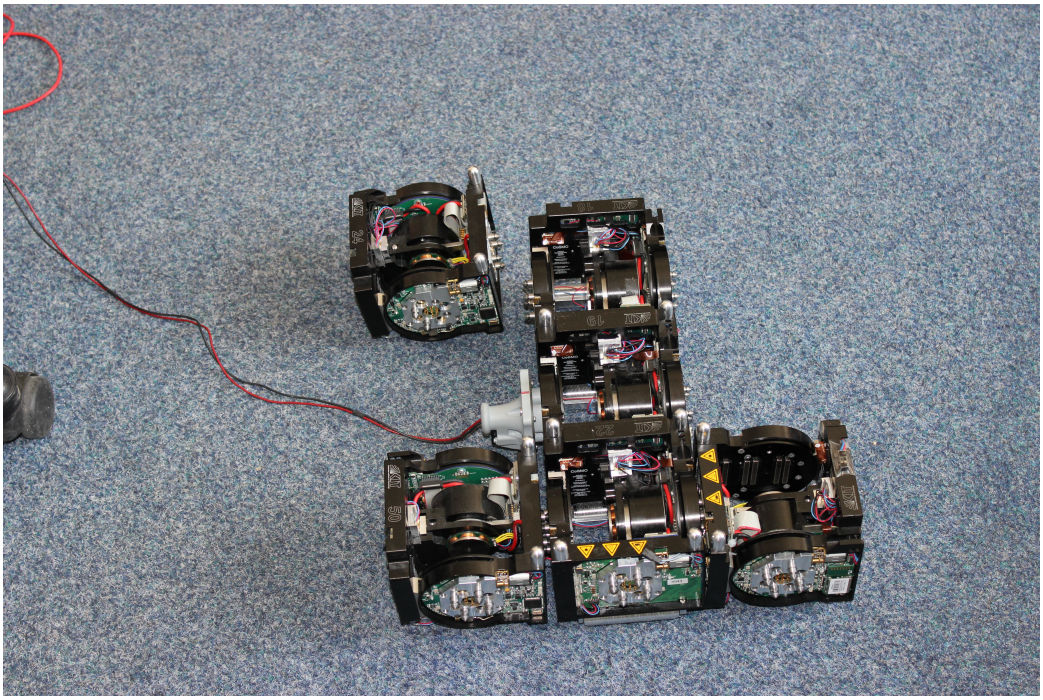


Figure 1.3: CoSMO robots forming an organism.

go through the environment and map it out, so when the humans entering the environment, they are already informed about the situation there [12]. They have to form organism for avoid or overcome obstacles as a wall, stairs

in building. In agriculture, the swarm of robots can take care of plants in a greenhouse. They can supply the plant with water and also crop from the plants [6]. They are communicating between themselves and helping each other. In civil engineering such robots can be use to help with harder tasks. For instance, for aqua robot, one robot finds the default position, connects to it and starts to emit a signal to form an organism, which can be used as a bridge for ground vehicles [14].

As you can see, there are many possibilities of use the swarm of robot and form an organism from them. In our work, we consider modular robotic platform CoSMO developed within EU project Symbrion/Replicator [9]. We can see them in Fig. 1.3. The CoSMO robots provide 3D locomotion using a powerful hinge and they can even move in 2D space using two screwdrivers. The 2D locomotion drive allows them to form the robotic organisms in 2D spaces. Forming on an robotic organism is necessary to allow the robots to achieve places, that would not be accessible by single modules. For example, to connect to a power plug mounted on a wall, a robotic organism is necessary.

Generally, the robots in swarm are not physically connected and they are executing their own tasks, but they can communicate with each other. If it's needed, one robot in the swarm can emit a signal to start forming of an organism. For the forming algorithms as DAM or A* are used and this process take a time. The duration of this operation is dependent on the complexity of the organism. After successful formulation to the organism, the robots in the organism can transfer energy, communicate with each other faster and can move only as a part of the organism. As the organism they can overcome large obstacles and reached a desired task.

Our task is to form different organisms in 2D, because every organism, event if they seems same, they can have different locomotion, in our case, in 3D and to form such an organism we have to connect all the robots. It's used 2D space, because it's much more difficult to form an organism in 3D space. Our work utilizes a simulation of the robotic swarms. The simulations allow to investigate a given problem much faster than conducting the experiments with real robots. We can test the algorithm in virtual world if it's actually possible to run the given algorithm in real world. Virtual simulation is also faster, enable to work with more robots, no maintenance needed and we can experiment in otherwise with hardware dangerous situation.

The goal of our work is to implement the informed search algorithm A* and stochastic DAM algorithm for formulation of an organism. Test them and try to compare the result of both algorithms. The purpose is to gain information and confirm an ideas about these two algorithms. By that we will better know which of the use in the given situation.

1.2 Basics

At the beginning, we should describe what is initial and goal state of the studied problem. The initial state are the positions of all robots, their angles of turning and identity numbers. The goal state is when all robot needed for desire organism are on their places in the organism. The robots themselves can move, rotate and communicate all the time. Also they can connect physically with each other and they can lift themselves up in the organism. The goal of our algorithms is to form a predefined organism from a given initial position.

Forming of robotic organisms from swarms can be achieved by two approaches. One approach works with static robots and before it makes a movement, the algorithm plans all the movements leading to the formulation of the desire organism. It's called deterministic algorithm and this type of algorithm is representing A* in our work. The second method works with robots which are moving and they can already working on a task. In the moment when a robot decides to form an organism, it has a plan, where it wants to connect a robot. The plan isn't distributed between other robots. And the algorithm within the robot has to wait until the second robot fulfil a conditions, so the robot can connect it. The algorithm ends when the plan of connection is done. This approach is stochastic algorithm and it is represented by DAM in this work.

A* is a deterministic algorithm, based on a informed search of state-space. That means that in A* all robots know how will the completed organism looks like and have information about how far they are to complete this task. In A* we call these robot Cluster. Every robot has its plan of moves, which they have planned as the best possible path to the goal position. But this "best" path is strongly dependent on function that they are using to compute it.

In contrary, DAM is a stochastic algorithm. As you can see, it has advantage of that it isn't dependent on any function and or it doesn't compute any data so we certainly save requirements on the hardware. But the DAM algorithm can be slower then A*, which depends on the initial conditions, the number of robots and the desired shape.

1.3 Platforms

Robots usually consist of several building blocks with uniform docking devices that allow transfer of forces, torques, electrical power and communication throughout the robot system [11]. Depending on the geometrical arrangement of their modules, modular self-reconfigurable robot (MSR) sys-

tems are divided into two groups. Modules in a chain-type system are arranged in a string or tree topology. Examples are the CONRO [1], the Polybot [10] and the Molecubes [16]. Lattice-type systems, such as the Atron [13] or the Catom [3] robot consist of modules which are arranged in a regular pattern, such as a cube or hexagonal grid. Then there are hybrid of these two types [9]. Example of Hybrid architecture is Superbot [2] and M-Tran [4]. The advantage of hybrid architecture it's able to complete different needs and it is able of multitasking.

Modules can be mobile, meaning that each module is able to locomote by its own without being connected to other modules. Such mobile MSRs are for example the CEBOT [15], the Swarm-Bot [5], the Sam-Bot [7] and Symbrion. Every of them is good for different tasks, as distance they can travel or they can former really an long organism and so on. Our work is based on CoSMO platforms. The platforms allows to move to all four sides and it can turn, which is due to the two wheels, which are driven by a motor. CoSMO robots can also communicate with each other and have possess much more computational power than other MSR systems. In fact, this is their big advantage above the other platforms we mentioned. The CoSMO module is a part of the heterogeneous Symbricator platform [3], which consists of more kinds of robots. CoSMO was designed to fulfil the role of building blocks for the organism and to act as its Backbone. It consists of a powerful hinge and has the ability to dock with every of its four sides to every side of every other Symbricator robot [9]. Each robot is endowed with 8 proximity sensors, 8 docking alignment sensors and 4 channel local communications.

Problems with real world robots are often the hardware ones, which we can hardly count with as error in hardware of robot or something can lock wheels of robots, it very depends on terrain in which they move. In laboratory environment with regular service we can decrease probability of this unwanted failure. Environmental problems can be different. For example, from strong electromagnetic fields, that can destroy communication. Communication can be also influenced by big enough obstacles, or here can be only dangerous zones for robots, or there may not be border for the arena and robots have to keep a distance to communication. Outside condition, mean weather, can be a big problem. At last hardware problem can be constriction for instance in precision of docking or the distance they can communicate with each other. With this problems of real world there would be more states and of course whole process of an forming organism would be longer. But the principle would be same as in virtual environment and here is the point of this work. Because the better given algorithm will be in virtual environment, the better performance can give in real world with right treatment of problems we have written above.

Chapter 2

Theory

2.1 Deterministic search algorithms

We have mentioned already that forming of an organism can be achieved using A*, so let take a closer look. The goal of A* is to create a plan to move all robots from their initial positions to a desired organism. It's always using at least two lists, one is called Open and the second is called Closed. In open list we have nodes, which we didn't visit yet, but we can visit from node we have already been. By nodes we mean states describing the organism. In our case, the states contains positions of the robots in the environment and their rotations.

Every edge has its value, which a robot has to pay for its usage. This cost is called tentative cost and it can be constant or function, which depends on our task and how we set the values/prices. If we have terrain which is homogeneous, we can set this tentative function to a constant value. This is common in laboratory conditions. If the terrain is diverse, the tentative cost can be a function, which is variable with distance from the start, for example linear function or it's variable, which can be hardly described as a function. Then we must find out price every edge/road. In practice is the most common the last case. The tentative cost is in the most cases marked as g. For instance in practice one way to choose g dynamically is:

$$g'(n) \quad = \quad 1 + \alpha(g(n) - 1). \tag{2.1}$$

Where g=1. If α is 0, then we have constant function (terrain costs are ignored), else if α is 1, we have tentative function of A*, which fully react on the travelled distance. We can set α somewhere between 1 and 0.

The second part, from which is the main function of A* composed is a heuristic function. Heuristic function is a function, that ranks our nodes in

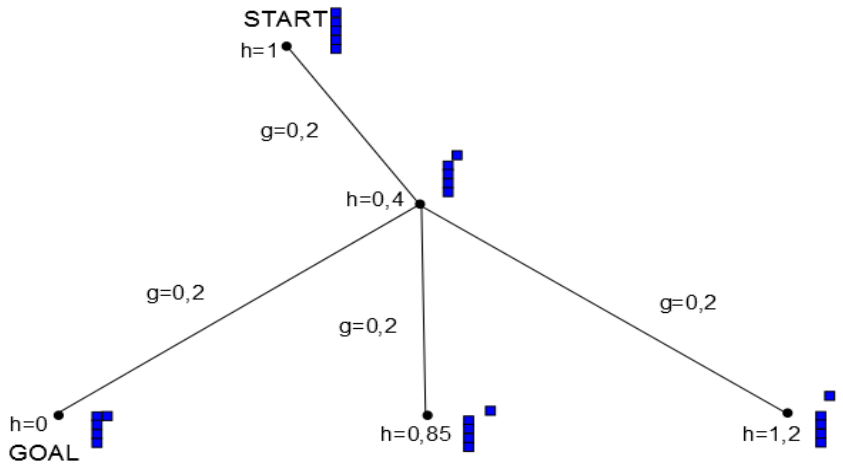


Figure 2.1: Example of problem forming an organism using informed algorithm A*. In the graph are marked tentative values as g and h are heuristic values. It's catch only part of the state space.

dependence on the distance from a goal. The distance can be computed by an different approach. One approach is called Euclidean distance. Here is the distance to the goal compute by Pythagoras formula, so the distance is direct, meaning aerial distance. Another is Manhattan distance, which needs a grid map. The grid map is a map where the robot can move in four basic directions as up, down, right and left. It can be used in other maps, which are not types of a grid map, but it can have negative impact on A* function and it isn't guaranteed the best possible path to the goal. In grid maps it really gives the best paths and the heuristic distance is then a real distance to the goal along given path. It's marked in most by the letter h. We can see such forming of an organism using A* in Fig. 2.1

Then we can get a greedy heuristic, for example if we make square of Euclidean distance. It's so called over-estimate heuristic. This is good for experiment or for certain special task, but generally we try to avoid it. If the heuristic is over-estimated, so in our algorithm plays the main role the heuristic function, which leads in path by which we don't know if it's the best possible path. For example if there will be U obstacle in the way, it it's very probable that our robot won't avoid the obstacle in time. U obstacle is meant an obstacle of shape U, where robot is above it and goal under the letter.

In contrast there is under-estimate heuristic, where the heuristic function is too small in compare with tentative function and so the main role plays

tentative function. This approach assure us the best possible way, but we pay for it with time. Sometimes can also be used this approach if we know, that the goal is only few node distant or if we don't mind it takes a lot of time to solve the problem. We should have on mind, that in certain more complex task we don't have to get solution. For instance, Dijkstra is under-estimate algorithm, which takes into account just travelling price. Then there are another heuristic approach, it depends on the problem we try to solve with A*. Normally we try to achieve the h is always equal to g, then A* will find the shortest path and expands only path nodes, makes it really fast. Combine advantages of both, under-estimated and over-estimated h. This is in practice very difficult to reach and are needed more pre-computation.

Sometime there can be more paths with the same length/price, which can lead to the goal. Then we have to decide, which path is optimal for us. This can be done by modifications of the functions g or h. We can scale h upwards slightly. Downwards it would expands the nodes near the starting point and it wouldn't be wished. We want expand to the goal a little faster. We can use the following formula:

$$h = h(1+p), \tag{2.2}$$

where the p factor goes from this formula:

$$p = min/maxlength, \tag{2.3}$$

where min means the minimum cost of taking one step and by $maxlength$ we mean the expected maximum path length. The second is the upper limit, which we're expecting our A* will not cross. Steven van Dijk suggests let the h passes to the comparison function, if f functions are equal, and here compare h and so break the tie. Another way is to prefer the paths, that are along the straight line from the start to the goal. Next way is for example construct A* priority queue, where new insertions are always ranked better. If this doesn't help, there are methods specially for grid maps as Rectangular Symmetry Reduction.

We have to take attention of scale too. If one function is in different scale than the second, it would slows down our algorithm or we wouldn't have to find the best path. For instance if we would have unit in meters and in hours. A* can be used on multiple goals too, only is needed to iterates it always in the state like it was before and then connects the shortest way. A* itself is controlled by functions g and h and we have to choose from the speed of an algorithm and the accuracy of a founded path.

2.2 Stochastic algorithm for organism building

In contrast to the above described deterministic approach, Distributed Autonomous Morphogenesis (DAM) [17] is a stochastic algorithm. DAM works as a finite state machine running autonomously on each robot. The basic two modes are Organism and Swarm. Each of these modes has states, which set the behaviour of the robot. These algorithms have to know the goal, have to have certain plan in which they will proceed along. This plan can be tree-like and the nodes are described in it by a certain rule. The rules can be more, it very depends on number of robots which will connect to an organism in one time and the number of robot with we are working.

The plan can be for example make graphically as a tree, where the root, or Seed, is the first node. The Root node contains information about a robot, which started the formation process, in practice of our formation of an organism is the first robot, which starts to build the organism. The Root's children are robots which are connecting to it. If we take a robot, which has three sides to which other robots can connect and one side by which is the robot connected to another robot, Fig. 2.2. This imagination is valid for all robots, with exception of the root robot. Left, right and middle child can point out the side to which the robot should connect. It can be as left child is the west side, middle child is the back side, right child is the east side of the robot, if we take in consideration that robot's front side is connected to its parent.

Different approach is to look on plan as a list, where are inserted both information, about side and robot to which we need to connect. When we have more robots, we can make a list, which is composed of objects, which have both informations coded as pair of numbers. For instance we can mark the sides of robot 0,1,2,3 which will mean the front, right, back, left side, in chronological sequence. After a robot is connected to the organism, the robot that started the connection changes its actual connection plan. If there is no other robots to be connected, the plan is sent to its child, which then continues with the connection process. Edited plan is a plan which doesn't contain information about structure of whole organism. This approach is discussed in [17].

In the case, that we will connect more robots at one time, in the literature is also described another approach [17]. We can set a list as a sequence of char, which will notify which side should be connected. And the zeros, or if we chose another integer, notify us about if it still should be used the same robot or another one. Sequence of char is set of letters from an alphabet, which we

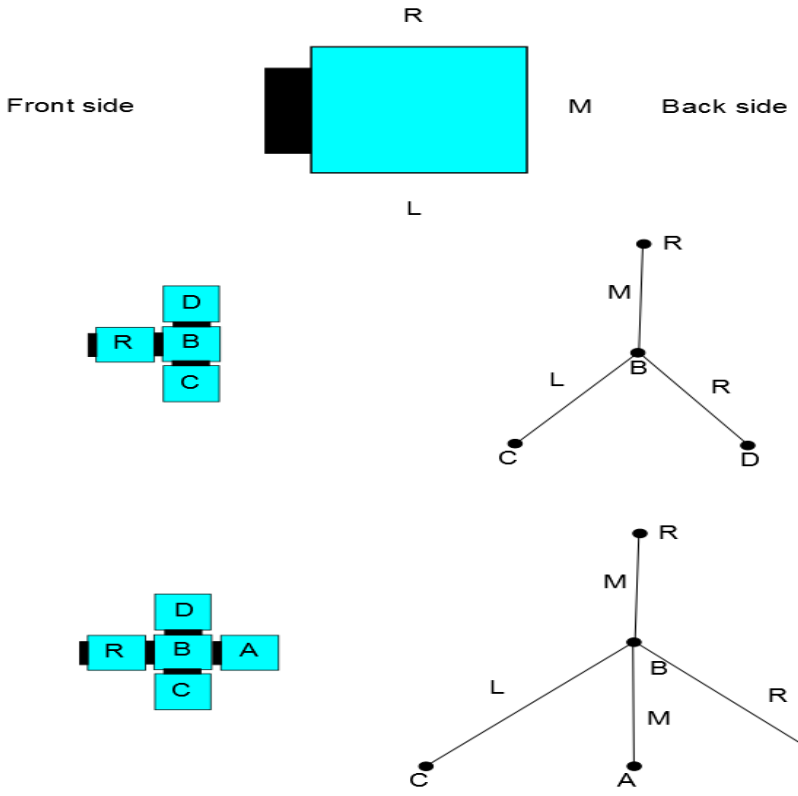


Figure 2.2: In the left side we can see an organisms composed from robots R,A, B, C, D, where R is a mark for root. The black rectangle is marking front side of a robot. In the right part we can see trees, where nodes are the robots and the edges are sides of a robot. Edges are marked M, L, R as middle, right and left side.

chose for the sides of a robot. For this recognition are used branches, where every robot can have maximal four branches. Every branch must have two nodes, one is the char and the second is the pairing zero. Every robot has to then searched in its list, which the robot gets, and finding out if every char in the order has its zero. If the number of zeros is same as the number of chars, we have a branch. We have to find how many branches we have, then we will connect along these branches. We will remove the branch after the connection and that pairs/branches under the branch sends to the connected robot, if there are pairs after (graphically under) this branch.

Every robot in this algorithm has its unique identification number (ID) and gets another identification number in an organism (ID_{Org}). These numbers are needed for communication and statistical purposes. ID_{Org} has to be

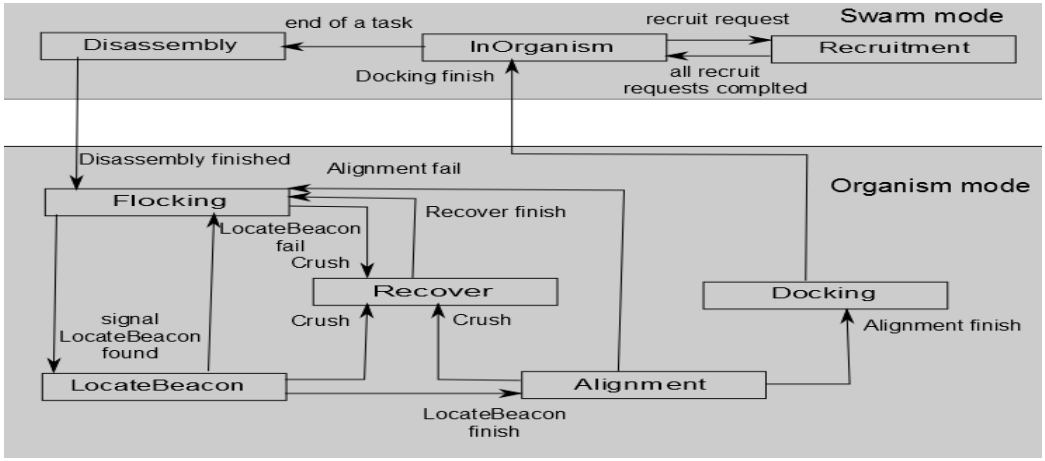


Figure 2.3: Finite state machine of Distributed Autonomous algorithm.

different, because it is a reference to the certain plan, if we are using method for connecting only one robot. Because we have to know if the robot wants to connect next robot or we can give Token to a different robot. Token is for us a privilege of a robot, which can be in Recruitment state, so it can connect with another robot. Now we take closer look to all states, we can see them as the finite state machine in the Fig. 2.3

- *Recruitment state* is in the Organism mode. In this state a robot starts to emit a signal, which attracts a robot, which is in Flocking state, on robot's certain side. The recruiting robot is the one who catch the signal and was in the Flocking state before. The whole DAM algorithm starts with the Recruitment state. In the beginning of the organism forming, only the seed robot is in the Recruit state. All other robots are in the Flocking state in Swarm mode.

- From Recruitment state robot can move only to the *InOrganism state*. This state is passive, it's just plays role if we decide about who will get Token. Otherwise, this robot are for us finished ones and play role as obstacles in the map. They are static and can be in this moment in a save mode, which will saves their energy. To this state the robot can also get from Docking state, we will discuses it later. From this state the robot also can go to Disassembly state, which is the second way, one is Recruitment state, where the robot can go.

- *Disassembly state* is the state, from which a given robot leaves Organism mode. This state takes care about disassembling an organism when it finished its task. The robot in this state unlocks its connection

with neighbourhood robots then it changes its state to Swarm mode to Flocking state. We can disassembly only few robots from organism or all. It depends on the algorithm and similarity of the organisms. Sometime a disassembly is enough to get a whole new organism.

- In *Flocking state* robots randomly move in a environment, in the directions their hardware is allowing them, and they randomly turning. They're also avoiding to each other, so no collision should happened and they also have to take in account borders of the environment. Also they have to avoid other obstacles in map, if there are some, or organisms.

- For these collisions there is special *Recovery state*. To this state the robot can get almost from every state in Swarm mode, every time a collisions happens. In this state can be implemented an algorithm for collision avoidance. For instance, a reaction algorithm, which in recruiting, or from LocateBeacon state to Docking, can drive around the given object and continue in the process again.

- *LocateBeacon state* means that the robot receives a signal from a robot in Recruitment state. In this state, the robot starts to behave deterministically as he tries to locate the source of the signal, which brings it to this state. In this state are generally used special communication channels as Infra Red (IR) [17].

- If a robot in the LocateBeacon state approaches close enough, the robot will get to the *Alignment state*. In this state, the robot tries to minimise the misalignment of two docking units. Docking units are devices, which are connecting the two robots. In this state the robot has to move very accurately. From this, if the robot doesn't collision, it moves to Docking state, these two states are in real world the most difficult tasks, because the process depends a lot on accuracy of hardware units too.

- *Docking state* is a mechanical docking procedure to physically connect a robot to an organism. After the completion of this, the robot moves to InOrganism state. This state is the only one from where the robot can't get to Recover state. In the Docking state are connected not only physical connection, but also communication connection in an organism. By the faster communication connections the robots hand over informations about it's successfully docked, gets its organism identity number and requests for permission to recruit or disassembly. Due the

physical and communication connection a robot can share events with whole organism and moves with an organism.

From the organism mode robots can return to Swarm mode just through Disassembly state. Another way is opposite and the way is from Docking state to InOrganism state. Sometime the literature evokes that there are two ways how to get from Swarm mode to Organism mode. The second way is from Flocking state to InOrganism state [17], but this way depends on our viewpoint and if we starting our task with all robots moving or the Seed is static from the beginning. Note that more Seed robots can exist at the same time in the whole system. Every organism has only one Seed robot. The Seed robots can be chosen by a certain conditions. For example a condition can be a high enough wall, or different obstacle or finding a electrical plug.

The next topic in this algorithm, aside of the internal states, are the messages by which the robots communicate with each other. Different messages are hand down by robots in an organism and different in swarm mode and different between this group of robots. In our task we don't need to use messages between robots in Swarm mode. The most frequently used communication is between robot in Recruitment state, recruiting robot and the robots in Flocking state. Here is the list of messages, which are normally used:

- *MSG-Recruitment* is message which indicates that the recruit process has started. This message is emitted to all robots by the robot in Recruit state and doesn't finish until a robot fulfil the given conditions as distance from the robot, which emits the recruit signal. And needs to be under certain angle from the emitting robot. The distance and angle are on consideration and can resume in different behaviour.

- *MSG-InRange* is transmitted by the robot in LocateBeacon state. This message is used just once and is given to all robots in Flocking state to inform, that MSG-Recruitment was stopped and so recruiting process has started. Now the robots in Flocking state can't go to LocateBeacon state.

- *MSG-Expelling* is a special signal broadcasted by the robot in Alignment state, to expel all other robots, which would be otherwise in its way and can cause a collision. This message reduces the interference between the robot in Alignment state and other robots to speed up the process of forming an organism.

- *MSG-DockingReady* is sent by the robot in the Docking state, when the docking units are fully in position to the recruiting robot. The robot

in Recruitment state stops to emit the beacon signal and starts to lock the docking units.

- *MSG-UnDocked* sends the robot in Disassembly state, when the undocking procedure is fully completed. The robot which was previously docked with the undocked robot receives this message.

Within organism, the communication can be implemented through a bus. It can be used to an undocking event or to the decision which robot should continue to emit recruit signal, which robot will get Token. It also takes care about handover the plan between the robots in the organism, the number of robots in the organism and other helpful information. For example, a notification when new robot has joined the organism can be sent to other members of the organism. This message is sent by a robot in Recruitment state and it is propagated by every neighbourhood robot in the organism. It can be also used to trigger the transition between InOrganism, Disassembly and Recruitment states as Token.

We have to take attention to competition between the robots, if they catch the emitting signal at the same time from the same source, they both fail in recruiting and can end up blocking each other. We can avoid this situation by sequential sending MSG-Recruitment to the robots and also controlling the execution of the conditions sequentially, so the first robot, which meets the condition is taken. If we're connecting more robots at same time, the situation might be saved by hardware solution, which is emitting signals via different channels at different interval, so our messages wouldn't be misspelled. Yet we have take care of collisions between the recruiting robots, we can either implement a react algorithm or gives priorities or plan the docking side of robots, which aren't in the danger of interference.

There are different algorithms for docking or algorithm building as for example mathematical modelling, exactly probabilistic model, where depend on the approach, it can be microscopic or macroscopic [19]. Another algorithm is Adaptation mechanism [18]. And more, very interesting algorithm, but we will take look on this in individual chapter.

Chapter 3

Experiments

In both experiments we used Java. The test was set on desktop computer, which has operating system Window 7 Home Premium, with Processor Intel(R) Core(TM) i5-3550 CPU with frequency 3.30GHz, number of cores is 4 and has 64-bit computing. The program can be seen in Fig. 3.1.

Every robot is in one of these group and every robot has six statistical data. It's robot's information, here are coordination, angle, identification number, identification number in organism of the robot. Then next distance the robot travelled overall, distance the robot travelled in Flocking state and distance the robot travelled in recruiting state.

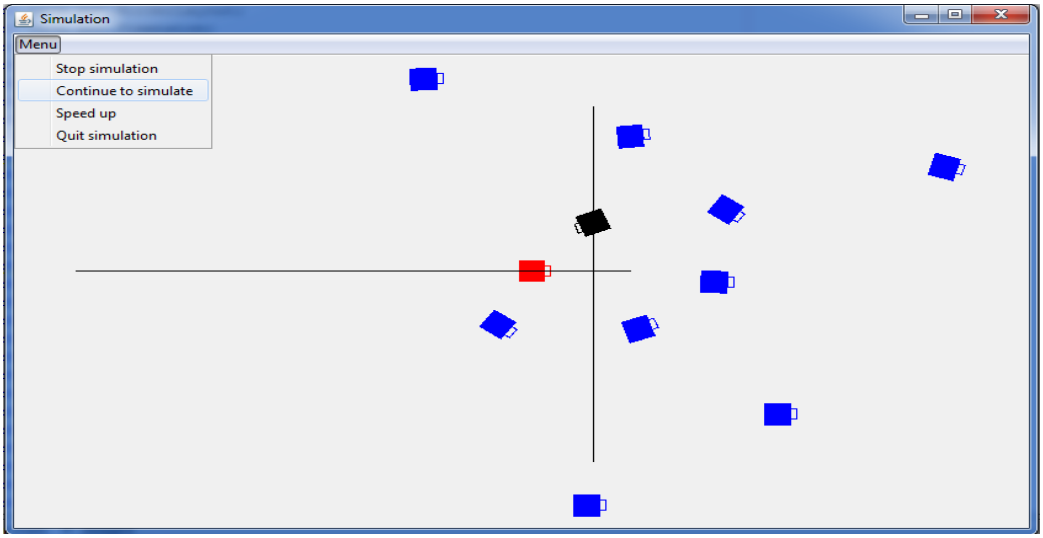


Figure 3.1: Screen shot of our Simularot with pop up menu.

Then next three data are times. Time which robot was in Flocking state and time which robot was in recruiting. The program doesn't count time in

organism, when they are static. The time is in seconds. Only one text file is different, which contains statistic about organism as a whole. In this file is five data, first line is the name of the made cluster, second is number of recruit's process, that is summation of successful and unsuccessful recruit's tries, third and fourth are summations of times, which robots spend in Flocking and Recruit states, respectively. And the last data is the overall time, which took the execute the task.

The program is also saving screenshots from important moments. Few of these screenshots you can see in Fig. 3.6 – 3.9. The screenshots are taken in an important moments as at the beginning of recruit process, at its end, when the recruiting robot was successfully docked or when robot crushed in recruiting process. All these moments are saved as .jpg files in this folder. As you can see on the pictures, there is always a black cross in the arena, which means where exactly is the given goal of the recruiting robot, from where the robot will be docked.

3.1 Distributed Autonomous Morphogenesis

3.1.1 Implementation

In this section we take a look at our implementation of the DAM method. In the beginning, we should say, that we will not use all states as were discussed in Theory chapter in section Stochastic algorithm for organism building. The reason is that the docking and alignment are rather hardware problems (concern of docking unit and sensors). From organism forming point of view, these technical problems are not important. In addition the problems in these states are hardly simulated with.

Our algorithm leads the recruiting robot to the goal position, which is in distance of our one-robot size in front of its docking size. In real word our robot would have 10 cm (∼3.9 inch). From this distance in our virtual environment we turn the robot with its docking side to side of the robot in Recruitment state to which the robot should be connected and then connect the recruiting robot. We're recruiting one robot at a time. The robot in Recruit state prefers to recruit robot with smaller ID number, if it has more robots on choice.

We also made simple Recovery state, which has two possibilities. If the robot gets there from Flocking state, he changes direction in time as the protection from collision. Collisions are dangerous because they can damaged robot or lock it. The second possibility is that the robot gets into the Recover state when it's in recruiting process(now only LocateBeacon state). In this

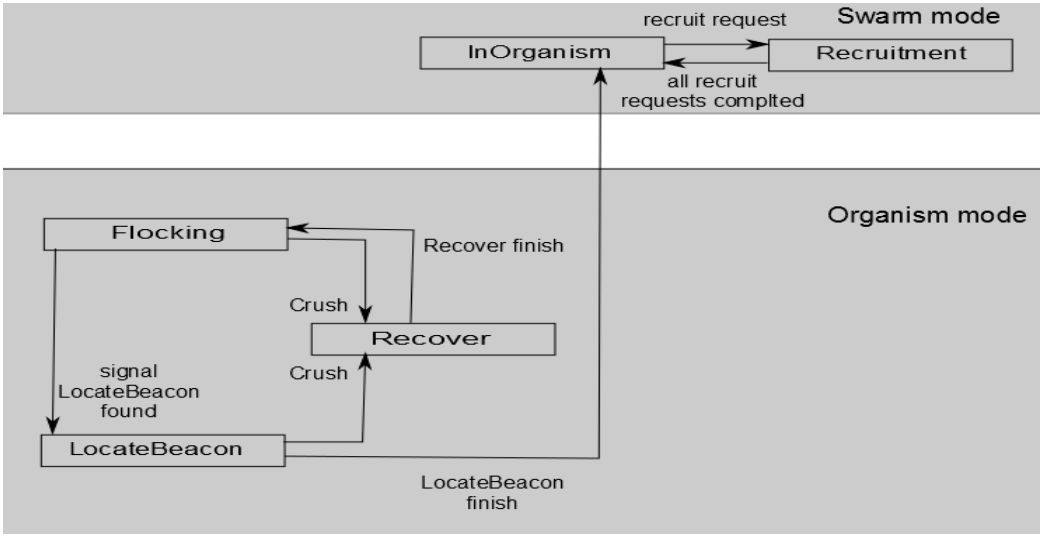


Figure 3.2: "Edited state" of DAM algorithm.

case we restart the robot, and change its state to the Flocking state. We can see the "edited state" of DAM algorithm in Fig. 3.2.

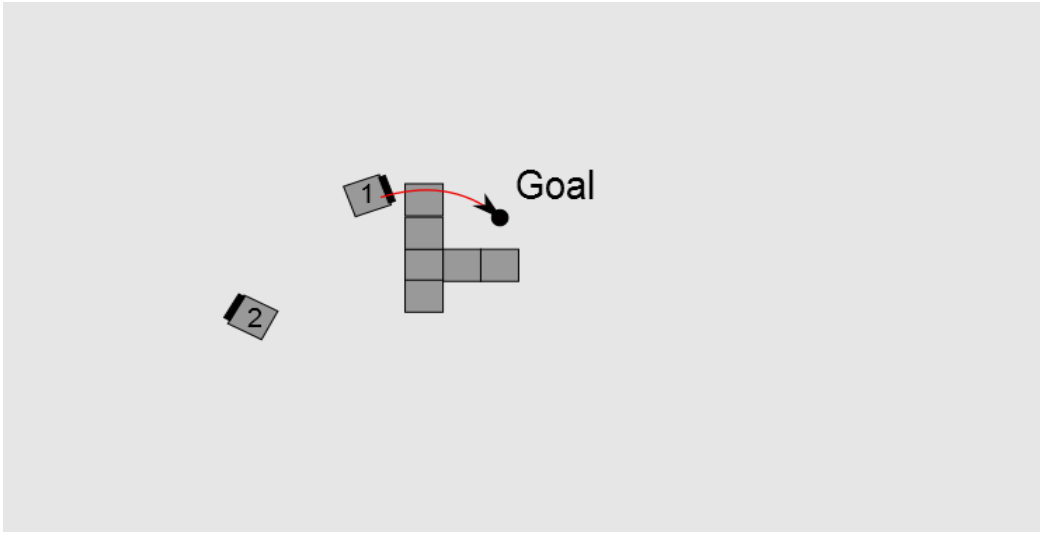


Figure 3.3: Lock of an robot with identity number 1 due the organism. It can reach the Goal position, so the robot will be restarted.

Algorithm then waits ∼3 seconds and then starts to emit new signal/message to recruit a new robot again. This delay helps the algorithm to repeat same situation and so lock itself in the situation as is shown in Fig. 3.3. The distance on which will robots react on each other is important here. In our

algorithm, we set it on 2.5 size of the robot. The distance may look too far, but if the robots are in the angle, in which their edge are the closest to each other, this distance is smaller, because we compute the coordination from the center of the robot. Example is in Fig. 3.4

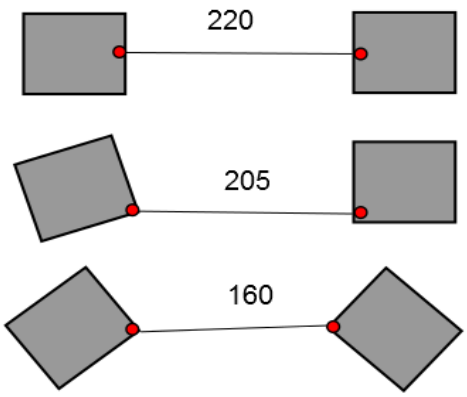


Figure 3.4: Presentation of how the turning of two robots can change distance between them. Red dots are to closest points to each other. The unit are in pixels.

To control motion of the robot, following differential-drive model is used:

$$\dot{x} = \frac{r}{l}(v_1 + v_2)\cos\phi, \tag{3.1}$$

$$\dot{y} = \frac{r}{l}(v_1 + v_2)\sin\phi, \tag{3.2}$$

$$\dot{\phi} = \frac{r}{l}(v_1 - v_2), \tag{3.3}$$

where r is the radius of the wheels and l is distance of the wheels. In our program both constants are 1. The position of the robot is (x,y) and it's rotation is ϕ. The three variables make the state vector $s = (x, y, \phi)$. Input signals v_1 and v_2 are velocities of the wheels of a robot.

To obtain a new configuration $s(k+1)$ after a control inputs are applied to the robot at state $s(k)$, the following numerical integration is used:

$$s(k+1) = s(k) + s(\Delta k) \cdot \Delta t, \tag{3.4}$$

where k is the number of step, $s(k)$ is a actual state, $s(\Delta k)$ is the derivation in point k, Δt is the time between these two steps. We can see the scheme of

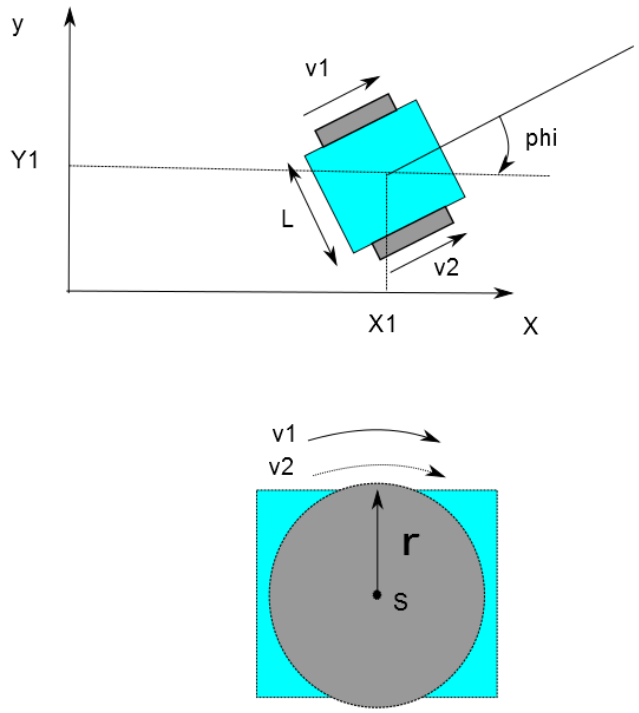


Figure 3.5: Scheme of the robot with all variables and constants.

the robot in Fig. 3.5 We implemented different colors of robots for Flocking state (blue), recruiting (black), Recruit state (red) and in organism (green). We can see starting and last step of our simulator in Fig. 3.6 – 3.9. with notification.

Due that we're using one thread for our program, if we don't count the treads that take care about graphic of the program, our messages are sending sequentially. It would seems as disadvantage due the speed, but in fact this saving us from the problem, that more robots get the recruit message at the same time.

We use two conditions for transfer from Flocking state to LocateBeacon state:

- The first is clear, our robot needs to be in a certain distance from a robot in Recruit state. This is caused by limited range of sensors. This distance we set adequately about 7 times size of our robot. This distance doesn't have to be good for other types of robots.

- The second condition is angle, by which the robot in Flocking state can get to the LocateBeacon state. The flocking robot has to be in the

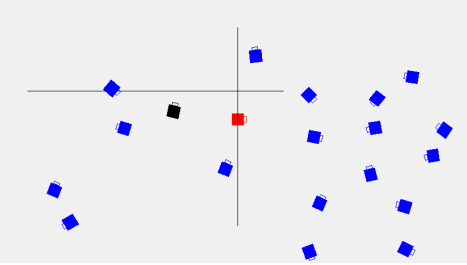


Figure 3.6: The start of a recruiting. Here the red is also Seed.

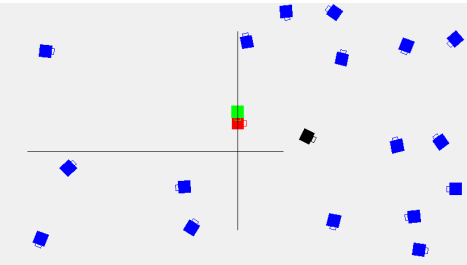


Figure 3.7: First successfully docked robot, Token stays by the Seed robot, it will continue to emit recruit signal.

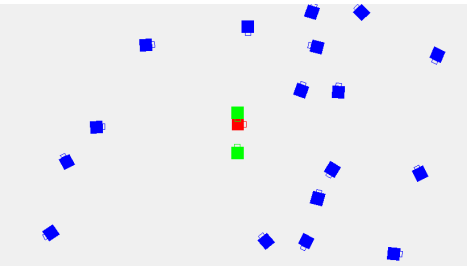


Figure 3.8: Successfully docking of the second robot, practically already in organism.

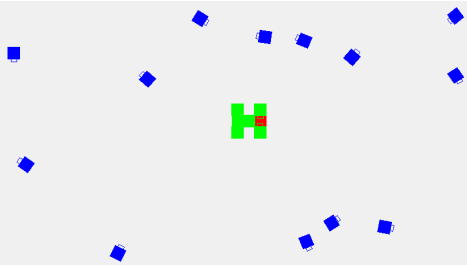


Figure 3.9: The end, robots successfully build a H-shape organism.

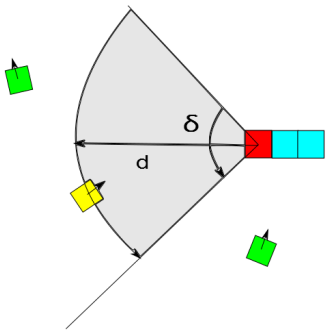


Figure 3.10: Area in which a robot can obtain recruiting message.

sector, which is made of set $< angle - \delta, angle + \delta >$, where $angle$ is angle of robot's side, where we want connect the robot, in Recruitment state and δ is the choice made of us. We chose the δ as $\pm$ 90 degrees.

The robot in the recruitment state asks for connecting of a robot, which is in front of the given side and under a certain distance limit. By this we made something as half-circle. We can see the example in Fig. 3.10. This condition is here to forbid the collision of the recruiting robot and the recruiter. This is our set, in some cases with a lot of robots, can be used a smaller sector, an smaller angle for recruiting. Then it can help with more complex organisms, for instance with more legs, which are close one to another.

The recruiting robot has few phases in our program, although it is only in LocateBeacon state. Firstly the robot is turning around until its front side is turned approximately to the robot in Recruitment state. Then the robot starts its way to the goal, which is in the front of docking side of the robot in Recruitment state in certain distance. The distance we chose big enough so the robot has a manipulative space, but small enough for the finish moving Alignment state. The move to the goal is controlled by the state-space equations again and the velocities aren't random any more. They are controlled by a proportional regulator:

$$\begin{aligned} \dot{P}_x &= (P_x - x)\cos\phi + (P_y - y)\sin\phi, \\ \dot{P}_y &= -(P_x - x)\sin\phi + (P_y - y)\cos\phi, \end{aligned} \tag{3.5}$$

where P_x and P_y are goal's coordination, ϕ is angle of the robot, x and y is actual robot position. Description of these equation we can see in Fig. 3.11. The derivation of $\dot{P}_x$ and $\dot{P}_y$ are used to computation of the two velocities

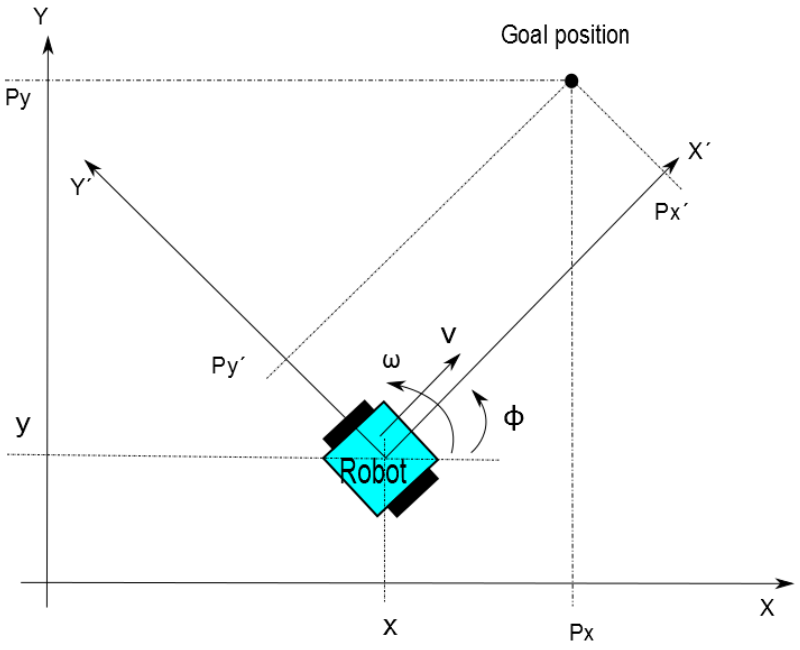


Figure 3.11: Description of the state-space equations navigate a robot to the goal in the space.

specify here:

$$v = 0.5\dot{P}_x, \tag{3.6}$$
$$\omega = 0.9\dot{P}_y. \tag{3.7}$$

Where v is the translation velocity of the robot and ω is angular velocity. The control velocities v_1, v_2 are computed as:

$$v_1 = \frac{v+\omega}{2}, \tag{3.8}$$
$$v_2 = \frac{v-\omega}{2}, \tag{3.9}$$

This velocities are used in the Eq. 3.1– 3.3. as usually. Now we have so called navigation equitation, which gets the robot to the goal (P_x,P_y).

This navigation equations get us to the certain distance from the goal's coordination. Then the robot would circle about the position in a spiral and starts to radically slow down until it would achieve zero velocity. With the knowledge of this problem, we should stop the movement driven by the navigation equations (we will call it Navigation mode) when the robot cross over

certain distance from its goal and start a different process. This distance should be large enough to keep the whole process of recruiting from redundant deceleration, which would make the process non-effective. Also this small distance can evoke a collision, when the robot starts make the spiral movements around the goal. On the other side, if we make the distance too large, we would also make the whole process slower, because in Navigation mode is the robot the fastest as in speed so in time. We tried make the distance optimal, which we found out experimentally. Our optimal distance is 2.5 size of robot.

Now comes the last phase after the Navigation mode. This mode makes a turning until our robot reached 180 or 0/360 degrees, depending where the goal is. Afterwards, the robot makes analogically the same process, only the degrees are 90 or 270. Now when the recruiting robot in on the goal's position, our robot with the help of knowledge about the robot in Recruitment state and its side and with help of trigonometric functions, it will turn the recruiting robot's front side toward the docking side of the robot in Recruitment state. After the turning is finished, the robot connects to the organism. If the recruit robot has another side to dock, it stays in the Recruit state, otherwise the edited plan gets the new robot, if it has a require for a docking. The plan works here as we describe in Theory, just it contains object, which has in itself coded information about the side and the robot in Recruitment state. When there is no other requirement for docking a robot, we reached our task for an certain organism. Here our task ends.

3.1.2 Pseudocode

In this small section we will show a few of the important and interesting functions of our program. One of them is the *recruiting process*, following code is repeated along the whole existence of the running program and basically works like this:

```
if(is in field && isn't in Organism && isn't end ){
 if(is recruiting){
  if(is collision){
   make_deafult_values();
  }else{
   if(is first run){
    make_goal();
   }
   if(robot doesn't turned round to the goal){
     turning();
```

```
   }else{
     watch_distance_to_goal_
     if_is_crossed_turn_and_even_up_with_X_coordination();
   }
   if(robot turned round to the goal){
    if(isn't close enough to goal){
     navigation_mode;
    } else if (robot is already on X coordination of goal){
       turn_and_even_up_with_Y_coordination();
       if(is on the goal){
        turn_round_to_the_docking_side_with_front_side();
        if(the turning to docking side was done){
         connect_to_organism();
        }
       }
     }
   }
  }
 }else{
 move_normally();
 }
}
```

That is the core code of our program. The next interesting piece of code is method for *receiving* a message and inserting it to the box of messages to process. Again, we will show it simplified:

```
void receiving(Robot robot){

 if(robot has recruit message which isn't in the box){
  if(robot in Recruitment state doesn't find another robot to
  recruit and doesn't have to wait on delay after collision){
   add_the_message_to_the_box();
  }
 }
 if(robot has message about restart of recruit and not in box){
  if(the message isn't send){
   add_the_message_to_the_box();
  }
 }
```

```
 if(robot has message about Token and it isn't in the box){
  add_the_message_to_the_box();
 }
}
```

And the last thing we will show is *process* of that message by the robots:

```
void process(Robot robot){

 if(is a global message){
  if(it's not robot in organism
  and we can process the global message){
   robot.process_message(message);
  }
  if(we start recruiting process with a robot){
   restrict_global_messages;
  }
 }

 if(message is about Token and not end of task){
   if(robot is the one which is marked in message){
    robot_setHasToken(true);
    robot_setPlan(message.getPlan);
    robot_setID_in_organism(message.getID_in_Organism);
   }
 }

 if( is the message set for robot
 and is about restarting of recruit ){
   inform_recruit_robot_about_collision();
   prepare_the_recruit_rob_for_next_emit;
 }

}
```

3.2 A*

The goal of this algorithm is forming an organism from a swarm of robots. In contrast of DAM algorithm A* is deterministic algorithm. It acts as a central planner, which will plan every move for every robot until desired

organism is formed. Before the planning process starts, the robots have to be static.

3.2.1 Implementation

In our work we implemented A*. We did statistics in simplified space and used the best heuristic for compare it with the Distributed Autonomous Morphogenesis. In A*, every Cluster has attributes as f function and g function, robots from Cluster is composed of, and Cluster called as Ancestor (the one from which our cluster was expanded/evolve). Cluster are all the robots, which participate on building an organism. Each robot knows its past, and the possible future position. The robot knows also its position. The robot can move and turn and connects to other robots. The robot can expand (or gain) positions around it and checks if the position is occupied by another robot or an obstacle. Then we have a map, which represents our work space with robots and obstacles.

Here we implemented heuristic function as Euclidean distance, Euclidean distance squared, Manhattan distance, Dijkstra algorithm, Manhattan upgrade, which invoke the Equation 3.12 in chapter Theory. Then we implemented special heuristic functions for Cluster as euklid_sum, euklid_max, euklid_avg and euklid_w, which are all based on Euclidean distance. As we wrote in chapter Theory, these are specialized to cope with more robots. With approach of summation, maximal value, average value and weighting as in the given section in the chapter Theory. The formula for euklid_sum heuristic is following:

$$h = \sum_{k=1}^{m}\sum_{i=1}^{n}(\sqrt{|X_{gi} - X_{rk}|^2 + |Y_{gi} - Y_{rk}|^2}), \tag{3.10}$$

where X_{gi} and Y_{gi} are different goal positions of goal Cluster, X_r and Y_r is position on computing robot and n is number of robots in the Cluster. For euklid_avg we divide this result with number of robots in Cluster:

$$h = \frac{\sum_{k=1}^{m}\sum_{i=1}^{n}(\sqrt{|X_{gi} - X_{rk}|^2 + |Y_{gi} - Y_{rk}|^2})}{n}. \tag{3.11}$$

For euklid_max we are storing the values in the list and the we are using following formula:

$$d(a,b) = euclideandist(M_a, M_b) \tag{3.12}$$

$$h = \sum_{k=1}^{m} max_{a \in R} d(M_k, a), \tag{3.13}$$

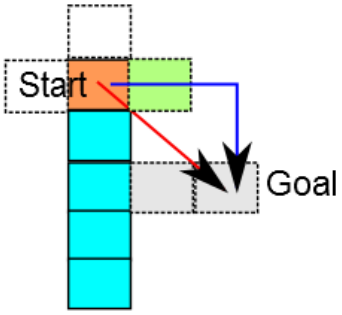


Figure 3.12: Comparison of heuristic functions in planning process. The orange is the chosen robot, grey fields are the goal positions, green fields are visited positions, blue are other robots. Manhattan distance is drawn with blue line and Euclidean distance drawn by red line. The robot is trying to moving along the line to the goal position.

where max is function for choosing maximal value from the list of distances, M is module and a and b are positions. How we discussed in Theory, it's pessimistic approach. The last formula is euklid_w is as classic Euclidean heuristic:

$$h = \sqrt{|X_{gi} - X_r|^2 + |Y_{gi} - Y_r|^2}, \tag{3.14}$$

where we only add to specific robots a constant, which will deceased them the probability of moving. Otherwise the central planner will not use the planning often.

3.2.2 Pseudocode

Here we describe closer a few of ours interesting or important functions used in A* algorithm. In the start we should just very generally take a look on A* as our basic process of the work:

1. Pick the best choice from Open list a give it to Close list, check if it's our goal, if yes stop.

2. Expand all nodes from our node, which we chose.

3. Check our expanded nodes. What is already in Close delete from expand list. What is in Open list either delete from expand list, if it's pricier, or, if node from Open list is pricier, delete this node from Open list.

4. What left in expand list we give in open list and we repeat this algorithm from step one.

First important function isSameCluster, which is a very busy function (has about 160 000 iterations). That's the reason that the function is so important for the global time of our algorithm. It compare two Clusters of robot and deciding if they are equal:

```
boolean isSameCluster (Cluster close/open, Cluster expands) {
 for (go through open/close list){
  for (go through expand list){
   if(coordination are same){
    counter++;
   }
  }

  if(coordination aren't there){
   break;
  }

 if(number of the same coordination is same as the size of the
 cluster)
 {
  return true;
 } else{
  return false;
 }
```

The break in that code is very important for the optimization and so for the time behaviour. Another important function is the basic expand for one robot, but for better imagination we will show it in Fig. 3.13 too. The pseudocode is brief:

```
ArrayList<Integer> expand (Map map)
{
```

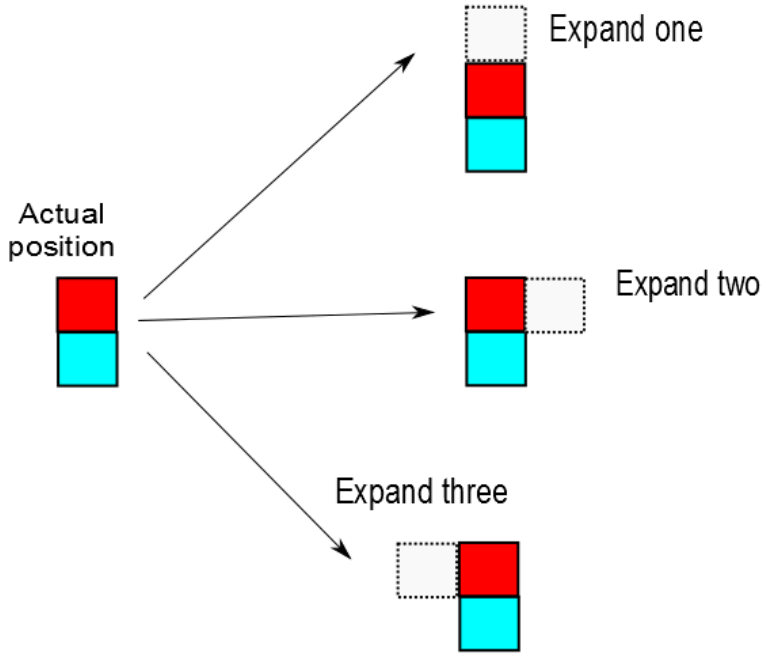


Figure 3.13: The red robot is the one which expanding positions. In the case of grid map the robot has three possibilities.

```
   if(margin conditions are fulfil){
     give_to_the_expand_list_given_coordination;
   }

  if( is given position occupied by obstacle or another robot)
  {
remove_the_expand_from_expand_list;
  }
 return expands;
}
```

Note that we're expanding the four sides, so the robot moves to the four directions, that can be consider as an approximation. Then there is very important function for finding a path, when we found the goal cluster, which gets us the format we can use as output:

```
LinkedList<Cluster> findAWay (LinkedList<Cluster> close,
 ArrayList<Robot> start_Cluster,
```

```
 ArrayList<Robot> actual_cluster)
 {
  if( is not actual_cluster same as start_Cluster ){
   for(go through Close list){
    if( we find a cluster same as our actual cluster){
     give_it_to_our_resulting_way_list;
     // recursive calling
     findAWay(Close, startCluster, actual.getParent);
     return way;
    }
   }
  }
 shift_the_nodes_in_right_sequence();
 return way;
 }
```

Last function I would like to show is the general heuristic function:

```
Cluster heuristic
(ArrayList<Integer> expands,
ArrayList<Robot> goal,
ArrayList<Robot> original){
if(heuristic is calling first time)
{
g=0;
}else{
g=value_of_original;
}
make_New_Clusters_from_expands;
g=g+price_of_step;
 for(expanded clusters){
  for(robots in each cluster){
   for(robots in goal cluster){
    h=some_heuristic_for_each_robot;
    f=h;
    add_f_to_list;
   }
  //here can be whatever
  //function min,
  //max, avg or so on..
  add_minimum_of_f_to_other_list;
```

```
  //now we have min distance given
  // robot to nearest goal
  Tot=sum_of_min_dist_whole_cluster;
  Tot=Total+g;
  this_cluster_to_list;
  }
 }
}
```

3.3 Results

3.3.1 Distributed Autonomous Morphogenesis

We have tested the algorithm with different organisms for statistics as distance, duration, collisions in different ways. We tested them on different number of robots, placing of an organisms and non-random placing of initial positions of the robots. We created organism as Cross, which representing us non-symmetrical organism, where east and south arms are longer by a one robot. Then we created Snake organism (as a line shape), Tshape, which looks exactly as T letter, Hshape which represent H letter, so same for Ishape (Hshape turned 90 degrees to right). Note that every of these shapes have either different number of arms or the orientation. They also vary in the number of robots, from which is the organism composed. Hshape, Ishape, Cross has composed from 7 robots, Snake and Tshape from 6. Hshape is built from left to right as Snake, Ishape from top to bottom as Tshape, Cross from the middle to the all size equally. All the shapes we can see in Fig 3.14– 3.18. Our maximum number of robots in the map was 30. Bigger number would slow down the start of our program. The minimum number was 10. When we tested the organism on a position, we meant testing the robot near the map's border. We placed them so the robots have enough of space for their movement, when they're trying to connect to an organism. The distance was about 3 times of the robot's size. When we placed the robots non-randomly, we placed them always at the same position, 10 robots to the vertical two lines. We will use statistics as maximum, minimum and average because in our case, when average is based on large amount of data, we can evaluate it as trustworthy.

Now we will make approach to the first statistic we have made. That is average distance of a robot with dependence on number of robots in the map. The units are meters. Based on the size of our robot. In the Fig 3.19 we can see the statistics, we can notice of the minimum value in all cases

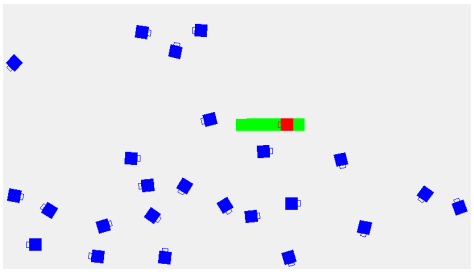


Figure 3.14: Snake shape.

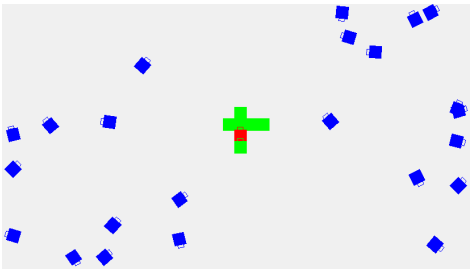


Figure 3.15: Cross shape.

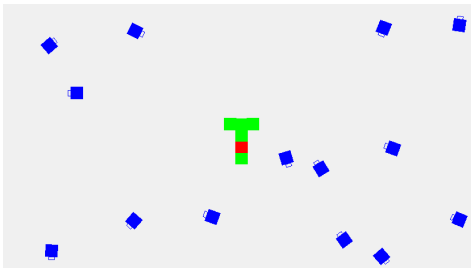


Figure 3.16: Tshape shape.

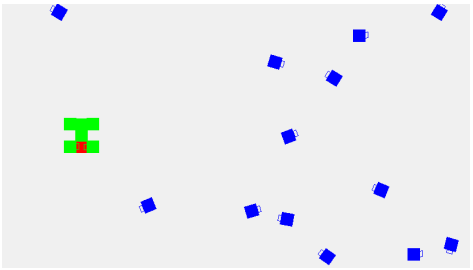


Figure 3.17: Ishape shape.

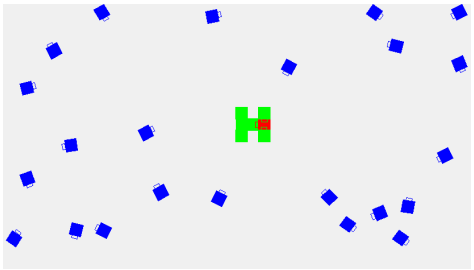


Figure 3.18: Hshape shape.

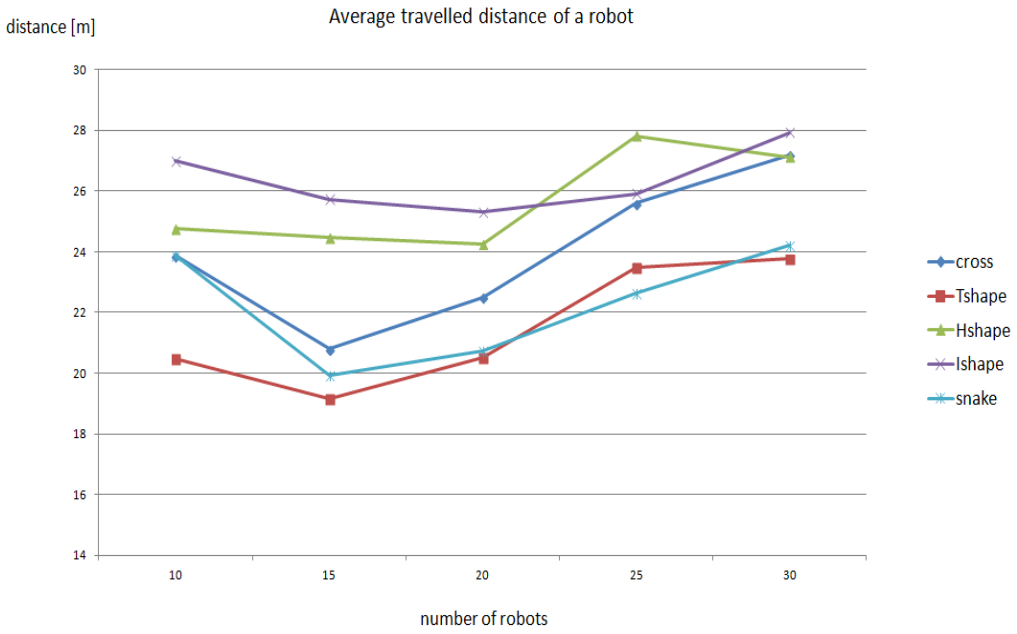


Figure 3.19: Average travelled distance of a robot controlled by the DAM algorithm in dependence on number of robots. Note that minimum is between 15 - 20 robots.

with 15 or 20 robots. And the high values in the end. We can also notice distinct behaviour with minimum number of robots depends on the type of organism.

Maximal values of distances were reached always by the biggest number of the robots in the map, apart of two exceptions, which were Cross and Snake shapes. In the situation of Cross the biggest distance was when Seed robot was in the left top corner with twenty robots in the map. In the case of Snake organism the maximum was reached when we placed the Seed robot to the top middle placing with twenty robots too. We can see it in Fig. 3.20 The minimum values have very distinct conditions. We marked them in the Table 3.1 by shortcuts as rb, l, t, b as right-bottom, left, top, bottom, retrospectively. The result of minimum distances are so given by the presence of the border near the organism, because the robot has distance restriction for the recruit process. Maximal values of distances are mostly given by the collisions, when in Recover state the robot for a while accelerate its speed and slows down the whole process when the other robots still were moving. Snake and Cross only show us the influence in which way we're building the organisms, but otherwise the causality is the same.

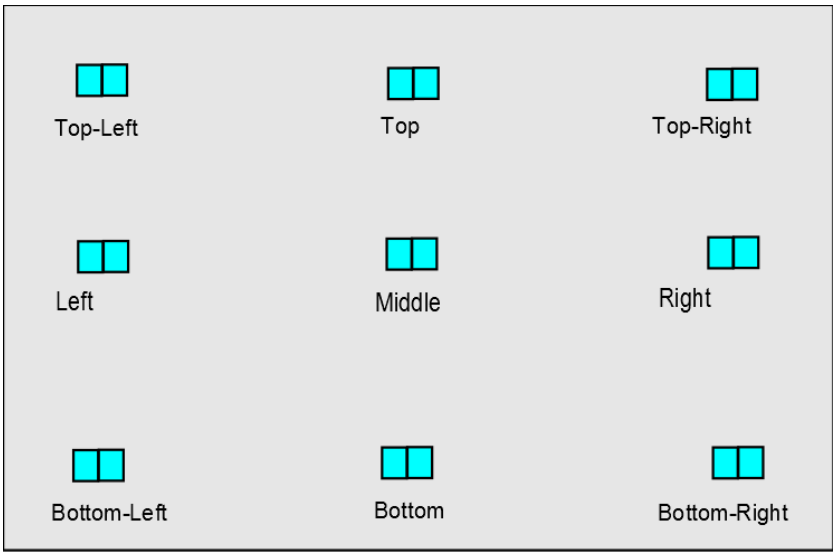


Figure 3.20: Description of positions of an organism in DAM algorithm.

Type of shape	**Max. distance [m]**	**Min distance [m]**
Hshape	$104,11$	$0,175$ (t)
Cross	$120,14$	$0,202$ (rb)
Tshape	$96,93$	$0,204$ (l)
Snake	$103,62$	$0,206$ (t)
Ishape	$113,76$	$0,212$ (b)

Table 3.1: Maximal and minimal distances of a robot with the shape, sorted by minimum. Used DAM algorithm.

In corresponding to this we also have made statistic for average time, needed to form an organism with certain number of robots, Fig 3.21. We can confirm that average time isn't dependent on average distance. If is important for us the speed of forming an organism, we should at least take 3 times number of the robots needed to form an organism, so we could approach to the minimum time needed to form the organism.

In our testing the program stopped after it reached of 400 seconds. It was used as protection from freezing in certain situation and furthermore we're looking for the best performances of the tasks, so we want show only the minimal times of fulfil the tasks in the Table 3.2. We can see a dependence on the best time performance in number of the robots and in complexity of the organism, so the Tshape and Snake had the best results. The best values of minims were given always around or exactly a number of robots given by rule Five. Rule Five says that ideal number of robots in the map is $5n$, where

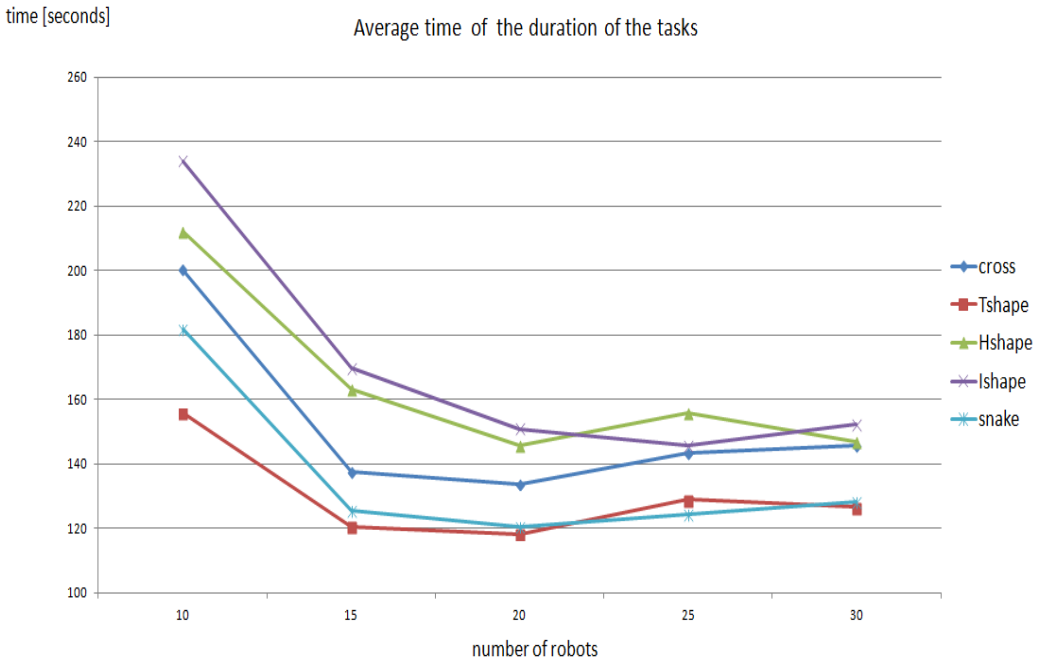


Figure 3.21: Average time that takes the task with the number of the robots in the map, units are in seconds. Used DAM algorithm.

n is the number of robots. In contrast Ishape was very time-good when the Seed robot started at the bottom too.

Type of shape	**Tshape**	**Snake**	**Ishape**	**Cross**	**Hshape**
Time in seconds	70	82	92	94	98

Table 3.2: The best time's performances for DAM algorithm in seconds, sorted from the left to the right from the lowest value

Another result is the number of trials to recruit a robot to an organism. We have made the statistics for every building of an organism and then make an average. Then we subtract the minimum needed number of recruit trials from these averages and gain the result, which is shown in Fig. 3.22. It shows how far from the ideal process of formulation of an organism are these situation. Ideal case would reached the zero value, so it would has done only the necessary recruits. This approach is independent on the number of the robots in an organism. It shows us the average number of collisions in recruit process, which are redundant. And independently on if the robot gets to the organism or not. The graph shows us how often we have to use the transition between Recover state and LocateBeacon state.

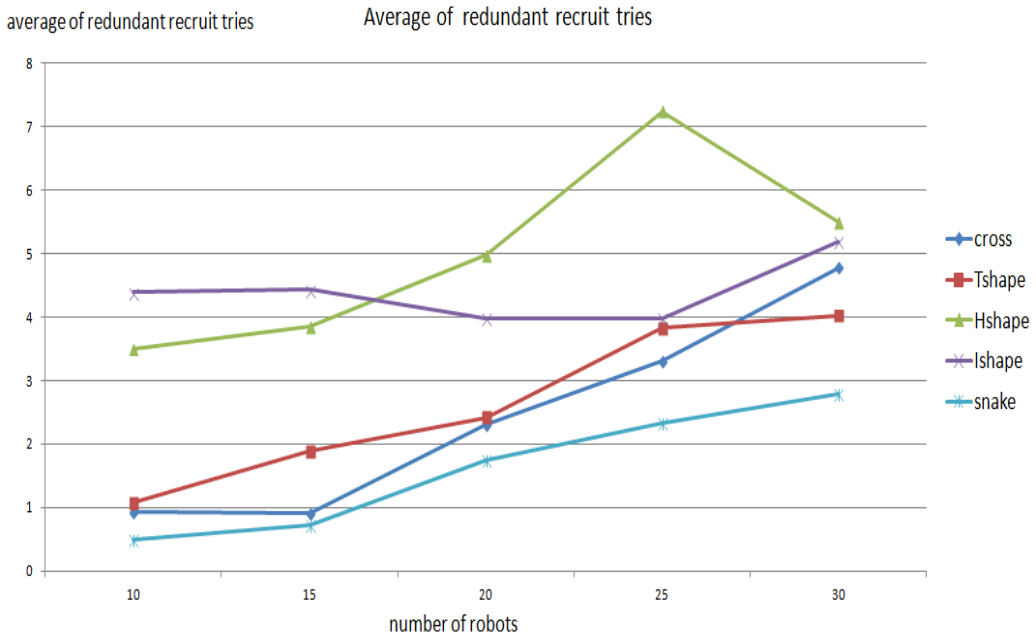


Figure 3.22: Average redundant number of recruits process in the DAM algorithm in dependence on number of the robots in the map.

The maximal values of the redundant recruits trials differ, the value depends on complexity of the organism. For example Snake and Cross have maximal values in distinct situation from 10 to 40 tries. More complex organisms as Hshape or Ishape have the values from 30 to 50. Tshape is between these two groups, it has values from 20 to 40. Dependence of the number of trials on the complexity of the shapes. We figure out the dependence on initial placing of the robots. We made tests for our shapes with ten robots, which were randomly placed and which were always at the same position. Every of the shapes with random and non-random placing we tested 40 times, we can see the placing in Fig. 3.23. We took that data and made an average from them.

Tasks with non-random placed robots of this amount were always faster to find a solution than the ones with randomly placed robot, in our cases, because we placed the robot adequately to the task. You can see it in Table 3.3. And the robots also have better result for distances, where the average distance of a robot was smaller than with a randomly placed robots, except Tshape where the random placed robots was a little better. This proofs the dependence of time and distance on the initial positions of robots.

Last thing we would like to mention is dependence on initial position of

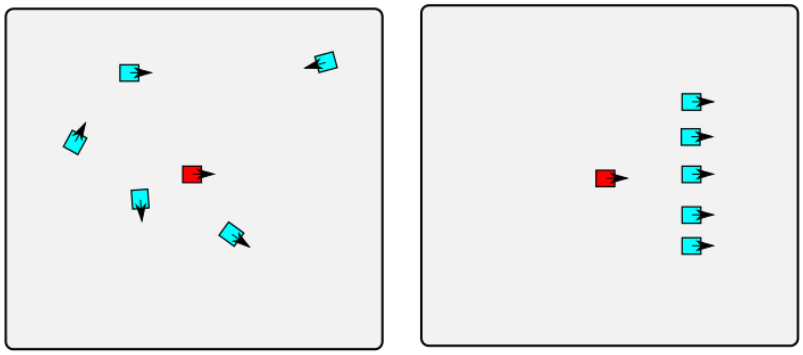


Figure 3.23: Random and non-random placing in DAM algorithm, the red robot is Seed.

Initial positions	**Tshape**	**Hshape**	**Ishape**	**Snake**	**Cross**
non-random	146	176	194	134	172
random	156	212	234	182	200

Table 3.3: Time performance of the shapes with initial random or non-random placing, unit are in seconds. Used DAM algorithm.

Seed robot. Until now all the Seed robots were in the middle of the map. We already noticed in minimal values of distances of a robot that dependence. The results indicate, that the initial positions of the robots influence speed of the organism formation. In the Table 3.5 we are showing an organism, where is the effect very significant. Cross perfectly fits as an example, due the asymmetry. We can find the effect is in all the organism.

Type of statistic	**Right**	**Left**	**Right-bottom**	**Left-top**
distance average [m]	23, 68	24, 57	28, 8	31, 18
time average [s]	138	145	169	179
time minimum [s]	101	97	110	101

Table 3.4: Cross performance in DAM algorithm on different positions ordered by best position from left to right. Note that the minimal time's value are close to each other, but averages are distinct. Tested with 10 robots.

3.3.2 A*

At the beginning we will show comparison between different heuristic functions we have made for forming of an organism and the reason of usage Euclidean distance in our experiments. In Table 3.5 we can see comparison the heuristic function in dependence on time. In one column we can see reconfiguration of an organism and in the second column also transposition and forming the desired organism on different position then was initial position. We can see stable results for Manhattan and Euclidean distance. The DNF means that the process Did Not Finish, so it means the process cross the limit of five minutes. Notice this is only two cases of situation, but for our case the main comparisons.

Type of heuristic	**reconfiguration** [s]	**transposition + formation** [s]
Euclidean distance	$0,27$	$0,15$
Euclidean distance squared	$0,10$	0.16
Manhattan distance	$0,09$	$0,15$
Manhattan distance upgrade	$0,09$	$0,14$
Euclid_ sum	$1,26$	DNF
Euclid_ avg	$0,75$	DNF
Euclid_ max	$9,07$	DNF
Euclid_ w	$3,7$	DNF
Dijkstra	DNF	DNF

Table 3.5: Table comparing different A* heuristic function with dependence on time. DNF means Did Not Finished. Heuristic function of A* algorithm.

We made experiments with five different shapes, same as we did with stochastic algorithm DAM. The organism differ here in the size, they composed from different number of robots. These are tested on two initial positions, which are as vertical and horizontal line. The statistics differ here, but even so we can still compare the results with different shapes and then compare A* results with DAM.

The first statistics we have made is the time dependence on the number of robots from which is an organism composed. In this program the statistics are constant, so they are not average values. In the case of A*, when we

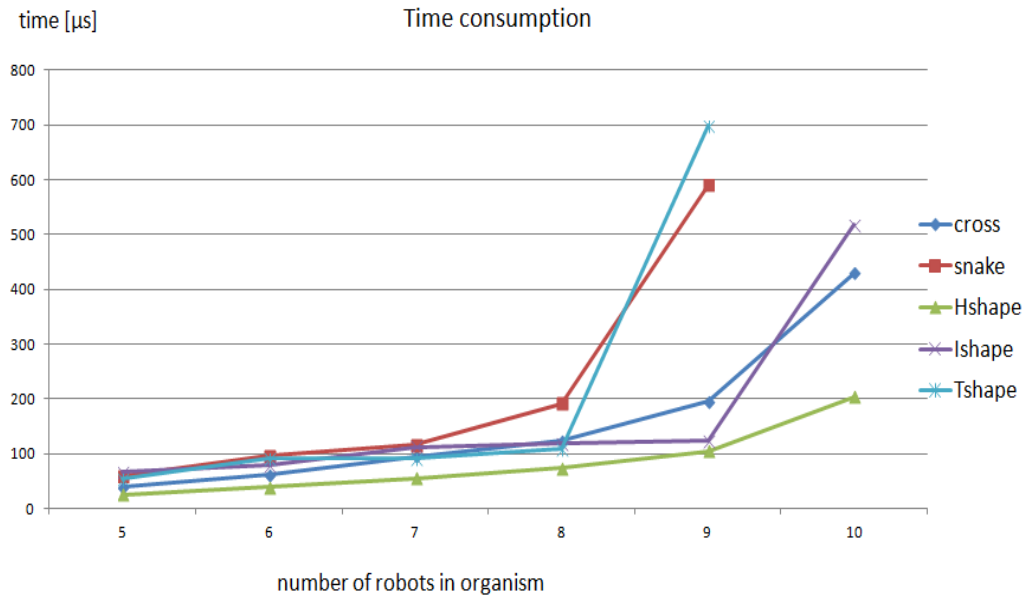


Figure 3.24: Time consumption in dependence on number of robots, unit are in micro seconds. Used A* algorithm.

move one robot at a time, the time would be the same. In this statistics, we cut off two values, the two greatest ones, because it would damage our graph. These values are for Tshape, where is the time value in μs 26326. And the second is for Snake, where this the time value is 3186 μs. These values are incomparable with others. Both are when is the organism biggest. We can confirm, that the weakness of A* is for organism with larger number of robots. We can see it in Fig. 3.24. The time depends on the initial positions of the robot, that is reason of the good performance of Hshape, because from the initial position only minimal number of steps is needed to achieve the desired organism. From this we can also indicate, that A* is more effective in the reconfiguration, but in contrast of DAM, its effectiveness is decreasing in cases, where the robots have to move more. A* is sensitive on movement to number of robots in an organism as its time consumption increases exponentially.

Now we can compare the time dependence on initial positions of Cluster in our algorithm as we did in DAM. We did the test with ten robots. Vertical placement means the robot are ordered as line from top to bottom of the map. Horizontal is analogical. We can see both in Fig. **??**. We will present it as Table 3.6.

There is advantage of DAM, which needs to have robots close to Seed robot. In A* if the initial positions are close, but the initial Cluster isn't similar to ending Cluster, it takes a lot of time. What is interesting about the Snake organism is that, if we move from horizontal position to vertical the planing takes different time that if we are moving from vertical position to horizontal. The reason would be that the one of Cluster wasn't placed exactly in the middle o the map as the second Cluster.

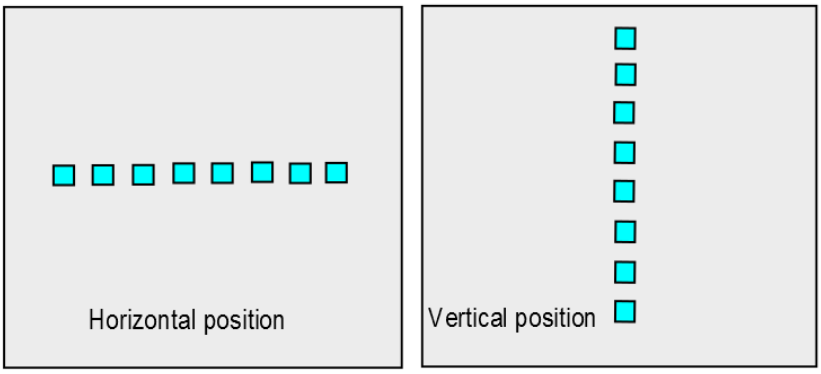


Figure 3.25: Horizontal and vertical initial positions in A* algorithm.

Initial positions	**Tshape**	**Hshape**	**Ishape**	**Snake**	**Cross**
vertical	701	424	166	2634	325
horizontal	26326	204	517	3186	429

Table 3.6: Time performance of the shapes with initial horizontal or vertical placing, unit are in μs. Used A* algorithm.

Except of time dependence we have also compared the price for using the given path, which would present the real effort to get from initial state of organism to the goal organism as is the distance in DAM algorithm. A* considers price for one step, which is price robot has to pay from on position to another, and it's 0.2. From this information we can get the number of the steps needed to achieve the goal. We made a graph of the dependence number of robots in Cluster on the price, Fig. 3.26. We can see, that A* is not only sensitive on initial position, but also on the goal (changing goal, same initial positions), for example in Tshape, when we remove one robot from the longest leg, it significantly changes the distance which the robot needed to travel. In contrast with DAM algorithm, the distance (price) is increasing with number of robots.

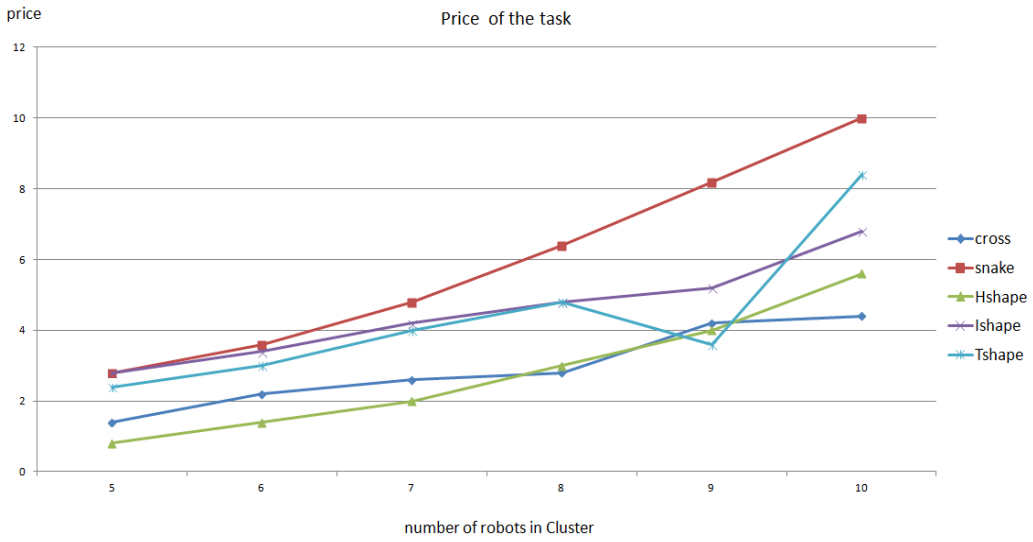


Figure 3.26: Price for the given task with dependence on number of robots in Cluster (in organism) in A* algorithm, the price doesn't have units.

The last we want to show is number of expansions. The number of expands is number of possibilities and so it can slow down the process. The number of the expands in space we show with dependence on number of robots in Fig. 3.27. We cut off last value in Tshape with ten robots, which was 20 533 expands, because it was incomparable with the other values. Ishape was the only one which decreased in one moment, exactly with nine robots. That is because we extremely changed the shape, in this case the Ishape looks as letter Z and that proofs again the sensitivity on the shape even in number of expands. Also we can see a similarity between DAM algorithm in redundant recruit and A* expands in space, both are exponentially-like increasing.

3.4 Conclusion

The results shows that in case of DAM algorithm the most influence on the forming of an organism in time and distance matter has number of robots in the map. Due that we we can radically change the sped of executing of forming an organism. The distance of a robot is dependent on the number of robot mainly in maximal values. In average value is distance of a robot more dependent on the shape of an organism. A* is also the most dependent on number of robots, even more, in time consumption. The difference is A*

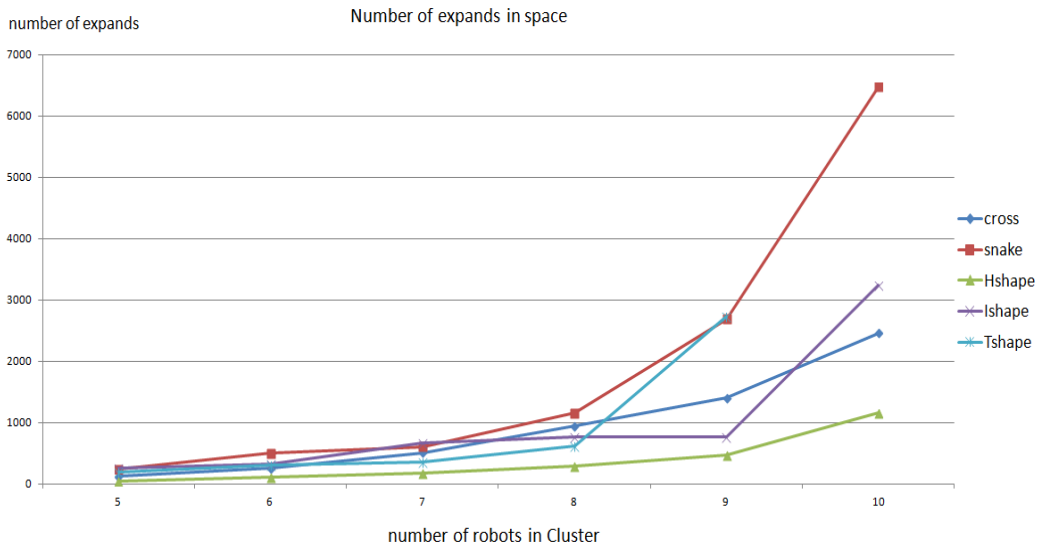


Figure 3.27: Number of expands in space with A* algorithm. Depend on number of the robots in Cluster.

is worse with growing number of robots. A* can meet big problems when it has to form an organism, where robots have to overcome bigger distance. So it's very sensitive on type of organism to form, more then DAM algorithm.

We can put up approach when is good to use the given algorithm. Performance of the A* algorithm is better for forming of small organisms. The performance is further increased if only few changes are needed to form a desired shape, i.e., where the modules are close the their goal positions. In contrast DAM algorithm exceed A* algorithm, when is needed to build bigger organism with more than six robots. DAM algorithm can exceed the A* event with less robots, if doesn't contains initial position. And then it depends on distance the goal organism from initial positions of the robots. The DAM algorithm is suitable for cases, where the initial position of the modules is far from the goal position.

Chapter 4

Conclusion

In our work we have reached to the following statements about effectiveness of both algorithms. One of them is that A* algorithm has very good results in case of time and expands with Cluster, which has less than six robots, but after crossing this number of robots, the trend started to be unfavourable to the A*, with exponentially-like function. In A* algorithm is also harder to set the condition so that the process would be the fastest possible, because it's very sensitive on initial conditions and goal's too. This constrictions are fulfil when the initial organism and goal organism cross a distinct limit of distance from themselves. A* is not good for forming an organism, in which the robots have to travel bigger distance from their initial positions.

The DAM algorithm has its weakness when it comes too little number of robots in the map. On the other hand, it gets better result with more robots in the map. More depends on the size of the goal's organism, because of the constrains of the map, but here we get good time performance with rule Five. The number is dependent on the size of our robot and the size of our map. This rule gives us the best rate between number of robots in the map and number of locks or collisions in the process of forming an organism. Due that we have enough of robots in the recruiting area, but not too many robots to cause slow down of the process, because of the bigger number of collisions. Good time performance doesn't mean good distance performance, that is better with lesser number of robots in the map then tell us rule Five, as for example 3 time of number of robots in the organism. Even here we have to take attention where we are building the organism, because in nearness of obstacles would slows down the whole process as did the boards in our case, if we look again on the statics with organism Snake placed in left-top part of the map in DAM algorithm. There would be interesting approach to test A* and DAM algorithms in the map with increasing number of obstacles and

comparing the results. Another advantage is that DAM algorithm isn't so sensitive on different organisms, that proof the results in time and distance performance, in contrast A* has very different results in time and distance performance depending on the shapes, they have all same trends but very different values.

We would also put up possible methods for upgrade DAM algorithm as for locking the process, which is evoked by sending messages from the robot in Recruiting state to robots in Flocking state sequentiality by their ID. The robot is then often more times transmitted from Flocking state to LocateBeacon though Recover state, but still is crushing. In that case we would make priorities for every robot and after every collision in LocateBeacon state we would decrease this priority. In A* algorithm we would also use Manhattan distance, but in other tests, we did in another work, we find out, that the result with Euclidean distance are very similar.

The comparison itself between A* and DAM algorithm was made successfully, but wouldn't be precise enough in some cases. We compared them on similar situations, but for A* we tested only planing process, but it isn't the problem, because the process itself in the time case is constant, which we can estimate from the simulator, where we tested DAM algorithm, if we take the same velocities of the robots as in that case. The problem would be the comparison on distances, because in A* we have them as constant without values. We have the trends of A* in distance mean, but it would depend on the transfer function from the cost to distance. In our map we could imagine, that the cost from one point to the second in A* is as 1/5 of the simulator map. Then we would get really comparable values and our result isn't affected.

Chapter 5

Appendix

5.1 Description of programs

We have used Java, which is objected-oriented, class-based program language, designed to have as few implementation dependencies as possible. This is what we need, because we want to run this application on whatever computer and whatever operating system it is needed. For this we used integrated development environment Netbeans version 7.3.1. with java development kit 1.7. Both application can be executed as java archive from command line with the given parameters. In both programs is implemented the help page, which user can open by parameter -h. The programs are treated on wrong number of parameters and give user a warning notification.

5.1.1 DAM

The program of the DAM algorithm has parameters as type of shape, which are defined in the program, number of robots and time out, or time until the process should end. If we insert zero as number of robots, it will behave along the rule, call it rule Five. This rule guarantee the appropriate number of robots for our organism, but not always the optimal number. In our program is also implemented a code for an input, which would presented the directory, where the user wants to save a statistic folder. We didn't add this possibility after all and we are creating the statistic folder in actual directory.

Every time our program is started up, it's created a new folder, which name is unique. It has similar name as temporary folder of operating system, but at the beginning is the name of the shape, which statistic are saved in it. Such a typical name is for instance "SNAKE15359956". In this folder there are text files, every containing a statistic information. They might be

separate to these four groups: statistic about created organism as a whole, statistic about robots, which don't affected the building of an organism, statistic about robots, which affected the building of an organism, but didn't get to the organism and finally robots, which end in the organism. The files containing these group of statistics are called "stattistic_Organism", identification number of the robot,"robotsCrushedRecruit" and "robotsInOrganism", respectively.

5.1.2 A*

In the command line user puts as parameters input a text file, a mode and a heuristic user wants to use. The mode means option by which we can chose from two possibilities presented by two numbers, either the output of the program will be path, which will be process graphically in command line as the zeros and numbers two or its output is coordination of whole cluster, juts clear numbers. The third choice is heuristic, which we implemented to our program and which will be used.

Output is always order as following, the first it's showed the start and the ending lay-out of robots. Then there are sequentially sorted steps of Cluster, steps of algorithm. And at the end of the outputs are showed statistics as total price we paid for the whole path of Cluster, number of expansion in whole space our algorithm has to make to achieve goal, total price we paid for travelling along the way, maximal travelled distance of one robot of Cluster, number of robots of Cluster, which don't move, number of needed steps for the Cluster to achieve the goal and what was the cost for one step, if it was constant.

5.2 Heap

We have implemented here so cold Max Heap and Min Heap. They are special dynamical lists, which are exactly called Binary Heap. They fit well to our case of A*. They are lists, to which we can insert objects,here important point is that they must be comparable, and after the insertion, the objects are always compared with the certain objects in the list,we will write later, how the objects, called parents, are determined. Here it's already important to divide the two approaches, sometime we want have the maximal value on the top of the tree or, else on the start of the list. And sometime we want have minimum value on the first position. That why is the first called Max Heap and the second Min Heap in our cases. Here are general steps for this algorithms:

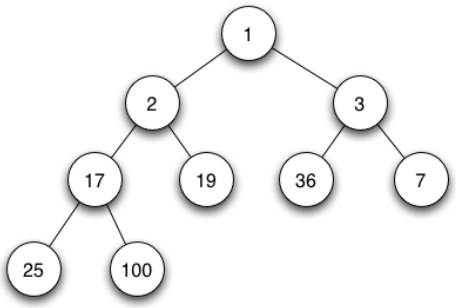


Figure 5.1: Example of Minimum binary heap.

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

Even the deletion of an element in the list must be special treated, we can't easily delete the given element, because when we delete an element, we would destroyed our tree structure. After a deletion we must check, if the elements are in the correct order. This method is known as bubble-down. Here is the algorithm of it:

1. Replace the root node of the heap with the last element on the last level.
2. Compare the new root (the one we took from last level) with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

This is theoretical description, which counting on that we are deleting the maximal or minimal value. Which we normally do, so in our program, because that's the advantage of the approach, but else for other values in the list it would be analogical. Another thing is how it's implemented as list in our source code. Here it's using a precise formula for the parents and its children. The formula for left children is following:

$$L = 2k + 1, \tag{5.1}$$

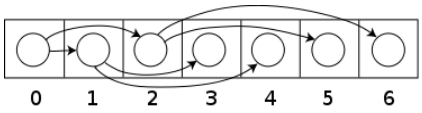


Figure 5.2: Example of Minimum binary heap in an array. Arrows show children.

and for right children it's following:

$$R = 2k + 2. \tag{5.2}$$

Where k is the index of a parent in the list. There is also formula for finding out the parent of a given children. We used it too, it's following:

$$P = \frac{k-1}{2}, \tag{5.3}$$

5.3 CD

The contain of the CD, which is appended to this work, is following:

- Electronic version of this work in pdf format as dip.pdf.
- Two source codes of A* and DAM. Sources code are in the file src.
- Acquired data from both programs are in the file Statistic.

Bibliography

[1] A. Behar A. Castano and P.M. Will. The conro modules for reconfigurable robots. IEEE/ASME Transactions on Mechatronics, 2002.

[2] M. Moll B. Salemi and W.-M. Shen. Superbot: A deployable, multifunctional, and modular self-reconfigurable robotic system. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006.

[3] Serge Kernbach et al. Heterogeneity for increasing performance and reliability of self-reconfigurable multi-robot organisms. workshop on Reconfigurable Modular Robotics, San Francisco, 2011.

[4] H. et al. Kurokawa. Self-reconfigurable modular robot m-tran: Distributed control and communication. international conference on Robot communication and coordination, RoboComm, 2007.

[5] G.C. Pettinaro F. Mondada, A. Guignard, I.W. Kwee, D. Floreano, J.L. Deneubourg, S. Nolfi, L.M. Gambardella, and M. Dorigo. Swarm-bot: A new distributed robotic concept. Autonomous Robots, 2004.

[6] Adwoa Gyimah-Brempong. Precision agriculture: Sustainable farming in the age of robotics, 2009.

[7] H. Li H. Wei, Y. Cai, D. Li, and T. Wangl. Sambot: A self-assembly modular robot for swarm robot. In IEEE International Conference on Robotics and Automation (ICRA), 2010.

[8] Larry Hardesty. Surprisingly simple scheme for self-assembling robots, 2013.

[9] Lutz Winkler Jens Liedke, Rene Matthias and Heinz Worn. The collective self-reconfigurable modular organism (cosmo). IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM), 2013.

[10] D.G. Duff M. Yim and K.D. Roufas. Polybot: a modular reconfigurable robot. Conference on Robotics and Automation, 2000.

[11] W.-M. Shen M. Yim, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G.S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. Robotics Automation Magazine, 2007.

[12] Jonathan Marcus. Robot warriors: Lethal machines coming of age, 2013.

[13] E.H. Ostergaard M.W. Jorgensen and H.H. Lund. Modular atron: modules for a self-reconfigurable robot. IEEE/ RSJ International Conference on Intelligent Robots and Systems, 2004.

[14] University of Pennsylvania. Robot boats rescue mission, 2013.

[15] H. Hosokai T. Fukuda, M. Buss and Y. Kawauchi. Cell structured robotic system cebot: control, planning and communication methods. Robotics and Autonomous Systems, 1991.

[16] A. Chan V. Zykov and H. Lipson. Molecubes: An open-source modular robotics kit. Self-Reconfigurable Robotics Workshop, 2007.

[17] Alan F.T. Winfield Wenguo Liu. Distributed autonomous morphogenesis in a self-assembling robotic system. Bristol Robotics Laboratory.

[18] Alan F.T. Winfield Wenguo Liu, Jin Sa, Jie Chen, and Lihua Dou. Towards energy optimization: Emergent task allocation in a swarm of foraging robots. Bristol Robotics Laboratory, 2007.

[19] Sin Sa Wenguo Liu, Alan F.T. Winfield. Modeling swarm robotic system: A case of study in collective foraging. Bristol Robotics Laboratory.