Automatic Discovery of Algorithms for Multi-Agent Systems

Sjors van Berkel
Delft University of Technology
Delft, The Netherlands
s.e.f.vanberkel@student.tudelft.nl

Dániel Turi
Delft University of Technology
Delft, The Netherlands
d.turi@student.tudelft.nl

Andrei Pruteanu
Delft University of Technology
Delft, The Netherlands
a.s.pruteanu@tudelft.nl

Stefan Dulman
Delft University of Technology
Delft, The Netherlands
s.o.dulman@tudelft.nl

ABSTRACT

Automatic algorithm generation for large-scale distributed systems is one of the holy grails of artificial intelligence and agent-based modeling. It has direct applicability in future engineered (embedded) systems, such as mesh networks of sensors and actuators where there is a high need to harness their capabilities via algorithms that have good scalability characteristics.
NetLogo has been extensively used as a teaching and research tool by computer scientists, for example for exploring distributed algorithms. Inventing such an algorithm usually involves a tedious reasoning process for each individual idea. In this paper, we report preliminary results in our effort to push the boundary of the discovery process even further, by replacing the classical approach with a guided search strategy that makes use of genetic programming targeting the *NetLogo* simulator. The effort moves from a manual model implementation to an automated discovery process. The only activity that is required is the implementation of primitives and the configuration of the tool-chain. In this paper, we explore the capabilities of our framework by re-inventing five well-known distributed algorithms.

Categories and Subject Descriptors

G1.4 [**Numerical Analysis**]: Optimization—*Unconstrained optimization*

Keywords

Agent Based Models, CUDA, Genetic Programming, NetLogo, Multi-agent Systems, Global-to-local compilation

1. INTRODUCTION

The current technological advances have lead to the creation of engineered computing systems characterized by high complexity in terms of software [14], system engineering [8] and expected quality of service [7]. Most of these systems are made out of a large number of structural elements that have complex dependencies and emergent properties. Some exhibit a tight-coupling between their structural elements split across a large number of layers [6]. Others are less hierarchical while being topologically distributed over large physical spaces [25].

The scale of the computing systems, measured in terms of the number of constituent elements, the complexity of the software stack required to control them, the tolerance to failures, etc., require novel programming paradigms. [3] Besides that, a well-known property of large-scale systems is that above a certain system size, global properties occur somewhat unexpectedly out of simple local interactions. In a very small number of cases, this *emergent behavior* is a positive property and can be put to good use by synchronization algorithms [23], clustering schemes [19], distributed feed-back mechanisms [5] etc., while in the large majority of cases it has mainly a negative effect [15].

In this paper, we propose a so-called "Global-to-local compiler" [24] to automatically generate local interaction rules between the constituent system elements starting from a global description of the system behavior. Since there is no linear mapping between the two levels, this has proven to be a very complex problem [16].

Looking at the problem from a constructive point of view, there is a high interest in programming in terms of emergent behavior, using generative local behaviors (both computation-wise as well as communication-wise), since such programs scale-up very well for large systems [2].

Our approach to solve the problem at hand is to use *Genetic Programming* in order to discover algorithms that fulfill the search goal. A large number of combinations of local rules needs to be evaluated in order to determine how well they map onto the desired global behavior (i.e. a good fitness for a given program). Doing so on a normal CPU is convenient, but it can take up a solid amount of computation time given that:

- The number of initial programs should be large enough to be sufficiently diverse in nature (population size);
- The programs that are generated through crossover and mutation should have enough iterations (number of generations) to evolve good programs.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

Obviously, it would be best if the user would get fast results without trading off the solution quality. To help diminish the computation time it takes to discover a good program, we accelerate the simulations on a graphics processing unit (GPU). Due to the parallel computation approach used by technologies such as CUDA and OpenCL, GPUs are a more natural fit to our problem domain than CPUs, hence obtaining a solution will take less time.

The paper is organized as follows. In Section 2 we present the related work while in Section 3 we present the system description. Experimental results are show-cased and discussed in Section 4. We conclude with Section 5 and show hints about future work.

2. RELATED WORK

Other research initiatives have already tried to automate the search for distributed algorithm targeted at multi-agent systems. One such project is called BehaviorSpace [22], and is included with the NetLogo [22] distribution. BehaviorSpace allows the user to simulate a model many times, by varying the input parameters automatically, and recording the results for each run. A user can provide multiple "reporter functions", that monitor the state information (agent properties) at every run. In this way, the user is provided with an overview on all possible outcomes of a model, rather than having to guess the model parameters. For models that have relatively low complexity, execution time is not a problem. Unfortunately, for a large number of parameters, due to the exhaustive type of approach, the runtime grows exponentially. Another similar project in [11] focuses on sampling the parameter space densely where the solution space changes rapidly, in order to precisely investigate the influence of the different parameters.

A project that is similar to ours is BehaviorSearch [21]. It uses genetic algorithms to search for optimum parameters for a given agent-based model. Unlike our approach, it doesn't change the structure (e.g., code) of the model. This is somehow restrictive since it presumes that the user already knows a promising agent behavior. What is left to be discovered is only the tweaking of the parameters.

ABM Meta-modeling Framework [13], splits the problem in two: bottom-up (also called forward-mapping problem) and top-down (also called the reverse mapping problem). Their approach to solving the top-bottom problem is to interpolate configurations using the forward mapping to approximate a smooth and continuous surface. This interpolated surface represents the space of configurations that would satisfy the system-level requirements set out by the user. Again the framework presumes that the user knows the exact structure of the model and what's left to be discovered are its parameters.

Our approach is to use Genetic Programming [1] for altering the internal structure of the code for the agents. This gives us more power for finding promising novel algorithms with limited knowledge about the agent's internal code structure. Out of many approaches we chose to use the Grammatical Evolution (GE) [17] to guide the evolution process. We further outline our motivations for using this approach in Subsection 3.1.

In [18], the Ant Foraging problem [9] is implemented using Genetic Programming. In order to do this, the Genetic Programming framework is supplied with a number of function calls and properties that may be of importance for finding a suitable algorithm. The Genetic Programming framework recombines these "building blocks", and is able to find a good solution for this particular problem.

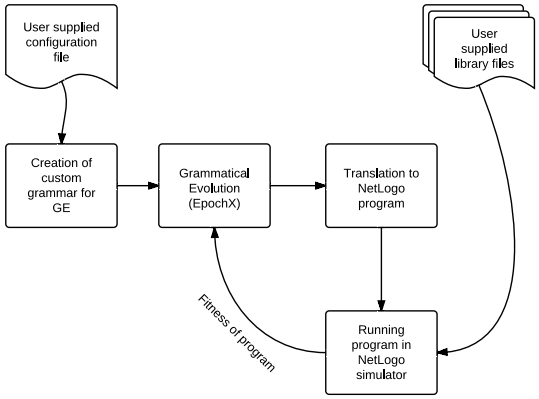

Figure 1: MetaCompiler design

3. SYSTEM DESCRIPTION

3.1 Genetic Programming

The main contribution of the paper is preliminary reporting on a tool called *MetaCompiler for Automatic Algorithm Discovery* (MAAD). Its architecture is shown in Figure 1. The input is a user defined configuration file in which he/she specifies the simulation environment, as well as agent-specific capabilities. These settings are expressed as functions and their code is located in the user-defined libraries. Additionally, it also contains the fitness function that is used to evaluate the quality of the generated programs. For instance, for the case of a synchronisation algorithm, a fitness function expressed as the standard-deviation of the clocks should be as close as possible to zero.

Once the configuration file has been processed, the MetaCompiler generates a custom grammar file that defines all the possible programs that are allowed to be generated by the genetic programming module. As mentioned in Section 2 we use a Grammatical Evolution approach. GE is the most flexible, widely applicable strategy out of all the GP alternatives; this is due to the fact that GE is able to generate programs from an arbitrary grammar. The actual interpretation of the generated programs can be done by a separate system. The GE tool generates a string representation of a program. Additionally, an inherent property of the fact that we generate a program from a grammar is that the generated code is by definition syntactically correct, given the correctness of the grammar. This removes the need for a program to be syntactically verified before pushing it for evaluation by the agent simulator (e.g. NetLogo).

For us, this flexibility made it possible to come up with a simple and general-purpose language we could fit to the specifics of distributed systems algorithms. Our choice has some advantages compared to other approaches:

- it generates only programs that are syntactically cor-

rect. Agents do not have to execute methods that make use of unknown or unsuited state information (e.g. a method that returns a list will not be assigned to a numeric state variable);

- it gives full control to the translation to other ABM languages besides logo-*;
- it allows the usage of multiple agent simulators than NetLogo, if necessary as a consequence of the modular design;
- it is easy to change the grammar from which code is generated;
- it offers multiple configuration options with respect to initial population growth strategies, fitness evaluation, experiment parameters, etc.

The Genetic Programming part of the MetaCompiler, is built using the EpochX library [4]. This is an open source genetic programming framework written in Java and is released under a permissive license. Among the advantages of EpochX is the fact that it readily includes implementations for the most popular GP tactics. Another advantage is the fact that it is built with extension and adaption in mind, making it easy to change each of the steps of GP to user's will without having to adjust and recompile the library. This way, we can leverage the power of genetic programming while maintaining both flexibility and community support.

Currently, we only use standard methods for the different steps vital to GP, provided by default in EpochX. For initialization, we use a Ramped Half-and-Half initializer [10]. This uses a mix of the full and grow initialization tactics that stem from the origin of GP [1]. We enhanced the initialization process, by allowing the specification of the maximum initial tree depth and maximum end tree depth. Selection is done through Tournament selection. This strategy repeatedly picks a subset of the programs in a population and picks the most fit program from that subset to be used in the mutation/crossover/reproduction stage of GP. Crossover happens through the One-Point Crossover technique. For both parents, a random point in the program representation is chosen, and the two parts of one parent are combined with the two other parts of the other parent. The problem with this approach is that programs with nested structures cannot be cut and reattached at any point and still conform to the given grammar. The reattached parts will be adjusted to suit the grammar, but this is where they lose a lot of their meaning. Hence, the second part of both new programs will likely be very different from their original value.

Mutation is done using Point Mutation, which picks a random structure (e.g. a statement or a function) from the current program and changes its value to some random one. Much like with crossover, the underlying structures of the changed structure need to be adapted to suit the grammar, and will most likely lose their meaning in the process.

The crossover and mutation methods we chose can be very disruptive to programs. In the future we would like to investigate alternative methods to make program transitions less disruptive. Additionally, we plan to give the user more power over the parameters of the methods. The ultimate goal of these steps being to smooth out the search space.

3.2 Agent behavior

To goal of this project is to generate global behavior based on local interactions between the agents. If we look at their structure, they have a reduced set of state variables. Due to the fact that there is no global coordination or accessible global state, we do not support NetLogo features that relate to global state. Hence, the first approach for our framework was to generate only local agent actions.

Even though it makes sense for the output program not to rely on global state, testing whether a program fulfills a global goal can be very hard without storing global state. Therefore, we introduced global variables in the consecutive version of the MetaCompiler. Global variables are also allowed to fulfill the function of immutable agent variables, reducing the memory consumption of the simulation. Other global variables can be hidden in order to restrict the agents from using global information.

Obviously, when porting the programs to a hardware platform for real-life execution, hidden global variables should be entirely dropped from the program, and immutable agent variables (that are configured as global variables) should be embedded at the agent level.

Another feature we added was the idea that we don't want to be entirely dependent on commands that are discovered at run-time. For example, sometimes you want to enforce an agent to move a little, or send out a particular signal, before starting the execution of a next round of code, or maybe after finishing the execution. For this, we introduced (agent) static primitives, actions that execute either before (begin) or after (end) the execution of "generated" code. In addition, we also enabled global static primitives, that happen either before the first agent static begin primitives, or after the last agent static end primitives. These global static primitives generally serve an administrative functions (e.g. updating some value to be used by the fitness function, or updating the network graph).

To enable the user to adjust the program between the initialization and execution, we have also included the ability to add initialization primitives. In conclusion, the outline of our output programs is as follows:

1. Initialization phase (executed once)
   - 1.1 Initialization code of both agent and global parameters
   - 1.2 User supplied initialization primitives
2. Run phase (executed multiple times)
   - 2.1 Global static begin primitives
   - 2.2 Agent static begin primitives
   - 2.3 Generated code
   - 2.4 Agent static end primitives
   - 2.5 Global static end primitives

3.3 NetLogo

The first agent language we use is NetLogo. Our choice is merely because of its flexibility and simplicity. Additionally, we have a lot of programming experience in NetLogo, such that the translation from our intermediate language is simple.

Since our intermediate language is mostly about the structure of the programs rather than using a lot of difficult language constructs, and a major part of the program consists

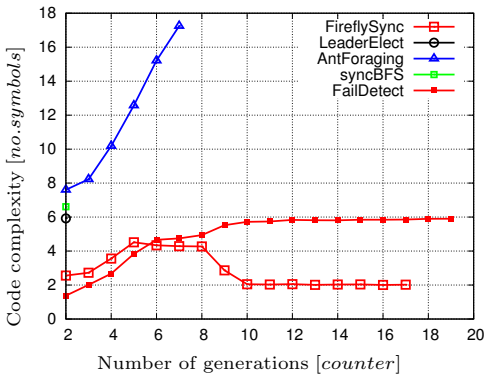

Figure 2: The influence of the code size to the fitness.

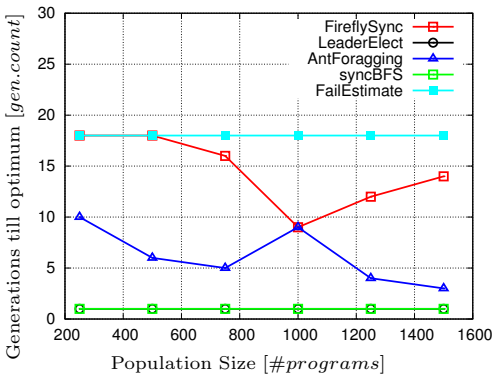

Figure 3: The influence of the population size to the fitness.

of library calls (that are already natively implemented in NetLogo), the translation to this language is fairly simple.

In fact, the only real separation is the difference between the actions in initialization and run phases (as described in Section 3.2). The rest of the NetLogo program is hard-coded, and resides in a template.

3.4 CUDA

In order to develop an algorithm with the GE method, usually a large number of experiments are required. Additionally, since the experiments involve random components (random behavior of the agents, execution order, etc.) each experiment has to be run multiple times. This requires a lot of computation and thus it can take a long time to reach a satisfactory fitness value. In order to accelerate the process, the inherently parallel nature of the problem can be exploited. However in most desktop computers the number of computational cores is low (between 2 and 8), thus the overall architecture is less suitable for massively parallel execution of programs.

Nowadays, the usual approach to accelerate these kind of computations is to use General Purpose GPU's, specifically on CUDA capable cards. The architecture of these devices is specifically designed to execute massively parallel programs, utilizing a large number of cores, a special memory architecture and a very large number of lightweight threads. As described above, the MetaCompiler takes the user configuration file, and generates a number of NetLogo source files. With the GPU-based acceleration these are then converted to a special bytecode that is executed by a virtual machine running on the GPU. The evaluation of the fitness happens on the GPU as well, after which a vector of fitness values (for every experiment respectively) is sent back to the CPU. This happens in order to minimize memory transfers between the host (CPU) and the device (GPU) memory. The evaluation of the fitness usually requires the state information for all the agents, and transferring these would incur a high latency.

For CUDA, the threads can share data, and can synchronize. The thread blocks run independently and it is not possible to make any assumption on their execution order.

3.4.1 Application parallelism

The GE problem includes two levels of parallelism that can be exploited. On one hand the experiments are independent of each other. On the other hand, the agents execute the same code with some interaction between each other. Thus each thread block is assigned to an experiment, and each thread in that thread block is an agent of the experiment.

Ld/st	Arith/Logic	Cntrl	1-hop
setLocalImmediate	add	IfElse	addNeighbor
getOthersShared	greaterThan	threadGuardOn/Off	getNrNeighbors

Table 1: Examples of VM instructions

3.4.2 The CUDA virtual machine

The CUDA virtual machine executes the agent code. This is needed since on the current architectures it is not possible to start multiple computational kernels simultaneously, yet a number of different programs have to be tested. In this case the kernel contains only the VM's executor code. The agent's program is downloaded as a vector of numbers, each encoding an instruction on the VM. Along these a vector of parameters are passed with the instruction. With these the VM can execute corresponding behavior.

In the NetLogo environment the state of the agents is stored in *turtle-* (position, color, etc.) and in *breed variables* (user defined). Apart from these it is possible to define local variables in the procedures carried out by the agents. Each agent has a number of private variables, that cannot be accessed by the other agents, but also and shared variables which can. The procedure local variables along with temporary variables (which e.g. store the output of arithmetic operations) are stored in the array of local variables. The shared variables store the turtle- and breed variables. Both of these type of memories can be accessed for reading and writing, so that agents can communicate with each other (*ask* primitives). Apart from these each agent has a data structure with stores the ID of those agents which it is neighbors with. Since every agent executes the same code, they have the same amount of memory allocated of every type. Additionally, there is a vector of global values which resides currently in the global DRAM. These variables can thus be used to communicate with the host computer. At this moment only scalar values of type double can be used as variables, other types (e.g. lists, strings) will be implemented in the future. Boolean and integer values are simply converted to double (1.0 for *true* and 0.0 for *false*).

Table 1 shows some examples of the VM instruction set. It contains the usual instructions (e.g moving data into vari-

ables, arithmetic operations, comparisons, branching operations), and special instructions which encode higher level behavior relevant to agent based simulations, as well. These special instructions are currently: switch on/off the thread guard. If it is on, only Thread 0 (in each block) execute the upcoming instructions until it is switched off again. Another special control instruction is synchronization, which creates a thread barrier to synchronize all the threads in the block. This is essential to ensure consistency in the shared memory operations. The 1-hop operations manage the neighborhood data structure, thus they add/remove neighbors, return the number of neighbors, etc. One important member of this family is the remote procedure calling, which translates NetLogo's *ask linkneighbors* primitive. It is necessary, otherwise agents would not be able to initiate communication. It is implemented via a special shared data structure, which includes a flag, and a variable to store the call address.

The CUDA kernel implements an execution skeleton, which consists of several phases:

1. Agent initialization: dynamically allocated variables are allocated here (the data structures storing local, shared and global variables, since their sizes are known only at runtime), Thread 0 initializes shared data;
2. Execution: a while loop takes the instruction which is pointed at by the instruction pointer and executes it unless it is a stop instruction (at which point the agent has finished its program;
3. Agent destruction: the dynamically copied data structures are destroyed.

In order to ensure the consistency of the shared variables, the CUDA threads synchronize with each other via synchronization barriers. Each instruction can be divided into a *read/execution* and a *write* phase. Thus each instruction has to synchronize twice once after reading and once after writing. This scheme ensures that no race conditions occur while accessing the same shared variables. Apart from this, the VM has the customary auxiliary data structures: instruction pointer, which is incremented by the instructions (or set by jump and branching instructions). It also has a call stack with a stack pointer.

3.4.3 Translation of NetLogo code into CUDA VM byte code

In order to run the genetically generated code on the GPU, the NetLogo program has to be translated into a special byte-code which is downloaded to a CUDA capable card, and executed there. In order to simplify the process of the translation, it is first compiled by the NetLogo engine, so that later stages can use the information extracted by the NetLogo compiler. The NetLogo engine merges the included files, compiles the procedures one-by-one into an intermediary format (*ProcedureDefinition*). Additionally, it sets up the so called *Workspace*, which contains information about the breeds of agents (which defines the shared agent-variables, i.e. those which are visible to other agents), and the global variables (along with their initial values). In order to seamlessly integrate with NetLogo, the GPU VM translation framework is implemented in *Scala*

The translation process is controlled by the *GPUBackend* class, which stores the necessary data extracted from the NetLogo engine (as mentioned above), and instantiates the classes which facilitate the different phases of the translation. The input of this process is a restricted subset of the NetLogo language which is transformed the bytecode as multiple vectors of numbers. The different stages of the translation take place in the following order:

1. AST Modification: the Abstract Syntax Tree (AST) representation of each procedure is extracted and modified, in order to make the later stages simpler. These AST's consist of an array of statements which store *Commands* (instructions that create changes in the environment of the simulation), which in turn can store *Reporters* (instructions which do not create changes in the environment of the simulation, instead they report values of variables in the workspace or constants), or command blocks which contain more commands. During the modification some of these are erased or replaced by special primitives. Since the number of agents is static in the current version, only one agent-creating command is allowed. The number of agents are extracted (to be used at CUDA kernel launch), and the command is discarded. If there are agent initialization commands included, these are merged to the *SETUP* procedure. The usual forever button that keeps calling the *GO* procedure, is simulated by appending a jump statement at the end of the procedure pointing at the beginning. A jump statement is appended to the end of the SETUP to ensure the continuous running of the simulation. Finally agent variables which are automatically set up by the NetLogo engine are initialized (e.g. the CUDA thread ID is stored in the *WHO* field which identifies the agent). Special primitives are inserted to facilitate the passing of parameters between agents in case of remote procedure calls (corresponding to *ask linkneighbors*).

2. Instruction mapping: in this stage the statements belonging to the procedures are iterated together, and the NetLogo primitives, which constitute the instructions in the AST are mapped to their GPU VM counterparts (this means sometimes multiple instructions). This is also the phase where temporary variables are allocated to store the result of each operation. Thus in order to chain together operations the consecutive ones take the predecessor's result as their argument. For the sake of simplicity those instruction who have command blocks (*ask*, *ifelse*) are treated as a function call, thus return instructions are appended to the end, containing references to the parent procedure.

3. Branch assembly: In this stage the hitherto separated code blocks of the procedures, *ifelse*, *ask* command blocks, etc. are fused together into a continuous array of instructions. Prior to this stage these blocks referenced each other (call, return, etc.) with their formal names, in this stages these names are changed to addresses, which are simply their position in the instruction array.

4. Variable renaming: In this stage the GPU instructions are iterated over and their formal names (*WHO*, *COLOR*, etc.) are encoded with two numbers: the first one designating the type ($0 \rightarrow$ *local*, $1 \rightarrow$ *shared*, $2 \rightarrow$ *global*), the second one is its identifier. Thus on

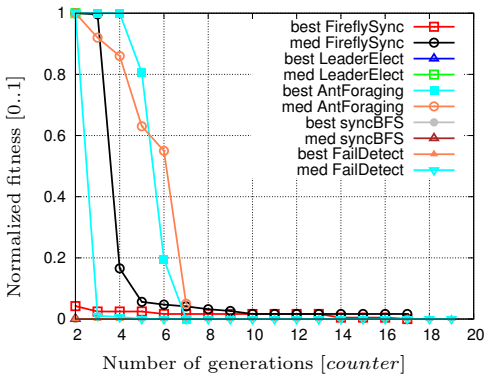

Figure 4: The influence of the number of generations to the fitness.

the GPU the separate types are allocated in separate arrays, the identifier denoting their position. At this stage one more global value is allocated after the existing ones, the fitness procedure (which has a conventional name) is selected and with special instructions its reported value is redirected into this global variable. Thus it has always a fixed place, and it can be easily retrieved after carrying out the simulation.

5. Instruction encoding: In this stage the instruction names which were hitherto kept in mnemonic form are renamed to identification numbers, which are understood by the CUDA VM's instruction executor.

Finally the *GPUBackend* class downloads the instructions to the GPU and starts the CUDA kernel, which executes the simulation and retrieves the fitness value. To ensure the seamless integration with the rest of the software, JCUDA was used.

3.4.4 Limitations

As this software is in its early stages, it suffers from severe limitations. As mentioned before, the number of the agents are fixed for each simulation. Also currently only one breed of turtles is supported. The neighborhood relations between the agents are decided solely by their ID, and not their position. The possible data types are floating point numbers and boolean values (thus e.g. lists are not allowed). The biggest limitation is currently on the language itself, as only a very subset is implemented.

4. EXPERIMENTS AND DISCUSSION

In order to validate our MAAD, we chose five well-known algorithms and tried to re-invent them with our framework. The first one is called *Firefly Synchronization* [23]. Compared to the synchronization algorithm used in sensor networks, our implementation uses a backwards time shift strategy. Our choice is determined by the increased complexity of the original paper. In short, our implementation works as follows: nodes adjust their phase backwards to the moment where they finished the TX interval when they sense a large number of neighboring nodes are sending at any RX moment. The execution happens in rounds and phase is initialized with random values. The primitives that are used are shown in Table 2.

The second algorithm is called *Leader Election* [12]. Nodes are placed in a ring with bidirectional links. The solution is an algorithm where the node with the highest id gets picked as the leader in a distributed fashion. The primitives that are used are shown in Table 2.

The third algorithm we implemented is Koza's famous *Ant Foraging* algorithm [9]. Ants collect food and carry it towards the nest from two food places. Along the path, they deposit pheromone in order to guide the others to good paths. The algorithm looks for strategies to maximize the amount of food carried around in a fixed amount of time. The primitives that are used are shown in Table 2.

The fourth algorithm is called *Fail Detect* [20]. It estimates in a distributed manner the amount of average packet loss in a large-scale wireless mesh network. The primitives that are used are shown in Table 2.

The fifth algorithm is called *Sync. BFS* [12]. It builds a tree in a geometric graph starting from a random node.

Figure 4 shows the evolution of the fitness for all the experiment generations (horizontal axis). The values have been normalized to [0..1] interval (vertical axis). The first two generations have been removed from the graph due to the fact that the fitness function in this part handles the expansion of the search space and not the search of the actual solution for the algorithm. *Firefly Sync.* and *Ant Foraging* show a steep decline of the fitness in the first seven generations, while the other algorithms have a more smooth transition towards the optimum. *Leader Election* algorithm is discovered (reaches a zero fitness value) in one generation.

The second metric we used to evaluate the programs is *code complexity*. It is defined as the number of symbols that occur in a program. Figure 2 shows the evolution of the code complexity (vertical axis) for all generations (horizontal axis). *Ant Foraging* has the highest increase in code size until it reaches the optimum at the seventh generation. All the others have a slight increase till the seventh generation and than remain quasi constant or show a slight decrease.

In the third set of experiments, we looked at the required number of generations for every population size that is required to reach the optimum (an individual program in one generation with a fitness value of 0.0). In Figure 3 we see that *Fail Estimate* never reaches a perfect optimum no matter the population size. On the other extreme, *syncBFS* and *Leader Election* find the optimum in one generation no matter the population size. *Firefly Sync.* requires a higher number of generations when the population size is small and a smaller number of generations when the population size is larger.

In the fourth set of experiments (Figure 5), we assessed the impact of the mutation rate on the process of finding a good algorithm. For these experiments, the mutation rate and crossover rate always added up to 1, such that either crossover or mutation was used to create a new program for the next generation (no reproduction was used). The evaluation is made for mutation rates of .1, .2, .3, and .4, and crossover rates of .9, .8, .7, and .6 respectively. All experiments in this series used 2 initial expansion generations (not displayed in the graph) to enhance the initial code complexity. It can be seen that the discovery of *syncBFS* was trivial in all cases, since the perfect solution was always found directly, something that also happened for *Leader Election* with mutation rate .1, and for *Firefly Sync.* and *Ant Foraging* with mutation rate .4.

Algorithm Name	Primitive	Params	Description
Firefly Sync.	resetClock()	none	sets back the clock phase
	countTXNodes()	none	counts the number of transmitting nodes
	RX()	none	signals that TX period has finished
Leader Election	sendPacket()	none	send a packet to the neighboring node in the ring
	sendMyID()	none	inform neighbors about own ID
	setAsLeader()	none	set node as leader by setting a flag to true
	isInMsgHigher()	none	check if input message is higher than own ID
	isInMsgEqual()	none	check if input message is equal to own ID
Ant Foraging	MOV-RANDOM()	none	agents (ants) move randomly
	MOV-TO-NEST()	none	agents (ants) move back to the nest
	PICK-UP()	none	pick-up the food if patch has food available
	DROP-PHEROMONE()	none	agents (ants) drop pheromone at the
	IS-FOOD-HERE()	none	branching condition based on food level on a given patch
	IF-CARRYING-FOOD()	none	branching condition based on the availability of food
	MOV-ADJ-FOOD-ELSE()	none	branching condition based on the adjacency of food
SyncBFS	addReqUnNeigh(nodeId)	number	adds a request to an unmarked neighbor
	clearParentRequests()	none	clear the parent request list
Fail Detect	multiply()	two numbers	computes the multiplication of the two input numbers
	timeWindow()	none	time elapsed since the start of the simulation
	multiply()	two numbers	computes the multiplication of the two input numbers
	oneMinus()	one number	computes one minus the input number
	oneOverAggregate()	none	computes the inverse of the aggregate

Table 2: Algorithm primitives and parameters.

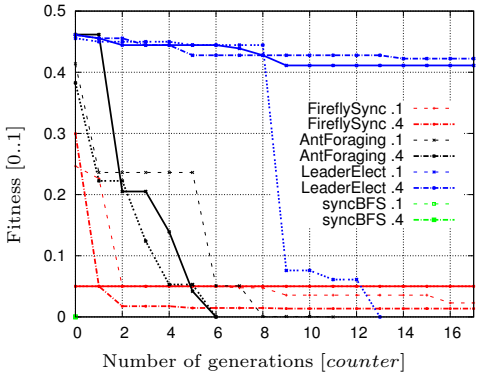

Figure 5: The influence of the mutation rate on the fitness.

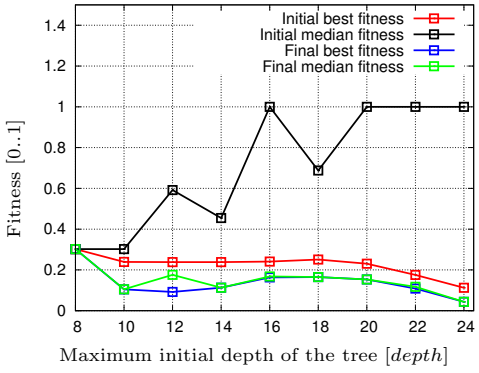

Figure 6: The influence of the maximum initial depth of the tree to the fitness.

In Figure 6, we compare the fitness of *Firefly Sync.* for different values of the maximum initial tree depth. The different maximum initial depths vary from 8 to 24, and we have divided the measurements into 4 different categories: best start fitness, median start fitness, best end fitness, and median end fitness. Each of the experiments was run 4 times to get a stable measurement value. Since we use Ramped Half-and-Half initialization, we also have a maximum end depth value to which the initializer tries to grow the trees, but for these experiment we fixed this value to 60, which is more than it can reach in the initialization phase. For this experiment we did not use any expansion generations, since they make the influence of the maximum initial tree depth less clear.

The complexity of the algorithms has an important influence on the performance of framework. Simple ones, like *Sync BFS* or *Leader Election* do not require a large population size or numerous generations in order to reach the optimum (at least one program in a generation with a fitness value of 0.0). More complex ones like *Ant Foraging* or *Fail Estimate* require much more computational effort as seen in Figures 2 and 3. An important role in the discovery process of the algorithms is the fitness function. Not only it has to guide the search process towards the solution algorithm but it also has to ensure we don't end up with trivial cases. For instance, for the case of the *Firefly Sync.* algorithm, a program that would trigger instantaneous synchronization of the clocks for all the agents is when they all set it to a constant value. We had to check that the clocks are actually incremented and penalize solutions that fail this test. For the case of the *Fail Estimate* algorithm, we had to make sure that the programs make use of the so called *conditional assignments*. Otherwise, constant values can be assigned that would sometimes provide very good, if not perfect estimations.

5. CONCLUSION AND FUTURE WORK

NetLogo has been used for many years as a teaching and research tool by which many scientists have tested their ideas. The task of inventing algorithms usually requires a tedious discovery process for every individual research idea. With our framework we push the boundary even further, by replacing the "classic" approach with a guided search. The potential user looks for novel distributed algorithms via an automatic process that makes use of genetic programming. The effort moves from manual implementation in NetLogo to a guided search where only the implementation of primitives is required. In this paper, we showcased our initial efforts towards a framework by trying to automatically re-invent five distributed algorithm and analyze its characteristics for different parameters such as population size, expansion generations, solution code size etc. Since this work is just a preliminary effort, the initial results are very promising. We plan to make use of better suited approaches for the Genetic Programming building blocks (mutation and crossover operators), as well as to further optimize the execution speed via e.g. the usage of GPUs. For the eventual usability of the MetaCompiler, it is important to investigate the complexity of algorithms it is able to discover. One way we plan to do this by making the building blocks for the current examples more granular and evaluate the difference in performance. We also plan to discover new algorithms and report on the corresponding results, as well as to report on the effort of creating building blocks versus the automatic discovery of algorithms.

6. ACKNOWLEDGEMENTS

We would like to thank Ana Varbanescu for her valuable help on the GPU acceleration implementation.

7. REFERENCES

[1] Genetic Programming: On the Programming of Computers by Means of Natural Selection, 1992.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] J. Bachrach, J. Beal, and T. Fujiwara. Continuous space-time semantics allow adaptive program execution. In *SASO'07. First International Conference on*, pages 315–319. IEEE, 2007.

[4] L. Beadle. *Semantic and Structural Analysis of Genetic Programming*. PhD thesis, PhD thesis, University of Kent, Canterbury, 2009.

[5] Y. Brun and all. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.

[6] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 2004.

[7] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Network, IEEE*, 2010.

[8] P. Fortescue, G. Swinerd, and J. Stark. *Spacecraft systems engineering*. Wiley, 2011.

[9] J. Koza. Evolution of emergent cooperative behavior using genetic programming. *Computing with Biological Metaphors*, 1994.

[10] S. Luke and L. Panait. A survey and comparison of tree generation algorithms. In *GECCO,2001*, pages 81–88. Citeseer, 2001.

[11] S. Luke and D. Sharma. Finding Interesting Things: Population-based Adaptive Parameter Sweeping. 2007.

[12] N. Lynch. *Distributed algorithms*, volume 872. Morgan Kaufmann, 1995.

[13] D. P. Miner. A framework for predicting and controlling system-level properties of agent-based models. 2011.

[14] D. Mishra and A. Mishra. Complex software project development: agile methods adoption. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[15] J. Mogul. Emergent (mis) behavior vs. complex software systems. In *ACM SIGOPS Operating Systems Review*, 2006.

[16] R. Nagpal and M. Mamei. Engineering amorphous computing systems. *Methodologies and Software Engineering for Agent Systems*, pages 303–320, 2004.

[17] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 2001.

[18] L. Panait. Learning ant foraging behaviors. In *Proceedings of the Ninth International Conference ...*, 2004.

[19] A. Pruteanu, S. Dulman, and K. Langendoen. Ash: Tackling node mobility in large-scale networks. In *SASO,2010*, pages 144–153, 2010.

[20] A. Pruteanu, V. Iyer, and S. Dulman. Faildetect: Gossip-based failure estimator for large-scale dynamic networks. In *ICCCN,2011*, pages 1–6. IEEE, 2011.

[21] F. Stonedahl and U. Wilensky. Finding forms of flocking: Evolutionary search in abm parameter-spaces. *Multi-Agent-Based Simulation XI*, pages 61–75, 2011.

[22] S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, pages 16–21, 2004.

[23] Werner-Allen and all. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Sensys*, 2005.

[24] D. Yamins and R. Nagpal. Automated global-to-local programming in 1-d spatial multi-agent systems. pages 615–622, 2008.

[25] F. Zambonelli and M. Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. *Autonomic Communication*, 2005.