Extension of an evolutionary design algorithm to complex tasks

Spring Term 2013
Contents

Abstract iii

1 Introduction 1
  1.1 Motivation ....................................................... 1
  1.2 Previous Work ................................................ 2
  1.3 Goals .......................................................... 3

2 Algorithm Improvements 5
  2.1 Reinforcement .................................................. 5
  2.2 Deflection ...................................................... 7
  2.3 External Load .................................................. 8
  2.4 Obstacle Avoidance .......................................... 9
  2.5 Other Improvements ......................................... 10

3 Analysis and Discussion 11
  3.1 Population Size ................................................ 11
  3.2 Force ........................................................... 13
  3.3 Weights in Fitness Function ................................. 15
  3.4 Parallel Computing Toolbox ................................ 19

4 Future Work 21
  4.1 Approach to buildability .................................... 21

5 Conclusion 23

A Algorithm Manual 25
  A.1 Organism encoding ........................................... 25
  A.2 List of functions ............................................. 25
  A.3 evolalg.m ...................................................... 27
  A.4 checkstability.m ............................................ 27
  A.5 group.m ........................................................ 28
  A.6 fitnessfunction.m .......................................... 28

Bibliography 29
Abstract

In this project, genetic algorithm for evolutionary design of cube structures was used. Its main purpose is to evolve structures that can reach a specific target. In order to make it more functional, improvements were made. A reinforcement procedure was implemented in order to obtain more stable and robust structures. Furthermore, deflection of structure was included in fitness function with respective weight. In order to get more realistic (and useful) structures external load and obstacles were also introduced. As a result, evolved structures were more stable, stiffer and realistic. Also, external conditions like obstacles and forces were simulated. In the end, analysis was done to inspect the behavior of the algorithm with respect to different input parameters.
Chapter 1

Introduction

1.1 Motivation

The science of robotics is expanding rapidly each day. Since it is a multidisciplinary field, new ideas and solutions from many branches like electronics, automation, mechanical design, etc., are providing the necessary ingredients for such fast development. Amongst many, one of the popular research topics in this field is evolutionary robotics, specifically robotic body extension. Designing and building robots that can change their functionality and adapt their own body to different situations, is a challenging and demanding task. This topic has already been exploited at BIR-Lab [1], [2], but this is just the beginning considering what can really be achieved with further improvements. In this project the main emphasis is on developing an algorithm to provide a robot with the design of structure for a defined task.

The design of structure would have to consider all the limitations that the environment imposes. This means that the design should be realistic (so that two points do not take the same space in defined coordinate frame, that is, that the structure is not colliding with itself), stable (if the structure is built it should be able to sustain its own weight) and adapted to the environment (if there are obstacles, the structure should go around them). Pollack and Funes [3], [5] managed to develop such an algorithm that provides the design of structure for a given task. They used Lego cubes as building blocks and for a developed design they built structures by hand, as shown in figure 1.1. For the left structure, the task was to reach the defined target point while for the right one, it was to sustain the external load applied at specified position. Developing an algorithm that can behave similarly was the goal.

Instead of building structures by hand, at BIRLab there is a robot that is capable of achieving this task. It is a 6DOF robot that uses wooden cubes as building blocks.
and hot melt adhesive to connect them together. In order to provide the robot with a design, the algorithm had to be adjusted to these conditions. The ultimate goal was to develop an algorithm that is able to provide the robot with a structure that is suitable for any defined task.

1.2 Previous Work

The initial evolutionary design algorithm was built by J. Seitz [6]. A genetic algorithm as shown in figure 1.2 was used in order to evolve different structures. Each organism represents one structure and each structure is encoded by three variables: matrix that contains starting vectors of each branch, matrices that contain directions where each branch evolves (for example \([1 \, 0 \, 0]\) means that the next cube is evolving in x direction) and vector containing parent branches (each new evolved branch has to be connected to one of the previous ones - its parent branch). More details on encoding of organisms are presented in J. Seitz’s Bachelor’s Thesis [6]. The algorithm starts with an initial (current) generation from which the best organisms (elite) are directly copied to a new generation. The rest of the organisms are mutated and cross overed. Afterwards, each new organism needs to be verified by checking stability and collision inside the structure. If these conditions are met, the new organism can be copied to a new generation. When the number of organisms reaches the population size, a new generation is sorted by fitness (so that the best organism is first in the generation) and it becomes the current generation. This process is repeated until the value of fitness of the best organism in current generation reaches the specified value.

In order to check stability of the structure, a truss algorithm was used [4]. Each cube is approximated by 28 elements (intersections of these elements are called
nodes, and there are 8 of them for each cube) and each connection (glue section) by 16 elements, as shown in figure 1.3. The algorithm outputs force in each element. From these forces equivalent stress in each section is calculated. If this value exceeds critical stress, the structure is unstable and in that case mutation or crossover is done again to get a new organism. This sequence is repeated until stability condition is fulfilled.

Finally, number of cubes, distance from given target and maximum stress in the structure were taken into account when calculating the fitness function.

Figure 1.4 shows a simple example of evolution process. For a specified target (red cube) the algorithm was supposed to evolve a structure that must be stable. In left figure there is only one branch evolved. After a certain number of generations the algorithm modified this branch in order to reach further (middle figure). Finally, the algorithm developed a structure that was able to reach the target (right figure).

1.3 Goals

In order to improve the existing algorithm for evolution of cube structures, the following goals were defined:

1. Implement a reinforcement procedure so that the algorithm can evolve more stable structures by connecting different branches, which results in lower equivalent stress for each HMA connection.

2. Include deflection into the fitness function and adjust weights accordingly in order to get better and more realistic structures.

3. Introduce obstacles in a simulated working space.

4. Simulate external load at a certain location of the structure.

5. Analyze the behavior of the algorithm (is it able to converge, how long it takes to reach the given goal, etc.) for different input parameters.

These topics are going to be discussed in the following chapters.
Chapter 2

Algorithm Improvements

In first chapter, improvements of the algorithm are introduced. Each improvement is presented separately and its influence on the algorithm performance is discussed. In order to demonstrate the following results more clearly and to save a lot of simulation time, 10 times higher density than one of wood was used. This way structures are smaller and more compact, and thus, it is easier to see the obtained results. This is just a matter of implementation, problem can easily be scaled to realistic values without any change in functionality of the algorithm.

2.1 Reinforcement

Figure 2.1: Lego structure with developed reinforcement, adapted from [5]. Branch 1 is the main branch with load applied to it. Branch 2 evolved as a reinforcement and connected with the main branch.

The first goal was to develop a procedure to build reinforcements since the existing algorithm was not able to. With reinforcements, evolved structures would be more stable, stiffer, could reach further and they could sustain much higher load. Figure 2.1 shows a structure with such reinforcements. Branch 1 is the main branch, with a certain load applied to it. In order to sustain that load branch 2 evolved and connected with branch 1. This way they form a stable unit that is able to carry the load. Branch 2 also acts as a counter weight in order to balance the whole structure. This kind of behavior is desirable and our evolutionary design algorithm should behave similarly.
Chapter 2. Algorithm Improvements

After analyzing the algorithm it was clear that it was possible to build reinforced structures but it actually never happened. Evolved structures just went straight for the target without any need of reinforcement. This was possible because of the wrong stress calculation. These structures were stable even though they should not have been. Figure 2.2 shows two examples for two arbitrary targets. No reinforcement was built in these cases. In order to improve this, stability check function was modified. Young’s modulus of wood was changed to $9 \times 10^9$Pa and Young’s modulus of glue to $8.9 \times 10^9$Pa. Normal and shear force component calculation was modified and equivalent stress calculation was corrected. Also, compression was neglected because we assume that the structure will only fail due to tension. As a result, reinforced structures evolved. Figure 2.3 shows the same two examples as figure 2.2, but with implemented improvements. In both structures, left and right, there is a branch that acts as a counter weight. In the left picture, the main branch is reinforced with another branch so that it can sustain higher load. In the right picture, there is a bridge connecting the main branch with counter weight branch, making the whole structure more stiff and stable.

With these improvements, the algorithm was able to develop structures with reinforcement, and by that these structures became much more realistic and useful for real-world application.
2.2 Deflection

If the target is far from the origin, the algorithm will develop structures with long branches. Even if the structure is stable, significant bending can occur. In order to minimize the deflection, the fitness function was updated. The truss algorithm outputs displacement of every node. It is possible to find maximum displacement and to include it in the fitness function with respective weight. This way evolved structures look more compact and bending of long branches is minimized.

Figure 2.4: Structure without deflection included in fitness function. Evolved structure (left) and the same structure displayed as a truss with blue lines representing the flexion (right), max. deflection: 0.0143 m.

Figure 2.5: Structure with deflection included in fitness function. Evolved structure (left) and the same structure displayed as a truss with blue lines representing the flexion (right), max. deflection: 0.0025 m.

Figure 2.4 shows a structure with reinforcement and with long branch that has deflection of 0.0144 m on the edge. In order to reduce this deflection, fitness function was updated as discussed above. As a result (figure 2.5, same target as before), the structure was bigger, with more massive counter weight and a bridge connecting it to the main branch. This way deformation was reduced 5 times.

In a real-world application, it is reasonable to want to minimize the deflection, especially if there are obstacles in the workspace (in order to avoid hitting one). With this update, evolved structures are more robust with bigger branches acting as counter weights and with smaller deformations. It is also possible to define
critical deflection (maximum allowable) and check if maximum deflection is higher than this value. If this is the case, the structure is declared unstable and therefore mutation or cross over is done once again. Also, this way the influence of deflection in fitness function is limited (by normalizing deflection with critical deflection and using this as a parameter in fitness function, the range of values it takes is between 0 and 1).

2.3 External Load

In order to make the algorithm even more realistic, a possibility to simulate external load was implemented. For a given input force and its position, the algorithm evolves a structure that can withstand that load. No force is applied if the specified position is not reached. If there is a cube at that position, the algorithm finds nodes which correspond to it and applies \(\frac{1}{8}\) of force (divided by 8 because each cube has 8 nodes) to each of them.

![Figure 2.6: Structure without external load (left), with lower force \([-0.4,-0.4,-0.4]\) applied at target position (center) and with higher force \([-2,-2,-2]\) applied at target position (right). Same target was used in all three examples (force is represented by the blue arrow).](image)

Figure 2.6 shows the same example but with different forces applied. Left structure was without any force, middle structure had lower force and right structure had higher force applied at the target position. In order to withstand higher forces, the algorithm evolved bigger structures with more reinforcements that are much stronger.

Clearly, external load influences the shape of the structure. With higher force applied, the structure becomes bigger and stronger. The possibility to simulate external load makes the algorithm more useful and applicable in a wider range of situations. For example, if such structure was used as an extension of a robot end-effector that has to pick up some kind of an object, the mass of this object could be simulated as an external load.
2.4 Obstacle Avoidance

In order to be able to simulate obstacles in the workspace, the algorithm was modified. For given obstacle positions, the algorithm checks for collision between structure and obstacles. If there is collision between them, the structure is discarded and a new one has to be evolved. This process is repeated until there is no collision.

![Figure 2.7: Structures evolved using obstacles. Structure with base surrounded by obstacles (left), structure reaching over a wall of obstacles (center) and planar structure with obstacles in plane in front and behind it (right). Blue arrow points to the target position.](image)

In the first example (figure 2.7, left) obstacles surrounded the origin of the structure. The algorithm was able to find a way to reach the target. Problem can occur if target position is below the origin and the starting number of cubes is not sufficient to reach out of obstacle formation. In this case the algorithm can only add cubes upwards but then distance from the target increases and number of cubes increases and thus fitness function of that structure decreases (due to larger number of cubes). This way the best structure in each generation is the one that only has a cube in the origin because it is closest to the target and its fitness is the highest one. This problem can be avoided by choosing appropriate parameters for initial number of cubes or adjusting the number of cubes that can be added in mutation (in both cases number of cubes has to be big enough so that the algorithm can exit the obstacle formation). In the second example (figure 2.7, center) obstacles made a wall and the target was behind it. In order to reach the target, the algorithm managed to evolve a structure that goes above the wall. It had a massive body over the origin in order to lower the deflection. In the third example (figure 2.7, right) obstacles were in plane in front of the structure, in plane behind the structure and in line below the origin of the structure and as a result planar structure evolved.

With possibility of defining obstacles, the algorithm can simulate a real-world workspace that contains different objects. Each object can be substituted by a set of obstacles. This way functionality of the algorithm is improved.
2.5 Other Improvements

Following improvements were also made:

- Parfor loops from parallel computing toolbox were implemented. This makes the algorithm evolve organisms (structures) from one generation simultaneously and thus it is much faster (more details in section 3.4).

- Writing of organisms in mat files was changed by storing them in one variable. Original algorithm saved each organism in a separate file and with this modification all of them are contained in one variable.

- Probability of using a certain organism for mutation or cross over was modified, so that probability distribution function is: 
\[ f(x) = \frac{3}{(N-1)^3} \cdot (x - N)^2 \]
where \( N \) is population size.

- Check for existing organisms in generation was added. After mutation or cross over, if the created organism already existed in the current generation, it was discarded and a new one was created. This only works when parfor loops are not used (more details in section 3.4).

- Stability check for initial population was added. Since stability of initial population was not considered, if one would choose a big number of starting cubes it could happen that every organism from initial generation is unstable. Then the algorithm would try to mutate and do cross overs to get a new generation but since none of the structures from previous generation were stable, evolving a stable structure would be almost impossible (highly improbable) and thus the algorithm would get stuck trying to evolve a new structure.

- Functions for timing were implemented.

- Control functions were added so that it is easier to use the algorithm. User can choose which option to use. More details in appendix A.3.

- Number of initial cubes was changed. Instead of specifying one number, a range is defined and the algorithm randomly chooses a number in that range (uniform distribution).

- Number of connections between cubes in structure was included into fitness function with respective weight. For smaller density of cubes the algorithm is able to build much bigger structures. This parameter was used in order to make structures more compact.

- Check for adding a new branch was added. For example, if the algorithm mutates an organism from figure 2.7 (right) by adding a new branch with more than one cube and if any of the cubes surrounding a hole in the structure (either of two holes) was specified as a place to start a new branch, the first cube would fill that hole, but then the second cube could not be placed anywhere. The algorithm would try to place it in any direction (randomly) for an infinite amount of time. In order to avoid this, check for adding a new branch was added. If the algorithm takes too much time (runs the same mutation for many iterations) this mutation gets discarded and a new mutation is used instead. Same problem can occur in 3D but it is less probable.

- Initial population fitness function calculation was corrected. In the original algorithm stability of initial structures was not considered so maximum stress and maximum displacement were always zero and thus fitness function value was not correct.
Chapter 3

Analysis and Discussion

After implementing all improvements, an analysis was made. Influence of different parameters like initial population size, external load magnitude and weights in fitness function were considered. Also, timing and convergence, with and without using parallel computing toolbox, were analyzed. The results are presented in this chapter. Again, 10 times higher density than one of wood was used for the same reasons as in previous chapter.

3.1 Population Size

The first analysis was about the influence of population size on the algorithm performance. Population size was varied from 10 up to 100 for same parameters and target position, and for each population size 10 algorithm runs were conducted. Worst case fitness, standard deviation of fitness and average values (for these 10 runs) of final fitness, convergence generation (generation in which the algorithm reached the desired fitness value) and timing are displayed in table 3.1. Target position [5 5 5] and 100 generations were used.

<table>
<thead>
<tr>
<th>Population size</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average convergence generation</td>
<td>98.20</td>
<td>74.90</td>
<td>69.40</td>
<td>68.30</td>
<td>65.00</td>
<td>60.20</td>
<td>66.60</td>
<td>54.70</td>
<td>51.10</td>
<td>56.60</td>
</tr>
<tr>
<td>Average fitness</td>
<td>81.48</td>
<td>96.59</td>
<td>95.97</td>
<td>95.60</td>
<td>97.10</td>
<td>97.08</td>
<td>94.45</td>
<td>98.20</td>
<td>98.10</td>
<td>94.50</td>
</tr>
<tr>
<td>Average time</td>
<td>187.9</td>
<td>223.2</td>
<td>340.2</td>
<td>387.9</td>
<td>534.7</td>
<td>486.8</td>
<td>609.7</td>
<td>708.8</td>
<td>518.9</td>
<td>755.93</td>
</tr>
<tr>
<td>Worst Case Fitness</td>
<td>55.92</td>
<td>82.36</td>
<td>87.30</td>
<td>82.38</td>
<td>87.28</td>
<td>87.39</td>
<td>82.62</td>
<td>98.06</td>
<td>98.00</td>
<td>76.15</td>
</tr>
<tr>
<td>Std of Fitness</td>
<td>11.51</td>
<td>5.00</td>
<td>4.55</td>
<td>5.78</td>
<td>3.45</td>
<td>3.41</td>
<td>6.10</td>
<td>0.13</td>
<td>0.09</td>
<td>8.07</td>
</tr>
</tbody>
</table>

Table 3.1: Algorithm performance with respect to different population sizes. Maximum generation is 100 and maximum fitness is 100.

The first row of table 3.1, showing the average convergence generation, is displayed as a histogram in figure 3.1. Average fitness was around the same value for all population sizes except for the smallest one. For population of 10, the algorithm performed badly (for this specific problem). In general, for a certain problem, there is a population size above which the algorithm will perform equally well (with respect to fitness function).
The second row of table 3.1 is displayed as a histogram in figure 3.2. As expected, with bigger population size, average convergence generation is decreasing until a certain value is reached. Population of 70 is an exception probably due to randomness of the algorithm (there is always a possibility that the algorithm might get stuck for a certain amount of generations, for example if a specific mutation is necessary in order to develop a reinforcement, the algorithm might spend few generations trying to find it). After population size of 70, convergence generation is decreasing until size of 100 when it increases again. If population size increased even more the result would probably be around the same value (55 generations). This is because the structure has to grow into one direction and growth rate is limited. Even if all the mutations and cross overs are “perfect”, there is still a minimum number of generations that the algorithm needs in order to converge.

Finally, the third row of table 3.1, showing the timing, is displayed as histogram in figure 3.3. It is expected that the time it takes the algorithm to converge will increase with population size. Again there can be some outliers (like population size 60 and 90) due to randomness of the algorithm, but in general, if population size is increased, total running time will be bigger.
The last two rows of table 3.1 represent worst case fitness value and standard deviation of fitness. In worst case scenario, all of the algorithm runs performed good compared to the one with population size of 10. It can also be observed that the algorithm managed to converge in every run only for population sizes of 80 and 90 and that it did relatively bad (compared to others) in one case for population size of 100. Again, these values are result of the algorithm randomness mentioned before.

For a specific problem that was used in obtaining these results (table 3.1), the population size of 20 was the best choice. This is because it gave reasonably good fitness value compared to the size of 10 and it was by far faster than any bigger population size. For a further away target, bigger population size would be more reasonable choice because all of these histograms would look slightly moved to the right.

Since this algorithm depends a lot on random processes, a certain population size cannot guarantee that it will be better than the other. In general, there is dependence between population size and the algorithm performance, but it differs for various problems.

### 3.2 Force

In order to analyze the behavior of the algorithm with respect to different external loads, two different tests were conducted. In the first one, force was defined as a function of generation. This means that the force was increased in each generation. Target position \([4, -4, 4]\) and population size of 80 were used. If the structure did not reach the target there was no force acting on it. When the target was reached, force was applied at target position. The result is displayed in figure 3.4. In this case no elites were used because the algorithm stores maximum stress and deflection for each organism and if that organism was just copied to the next generation maximum stress and deflection would stay the same although the force would change. That is why there are small drops in fitness in first 25 generations. From 25th to 45th generation the algorithm got stuck and could not reach the target. This is because it needed to develop reinforcement and in order to do that optimization of the structure (lowering maximum stress) was needed. Around 45th generation the
algorithm managed to reach the target. Then around 60th generation the structure was no longer able to withstand the load and reach the target (while satisfying stability conditions at the same time), since the force was increasing and the structure was not improving fast enough. Optimization was necessary once again and around 80th generation the structure managed to reach the target once again. Afterwards, no changes occurred. The algorithm managed to make necessary optimizations to withstand the increasing force (massive structure evolved). The 400th generation, which corresponds to force vector of [-12, -12, -12] was the final one. In the second test, only one specific force vector was used in each algorithm run. The first algorithm run was without any force applied. Then, in each subsequent run force was increased if the algorithm could converge and this sequence was repeated until it could not converge anymore (for a specific force). Population size of 40, target position [4, -4, 4] and maximum generation of 1000 were used. The results of this test, which was repeated 10 times, are shown in table 3.2. First column represents the algorithm run and second column force vector for which the algorithm was able to evolve a structure. Biggest force achieved was [-10, -10, -10]. This means that the algorithm performed worse than in the first test where higher force could be sustained.

Figure 3.4: Fitness function with developed structures for a force increasing with each generation. Structure 1 was not able to reach the target because no reinforcements were evolved. Structure 2 evolved reinforcements and was able to withstand the force [-1.8, -1.8, -1.8]. Structure 3 could not reach the target anymore because the force was too high. Structure 4 evolved reinforcements and was able to withstand force up to [-12, -12, -12].

When the algorithm tries to evolve a structure for a certain task (target and force applied) it is much more difficult to start from the beginning than to optimize an already existing structure. This is because the existing structure is much closer
to the solution (it already has reinforcements, which are hard to develop, and also maximum stress is lower). It is also possible to evolve a completely new structure, but then the algorithm would need much more time (generations).

Table 3.2: Maximum forces obtained for force increasing with each algorithm run. In each trial (left column), maximum force that the algorithm managed to reach is displayed in right column.

<table>
<thead>
<tr>
<th>Algorithm run</th>
<th>Force vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-10 10 -10]</td>
</tr>
<tr>
<td>2</td>
<td>[-99 -9]</td>
</tr>
<tr>
<td>3</td>
<td>[-8.5 8.5 -8.5]</td>
</tr>
<tr>
<td>4</td>
<td>[-8.8 -8]</td>
</tr>
<tr>
<td>5</td>
<td>[-8.8 -8]</td>
</tr>
<tr>
<td>6</td>
<td>[-7.5 7.5 -7.5]</td>
</tr>
<tr>
<td>7</td>
<td>[-7.7 -7]</td>
</tr>
<tr>
<td>8</td>
<td>[-6.5 6.5 -6.5]</td>
</tr>
<tr>
<td>9</td>
<td>[-6.5 6.5 -6.5]</td>
</tr>
<tr>
<td>10</td>
<td>[-6.5 6.5 -6.5]</td>
</tr>
</tbody>
</table>

3.3 Weights in Fitness Function

The final structure that the algorithm evolves depends greatly on how the fitness function is defined, not only on parameters used, but also on their respective weights. Varying these values can lead to completely different results. Which values to use depends on the purpose of the structure. This algorithm defines fitness function by using distance from the target, number of cubes, maximum stress in the structure, maximum displacement of the nodes and number of connections between cubes. For example, if structure needs to be stiff and stable, higher weight for maximum deflection and maximum stress could be used. If it also needs to be compact, then higher weight for number of cubes and number of connections could be used.

Population size of 25, maximum generation of 150 and target position [4,4,3] were used. Figure 3.5 shows an example of 8 different evolved structures for the same target position but with different fitness function parameters. Weights were chosen depending on a range of values that each parameter takes. Furthermore, table 3.3 contains results that the algorithm achieved with respect to each parameter.

\[
\text{fitness} = W_{\text{Distance}} \times (1 - \text{Distance}) - \ldots \\
\ldots - W_{\text{NumCubes}} \times \text{NumberOfCubes} - \ldots \\
\ldots - W_{\text{MaxStress}} \times \text{MaximumStress} - \ldots \\
\ldots - W_{\text{MaxDisplacement}} \times \text{MaximumDisplacement} + \ldots \\
\ldots + W_{\text{NumConnections}} \times \text{NumberOfConnections};
\]  

- \text{Distance} value is normalized (current distance divided by distance from the origin to the target) so it is in range from 0 to 1. Since 1 corresponds to the highest distance from the target, \(1 - \text{Distance}\) was used in fitness function. Thus, with weight of 100 its maximum contribution to the fitness is 100 (when the target is reached).
• *NumberOfCubes* is not a normalized value so it can be any number bigger than 1. It is desirable that structure has less cubes and that is why there is a negative sign in front of *NumberOfCubes* parameter. Since in this example the target position is not far from the origin, a weight of 0.1 was used. This means that when there are 100 cubes, contribution to the fitness is -10 (since target is close, it is expected that structure will not have more than 100 cubes).

• *MaximumDisplacement* can be any number from 0 to an upper limit determined by the maximum stress in the structure. Since displacement of nodes in the structure should be minimal, there is also a negative sign in front of *MaximumDisplacement* parameter. In this example, weight of 200 (when 0, *MaximumDisplacement* is not used) was used, so, for displacement of 0.05m contribution to the fitness is -10 (since target is not far, it is expected that displacement will not be bigger than 0.05m).

• *MaximumStress* value is normalized (maximum stress is divided by critical stress) and it is in range from 0 to 1. Again, maximum stress should be minimal so negative sign was used in front of *MaximumStress* parameter. Weight of 5 was used (when 0, *MaximumStress* is not used), which means that maximum contribution to the fitness is -5 (when maximum stress is equal to critical).

• Because of the need to evolve bigger structures (when real value of density of wood is used and the target is further away) an additional parameter was added, *NumberOfConnections*. The Value of this parameter is normalized with $6 \times \text{NumberOfCubes}$. In order to make structures more compact, number of connections between cubes should be maximal and that is why there is a positive sign in front of *NumberOfConnections* parameter. In these examples it was not used.

<table>
<thead>
<tr>
<th>Structure</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxStress [x10^5 Pa]</td>
<td>1.0386</td>
<td>0.4543</td>
<td>0.9883</td>
<td>0.4774</td>
<td>1.0743</td>
<td>0.6420</td>
<td>1.1966</td>
<td>0.7175</td>
</tr>
<tr>
<td>MaxDisplacement [m]</td>
<td>0.0217</td>
<td>0.0307</td>
<td>0.0021</td>
<td>0.0033</td>
<td>0.0086</td>
<td>0.0057</td>
<td>0.0053</td>
<td>0.0071</td>
</tr>
<tr>
<td>Number of cubes</td>
<td>66</td>
<td>101</td>
<td>76</td>
<td>51</td>
<td>9</td>
<td>39</td>
<td>18</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 3.3: Number of cubes, maximum stress and maximum displacement for structures from figure 3.5.

All structures in figure 3.5 were evolved with distance weight of 100 because the main goal was to reach the target. For the first structure no additional weights were used. Because of this, the structure was just randomly built without any consideration of stress, displacement or number of cubes. The only conditions were stability of the structure and its ability to reach the target. During evolution the first structure that reached the target was the best one in the end.

The second structure had also weight for maximum stress, in addition to distance weight. Because of this, a really high structure was built in order to minimize the stress. This structure had best result with respect to maximum stress (table 3.3) since it was the only additional constraint so the algorithm could specifically evolve a structure to minimize this value.
3.3. Weights in Fitness Function

The third structure had weight for maximum displacement instead of maximum stress. Because of this the structure did not have any long branches and it had best result with respect to maximum displacement (table 3.3). Again, this is because the algorithm could specifically evolve a structure just for this task.

The fourth structure had both weights for maximum stress and maximum displacement. It was a combination of the previous two structures, but it did not achieve such good results as they did individually. This is because it had to minimize two parameters, but still, the result was far better than for the first structure.

Next four structures are analog to the first four but with additional weight for number of cubes included in fitness function. Therefore, the fifth structure had only weight for number of cubes. It did not manage to reach the target. This is because maximum stress in the structure was not considered and in order to reach the target it needed to develop reinforcement. This was very unlikely because all of the structures had minimum number of cubes in each generation and developing a supporting branch was almost impossible. As a result this structure had the lowest number of cubes and went straight for the target but it managed to reach only so far until stability of the structure was still satisfied (adding one more cube at the end would make it unstable).

The sixth structure, which had weight for maximum stress and number of cubes, managed to develop reinforcements and reach the target with reasonable amount of cubes. It also achieved the best result (considering structures with weight on num-

Figure 3.5: Structures evolved with different parameter weights in fitness function. $W_{\text{Distance}} = 100$ and $W_{\text{NumConnections}} = 0$ for all the structures. $W_{\text{NumCubes}} = 0$ in the top row (1-4) and $W_{\text{NumCubes}} = 0.1$ in the bottom row (5-8). $W_{\text{MaxDisplacement}} = 0$ and $W_{\text{MaxStress}} = 0$ unless otherwise noted in the figure.

1) fitness = 100; 2) $W_{\text{MaxStress}} = 5$, fitness = 98.11; 3) $W_{\text{MaxDisplacement}} = 200$, fitness = 99.59; 4) $W_{\text{MaxDisplacement}} = 200$, $W_{\text{MaxStress}} = 5$, fitness = 97.35; 5) fitness = 76.91; 6) $W_{\text{MaxStress}} = 5$, fitness = 93.33; 7) $W_{\text{MaxDisplacement}} = 200$, fitness = 81.43; 8) $W_{\text{MaxDisplacement}} = 200$, $W_{\text{MaxStress}} = 5$, fitness = 92.69;
ber of cubes) with respect to maximum stress and in addition to that it managed to achieve very good result with respect to maximum displacement (as a result of minimizing stress).

The seventh structure, which had weight for maximum displacement and number of cubes, did not manage to converge. This is because it was trying to minimize the deflection and number of cubes, but not considering the maximum stress in the structure. As a result it was hard for reinforcement to develop. Still, it achieved the best result (considering structures with weight on number of cubes) with respect to maximum displacement.

Finally, the eighth structure, which had weights for all of the three parameters, did not reach the target. This structure represents the best trade-off between these parameters. It can reach the target and reinforcement can be developed, but the main problem is that it takes more time (generations) to optimize all three parameters.

Structures with real density value of wood will be bigger and longer, but even without using NumberOfConnections parameter, if the algorithm is left to run for a long time, evolved structures might become more compact (example is shown in figure 3.6). If NumberOfConnections parameter is used (with respective weight) this process of evolving compact structure might be faster.

![Figure 3.6: Structures with real value of density of wood, without NumberOfConnections in fitness function. 50 (left) and 500 (right) generations were used. Blue line points to the target position.](image)

There is no strict rule on how to define values for weights. It is usually based on intuition, but even then the results might be unexpected. For example, in figure 3.5, sixth structure managed to achieve better result with respect to maximum deflection than eighth structure, and it did not have any weight for deflection included. Even though sometimes the results can be unexpected and bad even for good weight values, generally, the algorithm will perform well with good values used if there is enough time (generations) for it to evolve and optimize the structure.
3.4 Parallel Computing Toolbox

There are two possible ways to run the algorithm. The first one is by using parfor loops (parallel computing toolbox) and the second one is by using regular for loops (see appendix A.3). Implementation of the algorithm with parfor loops is slightly different than the one with regular for loops. With parfor all of the organisms are evolved simultaneously and thus it is not possible to check for a specific evolved organism if it is already contained in the current generation (all of the organisms are created at the same time). This means that in each generation there can be multiple organisms with the same structure. In implementation with regular for loops, a check function was used in order to prevent this from happening. Because of this, the performance of the algorithm, in these two cases, is slightly different.

<table>
<thead>
<tr>
<th>Population</th>
<th>10</th>
<th>40</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>220.26</td>
<td>326.99</td>
<td>272.83</td>
</tr>
<tr>
<td>Fitness</td>
<td>85.92</td>
<td>82.11</td>
<td>91.84</td>
</tr>
<tr>
<td>Generation</td>
<td>95.30</td>
<td>96.30</td>
<td>76.70</td>
</tr>
</tbody>
</table>

Table 3.4: Algorithm timing.

Analysis was done for population sizes of 10, 40 and 70. Same target and parameters were used as in section 3.1. For each population size, the algorithm ran for 20 times and average values of fitness, convergence generation and timing were used. Table 3.4 contains obtained results. For population size of 10 both algorithm implementations achieved similar results in terms of fitness and convergence generation. As for the timing, parfor implementation took 32% less time. This was expected because with parfor loops all computation is done simultaneously in parallel as opposed to sequential computation of regular for loops. Obtained results for fitness and convergence generation are not representative because, as mentioned before, for this problem, population size of 10 is not a good choice. For population size of 40, implementation with regular for loops achieved slightly better results for fitness value and convergence generation. This is due to usage of existence checking function. Again, timing was much better for parfor implementation (56% better). Finally, for population size of 70, similar results were obtained. Implementation with regular for loops achieved better results for fitness and convergence generation and parfor implementation had better timing (51% better).

In general, if population size is reasonably big, with parfor implementation timing will be much better than in regular for implementation. Fitness value and convergence generation will be slightly worse, but in most cases this can be compensated with running the algorithm for more generations.
Chapter 4

Future Work

The next logical step would be to actually build the evolved structure. In order to do so, one would have to provide the robot with a building plan that defines what to build, how to build it, when to build it and how to connect it. This is not an easy task because there can be multiple solutions to the same problem. Also, buildability of the whole structure would have to be considered in every step (for example if two parts are connected one would have to check whether they can be connected to the other parts). A simple approach to this problem is given in the next chapter (4.1), although, it is just a starting point, the main part (defining the connection sequence) has not been developed.

As far as the algorithm is concerned, there are always possibilities for improvements. Optimization of the code would improve algorithm timing. Also, further adjustment of parameters (for specific problems) like weights in fitness function, number of mutations, elites, population size, ..., could lead to better results (discussed in the chapter 3). Furthermore, more realistic boundary conditions could be defined, for instance, adding floor as a boundary (not allowing the structure to go below the plane of the origin) and including interaction between the structure and obstacles or floor in stability function. This way the algorithm could develop more realistic and much more massive ground structures. Also, different building blocks could be introduced. For example, modular units [1] could be used and with active modules, the algorithm could evolve actuated ground structures. There is a great variety of possibilities for further improvements and that is why there is a big potential with what can be achieved by using this algorithm.

4.1 Approach to buildability

Since most structures are buildable (it is hard to find an example of a structure for which you can certainly say that it is not buildable), the main question is not whether it can be built but rather how can it be built. Defining a sequence of operations that a robot has to take in order to build a structure is a challenging task and it is beyond the scope of this project, although, one possible starting point is given below. The simple planar structure from figure 2.7 (right) was used as an example.

First step is to define a database of structures that the robot knows how to build (or can be taught how to build). In this example very simple structures (figure 4.1) were used. These structures act as building blocks of the whole structure. In gen-
eral, big database is desirable because it can simplify the process of assembling.

Figure 4.1: Database of building blocks that could be used for building a planar structure.

The second step is to cluster the cubes so that the evolved structure is composed just out of defined building blocks, figure 4.2. Simple K-means algorithm was used. It started with small number of clusters and randomly grouped the data (it used discrete coordinates of cubes as data). If any created cluster was not contained in the database, the algorithm repeated the clustering process. If it did not find a satisfying set of clusters after a certain number of iterations, the number of clusters was increased. When a good cluster configuration was found, the algorithm stored it and ran again from the start to get more (specified number) results. In the end, out of all results, the best one was chosen (in this example, the one with the lowest number of clusters was chosen, but this does not have to be the best choice). Defining the best result is also a challenge. One could try with defining a fitness function which would include number of connections between segments and number of clusters. More details on the clustering algorithm can be found in the appendix A.

Figure 4.2: Clustering example of a planar structure.

The last step would be to define the connection sequence. One possible strategy would be to start building everything that is above the origin. After that clusters could be added one to the left, one to the right, in order to keep the balance. Defining a strategy greatly depends on the type of structure, especially in 3D, and it is the most complex part due to robot limitations and many possible solutions for a certain problem. Connection sequence has not been developed for the example in figure 4.2.
Chapter 5

Conclusion

With the ability to build reinforcement, evolutionary algorithm gained a possibility to develop structures for much more complex and challenging tasks. Furthermore, developed structures look more realistic and physically feasible. In addition to that, external conditions like forces and obstacles can be simulated. All of this together improves the functionality of the algorithm and makes it a useful tool for autonomous design.

The main gain from this project is that these structures can actually be used in real applications and that external conditions can be transferred from real workspace to a simulation. Although there is still work to be done in order to construct these structures autonomously, the algorithm can already provide a useful design for specific, user defined, tasks.
Appendix A

Algorithm Manual

A.1 Organism encoding

\textit{parentVec} - vector containing parent branches.
\textit{startVec} - matrix containing starting coordinates of each branch.
\textit{bdir} - vector of matrices where each matrix is containing direction vectors of one branch.
\textit{maxStress} - maximum stress in the structure.
\textit{maxDisplacement} - maximum displacement in the structure.
\textit{numCon} - total number of connections between cubes in structure.

More details on encoding in J.Seitz’s Bachelor’s Thesis [6].

A.2 List of functions

\textit{evolalg.m} - main function.

\textit{addbranch.m} - function to add branch (one of the mutations). Takes number of cubes \((nCube)\) as input and outputs branch direction matrix \((bdir_{new})\) and control variable \((flag)\) to check if algorithm got stuck (chapter 2.4).

\textit{build.m} - function to cluster the data and output the connection sequence. Takes as input organism \((organism)\) and outputs connection sequence \((sequence)\). Not complete, only clusters data so far.

\textit{checkcollision.m} - function to check collision inside structure (rarely used because it is already contained in \textit{checkstability.m} function). Input is \textit{bdir} and \textit{startVec} of organism and output is 1 if there is no collision and 0 if there is \((\text{allright})\).

\textit{checkexisting.m} - function to check if there are already existing organims in the generation. Takes number of current organism in generation \((\text{current})\) and current generation \((\text{generation})\) as input and outputs 1 if current organism already exists in the generation, otherwise 0 \((\text{exist})\).

\textit{checkfitness.m} - function to sort all the organisms in one generation according to their fitness. Takes as input number of organisms in population \((nObject)\), one generation \((\text{organism}_{gen})\), target \((\text{destination})\) and fitness function parameters \((\text{useDeflection}, \text{useStress}, \text{useCubes}, \text{useConnections})\). Output is the same generation but with sorted organisms \((\text{organism}_{gen})\).
checkstability.m - function to check stability of the structure. Takes organism (bdir, startVec), external load (externalLoad, externalLoadPosition) and obstacles (obstacles, useObstacles) as input and outputs maximum stress (maxStress), maximum displacement (maxDisplacement), number of connections between cubes (num_con) and control variable (allright).

create.m - function to create new organism (used for creation of initial population). Number of cubes (nCube) and obstacles (obstacles, useObstacles) are input and organism (organism) is the output.

create_cube.m - function used for rendering cubes. Input is the origin of the cube (origin) and color of the cube (color). Output is plot of the cube.

fitnessfunction.m - function to calculate fitness of an organism. Takes organism (organism), target (destination) and fitness function parameters (useDeflection, useStress, useCubes, useConnections) as input and outputs value of the fitness function (fitness).

group.m - function to cluster the data provided by build.m function (uses K-means algorithm). Cube coordinates (coordVec) are input and centroids (minCent) and indices that indicate which cube belongs to which centroid (minCent) are output. Function output variables are taken from result with minimal clusters.

makeMovie.m - function to output sequence of pictures (best organism from each generation) in movieFrames folder. Takes organisms from all generations (organism), number of generations (nFrames), target (destination) and obstacles (obstacles, useObstacles) as input and outputs one picture for best organism from each generation.

mix.m - function to do cross over between two specified organisms. Takes two organisms (organismA, organismB), stability control variable (useStability), external load (externalLoad, externalLoadPosition) and obstacles (obstacles, useObstacles) as input. Output is the new organism (crossover).

mutation.m - function to mutate specified organism. Takes one organism (organism), stability control variable (useStability), external load (externalLoad, externalLoadPosition) and obstacles (obstacles, useObstacles) as input. Output is the new organism (mutate).

pdf.m - function to calculate probability of choosing an organism from generation. Input is number of organisms in population (nMax) and output is probability distribution (pdfValue).

render.m - function to display an organism. Takes organism (organism) as input and outputs a plot.

renderVideo.m - function used by makeMovie to output each frame. Similar to render.m with difference that obstacles and target are displayed in the plot. Input is an organism (organism), target (destination) and obstacles (obstacles, useObstacles).
A.3  evolalg.m

User defined values:
- maxGen - maximum number of generations.
- nCubeMin, nCubeMax - range from which number of cubes for initial population will be chosen.
- nPopulation - number of organisms in each generation.
- nElite - number of elites.
- mutationPercent - range from 0 to 1 which determines percentage of organisms to be mutated in one generation.
- destination - target position.
- externalLoad - vector of external force.
- externalLoadPosition - position of external force. It has to be at target otherwise algorithm will avoid specified position.
- obstacles - coordinates of obstacles.
- desired - number of organisms to render from last generation after computation is done.

Algorithm options (1 - use; 0 - do not use):
- useProfiler - option to use profiler.
- useMovieMake - option to use movieMake.m function.
- useParallel - option to use parallel computing toolbox (parfor loops).
- useStability - option to check stability of each organism.
- useObstacles - option to use obstacles.

Fitness function parameters (1 - include; 0 - do not include):
- useDeflection - option to include maximum deflection in fitness function.
- useStress - option to include maximum stress in fitness function.
- useCubes - option to include number of cubes in fitness function.
- useConnections - option to include number of connection between cubes in fitness function.

Output:
- mTime - total mutation time.
- cTime - total cross over time.
- iTime - total running time.
- organism - matrix of structures (size: maxGen x nPopulation) containing parentVec, startVec, bdir, maxStress, maxDisplacement and numCon.

A.4  checkstability.m

User defined values:
- nInternalConnections - number of connections within cube (default 28).
- nInterCubeConnections - number of connections between cubes (default 16).
- nNodesPerCube - number of Nodes for each cube (default 8).
- aCube - sidelength of cube in meters (default 0.02).
- tHMA - width of the glue layer in meters (default 0.002).
- tMargin - distance between outer surface and node (default 0).
- rho_cube - density of cube in kg/m³ (default 700).
- rho_HMA - density of glue in kg/m³ (default 0.97).
- E_HMA - Young’s Modulus of the glue in Pa (default 8.9 * 10⁶).
- E_wood - Young’s Modulus of the cube material in Pa (default 9 * 10⁹).
- criticalStress - critical stress in N/m² (default 1.2 * 10⁵)
Output:
allright = 1 - structure is stable.
allright = 2 - stress in the structure is too high.
allright = 3 - stress between origin and ground is too high.
allright = 4 - there is a collision in the structure.
allright = 5 - there is a collision with the obstacle.

A.5 group.m

User defined values:
\( K \) - number of clusters. As initial value, number of cubes divided by maximum sized element rounded to a closest integer is used.
\( numIter \) - number of iterations of algorithm. Corresponds to number of results.
\( maxSize \) - maximum size of cluster.
\( iter2 \) - number of iterations (when size of clusters was bigger than \( maxSize \)) before increasing number of clusters.
\( iter1 \) - number of iterations (when shape of clusters is not in the database) before increasing number of clusters.
\( max\_iters \) - maximum number of iterations of K-means algorithm to adjust centroids.
User has to define each structure from the database with appropriate conditions.

A.6 fitnessfunction.m

User defined values:
\( distanceFactor \) - weight that distance from target position has in the fitness function. Distance is normalized with distance from origin to the target.
\( weightFactor \) - weight that number of cubes has in the fitness function. Number of cubes is not normalized.
\( stabilityFactor \) - weight that maximum stress has in the fitness function. Maximum stress is normalized with critical stress.
\( displacementFactor \) - weight that maximum displacement has in the fitness function. Maximum displacement is not normalized.
\( connectionFactor \) - weight that number of connections has in the fitness function. Number of connections is normalized with \( 6 \times \) Number Of Cubes.
Bibliography


