Maastricht University
Department of Knowledge Engineering

Project 1.2.3 Report

Self-Reconfiguring Modular Robots

Authors:
Andrew Gold (i6126154)
Roel Hacking (i6141415)
Zsolt Harsanyi (i6136031)
Marion Meyers (i6136828)
Sander Post (i6140311)
Borislav Slavchev (i6137301)

29 June 2017

Supervisors:
Jan Paredis
Pieter Collins
Rico Mockel

Preface

This report is an exploration of virtual self-reconfiguring modular robots utilizing pathfinding algorithms to traverse obstacles and arrange themselves into target configurations. Modular robots are individual agents with simple capabilities that when combined into larger entities can perform complex tasks. Depending on the environment in which they are placed and the given tasks, modular robots are generally more dynamically capable and adaptable than a single monolithic robotic agent[1].

Abstract

Herein is described a comprehensive approach to configuring modular robotic agents that can search for a goal space and reconfigure themselves into that goal configuration. It includes a description of all algorithms used, the comparisons of such algorithms, and results from extensive testing. Finally the robustness and efficiency of each algorithms is discussed, and an exploration of which would apply in which situations is included.

Contents

1 Introduction

The concept of modular robotics is different than typical monolithic robots in that the modular concept takes many cues from nature. Instead of a single agent capable of performing a few specific tasks, modular robotic agents are comprised of many smaller modules capable of individually performing small atomic tasks, not unlike cells in a living organism. However when combined into a larger entity, their dynamic abilities shine as each modular agent plays a small role in a complex system capable of a variety of tasks limited only by the combined capability of their individual modules.
By giving individual agents specific tasks, the problem scale becomes boiled down to a much more simple form, and the whole modular agent becomes much more dynamically capable and independent of human control. It is far more likely that the robots of the future are going to mimic biological systems as nature has shown that efficiency and autonomy is key, and modular robotic agents are much more adaptable and dynamic than their monolithic counterparts[1].

1.1 Project Description

In this project the task was to create a virtual continuous environment in which to test simulated modular robotic agents, assigning agents a goal configuration, experimenting with various search algorithms to reach said goal, and test various obstacle environments to optimize these algorithms. The simulation's purpose is to have an agent begin in a user-defined configuration, traverse a field of obstacles, and reconfigure itself in a new configuration without any human direction.
In addition, a realistic representation of several laws of physics must be represented in this virtual environment. Gravity, friction, and impulses must be accounted for and realistically simulated. Agents are bound together by magnetic forces, and they must adhere to one another, as they cannot move independently. The mechanism in which agents move is defined as locomotion, where one moving agent relies on a stationary agent to move upon. It is impossible for agents to move along the floor or in the air without any support from its fellow modules.

1.1.1 Problem Definition

In a continuous environment, there is hypothetically an infinite number of moves an agent can make on its path towards a goal. Consequently, search algorithms must utilize a "distance-to-goal" heuristic to ensure that any movements made bring the agent closer to satisfying the goal configuration. Furthermore, the agent must be able to fully satisfy the goal configuration, meaning that leaving behind modules is unacceptable. Therefore, the agent must be able to identify objects that it cannot fully traverse and must find a more optimal path to the goal.
Finally, the concept of a distributed search algorithm based on pheromone trails modeling the famous ant colony optimization problem [9] is explored, however the results of such an algorithm are difficult to compare to more traditional search algorithms as the agents in a distributed search are unaware of the goal location. This means that the efficacy of a pheromone-based distributed search is incomparable with such traditional searches, and the results are evaluated separately from the other implemented search algorithms in this report.

1.1.2 Research Questions

With such a problem scope, many key questions need to be addressed. Is each module within the larger agent entity given a specific goal location, or should the agent as a whole move towards the goal space in any way possible and reconfigure based on its current configuration upon approaching the goal? Which of these methods is more efficient or effective? Given the same obstacles environment, which algorithms present the most efficient manner of traversing the obstacles, and which heuristic constraints perform best? Additionally, given extremely specific

obstacle configurations, which algorithms are robust enough to efficiently navigate the obstacle to the goal? Finally, how does the agent recognize impassible obstacle configurations and impossible goal configurations, and what does it do in such a situation? These questions are addressed throughout this report.

1.2 Report Structure

This report first contains an introduction to the concept of modular robotic agents in Chapter 1, followed up with a description of the given project and specific problems to be addressed. In Chapter 2 is an overview of the project software implementation including the structure and design of the program. Then in Chapter 3 the specific search algorithms and techniques implemented for this project will be discussed, including their specific implementations within the project domain. In Chapter 4 can specific obstacle and environment configurations be found, including a discussion regarding why these specific configurations were used as test cases for the agent. In Chapter 5 the reader can find detailed results of each algorithm's performance in specific environment configurations, followed by a discussion comparing each algorithms results and an exploration why each result was obtained in Chapter 6. In Chapter 7 the project results are concluded, and finally the bibliography and appendix can be found at the end.

2 Project Overview

2.1 Search Algorithms

For this project, a specific search algorithm "NotSoSimpleAnymoreAI" was an implementation requirement, and in addition three other well-known search algorithms were implemented. Finally, a pheromone-based distributed search swarm algorithm was implemented to explore the efficacy of such a concept. For a more in-depth explanation of the listed algorithms, please see chapter 3.

- **NotSoSimpleAnymoreAI (SimpleAI):** The NotSoSimpleAnymoreAI (heretofore referred to as SimpleAI) had specific implementation requirements that dictated how each module moved towards the goal. The purpose of such a simple greedy-based algorithm was to act as a baseline for comparison for other more complex search algorithms.
- **Global A*:** The Global A* algorithm is an A*-based algorithm that is applied to the entirety of the agent modules at once, giving the agent a goal configuration containing all possible goal locations at once, as opposed to the concept of giving each individual module a specific goal location.
- **Grouped A*:** The Grouped A* algorithm is another A*-based algorithm that splits the agent configuration into subgroups, giving each subgroup its own goal to pursue, which breaks the problem size into a more computationally manageable size.
- **Monte Carlo Tree Search (MCTS):** MCTS is applied to the agent module configuration as a whole, given an entire goal configuration space as a target destination. MCTS is a well-known and researched search algorithm for robotic searches, and the purpose of implementation within this project is to provide a contrasting search algorithm to A* to provide context within algorithm comparisons.
- **Distributed Greedy Pheromone Search:** The Distributed Greedy Pheromone Search (heretofore referred to as Pheromone Search) is a fundamentally different approach to the problem posed by this project. As a distributed search algorithm, the purpose was to demonstrate the concept of swarm search techniques, and not to compare efficacy to any other type of search algorithm contained in this report.

2.2 Simulation of Physics

2.2.1 Collision Impulse

To model the actual physical results of a move, a simple physical simulation was implemented. For the simulation, all agents are added as bodies with a mass of 1 and all obstacles and the floor are added as bodies with a mass of 0 (which represents a static object). For every tick of the simulation, the physical world is simulated in steps of 0.005 seconds. At the start of such a step, all collisions are calculated. The collision impulse for a body A and a body B is calculated as follows[4]:

$$j = \frac{-(1+e)((v_B - v_A))}{im_A + im_B}$$

where j is the collision impulse magnitude, e is the collision restitution, v_A is the velocity of body A, v_B is the velocity of body B, n is the collision normal, im_A is the inverse mass of body A (or 0 if the mass is 0) and im_B is the inverse mass of body B (or 0 if the mass is 0). The restitution e of a collision is determined by[4]:

$$e = min(e_A, e_B)$$

where e_A is the restitution for body A, and e_B is the restitution for body B. The velocity v of a body is determined by[4]:

$$v_a := v_a - (n)(j)(im_A)$$
$$v_b := v_b - (n)(j)(im_B)$$

where n is the collision normal.

2.2.2 Friction

Finally, the friction is calculated and applied. In order to determine the required friction, the collision tangent t first has to be calculated[4]:

$$v_R = v_A - v_B$$
$$t = v_R - (v_R(n))(n)$$

where v_R is the relative velocity. Using this tangent, the friction impulse magnitude jt can be calculated as[4]:

$$jt = \frac{-((v_B - v_A)(n)}{im_A + im_B}$$

The collision friction coefficient μ between two bodies A and B is calculated as[4]:

$$\mu = \sqrt{\mu_A^2 + \mu_B^2}$$

To determine the actual friction impulse Jt, the following formulas are used[4]:

$$|jt| < (j)(\mu) \rightarrow Jt = jt(t)$$

else

$$Jt = (j)(t)\sqrt{\mu_{kinA} + \mu_{kinB}}$$

These are all calculations required for the collision resolution.
After these calculations have been performed, gravity is applied to all bodies and the velocities and positions of the bodies are updated according to the previous calculations. All of these calculations are then repeated until the provided time has been simulated.
In order to make use of this physics simulation, special action types (moves) were added for impulses and climbing. As a climbing action is always the same, this action simply teleports the module to the correct location. The impulse action on the other hand uses the physics simulation by applying a certain impulse. After every turn the physical world is simulated for one second.
The described system allows the program to provide a more realistic model. As a result, the developed algorithms are more applicable to real-life implementations. Of course, adding such a system also comes with a lot of overhead for almost every algorithm, but without a sufficiently realistic simulation the algorithms would hardly be useful in real-life.

3 Algorithms in Depth

3.1 NotSoSimpleAnymoreAI

NotSoSimpleAnymoreAI is a greedy-based algorithm that only allows for the movement of individual modules. It is divided into 3 different phases: selection, greedy movement and reconfiguration. A loop of the selection and greedy movement phases is repeated until a certain condition is reached, which will lead to the third phase, reconfiguration. Before starting with the selection method, some preliminary work has to be done. Indeed, there are many different ways of fulfilling the goal configuration but SimpleAI2 focuses on filling it from back to front. Therefore, all agents are initially assigned the furthest position in the goal configuration as a target.
Selection is the phase during which the next agent is chosen to move. This choice is based on a backward-forward decision rule: the furthest agent from the goal configuration is selected first.
Next, the greedy movement part of the algorithm is called once the agent has been selected. Since each module stores its own target, this agent can move in a greedy way towards its goal, until it finds itself in a position where no more moves are valid. If the final position of that agent is not part of the goal configuration, the selection phase is called again and the loop is repeated.
The reconfiguration phase is triggered when an agent has found a goal position. This method is based on the assumption that once an agent has found a goal position, it will never be allowed to move anymore. Therefore, this specific goal position should never be assigned to any other agents. The detailed process is the following: once an agent finds a goal position, it is stored as a fixed agent. The rest of the agents then get assigned a new goal position, in order to ensure that they will not try to fulfill a goal position that is already occupied.
Finally, there is a method that copes with unsolvable cases. There exist situations in which the initial SimpleAI2 algorithm from the previous project phase would be unable to find a solution because none of the agents could move to a better position. When such situation occurs, agents are told to move in any direction possible until the configuration is again solvable with normal SimpleAI2.

3.2 Distributed Greedy Pheromone Search (DGPS)

With the distributed greedy pheromone search, the concept of a swarm based algorithm has been well-researched[3,9], though adapting known methods to a modular robotics simulation required out of the box thinking. Because agents cannot move on their own, agents were paired together in twos and each pair was treated as a single agent. Given a goal, the agents move according to a few given rules based on simple sensory input gathered by each agent. Each agent was given two "senses" simulating the sensor systems that might exist on a real robot. The sense of sight was given, allowing each agent to perform visual searches by casting rays up to a certain distance to determine if an object lay in its current path, or if objects lay to either side of the agent. If an object is detected, the agent moves towards that object until it is able to detect what that object is with its more precise olfactory sensors. These sensors allow the agent to detect if an object is an obstacle, another agent, or a goal. If no object is detected, the agent moves forwards until an object is detected by its sensors or it encounters an inactive pheromone trail of another agent. In the case of an inactive pheromone encounter, the original agent knows to avoid this trail as it knows that this trail does not lead to the goal. Thusly, fewer locations are explored by multiple agents leading to a greater area of distributed search by the agent swarm.
As an agent moves, it lays a simulated pheromone trail, containing a series of locations that it has already been[9]. This trail is inherently inactive up until the point where the agent comes close enough to a goal to use its olfactory senses to confirm the goal's presence; upon this confirmation the agent's pheromone trail is activated. Once another agent encounters an active pheromone trail, it knows that this trail leads to the goal in some way and begins to follow it. Furthermore, to aid other agents in finding an active trail, this agent activates its own pheromone trail, effectively doubling the size of the pheromone trail and increasing the chances of another agent reaching the goal. This distributed teamwork allows agents to remain completely unaware of the world state besides their immediate surroundings, yet they can still communicate indirectly to reach the goal.

3.3 A* Variants

3.3.1 Global A*

In the Global A* all agents are considered at once, meaning each node of the graph traversed by A* is a configuration, i.e. the set of all positions of the agents, and the neighbors of these nodes are determined by all possible moves of all agents. The heuristic score for global A* is calculated as follows:

$$\sum_{i=0}^{n} ||a_i - g_i|| + 0.5d_i = S$$

Where S is the heuristic score, n is the number of agents, a_i is the position of the agent in index i, g_i is position of the goal for the agent in index i, and d_i is 1 if the agent in index i is disconnected and 0 if it's connected (an agent is connected if it has at least one neighbor).
Because the global A* considers all possible moves, it is guaranteed to find a solution, though the possibility space can grow quite large.

3.3.2 Grouped A*

Unlike the Global A*, the grouped A* algorithm does not consider all possible moves at a time, but it does consider a group of modules at a time. The grouped A* algorithm uses the same heuristic H as local A*. The grouping of the modules is done by selecting the module with the lowest score according to H and then greedily expanding from this module until a group with a specific group size has been formed. After this, a new group is created (again, starting from a remaining module with the lowest score according to H) and the process is repeated until

there are no longer enough modules to form a group. The remaining modules are then assigned to neighboring groups. After all of the groups have been finalized, A* is run on every group in sequence and the modules are moved according to the computed path. These groups are usually still capable of getting to the goal themselves, and as a result the groups can simply move in sequence, though this needn't be the case for every possible configuration and it's highly dependent on the preferred group size

3.4 MCTS

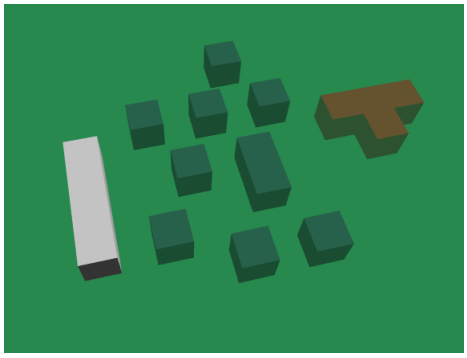
Monte-Carlo tree search is a search algorithm that focuses on finding the most favorable path to the goal, growing the search tree by randomly sampling legal actions for the robot modules[5]. The algorithm, hereby, builds a search tree, where the nodes of the tree represent the state of the world and the edges are representative of actions that cause a change in state. The essence of the algorithm becomes clear when split into four distinct functions the algorithm should carry out to find the best path to the goal.
The first function in the sequence is the selection of nodes in the search tree, starting from the specified origin of the modules, selecting the most favorable actions leading to the most promising state of the modules until an unexpanded leaf node of the tree is found.[5,7]
The second is expansion where you create child nodes of the current leaf unless the state stored in that specific leaf has already fulfilled the desired goal configuration. The third function is simulation where the algorithm performs a "roll-out" on each created child node, which randomly selects successive legal actions to take for each agent and then stores the score of the end configuration in said child node.[7]
The final function of the algorithm is a back up function which serves to use the information gathered from each sequence of valid actions (or simulation) to update the score of the nodes of the search tree all the way back to the root node (module origin) of the search tree[5,7]. Hereafter, you recommence with the first function creating a cycle of these four key steps. Once the goal configuration has been reached, the cycle is stopped and the path is constructed by moving down the tree from the root node choosing the most promising child node at each level based on its score. Using a tweaked UCT algorithm tuned to the optimal exploitation/exploration ratio for the project scope, where minimization of the score is favorable, we calculate the score based on the Manhattan distance of the modules in the end configuration of each roll-out to the desired goal location.

4 Defining the Environment

To demonstrate the efficacy of each algorithm, several precise test environments were created to challenge specific functions of each algorithm. Four main test environments were created, ranging in difficulty from trivial to highly complex. These four environments were tested on SimpleAI, A*, and MCTS. For the Distributed Greedy Pheromone Search, a separate test environment was created as swarm techniques require a larger search space to accommodate the large number of agents in the environment.
Note: The agents are viewed as white cubes, the obstacles are green cubes, and the goal configuration is in red.

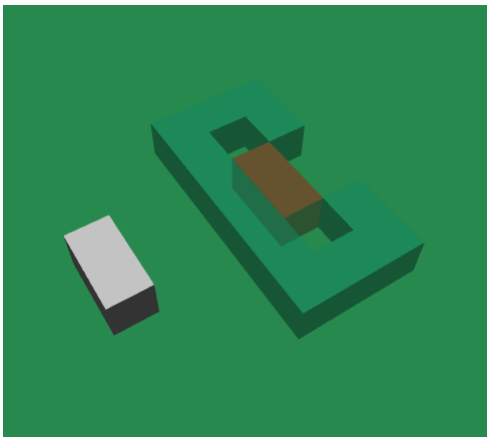
4.1 Test Environment 1: Minefield Traversal (Trivial)

This environment consists of four agents, a series of evenly spaced obstacles of dimension 1x1x1, and a goal of a max distance of 10 grid units. The minefield is meant to act as a simple obstacle field that demonstrates the ability of the agents to identify an obstacle, evaluate around the obstacle, and move closer towards the goal.



4.2 Test Environment 2: Horseshoe Entrapment (Moderate)

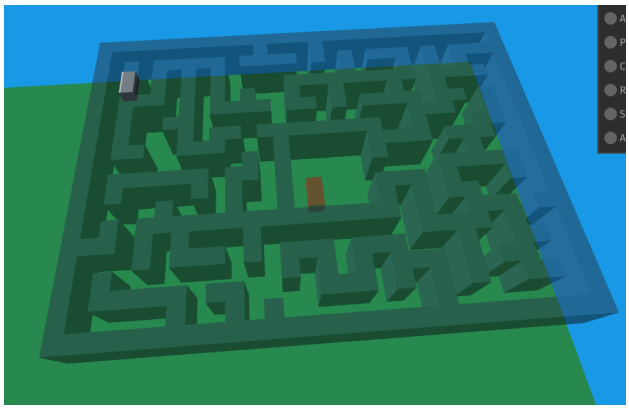
The Horseshoe Entrapment configuration places the goal location inside an obstacle configuration that surrounds the goal from three sides, and the starting agent configuration on the other side of the obstacle. The only way for an agent to reach the goal is to determine that it cannot climb the obstacle towards the goal, and must head further away from the goal by moving around the horseshoe towards the rear before entering the horseshoe towards the goal.
This configuration tests the robustness of the algorithm's ability to determine impassable obstacles, as it naturally would want to move closer towards the goal. Since that is initially impossible in this case, it leads to a reevaluation of possible moves which should then guide it first around the entrapment before moving towards the goal.



4.3 Test Environment 3: Labyrinth Navigation (Complex)

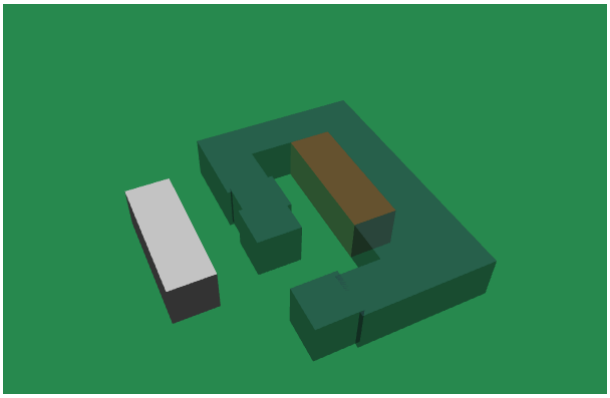
The Labyrinth Navigation environment tests each algorithm's ability to determine a highly specific route. The labyrinth is constructed in a way such that there is exactly one correct path of length 84 leading to the goal, With other possible "dead-end" pathways typical of a labyrinth maze. This specifically challenges algorithms such as MCTS, whose relative weakness in pathfinding exists in domains where there is a small number of correct paths amongst a larger number

of incorrect combinations. Because MCTS explores all possible combinations of moves and assigns scores to each move iteratively, it must explore an extremely large number of cases before reaching the correct combination of moves.



4.4 Test Environment 4: Non-Grid-Aligned Obstacle Gate (Complex)

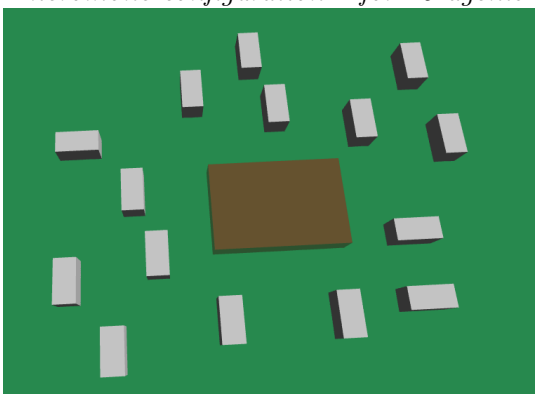
This obstacle environment tests the continuous space facet of the simulation, as the obstacles are not aligned with the grid upon which agents begin the simulation. The obstacle is similar to the horseshoe trap listed above, with an entrance of width 1x1 that prevents the agent from reaching the goal in any way except by aligning itself to the obstacle itself before entering the goal space. This challenges the algorithm's ability to identify misaligned obstacles, and to realign itself by moving less than a full grid unit - something that other test environments do not require.



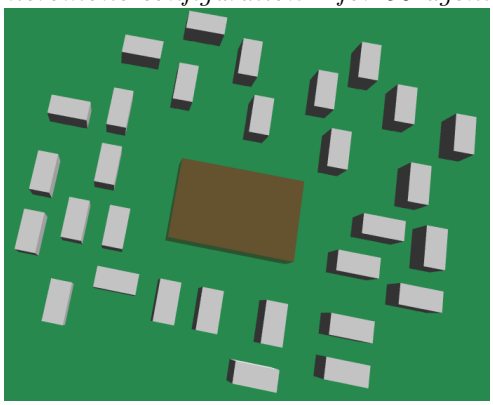
4.5 Distributed Search Environment: Pheromone Search

Consisting of two test environments, these are exclusive to the DGPS swarm algorithm in that they consist purely of agents in an open space with no obstacles, who must navigate based upon their senses towards the goal. This is meant to simulate the example given in the project 1.2 booklet that describes the possibility of modular robotic agents floating freely in outer space or on the surface of an alien planet, where agents are completely unaware of the goal location and must for themselves determine what is worth investigating, and what is negligible.
The first environment consists of a goal space surrounded by a number of agents, either 15 or 30 agents. The second environment consists of four goal spaces on the "edges" of the environment, with either 15 or 30 agents beginning in the center of these goals. These environments are meant to contrast each other drastically for comparison purposes.

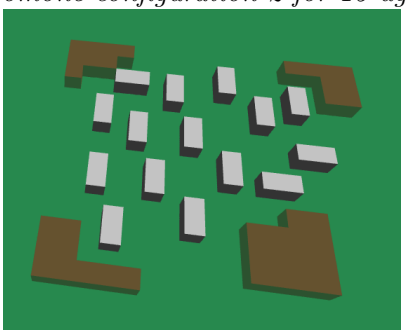
Pheromone configuration 1 for 15 agents



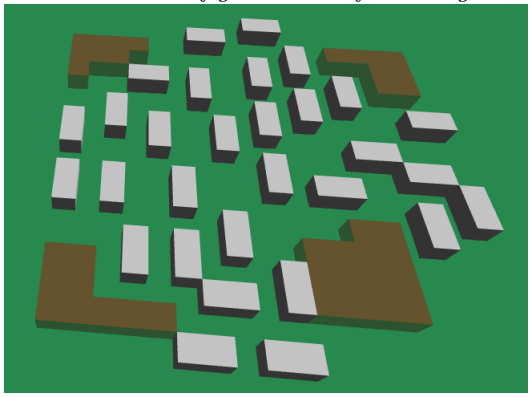
Pheromone configuration 1 for 30 agents



Pheromone configuration 2 for 15 agents



Pheromone configuration 2 for 30 agents



5 Test Environment Results

5.1 Test Environments 1-4:

Table 1: The results given from testing four pathfinding algorithms within the given test environments. Computation time is in seconds. Steps taken is the collective number of moves made by all modules within a great agent entity. The Manhattan distance is defined as the minimum number of grid length steps between the closest agent and the goal. *TO* indicates an algorithm timed out by reaching its maximum number of exploration iterations, or by reaching its maximum time limit of 600 seconds.

Test Environment	**Algorithm**	**Number of Agent Modules**	**Computation Time** *(seconds)*	**Steps Taken**	**Minimum Distance**
Minefield	SimpleAI	4	2	89	8
Minefield	Global A*	4	2	42	8
Minefield	Grouped A*	4	2	44	8
Minefield	MCTS	2	<1	18	8
Horseshoe	SimpleAI	2	*(TO)*	-	14
Horseshoe	Global A*	2	2	29	14
Horseshoe	Grouped A*	2	2	29	14
Horseshoe	MCTS	2	340	32	14
Labyrinth	SimpleAI	2	*(TO)*	-	84
Labyrinth	Global A*	2	50.38	165	84
Labyrinth	Grouped A*	2	50.38	165	84
Labyrinth	MCTS	2	*(TO)*	-	84
Mis-Aligned	SimpleAI	3	*(TO)*	-	6
Mis-Aligned	Global A*	3	5.6	32	6
Mis-Aligned	Grouped A*	3	5.6	32	6
Mis-Aligned	MCTS	3	*(TO)*	-	6

5.2 Distributed Pheromone Greedy Search Results:

Table 2: The results from DPGS with both 15 and 30 agents, given environments 1 and 2 as defined in Chapter 4.5. A time limit of 540 seconds was given to all test cases except for test environment 1 with 30 agents, which was tested with both 540 and 780 seconds to compare the results of giving more allowed computation time. Let it be defined that "Almost Reached Goal" means agents are within a 3 block unit distance from the goal, and are sitting on an active pheromone trail.

Test Environment	**Number of Agent Modules**	**Computation Time** *(seconds)*	**# Agents that Reached Goal**	**# Agents that Almost Reached Goal**
1	15	540	9	3
1	30	540	15	8
1	30	780	18	5
2	15	540	14	1
2	30	540	30	-

(Diagrams of the end positions of the agents are appendicized. Lines in orange going through agents at the end position show the active pheromone trails that lead one or more pairs to the goal.)

6 Discussion

6.1 Simple Minefield Obstacle

In this simple configuration, 3 algorithms have satisfying results. Global A*, Grouped A* and SimpleAI found the goal while MCTS left an agent behind, which prevented it from fulfilling the goal configuration completely. The path taken by the SimpleAI is considerably longer than both of the A* algorithm paths. As for the two A* algorithm comparison, their path length is very similar with a slight advantage for Global A* by not dividing the set of agents into groups right from the start.

6.2 Horseshoe Entrapment Obstacle

In this test case, it is clear that the A* algorithms works the best, as it finds an optimal path in a short amount of time. This is due to the small amount of agents and the relative closeness of the goal. Despite these factors however, this configuration proved to be challenging for MCTS. It did find the goal, but it took a lot longer than what would be acceptable. A possible reason for this result may be that the algorithm is too greedy and therefore does not try possibilities that don't bring immediate results (e.g. moving to the side). As expected, the simple AI does not work in this case, because all it tried is to move closer to the goal, which was deliberately made impossible in the configuration.

6.3 Labyrinth Obstacle

In the labyrinth configuration, the only algorithms that managed to run were Global A* and Grouped A*. In this case however, there is no real difference between the two as the configuration was too big to test more than two agents. SimpleAI does not find the goal and stops after taking 30 steps. In the case of MCTS, the number of objects limits significantly the speed at which calculations can be made. The inherent operation of MCTS requires the algorithm to explore all possible combinations of moves, and since only one possible combination leads to the goal there is an exponential growth of combinations that leads to incredibly inefficient computation speed, and therefore tests were inconclusive.

6.4 Non-Grid-Aligned Obstacle Gate

MCTS times out in this environment since the current implementation of MCTS is not compatible in a continuous environment. While the SimpleAI also works in a continuous environment, it had a tendency to leave behind modules. SimpleAI does not successfully reach the goal since a greedy exploration of the goal configuration is virtually impossible. The Global A* and Grouped A* algorithms were the only algorithms that were able to successfully cope with the necessary realignment in a continuous environment. The only algorithm that did not work at all-in a continuous environment was the MCTS algorithm, as the implementation did not use the special actions required for operating in a continuous environment. These actions allow the modules to align themselves with other modules or obstacles, allowing them to pass through the obstacle towards the goal.

6.4.1 Pheromone Results

Different configurations lead to different performances for this algorithm. In the first configuration, approximately 55% of the swarm found the goal in each case and in the second configuration, more than 90% of the swarm reached it. The closer the swarm starts from the goal, the faster it will get there. The number of agents in the swarm itself also has an impact on the performance. Increasing the size of the swarm has a positive impact as long the agents stay close to each other.

Indeed, if agents start from a considerable distance to each other, their communication ability decreases, which will lead to worst global results for the algorithm.

7 Conclusions

Different conclusions can be drawn from those test cases results. First of all, three out of the four algorithms work in continuous environment. Second of all, each algorithm can cope with a different level of configuration complexity. MCTS is for the moment only that efficient when there are only two agents, SimpleAI can cope with misaligned blocks but is only applicable in relatively easy test cases such as the first "minefield". Grouped A Star and AStar on the other hand work really well in complex test cases such as the labyrinth or the horseshoe and are always guaranteed to find the goal if given enough time.
MCTS doesn't perform as efficiently as the A* variants mostly because it analyses a lot more possibilities. MCTS by definition builds a tree of possible states and when the branching factor of the tree expansion is as big as the size of the valid moves for a certain configuration, the time needed to find a path becomes too important. And that huge computation time is the main reason why MCTS's results are worse that A*. Indeed, an A*-based algorithm only analyses a limited number of states, the ones that look more promising, which allows it to find a path in a way shorter amount of time.
Future improvements could include optimizing the efficiency with which the environment is simulated as well as enhancing the MCTS algorithm in order to ensure its success in cases where more than two agents need to find a goal position.

References

[1] Meyers, Robert A. (2009). Encyclopedia of Complexity and Systems Science. *A Guide to Graph Colouring: Algorithms and Applications*. Springer International Publishing.

[2] Bonardi et al. (2013) Collaborative Manipulation and Transport of Passive Pieces using the Self-Reconfigurable Modular Robots. *Roombots, page 2*. Springer International Publishing.

[3] S.Garnier et al (2007). *Alice in Pheromone Land: an Experimental Setup for the Study of Ant-like Robots*.

[4] Dr. J. B. Tatum. *Physics in Game Space*. http://astrowww.phys.uvic.ca/ tatum/classmechs/class5.pdf

[5] Chaslot et al. (2006). Monte-Carlo Tree Search: A New Framework for Game AI. *https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf*. Universiteit Maastricht

[6] Kocsis, et al. (2006). *Bandit based Monte-Carlo Planning*. Computer and Automation Research Institute of the Hungarian Academy of Sciences.

[7] Magnuson, (2015). *Monte Carlo Tree Search and Its Applications*.

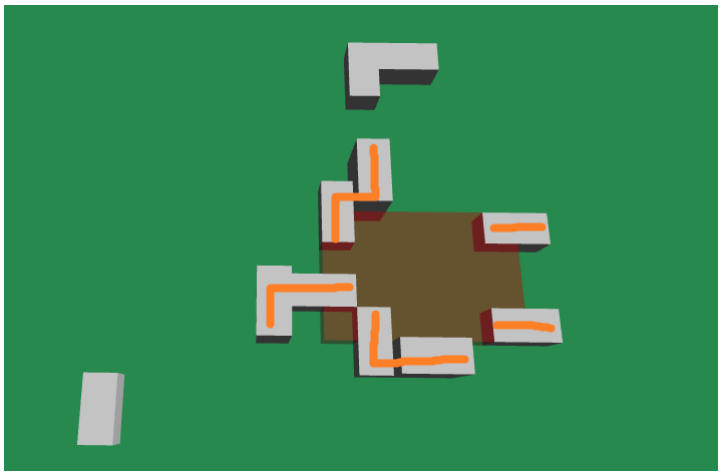
[8] Nillson (1968). *A Formal Basis for Heuristic Determination of Minimum Cost Paths*. (16-7-2017) http://ai.stanford.edu/ nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf

[9] Xiao-Min Hu et al. (2008) *Orthogonal Methods Based Ant Colony Search for Solving Continuous Optimization Problems*. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY: 23(1)

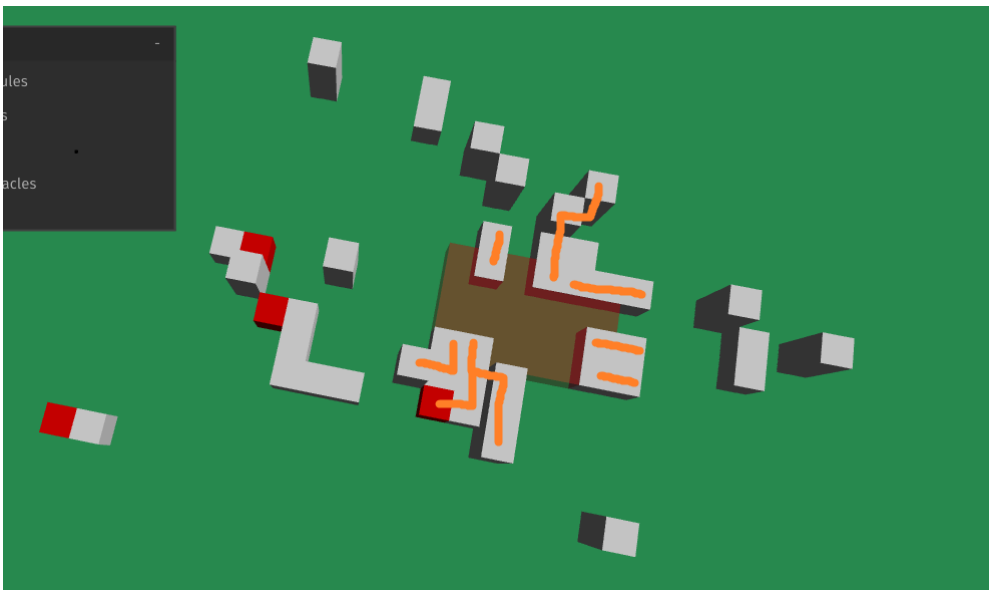
Appendix

Appendix: Final Configuration Results

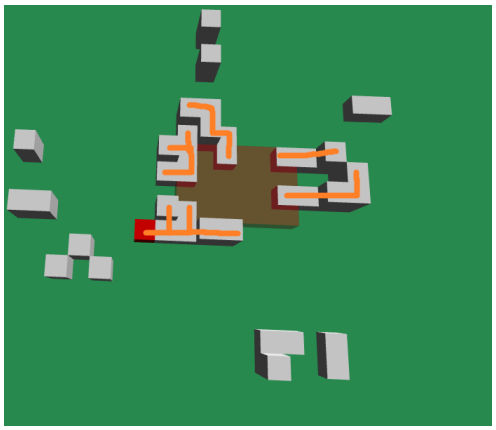
Final Configuration 1 (15 agents):
Final positions of the swarm in the configuration 1, with 15 agents and 9 minutes of allowed computing time. (2 agents are left out of the pictures because they ended up too far from the goal).



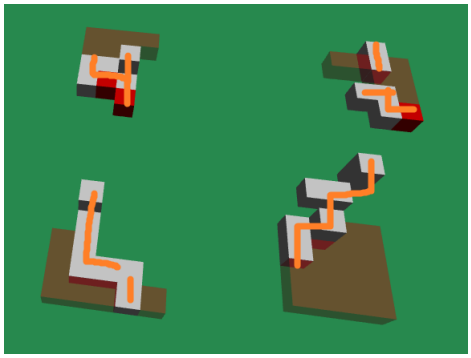
Final Configuration 1 (30 agents):
Final positions of the swarm in the configuration 1, with 30 agents and 9 minutes of allowed computing time.



Final Configuration 1 (30 agents - 13 minutes):
Final positions of the swarm in the configuration 1, with 30 agents and 13 minutes of allowed computing time.



Final Configuration 2 (15 agents):
Final positions of the swarm in the configuration 2, with 15 agents and 9 minutes of allowed computing time.



Final Configuration 2 (30 agents):
Final positions of the swarm in the configuration 2, with 30 agents and 9 minutes of allowed computing time.