Self-reconfiguration for Adaptive Furniture

Jocelyne Lotfi

January 14, 2009

Contents

Abstract

Roombots are modular robots developed at the BIRG in EPFL. These robots have a much wider possibility of movement due to their three degrees of freedom than previous YaMoR[3] modules, a former modular robot previously used at the BIRG, and can be used for different tasks. One of these task is to be able to reconfigure themselves into different shapes of furniture.

This project presents an algorithm for self-reconfiguration for roombots. It is based on the work of Masoud Asadpour[1] that builds an algorithm capable to define a list of actions that will reconfigure a group of YaMoR modules. His algorithm represents spacial configuration as graphs and uses graph edit distance and graph signature.

This master project presents an adaptation of Asadpour's algorithm for the roombot modules. It also extends Asadpour's algorithm by changing the definition of a configuration.

Chapter 1

Introduction

Modular robotic is a growing field constantly improving as to give answer to some of the main issues of the future of robotics, like locomotion in unpleasant environment (uneven ground, forest, snow, etc.) or quick reconfiguration (change of shape to be able to use certain tools or do specific actions).

This field faces many challenges among which we have the capacity to determine the best suitable shape that is needed to be taken to accomplish a task. Another issue is general robustness of the robotic systems. Being able to deal with a large amount of units and with eventual failures is a future task to solve. Having a good communication system between units is also an issue.

This project concentrates on one of these challenges: the optimal reconfiguration of a structure using self-reconfigurable modular robots as basic building blocks. It is part of a bigger project developed at the BIRG (Biologically Inspired Robotic Group)[24] at the Swiss Institute of Technology (EPFL) that aims at developing adaptive furniture called roomware[20]. This idea of having robots hidden in our environment was already proposed by M.Weiser[2] and is one of the field in which modular robotic can be applied.

1.1 Background and literature review

Modular robots are robots made of a lot of simple small and identical modules that can attach and detach themselves from the whole structure. The main idea is to get a robot that can change its shape depending on the task to be solved. One of the task in which these robots could be useful is in creation of adaptive furniture, that is, pieces of furniture that change their shape depending on the need of the users. An example could be to have a table that is able to move and change itself into a chair if necessary and, once it is no more needed, puts itself in a corner taking the form of a block.

This is only one specific example, but modular robots are expected to be useful in many areas of human activities like space exploration, helping to search for survivals in case of catastrophe or even mining under sea.

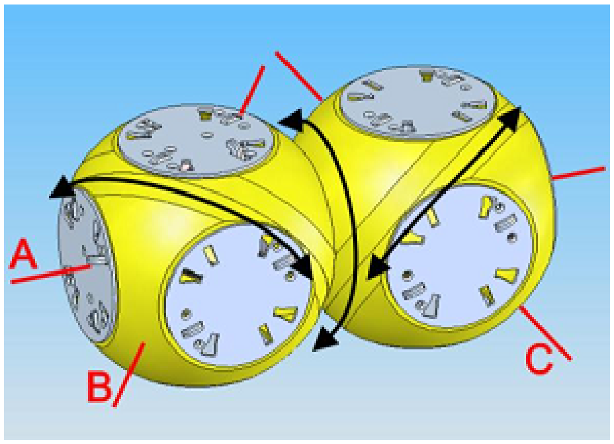
The main advantages expected from modular robots are:

- Robustness: as robots are made of many small identical modules, any part can be replaced easily; this could lead to self-repairable robots.
- Versatility: The capacity of adaptation of such robots allows them to create new morphologies better suited to execute multiple different tasks.
- Low Cost: Such small identical modules are well suited for mass production, which will inevitably reduce the overall cost of building modular robots. Moreover, such a modular robot has a general design to accomplish multiple tasks, other than a monolithic robot design, where many different robots are used each for a specific task.

Modular robots are generally divided into two categories, lattice and chain type. Lattice type use cluster-flow reconfiguration and movement. It means that it uses attachment and detachment over a lattice of other modules to progress, it “flows” over them. Some examples could be the Cristal robot[5] and the ATRON[7]. The chain type uses its own force to move and reconfigures itself only to adapt to a new task or environment. This category includes YaMoR[3], M-Tran[4] and Polybot[19].

Another way to classify modular robots concerns the way they reconfigure. They can use a deterministic or stochastic reconfiguration. In a deterministic

Figure 1.1: a roombot module



reconfiguration, the position of all modules are always known and movements are directly controlled. On the other hand, stochastic reconfiguration ensures only statistical probabilities on time needed to reconfigure or the position of different modules.

This project was done with a new type of robot called roombot[13]. This robot looks like two dices attached one to the other and having three degrees of freedom, one between the dices and two other that go through the biggest diagonal of each dice, its design was inspired from the molecube module[18]. Figure 1.1 presents a picture of such a module. It can be considered as an hybrid between lattice and chain type robots. Roombot modules can both move themselves on a lattice respecting a 3D grid or behave like chain robots to locomote with legs for example. It is not the first attempt to mix both types, Superbot is another example of such an hybrid modular robot[12].

The specific problem studied here is the problem of Self-Reconfiguration Planning. The reconfiguration problem consists of determining the different actions that should be undertaken by a group of attached robots so as to pass from a certain configuration in space to another.

This problem represents a challenge as the configuration space grows exponentially when new roombots are added. So, heuristic methods need to be used to obtain some solutions. The goal is to be able to find a viable solution, as optimal as possible, in a reasonable computation time.

Different methods were used to tackle this problem. P. Brat and al.[17] propose a hierarchy of sub-structures. Their approach consists of a base planner that computes an optimal solution for a few numbers of modules, composed of a group of units, and a hierarchical planner that calls this base planner or reuses pre-computed plans at each level of the hierarchy.

A more generic solutions for what is called "meta-modules", groups of modules in identical specific configurations, is presented in [14]. A "meta-modules" can do a set basic moves which are pre-computed and the planning is done entirely on "meta-modules" instead on units composing it.

These approaches tend to reduce the computation load by repeating at different level the same structure. But a first step is needed before applying these methods: the computation of a specific reconfiguration of a small group of modules.

On this smaller scale, reconfigurations often use stochastic method, like Genetic Algorithm or Simulated Annealing, combined with some heuristic to search more efficiently the space of solutions. Murata et al.[9] uses, for example, weighted probabilities based on potential fields to guide the search procedure.

The notion of metric distance was also introduced the first time by Pamecha et al.[10]. Metrics are useful tools to build efficient heuristics used to guide search. [15] introduces several of them.

In this project, we will adapt and improve the reconfiguration algorithm done by the team of Masoud Asadpour[1], who worked with YaMoR[3] and M-Tran[4] modules and presented a deterministic centralized reconfiguration algorithm and a specific metric based on graph theory to guide search. The algorithm of M. Asadpour is able to handle up to 8 YaMoR modules in reasonable time.

This report will first present a summary of the work of M. Asadpour in chapter 2; chapter 3 explains the different changes and improvements done to the original algorithm and the reason of them; chapter 4 contains a more detailed description of the implementation; chapter 5 includes results obtained with the new algorithm and a comparison to the previous one and finally chapter 6 presents some future works on the subject.

Chapter 2

Previous Work of Masoud Asadpour

This master project is based on the work of Masoud Asadpour. He creates an application that calculates the attachment and detachment necessary for a group of modular robots to pass from a certain configuration to another. His work was tested with YaMoR modular robots, so all pictures in this section will have figures with YaMoR modules.

Asadpour's algorithm can be decomposed in five main parts:

1. The representation of the configuration, meaning how we define a configuration and what it is representing in the real world. Section 2.1.

2. The comparison method, that is how do we compare two configurations and when they are considered the same or not. Section 2.2.

3. Distance metric, which represents the distance to the goal, how it is computed and how close it is to the real distance observed. Section 2.3.

4. The search strategy, which explains the way we choose the path in the configuration search tree. Section 2.4.

5. And finally, how all these components are assembled together to give a full algorithm. Section 2.5.

2.1 Configuration

This first question we have to answer is what is a configuration. It is defined here as a particular arrangement between independent modules. A configuration can be represented by a graph where each node represents a module and each edge a connection between them, meaning where they are attached one to the other. Fig. 2.1 shows both the robots and the graph representation of their configuration. In addition, each edge has a label that expresses how both modules are attached. This label is computed using connector numbering and orientation between modules. This means that each connector has a certain number, and a label is determined by the source connector, the target connector and the rotation between them, Fig. 2.2 represents the labelling of connectors used on YaMoR module. We can see that the module behind is connected by the connector 0 to the connector 4 of the front module with no rotation.

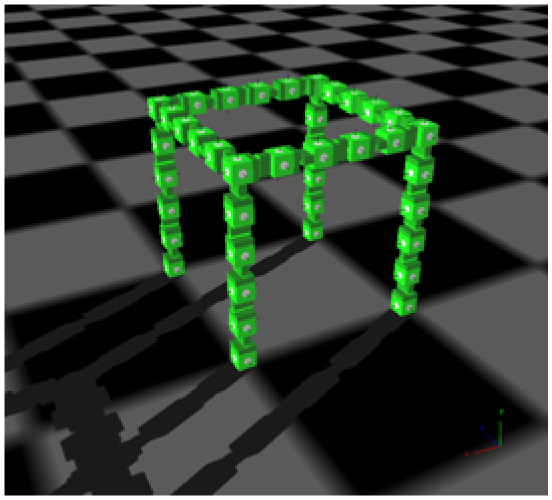
It is to be noticed that this way of labelling doesn't take into account the servo's position of the modules. Figure 2.3 shows an example where the same graph has different representations in space due to servo moves.

2.2 Signature

The next step is to explain the comparison tool used. The idea was to use a string representing the graph. This string is called a signature. It is via this signature that we can know if two graphs are identical or not. From a signature a graph can be reconstructed, but it is not a bijective application, more than one graph can have the same signature. The goal of the signature is to recognize that two graphs are the same and so represent a similar configuration of connectivity.

The graph isomorphism problem is an NP-hard problem[16], but for some special type of graphs polynomial algorithms permit to determine if two graphs are isomorphic or not. Among them Jiang and Bunke[22] proposed a quadratic-time isomorphism test for ordered graphs. This test was used to calculate a signature for the graph. First, configuration graphs were transformed into ordered graphs which was done by sorting the edge by their label values.

Figure 2.1: graph representation of a configuration, taken from Asadpour's[1]



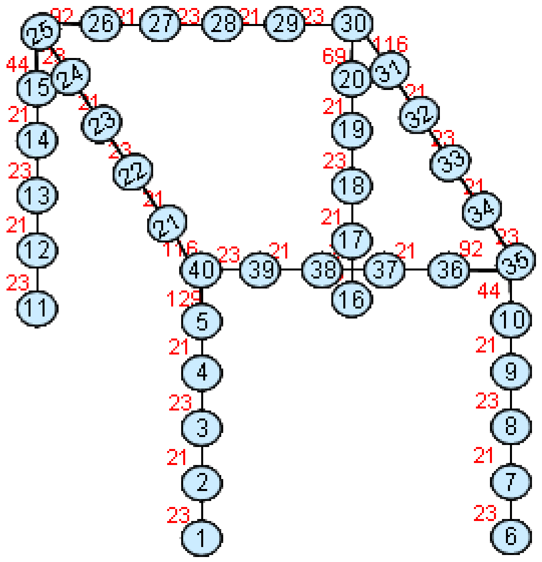


Figure 2.2: labelling of an YaMoR module, taken from Asadpour's[1]

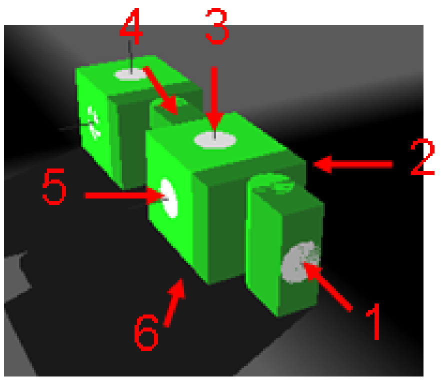
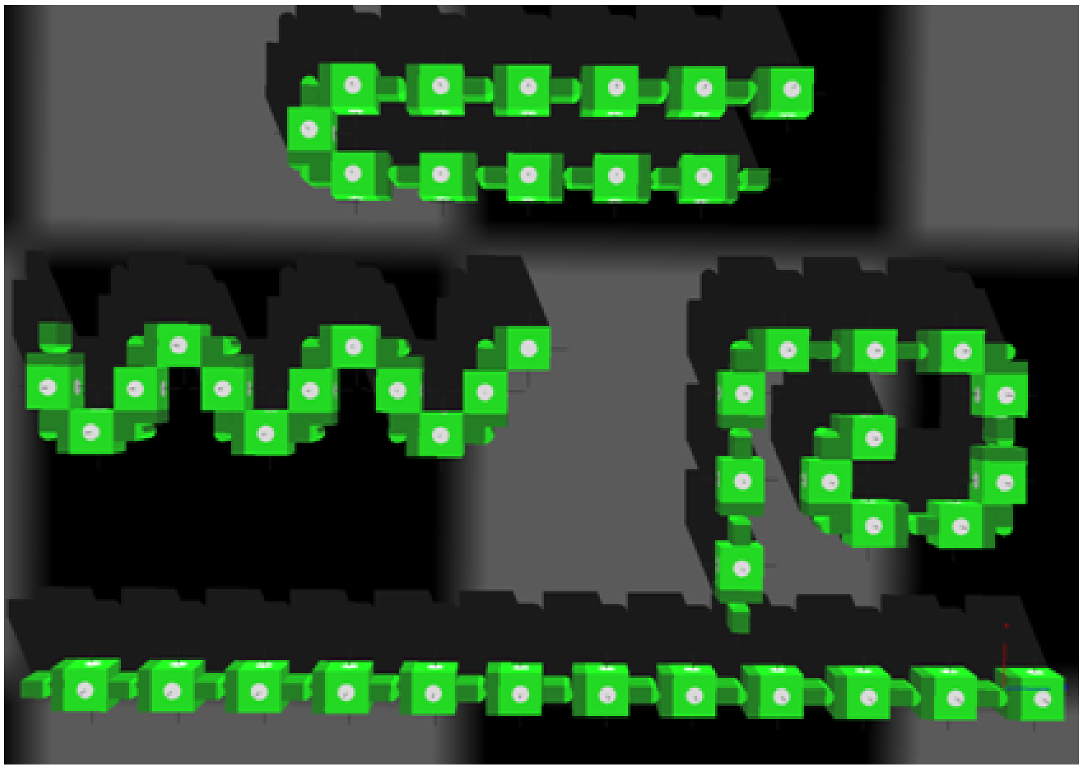


Figure 2.3: different configuration in space for the same graph, taken from Asadpour's[1]



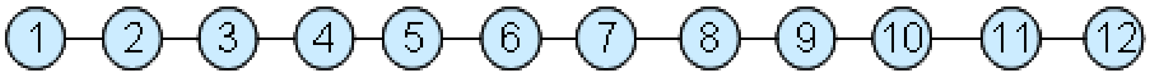
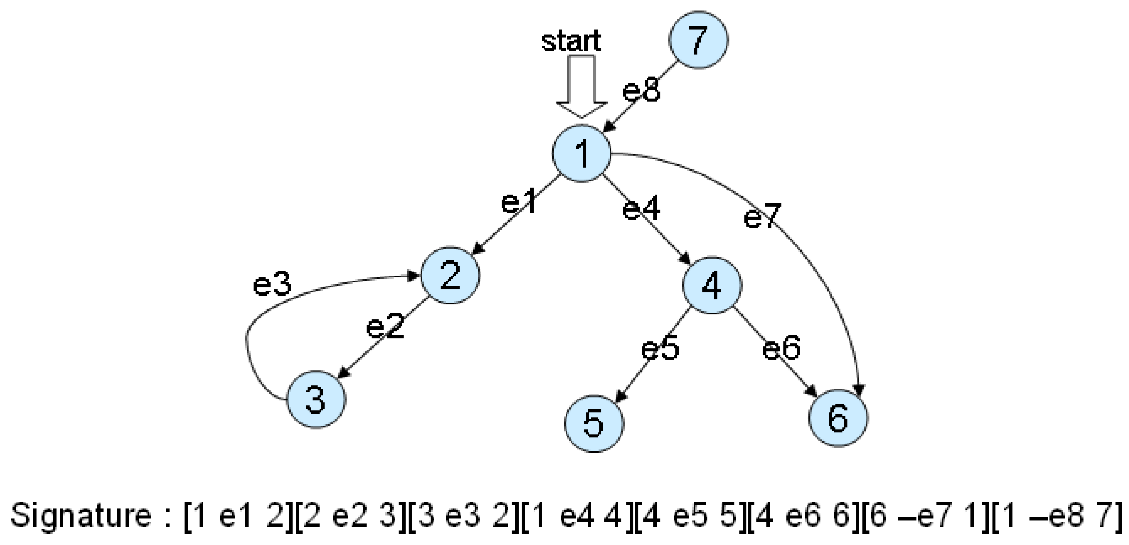


Figure 2.4: a graph and its signature, taken from Asadpour's[1]


start
7
e8
1
e1
e4
e7
2
4
e3
e2
e5
e6
3
5
6
Signature : [1 e1 2][2 e2 3][3 e3 2][1 e4 4][4 e5 5][4 e6 6][6 –e7 1][1 –e8 7]

Therefore, the calculation of the signature is done as followed: considering that we have an oriented weighted graph, the signature process is started from every single node and a depth first search is done on the nodes following the order of the edges. The signature will be a sequance of items containing the start node, the target node and the label. If the search process has to go through an edge in the reverse direction (in-edges), a minus sign will be added. From all computed signatures, the biggest possible one is kept as the signature of the graph.

If the module has symmetry axes and can be reversed without change in the configuration, it is the biggest possible label that is taken into account. Fig. 2.4 shows a graph and its signature. It was computed as followed, assuming that the edges are sorted according to their label, out-edges in descending order and in-edges in ascending order. The vertex shown as start point is indexed with 1. The out-edge e_1 is traversed first. The newly visited vertex is indexed with 2 and $[1, e_1, 2]$ is added to the signature. Then vertex 3 is visited via e_2. From vertex 3, vertex 2 is reached via e_3. Vertex 2 was already visited, so we back-track to 3. From vertex 3 we have no other edges to traverse. We have to back-track to 2, and then to 1. In vertex 1 the next edge i.e. e_4 is followed. Similar steps are repeated until vertex 6. It has two in-edges, e_6 and e_7, and no out-edge. As this node was reached by going though e_6, the next unvisited edge would be e_7. Since we are traversing e_7 in reverse order, $[6, -e_7, 1]$ is added to the signature. The procedure is continued until all vertices and edges are visited once and only once.

2.3 Distance metric

The third point is the distance metric chosen. This metric is based on the graph edit distance between two graphs. It is the minimal number of additions or removals of nodes and/or edges necessary to go from a certain graph to another. It's mathematically proved to be related to the Maximum Common Sub-graph (MCS) between two graphs by

$$\delta(I, F) = 1 - \frac{|MCS(I, F)|}{max(|I|, |F|)}$$

where I and F represent respectively the initial and final graph and $|MCS(I, F)|$ is the number of edges in the Maximum Common Sub-graph between I and F. $|I|$ and $|F|$ are respectively the number of edges in the initial and final graph. The problem comes from the fact that $|MCS(I, F)|$ is not computable in reasonable time so an upper-bound approximation is used instead. This approximation, proposed by Asadpour[1], is

$$\sigma_{UB}(I, F) = \frac{\sum_{l=0}^{l_{max}} min(C_l^1, C_l^2)}{max(|I|, |F|)}$$

where C_l^1 and C_l^2 are respectively the number of edges in the initial or final graph that have the label l.

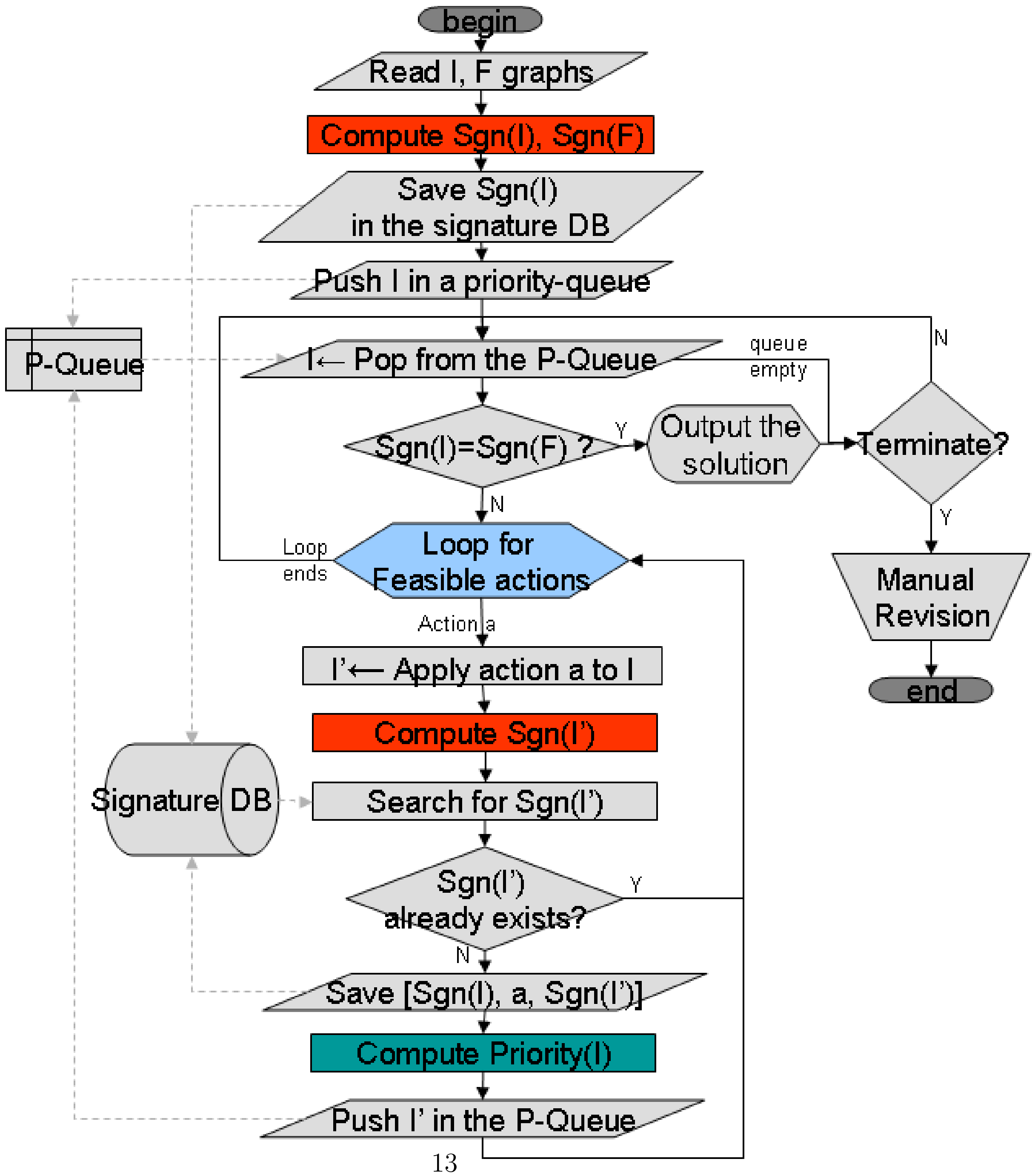
2.4 Search strategy

The last thing is how the search is done. First, each graph, once computed, is pushed in a pile sorted by distance to the final graph. It means that we always check first the graph that is the closest to the final configuration. It doesn't guarantee to find the optimal solution but it ensures the progression to the goal. Ensuring to have found the optimal solution would require to go through the whole search tree, which is impossible as it grows exponentially. Randomness comes when two or more graphs have the same distance to the final graph, in which case the next graph to examine is chosen completely randomly among them, this can happen quite often especially with small graphs.

2.5 Full algorithm

The full algorithm is given in Fig. 2.5. First, the signature of the initial and final graph are computed. If they are the same nothing happen, otherwise the initial graph is pushed in the priority queue sorted by distance to the final graph. For each graph popped from the queue which is not equal to the final graph, the algorithm searches all feasible actions. For each action, a new configuration is constructed and the signature of this new configuration is computed. The algorithm checks if the signature is in the database. If it is the case, the action and subsequent graph are discarded, otherwise the signature is saved in the database and the new graph is pushed in the queue. This loop is done until a graph is found to be equal to the final graph. The algorithm finishes when a solution is found or all different reachable graphs are visited without finding a solution.

Figure 2.5: a representation of the full algorithm used by the team of Asadpour, taken from Asadpour's[1]



Chapter 3

Improvements

This chapter contains the different improvements that was added to the code done by Masoud Asadpour. Firstly, the code was adapted for the roombot modules, secondly "Move Action" was introduced (section 3.2).

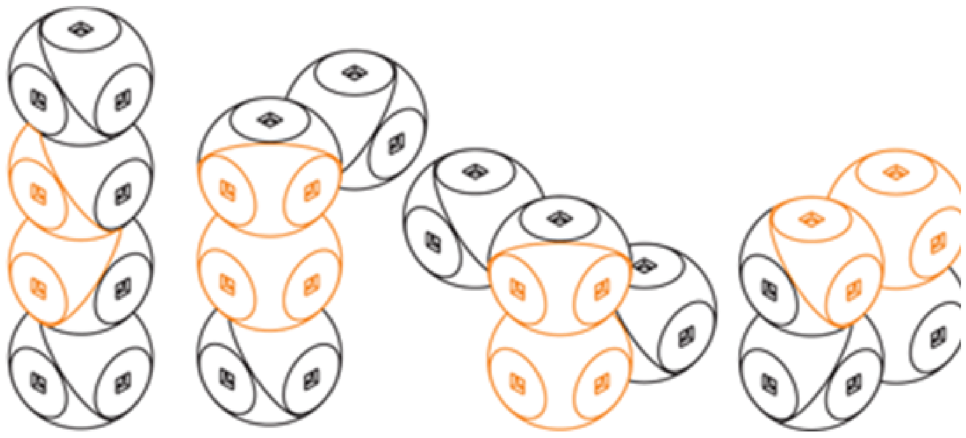
3.1 Adaptation for Roombots

The task of this master project was to adapt Asadpour's code to the current roombot modules. This mainly concerned two parts of the code: the model and the symmetries.

A model is simply the representation of the hierarchy of parts (servos or connectors) and their relative positions, given as transformation matrices, from the center of the module or from their parent part (if they have one) to themselves. The model is mainly used to detect the position of the different modules in space and helps on vital parts of the code (next possible movement, checking of collision, possibility of connection, etc.)

In the signature process, the symmetries of the module were changed to fit the current roombot module. The previous code was based on YaMoR modules which have a different shape (and less symmetries) than the roombots. This part brought to increase the computation time of the signature process for roombots as a signature is recomputed for every node and every possible starting position for the first module, which double, passing from 4

Figure 3.1: different configurations that would have the same graph



in the YaMoR module to 8 for the roombots.

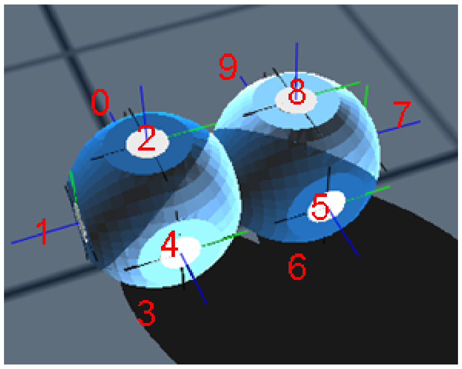
3.2 Move Action

We introduced an additional step: “Move Action”. As it is explained before in this report, the algorithm of Masoud Asadpour is based on the Edit Action Distance and servo moves are not taken into consideration.

With roombot modules that have three degree of freedom we can move in a 3D space a lot and still being considered as in the same configuration. For example, the four picture shown in Fig. 3.1 have the same graph and are considered the same and this is only with two roombots. The main difference between the old YaMoR module and a group of roombots is the possibility for the last one to move in 3D space without having to attach or detach. The YaMoR module having only one servo, meaning one degree of freedom, was able to move only in a 2D space.

The goal would be to consider configurations with distinct position in space as different. This will increase the number of possible configurations compared to [1]. For that, another way to label edges between modules is introduced. We don’t consider anymore the connector itself but more

Figure 3.2: labelling of roombot's positions

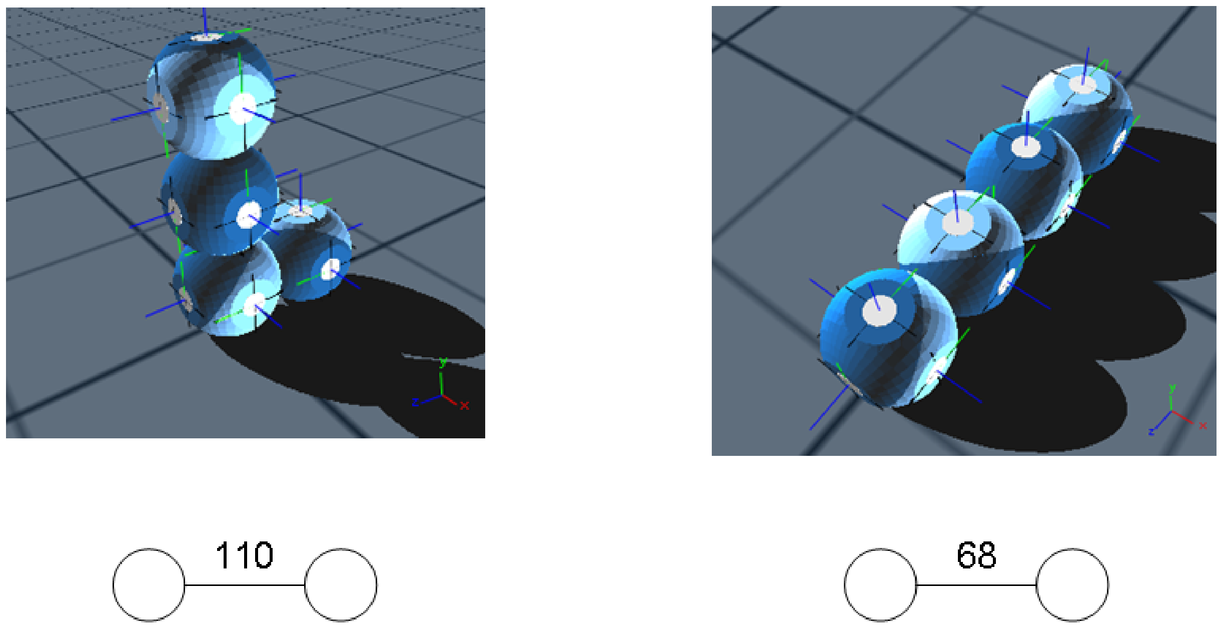


the position in which this connector is placed and the label is computed depending on between which positions modules are connected and no more between which connectors. Fig. 3.2 gives an example of position labelling on a roombot, it looks like Fig. 2.2 but the big difference is in the meaning of digits, we are no more considering the connector itself but the position in which it is. Due to architecture of a roombot, connectors can move from one face to another without changing the shape of the module, this was not possible with YaMoR where connectors were linked directly to the main body of the module. It means that when a connector moves from one face to another face of the robots, the position in which it is changes and so the label between both roombots also.

Fig. 3.3 gives an example of such a movement that implies a change in edge label. In the first picture the roombots are connected between an end position to a side position and when the left servo of the right roombot moves, we obtain the second picture on which roombots are connected by the same connectors but by both end positions. Exact computation of the label in the current implementation is given in the chapter 4.

This representation increases the number of configurations and so the precision of the result without violating the underling rule on which this representation is built. In fact, a Move Action, which is basically a change of label on an edge, can be considered as a detach action followed by an attach action between the same nodes. It is important because it permits us to re-use the distance metric of Edit Distance previously described.

Figure 3.3: graph representation of a configuration



The efficiency or usefulness of such a method was not proved here, but only being empirically tested, with the following theoretical background.

One issue that arises is that it produces more signatures than the precedent version. A configuration will have a number of move actions proportional to the number of movable servos. The signature process is one of the most time consuming process of the code right now. It should also increase the number of graphs that have to be examined by the algorithm. On the other hand, the current implementation of the code checks a lot of possible servo movements before applying an attach action, but without calculating the signature.

Increasing the number of possible configurations will also increase the size of the search tree as there are new actions and more possible graphs considered, but most of them will be redundant and thus will not dramatically increase the total number of graphs for which children graphs would be searched.

It is also interesting to notice that it will not consider the servo position in the signature but only the labels and the positions of the different modules to get the biggest ever possible signature. This means that we don't care which part of the modules are connected, which actual connector are there, but only the shape in space. Once a certain shape in space is assessed to find next possible graphs, this configuration will not be anymore looked at, even with different servo position that could bring to different next possible graphs. We are cutting ourselves from a possible way to get to the goal. This can prevent us to find some solutions.

Chapter 4

Implementation issues

This chapter explains the main parts of the implementations, it aims at explaining to the next user how the code is implemented for a future work on the subject.

4.1 Architecture

The code was written in C++. We use the C++ library boost[26] for two main things, the representation of graph via the Graph library and the shared pointers that are pointers which are garbage collected and don't need to be explicitly deleted.

The main class is the *Planner* class. The method *Reconfigure* contains the main loop of the algorithm with a priority queue sorted by distance to the final graph. It examines the current graph on the top of the queue, computes all possible single actions from it, puts them in the queue and tests if the loop is finished. It outputs also results in files for statistics and debugging purposes. The initial and final graphs are written under the .dot format in .graph files, a .model file can be also given to the planner but it is not used for roombots as its model is hard-coded. The class *Main* is here to parse input files and gives them to the planner.

Graphs are represented as objects of the class *ConfigGraph*. This class implements the boost generic template so as to represent an oriented weight graph as needed in the algorithm. It has a subclass *ConfigGraph3D* which adds to it a certain number of fields related to a graph in the algorithm like the signature of the graph, the depth in the search tree, the distance, the model of the module and some other fields needed to compute the signature. This class contain the computation of the distance in the method *MCSUpperBound* and the computation of the signature via the method *UpdateSingature2*(the 2 is here as it is the second version of it).

The *Model* class contains the model of the module. The model is here to express the physical reality of the module. For that, it contains a list of *Part* which can be either *Servo* or *Connector* and physical quantities as the width of the module, the rotation tolerance, the step size of servos and connectors and the number of rotation possible plus a map, called *label_position_map*, that links connectors' positions and a certain label. For the roombots, there is a method *CreateModelDices* that constructs the model. Some conventions where followed when creating this model:

- the rotation axis of every servos is the y axis
- the rotation axis of every connectors is the z axis, with out-going direction. Figure 4.1 shows the coordinate system used both for the center of the module (which is the same as the central servo) and for a specific connector.
- the order of position is: left-back, left-end, left-up, left-down, left-front, right-front, right-down, right-end, right-up and right-back, and correspond to x1, y1, z1, y2, z2, x3, z3, x4, y4, z4 in the webots tree model in the default position. Fig. 4.2 illustrate that. The order is important for the computation of the label.

A *Label* is simply a short int and the method *IndexOfSource*, *IndexOf-Target*, *IndexOfRotation* are here to retrieve respectively the source position index, target position index or rotation index from the label. Position indexes run from 0 to 9 depending on the order stated above, rotation goes from 0 (same rotation) to 3 and each unit represents a $\pi/2$ rotation between axes of both positions. Labels are created by a simple loop on all possible source, target and rotation. The label in computed as followed: $40 * source + 4 * target + rotation$. The number 40 and 4 comes from the fact that we have for each source 10 possible target connectors that have

Figure 4.1: coordinate system used

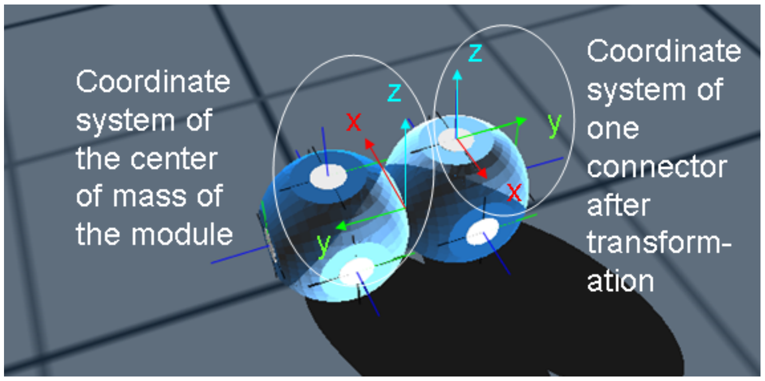
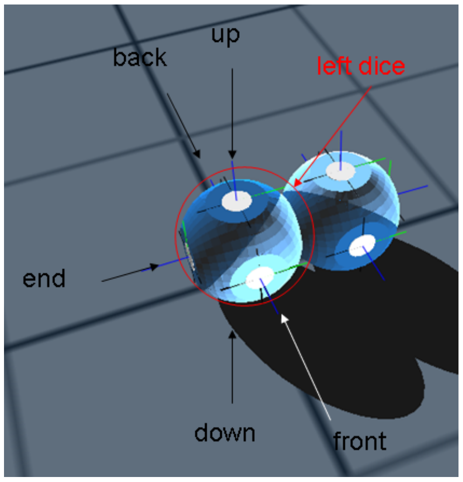

Coordinate system of the center of mass of the module
z
x
y
z
y
x
Coordinate system of one connector after transform-ation

Figure 4.2: convention on the naming of positions



themselves 4 possible rotations. In Fig. 3.3, the label of the left picture is computed as followed: it goes from the left-up connector (the connector labelled 2 of the roombot on the floor) to the right-end connector (the connector labelled 7 of the roombot above) with a rotation of 2 (a half-turn) so we have: $40 * 2 + 4 * 7 + 2 = 110$.

Every *Part* (*Servo* or *Connector*) has an index related to it but also two transformation matrices to represent its position compared to the center of mass of the module (called *tactual*) and to its father part (called *tdefault*). Each *Part* has a step size and a father part that can be null (if the part is static compared to the module's center of mass). The *Connector* has three more fields, *num_rotations* which in the case of the roombot is always four, *gender* which is not used for roombots as they are genderless connectors, and *position* of type *Position* which is simply a char representing one of the tenth possible connector's position on a roombot. *Connector* has also a *pos_trans_map* that maps each *Position* to a *Transformation* form the center of mass to the connector's position it represents. It's initialized with indexes and transformations of connectors in the default state, i.e. the *Connector* with index 3 that has a *Transformation tactual t* at the beginning, will be in the position 3 and the position 3 will have as transformation *t*. The *pos_trans_map* is constant once initialized.

The code is done to have only one set of *Servo* and *Connector* objects even if the current graph contains many nodes. So whenever a computation has to be done on a specific module, the method *UpdateModel* from the class *ConfigGraph3D* has to be called to update the different transformations to fit the current servo position of this specific module.

Each action is represented by the class *EditAction* which contains four fields, the type of the action (attach, detach or move), the source vertex, the target vertex and the label. The class *EditActionFinder* contains all methods to compute all possible actions from a graph and the resulting graph after an action is done. The key method is *Find* which called respectively *BuildKinematicGraph, FindMoveCandidate, FindAttachCandidate* and *FindDetachCandidate.*

The kinematic graph is where each module, servo and connector is represented by a node and permits to compute a load factor to know if the roombot will be able to lift or not the parts. A load map is computed from it, giving a minimal load number for each edges of the kinematic graph.

The detach candidates are found by searching loops in the kinematic graphs, i.e. which edge in the load map have an infinite weight and create a detach action for each case.

The attach candidates are found by checking if two connectors are on the same position in space and if they are not connected, we can do an attach action.

The move candidates are more difficult to find as we have to compute a full model of the current structure in space, called *vertex_trans* that contains a list of matrix representing the transformation between the center of mass of a module (not always the first one) to all others after moving a servo. From this point we check for collisions, simply by testing that two dices doesn't occupy the same space, and if not, we get the new label(s) from new position of connectors after the move.

Signature objects are a list of *SignatureItem* that records the source, the target, a direction and the label. It is computed recursively from every node in the method *Calculate2* in the inner-class *SignatureCalculator* of *ConfigGraph3D*. The Depth First Search is in fact done in the method *Visit2* from the same class. The signature is computed many times, once per node and for each node once per symmetries of the module, and roombot modules have eight symmetries. It's the *Canonizer* class that handles this by changing labels in the graph depending on the new position of a module once turned around a symmetry axis. For roombots it is contained in three methods *DiceRotatePiHalfAroundY*, *DiceRotatePiHalfBackAruondY*, *DiceRotatePiAroundZ*, combining those methods permits to create all possible symmetries.

In this architecture, the main contributions of this project are on the creation of move candidates which includes a radical change in the attach candidates computation, the creation of a new model specific to roombots which leads also to change radically the *Canonizer* class and the addition of *Position* which includes the field in the *Connector* class but also the computation of labels from positions and no more from connectors which brings to numerous changes everywhere this was used.

4.2 Webots

To test and confirm the quality of the solutions we used Webots[25], a robot simulation software, to see how the roombots would behave with the solutions found. The architecture proposed by Mikaël Mayer[21] was used, it consists of having a central supervisor that sends messages to roombot modules and each module has an identical controller that basically does only one thing, waiting for message and execute the order associated once received. In our case, we use three type of messages, *lock*, *unlock* and *setPosition*. These controllers are called *sup* and *dice*. Mikaël Mayer also proposed a way to check if a move is possible in Webots in both directions in case an obstacle is on the road. For more details see his semester project[21].

Moreover, the algorithm outputs a .sol file for each solution found that contains a list of actions stating the type of action (M, A or D) the index of the roombot involved, and the connector for Attach or Detach actions or the servo positions for a Move action. This file is afterward parsed by the supervisor.

The algorithm outputs also two other files per run a .stat that contains some statistics like the number of graph visited, the size of the queue and many others, one line being creating per graph examined and a .summary that presents the number of actions in each solution found and the list of distances to the goal for each graph in each solution.

M. Asadpour gives us also another C++ code called “Configurer” that takes an xml file containing information on how modules are connected to each other and creates a .wbt(Webots file), .graph and .model for the specific connectivity stated in the xml file. This tool is really useful to create easily simulation situations and test them.

This method permits us to see in “reality” what would happen and make sure the solutions found are correct and doable.

Chapter 5

Results

This chapter covers the results obtained with the new version of the code. The first section includes what was done and which metric are used and why. The second section contains general comments on the performance and results obtained. And the last one make a comparison to the previous work of Masoud Asadpour.

5.1 What was done

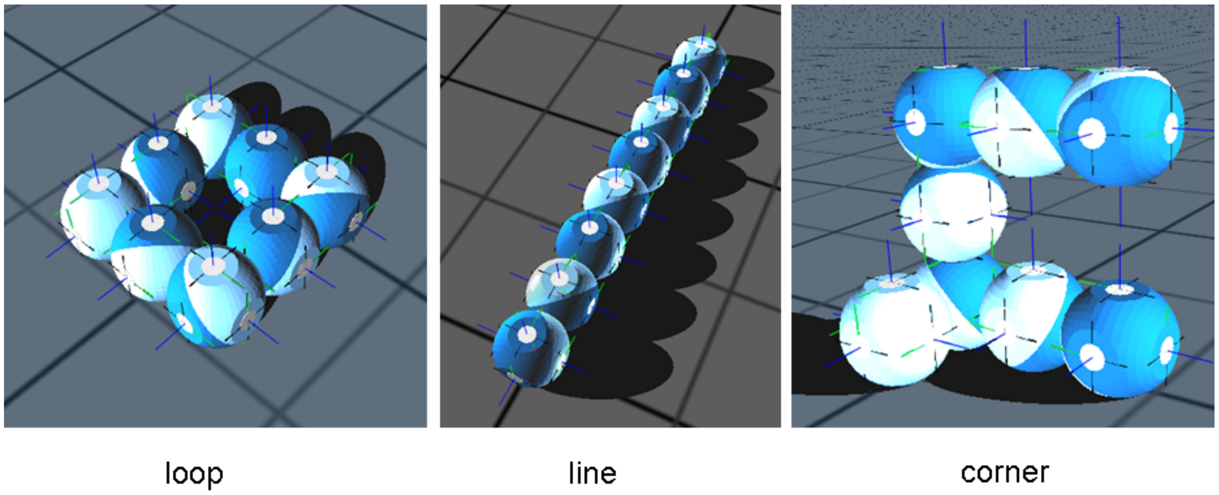
To test our algorithm, different configurations were chosen, a line, a loop and a shape with corners all of them are represented in Fig. 5.1. A quadruped shape was also chosen to be compared with the results obtained by Masoud Asadpour[1].

Four situations were chosen : form a line to a loop, from a loop to a cornered shape, from an cornered shape back to a line and finally from a quadruped to a line.

To measure the efficiency of the algorithm four measurements where done, the number of graphs visited before a solution is found, the number of graphs visited before the best solution is found, the number of actions in the first solution and the number of actions in the best solution. These measurements shows three main things:

- the quality of the solutions with the number of actions found

Figure 5.1: the three different configurations used in tests



- the time consumed with the number of graphs visited, it is preferred to a raw time measurement to free the result from a specific hardware
- the necessity to continue after the first solution found, wondering if we likely to find a better solution if we continue to run the algorithm or not

Every tests was run 50 times to have an average count of graphs and actions in both first and best solutions. All situations are run until the algorithm has check all possible shapes in space and has no new graph in the waiting queue. These experiments where done once with 3 modules (meaning 9 degree of freedom) and once with 4 modules (12 degree of freedom). Some result were obtained with a 5 modules configuration but the computation time to get to the end is too long to have 50 different runs completed.

5.2 Generalities

The Fig. 5.2, 5.3, 5.4 and 5.5 are the graphs representing the results obtained after running the algorithm in the different situations described in the previous section with four modules. The results are similar with three and five modules. The number of graphs visited before finding the first solution is in the left-up corner, the number of graphs visited before finding the best solution is in the right-up corner, the number of actions in the first solution is in the left-down corner and the number of actions in the best solution is in the right-down corner.

Remarks can be divided in different sections.

5.2.1 Difference between first and best solution

Most of the time the first solution is the best found. This is due, for a certain part, to the good distance metric that is used to direct the search but mainly to the fact that once a certain graph is checked and reached with a particular path, this path cannot be improved by further search. It is due to servo movements that need to be taken into account in this new version of the code and a graph doesn't say anything on how servos are oriented. Let us imagine we find a configuration c, we will compute the children nodes of c, derived from the different servo movements possible from c and find a solution that contain c and c' one of the children of c. Let's call a the action that brings c to c'. If we found a shorter path to c but with different servo positions we have no guarantee that we will be able to reach the configuration c' with these new servo positions and the action a will not more bring c to c' but to c'' a configuration that was not previously in the set of children of c and it will result in a unacceptable solution.

5.2.2 Difference between solutions in the same run

It is to notice that, in the results, most of the solutions found during one run are close one to the other. This is due to the fact that a graph cannot be seen more than once which cuts ways to perhaps find another solution to the same shape in space with a complete different list of actions. Such a solution would probably have to pass by a configuration that was already seen.

Figure 5.2: results given by running the reconfiguration algorithm from a line to a loop 50 times

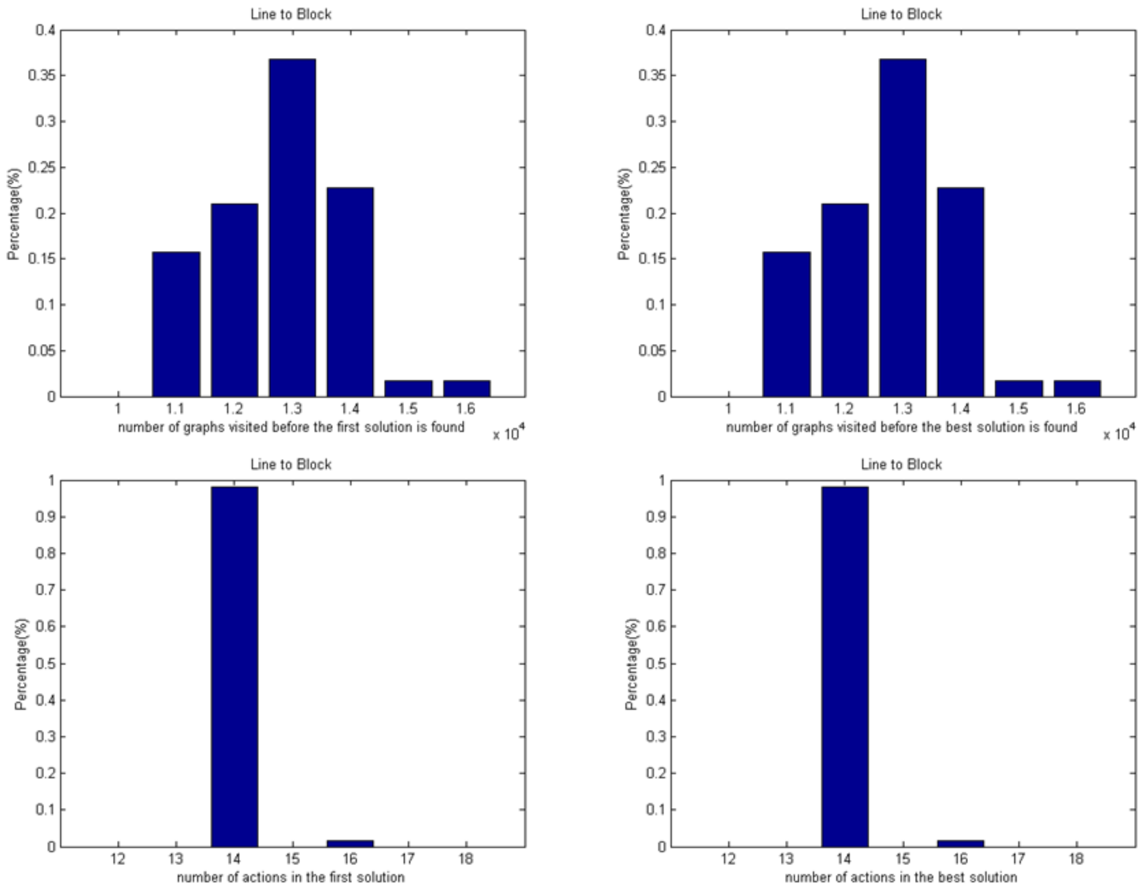


Figure 5.3: results obtained by running the reconfiguration algorithm from a loop to a cornered shape 50 times

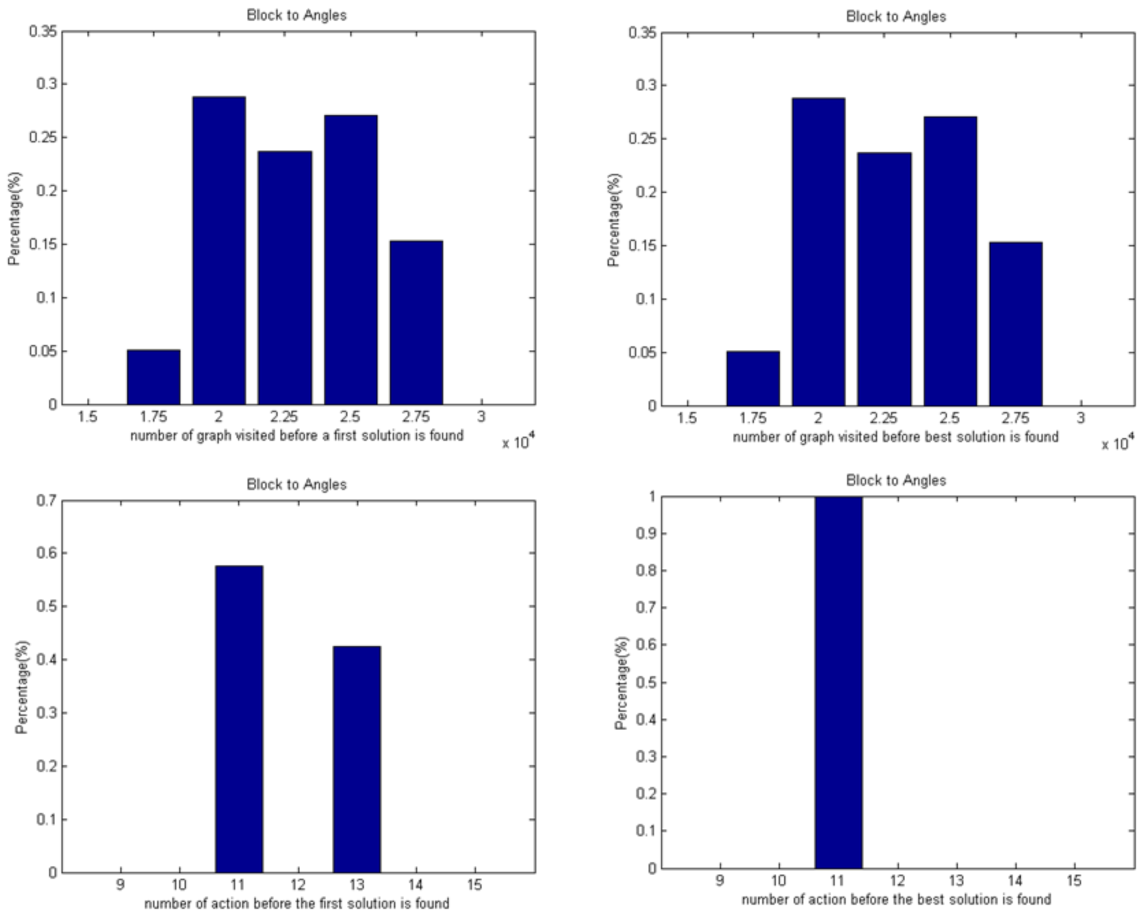


Figure 5.4: results obtained by running the reconfiguration algorithm from a cornered shape to a line 50 times

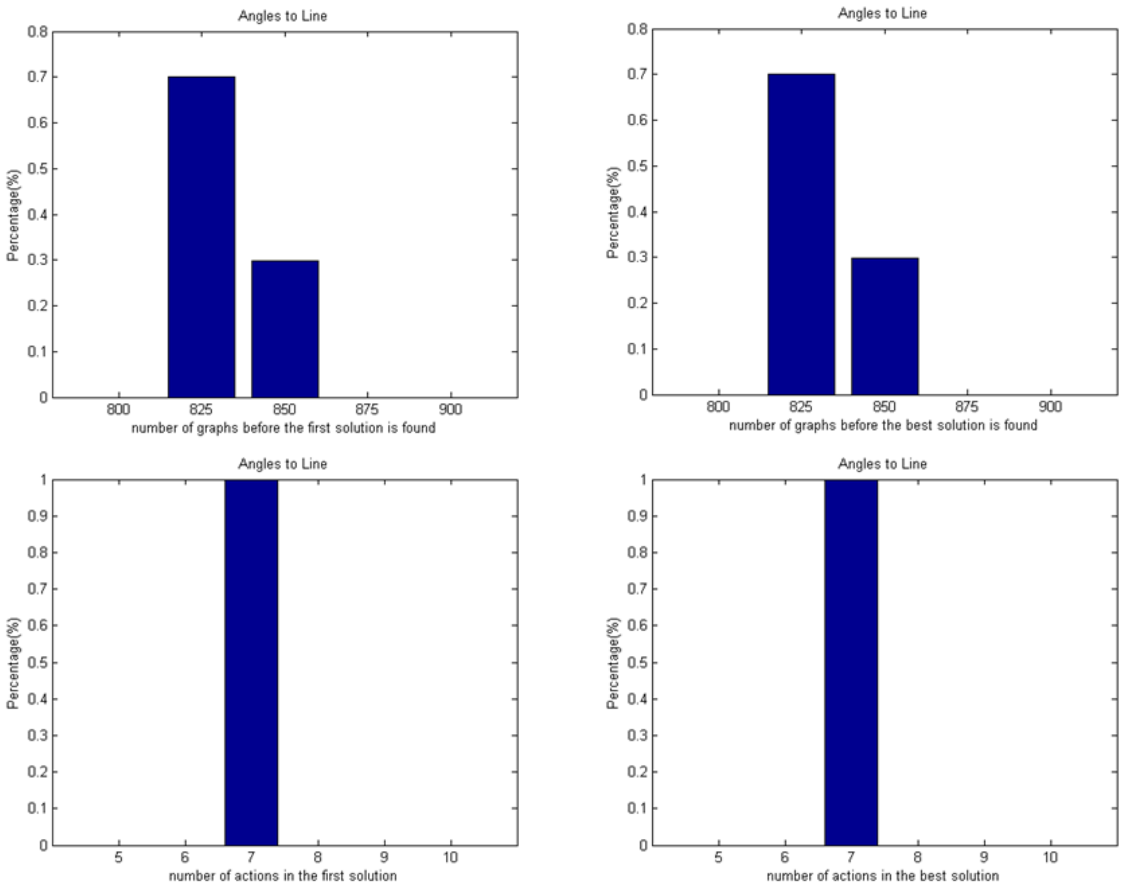
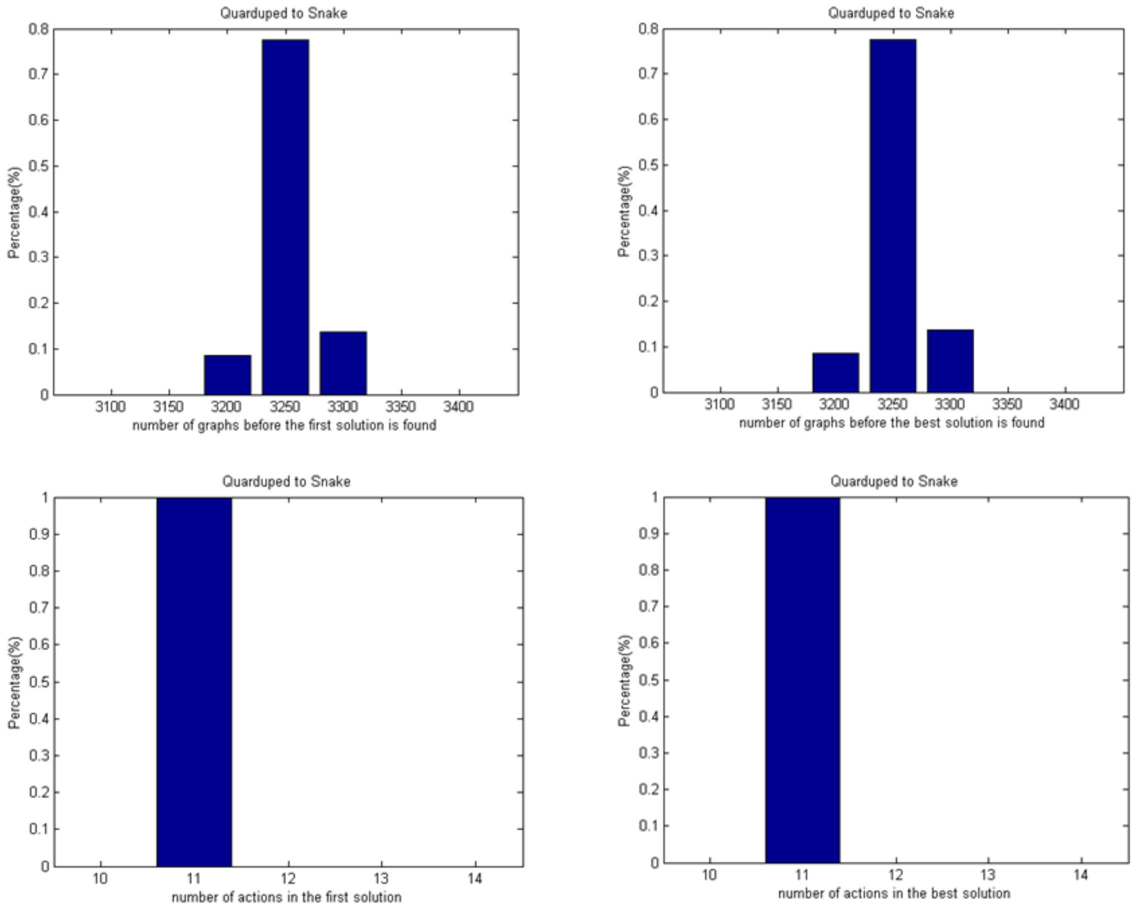


Figure 5.5: results obtained by running the reconfiguration algorithm from a quadruped to a snake 50 times



5.2.3 Time to reach a solution

The first solution is also found in very short time compares to full search done. If we look at the scale of the graphs we can see that the first solutions are found after looking at 10'000 graphs or even less than 1'000 in the case of the loop transforming in a line when more than 150'000 are checked in all cases. This shows the efficiency of the metric used as we have to visit less than $1/10^{th}$ of the configurations to find a solution.

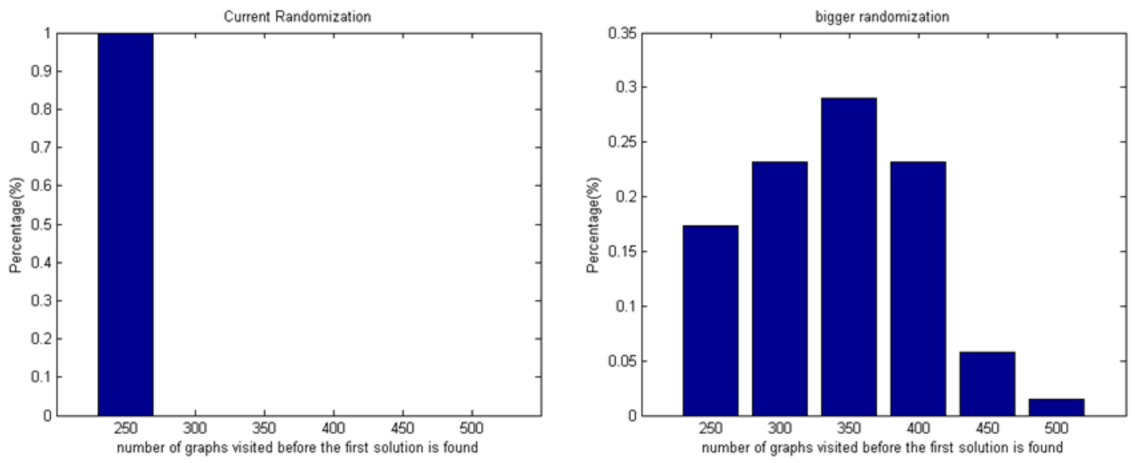
5.2.4 Quality of the solutions

But what can we say about the quality of the solution found? Is it close to an optimal or not? Well, the optimal cannot be computed due to the far to large search space, but the solutions found by hands are close to the one found by the algorithm. In the case of the quadruped to snake, the best solution by hands is exactly the same as the one found by the algorithm. In the case of the line that transform itself into a loop, the best solution by hand is 6, the algorithm found only a solution in 14 steps. These are both extremes, but it shows that the algorithm behaves as expected, not optimal but close to it in most of the cases.

5.2.5 Randomness

Another remark concerns the randomness factor, which is what changes the solutions found from one run to the next one. The only factor that change radically the kind of solution found is the randomness factor. Even with that, the quality of the solutions are quite similar from one run to the next one. This is basically a surprise but the solutions found are often similar one to the other and can be classified in two or three categories of solution easily recognizable. All solutions have also a quite similar number of actions in every case. This comes from two factors. First, the randomness is selective and happens only if there are identical distance between graphs, at a certain point when this is not the case, the algorithm chooses always the same way and cuts itself from a lot of possibilities even if the random choice rate always falls around 50% to 60% of all choices made. Second, we cannot find a shorter sub-problem and use it in the solution as explained before.

Figure 5.6: comparison between the current situation, on the left and the situation with more randomization on the right, the graphs are the number of graphs visited before the first solution is found



To see to which point changing the randomness factor could permit a wider number of solutions, we ran the algorithm with two new situations, the current one and a situation where the comparison function called to place every new graph in the queue return an incorrect result with a probability of 1/3. Fig. 5.6 gives the difference between both on the number of graphs visited before finding a first solution from a loop to a line with three modules.

It's clear that having a bigger randomization can influence the variety of solutions a lot and in the same time we can still notice, even the high probability chosen, that a large part of first solutions are in the left part of the graph, meaning it still give good enough solutions. A thorough search in this direction was not done in this project due to lack of time.

5.2.6 Running time

The last thing to say is that the running time is exponential to the number of degree of freedom and running a configuration with 3 roombots takes around

10-15 second, with 4 roombots it finishes in 10-15 minutes. With 5 roombots it's much more longer due to the fact that we begin to be a the limit of the RAM and working with virtual memory slow down incredibly the process.

5.3 Comparison to Asadpour's algorithm

The comparison with the work of M. Asadpour is not so easy because our results contains two mayor changes, the adaptation to roombots modules with their three degees of freedom and the addition of the Move Action. We can anyway say a certain number of things.

This work is based on the code given by M. Asadpour and the statistics collected are exactly the same, so comparison between quantities can be relevant if we keep in mind the transformations that was done and that can influence greatly these quantities.

First, the number of graphs looked at before finding a solution is far less than in the previous work. If we take the example of the quadruped to snake given in the paper[1], it's interesting to see that the scale is 100'000, and on the other hand, in our case, a solution is found in general around 10'000 and we search up to 150'000 with four roombots. Our algorithm is basically finding a solution in a shorter time which permits us to go up to 12 degree (4 roombot modules) of freedom where M. Asadpour stopped at 8 (8 YaMoR modules).

We don't know if this comes from:

- the fact that there is less robots meaning less nodes in the graphs
- the fact that a smaller number of actions are needed to reach the solution due to the new freedom given by the roombots
- the configuration space chosen.

Any further comparison would be difficult, due to the big change in the code.

Chapter 6

Future Works

This chapter contains some possible future works that would improve or use with more efficiency the current program. These include the improvement of the collision detection which is really simple for the moment, a possible division of the structure in small problems that could be compute in less time and an improvement of the search strategy to have more differentiate solutions.

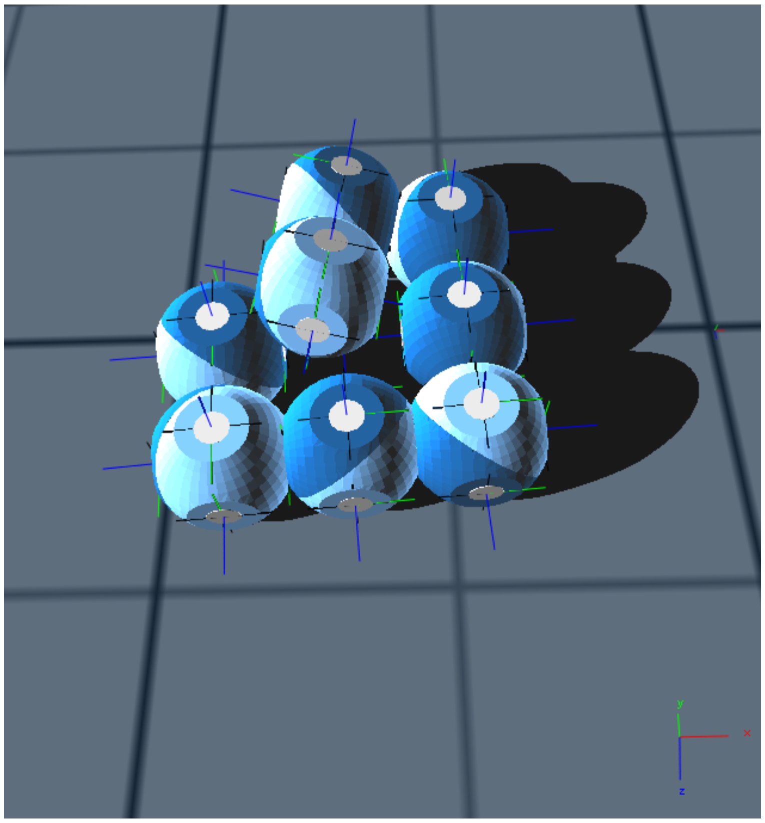
6.1 Improvement of Collision Detection

The collision detection is, for the moment, done by checking that the final position in which a modules try to reach is empty. There is no check to see is the movement is effectively doable and that there is no obstacle between the start and end positions.

This small problem brings solutions that basically are not doable in practice. Fig. 6.1 gives a simple example of a situation where the roombot cannot move due to the fact that on both directions there are obstacles (the other roombots) but the place where it goes is free.

There would be several ways to handle it. The problem comes from the fact that the computation of collision is done many time during the computation of possible movements. We chose to let it simple to not over complicate the algorithm but as we know precisely where each module is we

Figure 6.1: example of a collision between roombots during the execution of a found solution



can use a known algorithm for this kind of problem.

This situation could be also handled when the solution is running, as obstacles are also roombots, we could try to remove them from the way and put them back after. In the Fig. 6.1 this could include to lift the left roombot, so the movement in progress can be finished and put it back afterward.

Another problem that arises in simulations was the fact there is a ground and gravity. The code computes the different possible movements as if there is no obstacles nor gravity, it checks if the number of roombots lifted is not too large, nothing more. The ground can be an obstacle for some movements which is never taken into account.

6.2 Division of structure

One possible way to improve the time of computation is to divide the current structure in smaller problems. It will take less time to compute two groups of three roombots than a group of six. The problem becomes where to cut and how to be sure that the structure will be attached correctly to the rest of the modules.

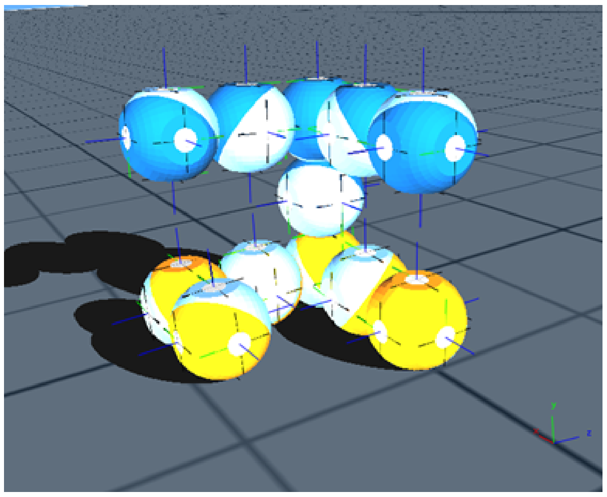
In the code, this means that we should consider that a certain connector in not available, because it is attached to the second part of the structure. In the same time we have to be sure that the structure will be oriented correctly compare to the other part. One possibility would be to add a node in the graph that represents the structure to clearly say where is the link between both parts. Fig. 6.2 shows the representation of the graph and structure in reality.

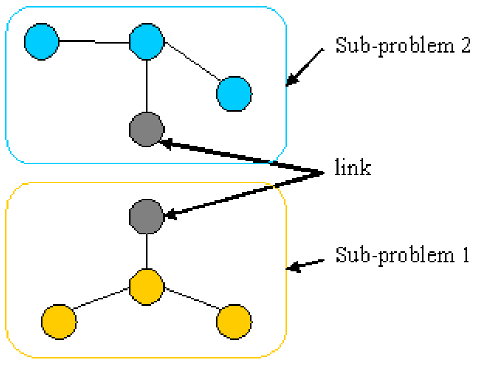
It would be interesting to adapt the current code to take that into account but also to see how a structure can be cut into parts automatically and if this method effectively permits to find a solution and at which cost.

6.3 Improvement of search strategy

A third possible improvement will be to change the current search strategy. For the moment, as describe in the first part of this report, the search strategy

Figure 6.2: On the left, the roombots in a certain configuration in space and on the right, two graphs representing a way to cut this into two part





consists of taking the closest possible graph to the goal as next one and use randomness when there are more than one possible choice.

The results given in the previous chapter shows that even with different random seeds, we got similar results in a certain number of cases, but that a simple change in the randomness can widely change the range of solutions.

One way to follow would be to change the search strategy and include some stochastic method like Simulated Annealing or Genetic Algorithm to get more randomness and search part of the tree that could be discard other way, without completely destroying the current quality of the solutions.

Chapter 7

Conclusion

This work shows a method adapted for roombot modules an existing approach to reconfigure a group of modules by two major changes. This approach is a centralized approach, still based on graph edit distance but trying to take into account specific positioning in space. This brought to redefine what was a configuration.

Empirical results show that our adapted algorithm can cope up to 12 dof/4 modules in reasonable time (x hours). Different remarks where done, especially the closeness between first and best solutions and between solutions of different runs due mainly to the randomization chosen. The quality of the solutions was also discussed and shows that we get near optimal solutions in many cases. Our adaptation is also able to handle more dofs than Asadpour[1](12 dof/4 modules against 8 dof/8 modules).

Possible future works could concern a more efficient collision detection, different search strategy or a way to break a structure in different parts that could be computed separately to get faster results.

This work is another step in the direction of a bigger project/aim: having "intelligent" pieces of furniture that are able to reconfigure autonomously.

Chapter 8

Acknowledgements

I thank the BIRG and especially my supervisor Alexander Spröwitz for the support and help given during this work and of course Masoud Asadpour and his team for giving me the opportunity to work on their project. I would like also to thank Mikaël Mayer and Alexandre Tuleu for their help and support through all this work.

List of Figures

Bibliography

[1] M. Asadpour, A. Spröwitz, A. Billard, P. Dillenbourg and A. J. Ijspeert, “graph signature for self-Reconfiguration Planning”, 2008.

[2] M. Weiser, ”The computer for the 21st century,” *Scientific American*, vol. 265, no. 3, pp. 66-75, Sep 1991.

[3] R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, and A. Ijspeert, “Exploring adaptive locomotion with YaMoR, a novel autonomous nodular robot with Bluetooth interface,” *Industrial Robot*, vol. 33, no. 4, pp. 285-290, 2006.

[4] H. Kurokawa, K. Tomita, A. Kamimura, S. Murata, Y. Terada, and S. Kokaji, “Distributed metamorphosis control of a modular robotic system m-tran”, in *Distributed Autonomous Robotic Systems(DARS) 7*, Springer, 2006.

[5] D. Rus and M. Vona, “A physical implementation of the selfreconfiguring crystalline robot.” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2000, pp. 17261733.

[6] E. Klavins, S. Burden, and N. Napp, “Optimal rules for programmed stochastic self-assembly” in *Proc. Robotics: Science Systems 06*, Aug. 2006.

[7] E. H. Ostergaard and H. H. Lund, “Evolving control for modular robotic units,” in *Proceedings of CIRA03, IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Kobe, Japan, 16-20 July 2003, pp. 886892.

[8] Preethi Srinivas Bhat, James Kuffner, Seth Copen Goldstein, and Siddhartha S. Srinivasa, "Hierarchical Motion Planning for Self-reconfigurable Modular Robots," In *IEEE/RSJ International Confernce on Intelligent Robots and Systems (IROS)*, October 2006

[9] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji, "A 3-d self-reconfigurable structure" in *Proc. of the IEEE Int. Conf. on Robotics and Automation*

[10] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian, "Useful metrics for modular robot motion planning" *IEEE Trans. on Robotics and Automation*, vol. 13, no. 4, pp. 531545, 1997 (ICRA), 1998, pp. 432439.

[11] R. Fitch, Z. Butler, D. Rus, "Reconfiguration planning for heterogeneous self-reconfiguring robots", in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003 (IROS 2003)*, pp. 2460- 2467 vol.3, Oct. 2003

[12] B. Salemi, M. Moll, and W.-M. Shen, "SUPERBOT: A deployable, multi-functional, and modular self-reconfigurable robotic system," in *Proc. 2006 IEEE/RSJ Intl. Conf. Intelligent Robots Systems*, Oct. 2006, pp. 36363641.

[13] A. Spröwitz, A. Billard, P. Dillenbourg and A. J. Ijspeert, "RoombotsMechanical Design of Self-Reconfiguring Modular Robots for Adaptive Furniture", 2008.

[14] D. Dewey, M. Ashley-Rollman, M. De Rosa, S. Goldstein, T. Mowry, S. Srinivasa, P. Pillai, J. Campbell,"Generalizing metamodules to simplify planning in modular robotic systems", in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 1338-1345, Sep. 2008

[15] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Trans. on Robotics and Automation*, vol. 13, no. 4, pp. 531-545, 1997.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theroy of NP-Completeness.* San Francisco, W. H. Freeman, 1979.

[17] P. Bhat, J. Kuffner, S. Goldstein, S. Srinivasa, "Hierarchical Motion Planning for Self-reconfigurable Modular Robots", 2006.

[18] V. Zykov and P Williams, Computation Synthesis Lab, Cornell University, http://www.molecubes.org, 2008

[19] D. Duff, M. Yim, and K. Roufas, "Evolution of polybot: A modular reconfigurable robot," in *Proc. of the Harmonic Drive Intl. Symposium and Proc. of COE/Super-Mechano systems Workshop*, Japan, Nov 2001.

[20] Norbert Streitz and Peter Tandler and Christian Mller-Tomfelde and Shiníchi Konomi, "Roomware: Towards the next generation of human-computer interaction based on an integrated design of real and virtual worlds," in *Human-Computer Interaction in the New Millenium*, J. Carroll, Ed. Addison-Wesley, 2001.

[21] M. Mayer. "Roombot modules - Kinematic Considerations for Moving Optimisation", 2009.

[22] X. Jiang and H. Bunke, "Optimal quadratic-time isomorphism of ordered graphs." Pattern Recognition, vol. 32, no. 7, pp. 12731283, 1999.

[23] A. Tuleu. "Cpg locomotion for roombots", 2009.

[24] Biologically Inspired Robotic Group, EPFL, http://birg.epfl.ch.

[25] Webots. Commercial Mobile Robot Simulation Software, commercialized by Cyberbotics, http://www.cyberbotics.com.

[26] Boost C++ Libraries. http://www.boost.org.