Comparing Finite State Automata Representation with GP-trees

Pavel Petrovic
Norwegian University of Science and Technology
IDI Technical Report 04/06
ISSN: 1503-416X
petrovic@idi.ntnu.no

Abstract

Genetic Programming and Evolutionary Programming are fields studying the application of artificial evolution on evolving directly executable programs, in form of trees similar to Lisp expressions (GP-trees), or Finite State Automata (FSA). In this exercise, we study the performance of these methods on several example problems, and draw conclusions on the suitability of the representations with respect to the task structure and properties. We investigate the role of incremental evolution in the context of FSA representation. We also present an evolutionary software package for educational programming environment Imagine, which can be used for introducing the principles of evolutionary computing to wider young audience.

Keywords

Evolutionary Programming, Finite State Automata, Incremental Evolution

1 Introduction

In our previous experiments[21], we have used augmented finite state automata (FSA, similar to finite state machines, FSMs) for behavior-arbitration in behavior-based mobile robot controllers. We have been designing these controllers by the means of evolutionary computation. Our main motivation for choosing the state-based representations was their structural similarity to the structure of the robot controller tasks: the robot performing some activity is always in some state while it reactively responds with immediate actions or it proceeds to other states as a response to environmental percepts – thus the activity of a robotic agent can be modeled by a state diagram accurately. We believe that state-diagram formalisms can in fact steer controllers themselves and be the back-bone of their internal architecture. Secondly, we believe that the state automata are easier to understand, analyze, and verify than other representations, for example neural networks. Thirdly, we believe that state automata are more amenable to incremental construction of the controller, because adding new functionality involves adding new states and transition, and making relatively small changes to the previous states and transitions. On the contrary, neural network architectures often need to be dramatically modified, unless some modular approach is used. However, research in modular neural approaches is still in its very early stages.

While the focus of our previous experiments lied in the issues of incremental evolution and evolving the arbitration itself, this study pays attention to evaluating the performance of the state-based representations as such. The purpose is to analyze the performance of the state-based representations and compare it to the performance of the GP-tree representation. We study the performance on several artificial tasks of various kinds with the intention to understand the set of tasks, where the state-representation might outperform the GP-tree representation, but also to identify the tasks, where the state-representations are less efficient. For the purposes of performing these experiments, we have designed a package for evolutionary computation experiments for educational programming environment Imagine Logo [18] that has an interface to control both simulated and real robots [22][1].

In the following sections, we review the related work on evolving state-representations, describe the evolutionary software package we developed, describe the tasks we have experimented with, and provide the empirical evidence of the performance of the state-based and GP-tree representations. We conclude the paper with discussion of the results and ideas for further investigations.

2 Related work

In [10], Fogel et.al. draw a strict distinction between evolutionary approaches where the evolution is modeled as a *genetic process* and the approaches where the evolution is modeled as a *phenotypic process*. The question is the one of the representation of the individuals. Strictly seen, and pertaining to a biological relevance, the genetic processes deal with genotypes, i.e. encodings of genes that influence the shape, behavior, or other properties of the individual, whereas the phenotypic representations attempt to encode directly the complete individual, its shape, behavior and other properties in their final form. We make a point here that this biologically-inspired distinction (that by the way leads to a heavily studied field of evolutionary development) is not very well-grounded in the computational context.

First, the phenotypic representation inherently implies some form of encoding, i.e. storing the usually complex patterns of interactions of the individual with the environment or with the details of the particular task instance, where the evolved or evolving solution is applied. This encoding does not explicitly list all these interactions, and properties, not even all the mechanisms that participate in these interactions. They are rather implied by and contained in the whole system that executes the solution. Isn't rather that whole system the phenotype? Where is the border line between the *body* of the individual and its environment in the computational context? What is the difference between interpreting a FSA in order to produce a run-time behavior of an individual in the computational context and interpreting a DNA of a biological organism in order to generate chemical compounds that are reacting to the chemicals found in the agent's internal and external environment? While the former is supposed to be a phenotypic process, the second clearly is a biological genetic process. However, in both cases, there is an encoding that is translated by the "operating system" in order to obtain the actual behavior of an individual. And even if the encoding involves a directly executable machine code, it is still being interpreted by the CPU to obtain the actual run-time behavior and execution of the CPUs own microcode. We argue that there is no evolutionary process in the computational context that is modeled as *phenotypic process*. Please also note and do not confuse the difference between the distinction of a sexual and an asexual process and the distinction between phenotypic and genetic process. The complex interactive individual must be frozen into some sort of a *static* encoding in order to be manipulated by evolutionary operators and stored in a population. In other words, in computational context, it is difficult and possibly misleading to try to identify what is a phenotype. In biological terms, body of an individual is distinguished by its spacial properties, however, those (if there could be any analogy in the computational world) are rather task-related than representation-related. Does the phenotype include all the bits of the software, where the solution is being run? Including the operating system, and all the

[1]A secondary objective for producing this software package was to provide an educational platform for popularizing and experimenting with evolutionary computation, which is supported by wide use of Imagine in the schools on elementary and secondary level. This objective is treated in a separate report.

low-level implementation of the hardware of the computer, which the solution is interacting with and utilizing? Or does it include only those parts of the evolved solution that can be changed in the evolutionary process? But isn't that then the genotype itself?

Secondly, in many instances of the evolutionary algorithms that are rendered by the authors as a *genetic process*, solutions are represented directly in the genotype – just recall the popular `maxbit` sample problem for instance. The implications of our remark for the field of evolutionary development are not dramatic, we only suggest that many research questions relate to the *genotype representation*, and we consider the GP-trees, FSA, and virtually any evolved representation to be genotype, rather than phenotype evolved in a phenotypic process. In this respect, however, one has to be very careful about any analogies drawn to the biological process of development. In the computational context, transformations and configurations of the genotype itself are possible, and their usefulness has little to do with the developmental process in nature even if it might share some [but most likely forced] structural similarities. The research questions of what transformations on the genotypes and how they can be performed, however, are still valid, interesting and worth investigation.

FSA or FSMs[2] have been used as genotype representation in various works, although this representation lies on the outskirts of the evolutionary algorithms research and applications.

Evolutionary Programming, [12, 8, 9, 11, 10] is a distinguished evolutionary approach that originally used FSA as the genotype representation[3]. EP does not utilize recombination operators[4], and relies on mutations. The original work addressed the tasks of prediction, identification and control.

In [6], Chellapilla and Czarnecki introduce modular FSMs, which are in fact equivalent to non-modular FSMs, except that the topology is restricted – in particular, the FSMs are partitioned into several encapsulated subparts (sub-FSMs), which can be entered exclusively through their starting states. The authors use modular FSMs to evolve controllers for the artificial ant problem, that was previously successfully solved by evolving binary-string encoded FSA[17], they provide evidence that modular FSMs perform better on this task than non-modular FSMs, and they also provide evidence that direct encoding with structural mutations of non-modular FSMs perform better than binary-string encodings used in [17]. This idea of modular FSMs has been adopted also by Acras and Vergilio [1], who develop a universal framework for modular EP experiments and demonstrate its use on two examples.

In [2], Angeline and Pollack are experimenting with automatic modular emergence of FSA. They suggest to freeze and release parts of the FSA so that the frozen (or "compressed") parts cannot be affected by the evolutionary operators. The compression occurs randomly and due to the natural selection process, it is expected that those individuals where the compression occurs for the correctly evolved sub-modules will perform better and thus compression process interacts with the evolutionary process in mutually beneficial manner. Indeed, the authors document on the artificial ant problem that the runs with compression performed better than equivalent runs without compression. They reason: "An explanation for these results is that the freezing process identifies and protects components that are important to the viability of the offspring. Subsequent mutations are forced to alter only less crucial components in the representation."

[2]The difference between FSA and FSMs in the evolutionary literature seems to be that the former refer strictly to the formal computational model as originated sometimes in the middle of the 20th century and intensively formalized and studied for example by [14], while the latter usually refers to models where control actions are performed when transitions are followed. Other computer science literature, however, often makes no distinction in these two names, while various other names (Moore, Mealy) are used for different flavors of the formalism. The core of all representations, however, are the FSA, and we refer to the extensions in this article as augmented FSA, or FSA for simplicity. When referring to previous work, we attempt to use the same term as the author.

[3]Further developments of EP moved from the FSA to real-value parameters representation, where the Gaussian mutation is applied to alter the parameters from generation to generation.

[4]Even though later the annual EP conferences included all works relevant for Evolutionary Algorithms, and later have been integrated into Congress on Evolutionary Computation – CEC.

In [20], Lucas is evolving finite state transducers (FSTs), which are FSA that generate outputs, in particular, map strings in the input domain with strings in the output domain. FSTs for transforming 4-chain to 8-chain image chain codes were evolved in this work, while three different fitness measures for comparing generated strings were used: strict equality, hamming distance and edit distance.

An interesting piece of work by Frey and Luagering [13] considers FSMs as controllers for several 2D benchmark functions and the artificial ant problem. In their representation, the whole transition function is represented as a single strongly-typed GP-tree – i.e. a branching expression with conditions in the nodes that direct execution either to the left or to the right sub-tree, and finally arriving to a set of leaves that list the legal transition pairs (old state, new state).

In the PhD thesis [16], Hsiao is using evolved FSA to generate input sequences for digital circuits with the purpose of their verification, and fault detection. The author achieves best fault detection rate on various circuits (as compared to other approaches), except of those that require specific and often long sequences for fault activation.

In [15], Horihan and Lu are evolving FSMs to accept words generated by a regular grammar. They use an incremental approach, where they first evolve FSMs for simpler grammars, and gradually progress to more complex grammars. They use the term *genetic inference* to refer to their approach of generating such solution.

In [7], Clelland and Newlands are using EP with probabilistic FSA (PFSA) in order to identify regularities in the input data. The PFSA is a FSA, where the transitions are associated with probabilities as measured on input sequences. The EP is responsible for generating the topology of the FSA – number of states and how they are interconnected, and the transitions in PFSA are labeled according to their "fire rate". This combination can be applied for rapid understanding of an internal structure of sequences.

In [3], Ashlock et. al. are evolving FSMs to classify DNA primers as good and bad in simulated DNA amplification process. They evolve machines with 64 states in 600 generations. They used the weighted count of correct/incorrect classifications as their fitness function – however, they sum the classifications made in each state of FSM throughout its whole run, they argue that if classifications made in the final state only were used, the performance was poor. In addition, this allows the machine to produce weighted classification – how good/bad the classified primer is. The best of 100 resulting FSMs had success rate of classification of about 70%. Hybridization, i.e. seeding 1/6th of a population of an extra evolutionary run with the best individuals from 100 previous evolutionary runs improved the result to about 77%. This work was continued in [4], where the FSM approach was compared to more conventional Interpolated Markov Models (IMMs), which outperformed FSMs significantly.

In an inspiring study from AI Center of the Naval Research Laboratory [23], Spears and Gordon analyze evolution of navigational strategies for the game of competition for resources. The strategies are represented as FSMs. Agent moves on a 2D grid while capturing the free cells. Another agent with a fixed, but stochastic strategy is capturing cells at the same time, and the game is over when there are no more cells to capture. Agents cannot leave their own territory. Authors find that the task is vulnerable to cyclic behavior that is ubiquitous in FSMs, and therefore implement particular run-time checking to detect and avoid cycles. They experiment with the possibility to disable and again re-enable states (as contrasted to permanent and complete state deletion). They also compare evolution of machines with fixed number of states and evolution of machines, where the number of states changes throughout the evolutionary run. They discover that in the case of varying number, the machines utilize the lately-added states to lesser extent, as well as that deleting states is too dramatic for performance, and thus suggest to merge or split states instead of deleting and creating states. Due to the stochastic algorithm of the opponent agent in the game of competition for resources, the fitness function must evaluated each individuals in many different games (G). Authors disagree with others claiming that

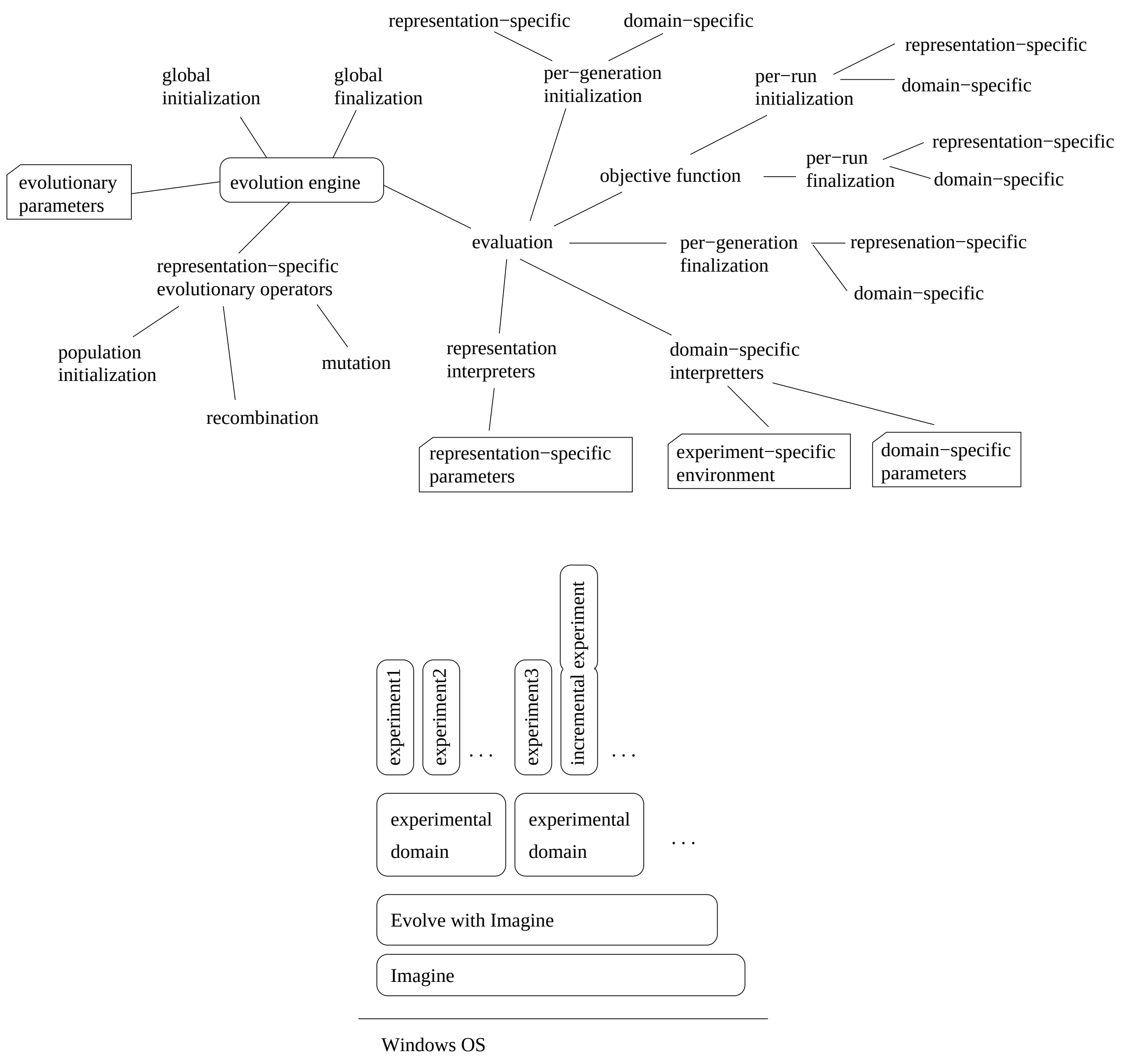


Figure 1: Architecture and deployment of Evolve with Imagine package.

keeping G low can be well compensated by higher number of generations and conclude that it results in unacceptable sampling error. The authors therefore evaluate all the individuals on fewer games (500), and if the individual should outperform the previous best individual, they re-evaluate it on many more (10000) games.

Some further FSM-relevant references can be found for example in the EP sections in the GECCO and CEC conferences.

3 Evolutionary algorithm and the EI software package

In order to perform experiments with the state representation, we have designed an evolutionary computation software package EI (Evolve with Imagine). Figure 1 outlines the architecture of the EI package.

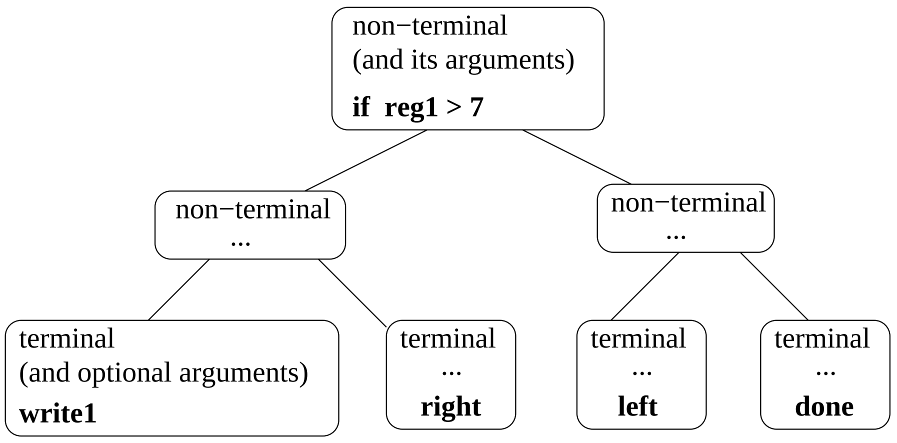


Figure 2: Illustration of GP-tree representation: nodes contain non-terminals that have two sub-trees, terminals are in the leaves.

3.1 Representation

The software package supports two different representations – GP-tree representation and FSA representation.

In the GP representation, the evolved program is a binary tree with non-terminals in the nodes and terminals in the leaves. Non-terminals and terminals are symbols with the semantics of a procedure (not a function as often is the case in GP implementations; here, the state of the computation is contained completely in the registers and in the state of the environment). Non-terminals have two sub-trees[5], which themselves are binary GP-trees. Both terminals and non-terminals may contain additional arguments: constants of various ranges, register references, predicates (or conditions). Each problem domain is thus defined by the syntax and semantics of, see also figure 2:

- Set of terminals T, $|T| = N_T$
- Set of non-terminals N, $|N| = N_N$
- Number of registers N_R, and possibly their semantics (such as coupling to some sensors)
- Binary relations Rel, for example `<, >, ==`, etc. that can be used by non-terminals.
- Definition of terminals arguments, a function[6] $ArgT : T \rightarrow ArgTypes^*$, where $ArgTypes$ is a set of possible argument types: {`constant, interval, register reference, relation`}, where `constant` can be instantiated to any integer number given globally specified range, `interval` is specified as `[min max]` and can be instantiated to any integer from this interval, register reference can be instantiated to any of the registers $R_1, \ldots, R_{N_R}$, and `relation` is instantiated to a member of Rel. For example, a non-terminal `if` typically has arguments `(register relation constant)`.
- Definition of non-terminals arguments, a function $ArgN : N \rightarrow ArgTypes^*$.

[5]For the reasons of better topological compatibility of all nodes with respect to evolutionary operators, our system currently supports only non-terminals with two sub-trees.

[6]With the notation M^*, we refer to all the possible tuplets $(m_1, m_2, \ldots m_k)$, $k \in \mathbb{N}$, $\forall i(1 \leq i \leq k \rightarrow m_i \in M)$, including tuplets with no members.

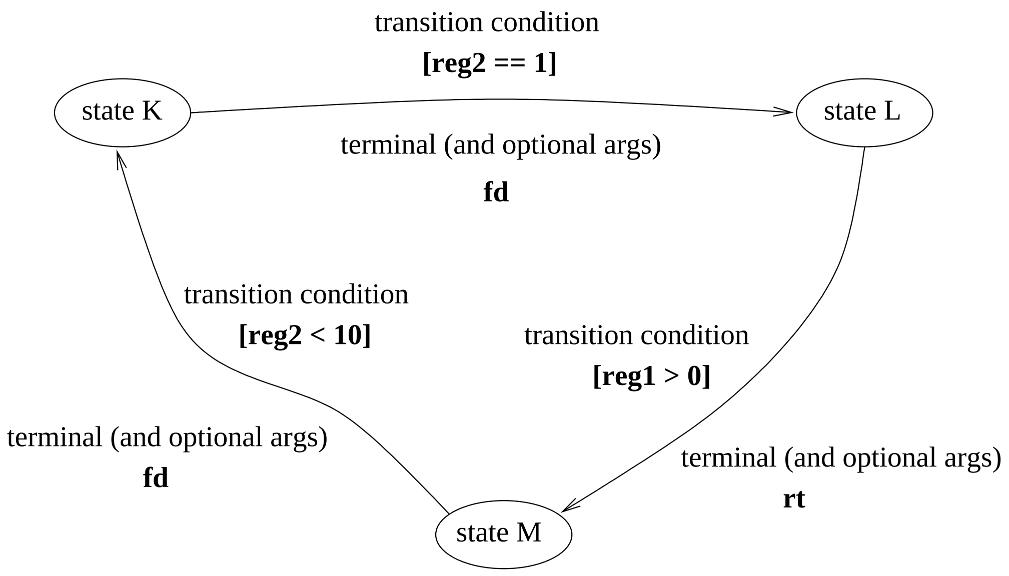


Figure 3: Illustration of FSA representation: transition conditions correspond to GP-tree non-terminals, however each transition has also an associated terminal. The state transitions can lead to the same state where they originate, and multiple transitions between the same two nodes are allowed (although only one is used). Sequences of state transitions can create loops.

In our FSA representation, the programs are augmented state automata, formally defined as, see also figure 3:
$\mathcal{A} = (N_S, N_R, N_{Trans}, Rel, T, ArgT, condition\_syntax, F, max\_steps)$, where

- N_S, is the number of states of the automaton, states are numbered $S = \{1 \dots N_S\}$, and 1 is always the starting state
- N_R is the number of registers of the automaton
- $N_{Trans} : \{1, \dots, N_S\} \to \mathbb{N}$ is a function returning for each state the number of transitions leading from that state
- Rel is a set of binary relations that can be used in the transition condition
- T is a set of terminals, $|T| = N_T$
- $ArgT : T \to ArgTypes^*$ defines terminals argument types
- $condition\_syntax \in ArgTypes^*$ is a tuple defining syntax of conditions that can trigger the transitions between states, for example `(register relation [0 3])`
- $F : S \times \mathbb{N} \to ArgTypes^* \times S \times T \times ArgTypes^*$ is the transition function specifying transitions in all states, including the conditions and actions. One condition and one action are associated with each transition. The transitions leading from states are ordered. Each transition leads from some state to another state and can be followed, if its associated condition is satisfied. When the program is in state s, only one of the transitions leading from the state s can be followed, and it is the one that is satisfied and has the lowest order. Transitions terminating in the same state as they originate are allowed. When a transition is followed, the associated action represented by a terminal and its arguments is performed.

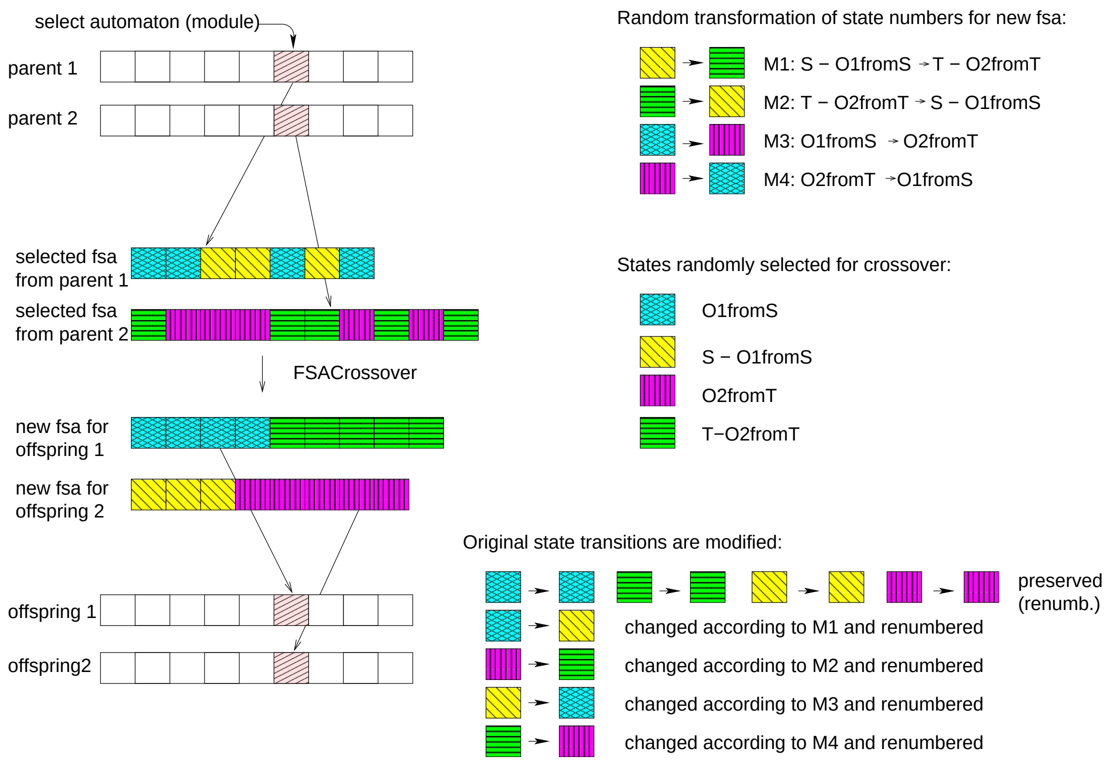

select automaton (module)
parent 1
parent 2
selected fsa from parent 1
selected fsa from parent 2
FSACrossover
new fsa for offspring 1
new fsa for offspring 2
offspring 1
offspring2
Random transformation of state numbers for new fsa:
M1: S – O1fromS → T – O2fromT
M2: T – O2fromT → S – O1fromS
M3: O1fromS → O2fromT
M4: O2fromT → O1fromS
States randomly selected for crossover:
O1fromS
S – O1fromS
O2fromT
T–O2fromT
Original state transitions are modified:
preserved (renumb.)
changed according to M1 and renumbered
changed according to M2 and renumbered
changed according to M3 and renumbered
changed according to M4 and renumbered

Figure 4: Overview of topological crossover for FSA representation.

- *max_steps* – the maximum number of steps of the automaton to execute. In our FSA, there are no final states, anytime the automaton arrives at a state when none of the outgoing transitions is satisfied, the automaton terminates. Otherwise, it terminates only after *max_steps* are performed (or for another reason defined by the problem domain).

EI implements also the third representation type – weighted FSA (labeled HMM due to its similarity to Markov Models), where each transition is associated with a real number w_t, the weight that implies the probability that the transition will be followed. In the case of FSA, the transition that is satisfied and has the lowest order is always followed, even if there are multiple satisfying transitions. In case of HMM, each satisfied transition in the current state can be followed, and it is stochastically chosen based on its weight. The probability for a transition to be chosen is linearly proportional to the weight of that transition.

3.2 Recombination

EI provides recombination operator – crossover for both GP-tree (usual GP-tree node exchange crossover), and for FSA representation. For the FSA representation, the crossover is the same as we used in our previous Evolutionary Robotics experiments, see figure 4. We randomly select states in both parents for the first genotype, the remaining states form the second genotype. All transitions that can be preserved are preserved. The remaining transitions are redirected according to a random projection from the states that were given up to the states that are imported. For more details, please see [21].

In the GP-tree representation, the crossover operator exchanges two random sub-trees of two selected individuals (or with a probability *pcross_combine* it simply merges the two individuals in one sequence (using the `seq` non-terminal)[7]. Individuals that exceed the maximum allowed

[7]if the `seq` non-terminal is not used, *pcross_combine* should be set to zero

depth are always trimmed immediately.

In addition to the standard GP-tree crossover, EI implements GP-tree homologic crossover, which exchanges only nodes in the two program trees that are topologically at the same location. This takes the inspiration from nature, where crossover occurs in highly topological way – only the genes of the same type can be exchanged. It reduces the bloat (useless offspring), and smooths the fitness landscape – although also slightly decreases the discovering potential of the search. The ratio of the homologic crossover can be controlled using *phomologic_crossover* parameter.

In both representations, EI supports brooding crossover, i.e. generating several broods of offspring from each selected pair of parents, and evaluating them only on a stochastically selected subset of testing sample (in order to avoid too high growth of CPU demand). Only the best two of all the generated offspring are chosen for the new generation. The parameter *crossover_brooding* determines the number of offspring pairs generated, and the parameter *cross_brood_num_starts_q* determines the relative size of the subset of the training set for the brood evaluation (i.e. 1.0 means to use the whole subset). Brooding increases the success rate of the crossover, since normally the crossover generates estimated 75% of non-functioning individuals, which should not even be considered. On the other hand, brooding decreases the creativity of the search slightly, while some innovative offspring will perform worse than some other offspring of the same parents – in particular those that perform almost or exactly the same as their parent. When they perform the same, it is often due to the fact that the difference in the genotypes of the parent and the offspring is irrelevant for the resulting behavior of the offspring. And while the parent already has a relatively high fitness, the new useful innovative genotypes with somewhat smaller fitness than the parent are not accepted.

Therefore the brooding crossover can either be limited by the parameter *pbrooding_crossover*, or the user may require that the offspring with the same fitness as the parent genotypes be discarded (parameter *strict_brooding*), which, however, works well only for deterministic objective functions (i.e. the same genotype always has the same fitness).

3.3 Mutation

For the GP-tree representation, EI utilizes the following operators:

- *mut_change*: changes a random node (terminals → terminals), non-terminals → non-terminals, or changes an argument of terminal/non-terminal, if any;
- *mut_exchange*: exchanges two arbitrary nodes within the individual;
- *mut_insert*: inserts a non-terminal node with a full random sub-tree somewhere inside of the individual;
- *mut_remove*: removes random node/sub-tree within individual;
- *random_node*: replaces the whole individual with a completely new individual;

For the FSA representation, EI utilizes the following operators:

- *mut_change*: changes a random transition: either by changing terminal, destination state (either randomly or by following another transition from the original destination state), changing relation, or splitting the transition to two and inserting new state in the middle; alternately, it randomly reorders the transitions in a single random state;
- *mut_exchange*: picks two states A, B, and redirects all transitions leading to A and to B to point to the other one of the two states instead;

- *mut_insert*: either inserts a new random transition between two existing states or inserts a new state randomly connected to existing states by at least one incoming and one outgoing transition.
- *mut_remove*: removes a random transition or a random state;
- *random_fsa*: replaces the whole individual with a completely new individual;

The probability of each mutation type is by default the same, except that the *mut_insert* is by default applied with 3-times higher probability. However, if the selected mutation operator can not be applied to the genotype (for instance the maximum number of transitions has been reached), another operator is selected (and this is repeated at most three times).

3.4 Selection and the remaining parameters and features

EI supports tournament selection with adjustable tournament size and the probability of selecting the tournament winner (if less than 1.0, there is a chance that the best individual of the selected group will not be selected – this is to make the search more stochastic, and improve chances to escape local extremes). The user can also require that the selected individuals are removed from the population to ensure that all members of the population participate in the selection.

Alternately, fitness-proportionate selection (i.e. roulette wheel) can be used.

Optionally, the fitness can be normalized to interval [0,1] and squared in order to increase selection pressure. The squared normalized fitness is obtained by the following formula:

$$NormalizedFitness_i = \left(\frac{Fitness_i - MinFitness}{MaxFitness - MinFitness}\right)^2$$

EI supports two types of elitism – either the best *num_elitism* individuals are automatically copied from the previous generation, or alternately the best *num_elitism different* individuals are copied. Requiring that the elites be different improves the performance of the algorithm significantly, especially in the cases when the objective function is not deterministic, and the training set used by the fitness function is random. In those cases, the better individuals can temporarily in one or very few generations perform worse, and be lost when defeated by a lucky genotype tailored for a set of the random special cases chosen for evaluation in those generations. This requirement of difference contributes also to premature convergence prevention.

The EI package has support for pretty-printing of genotypes, loading and saving environments, and easy addition of new experimental platforms. It allows saving and restoring the state of the evolution anytime during the run, and this can be performed automatically periodically.

EI was designed for easy extensibility with new experiments. Please consult the software documentation for more details. The software is an open-source project, freely available from [24]. The list of all parameters appears in Appendix A.

4 Experimental tasks

In order to asses the performance of the state representations we have designed five tasks of different nature, falling in two categories: controlling a robotic agent in a two-dimensional environment, and processing symbolic sequences prepared on a one-dimensional bi-directional, possibly infinite tape.

4.1 Experiment "bit_collect"

This task is designed to verify the ability of the evolutionary algorithm to encode algorithmic structures in the chosen representation. The input to the system is a word consisting of bits (0s and 1s) printed on a tape. The start and the end of the word can optionally be marked by surrounding -1. The tape can either be infinite, or finite. In the latter case, moving outside of the tape causes the program to terminate. There is a current read/write pointer that points to one symbol on the tape. The evolved program can perform the following operations:

- **left** – move the current read/write pointer one symbol to the left
- **right** – move the current read/write pointer one symbol to the right
- **write0** – write zero to the tape at the current read/write pointer
- **write1** – write one to the tape at the current read/write pointer
- **done** – task completed, terminate

The program has at disposition the symbol of the tape at the position of the current read/write pointer (register `R1`). The task for the program is either to fill all holes (tape positions containing zeros) with ones – in the easy version, or in a difficult version to pack – move the symbols at the tape in such a way that the remaining word will consist of a continuous sequence of ones, the same number as the total number of ones in the input word. The program can write arbitrary number of zeros on both sides of the output word. For example, the input:

```
10111001010001
```

would be transformed by a correct program to:

```
11111111111111
```

in the easy version, or to (for example):

```
11111110000000
```

in the difficult version of the task. The computing platform in this task is similar to the Turing Machine. The performance of the program is measured in terms of the number of errors – each extra "1" as well as each missing "1" is penalized by one point. In addition, all remaining holes – symbols "0" – are penalized by one point each. Fitness function:

$$fitness = B - s \cdot q_s - \sum_{i=1}^{n_{starts}} (r_i \cdot q_r + \frac{h_i}{H_i} \cdot q_h + \frac{o_i}{O_i} \cdot q_o)$$

where n_{starts} is the number of random input words presented to the program, H_i and O_i is the number of holes and ones in the i^{th} input word respectively, h_i is the number of holes remaining in the output word, o_i is the difference in the number of ones expected (either too much or too little), r_i is the number of execution steps, s is the size of the genotype, and q_s, q_r, q_h, and q_o are weight constants. Coefficients q_h and q_o were strictly more significant than q_s and q_r, and in balance (we used $q_o = 2q_h$).

4.2 Experiment "(abcd)^n"

In this task, we test the ability of the representation to encode repetitive structures. Using the same computational model of the tape state machine as in the previous task, the goal is to replace a continuous sequence of symbols "1" on the tape of random length with a repeating sequence of symbols `a, b, c, d`. For instance, the input:

```
11111111111111111111
```

would be transformed by a correct program to:

```
abcdabcdabcdabcdabcd
```

The allowed operations are the same as in the previous task, with the addition of the operations that write the symbols `a, b, c,` and `d`. In a simplified version of this task, we require the sequence (abc)^n. Fitness function:

$$fitness = 1 - s \cdot q_s - \sum_{i=1}^{n_{starts}} \left(\frac{\frac{e_i}{l_i}}{n_{starts}} - r_i \cdot q_r \right)$$

where e_i is the number of incorrect symbols in the output word (including extra placed or missing symbols) in the i^{th} input word, l_i is the number of symbols in the input word, and the meaning of the other symbols is the same as in the previous task.

4.3 Experiment "switch"

This is a task with a structure that shares properties with the structures of robotic tasks where the robot reacts to environmental percepts depending of its current state, and enters other states when triggered by some input data. The computational platform – a tape machine – is the same as in the previous two tasks. The task for the program is to replace all zeros on the tape with numeric symbols 1, 2, 3, and 4. The input sequence on the tape determines how the zeros are to be replaced: the input contains random symbols 1, 2, 3, and 4 that are interleaved with sequences of zeros, each 0-sequence containing up to 10 zeros. The program should replace the zeros with the closest non-zero input on the left. When the program leaves the tape, it is automatically terminated. For instance, the following input:

```
1000403000020001300400000000003000020
```

should be transformed to:

```
1111443333322221333444444444443333322
```

The performance of the program is measured as the sum of errors from the expected string. The allowed operations are the same as in the previous tasks, except that the program is allowed to write the symbols 0, 1, 2, 3, and 4. In a simplified version of the task, the input may contain (and the program can write) only the symbols 0, 1, 2, and 3. The fitness function is the same as in the task abcd^n. We have experimented with incremental versions of this task, which are described in the results section below.

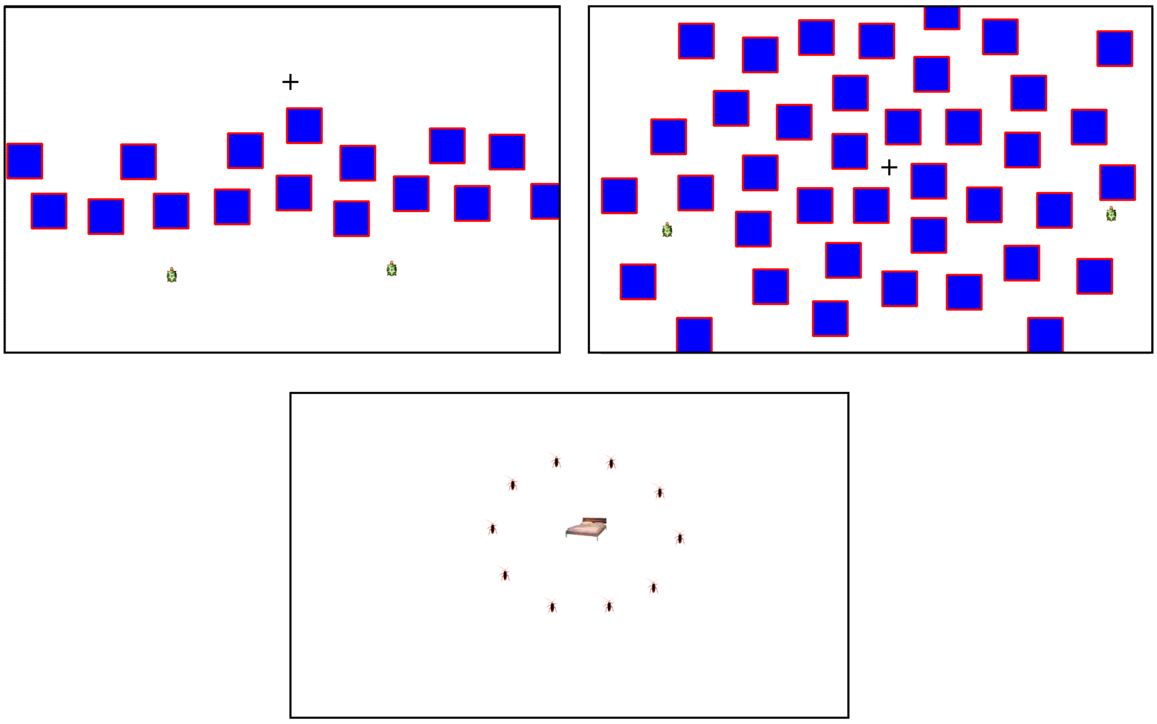


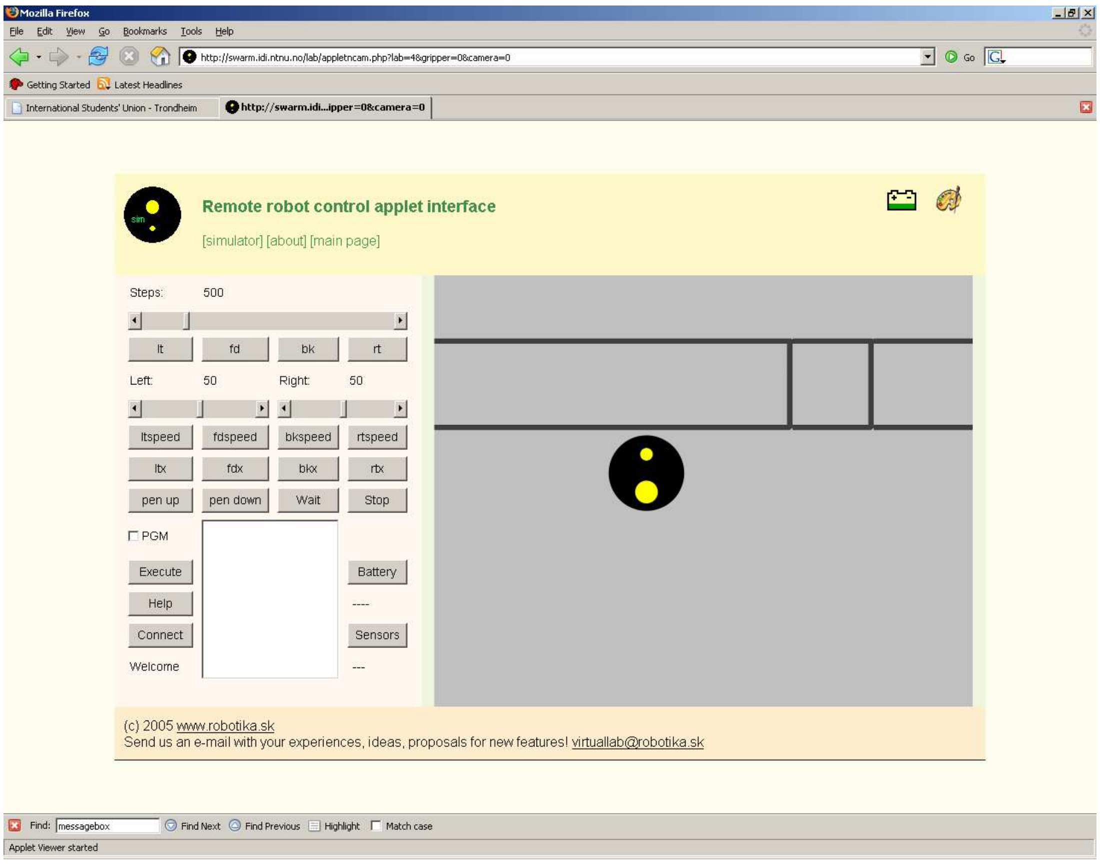
Figure 5: Different environments for the "find_target" task (from top-left: `experiment_fence`, `complex_environment`, `ten_around`). In the first two, the robot (depicted by a turtle) navigates the environment towards target marked by the cross, there are two different starting locations. In the last one, there are no obstacles, and 10 different starting locations, all heading upwards.

4.4 Experiment "find_target"

In this experiment, a robotic agent is placed in a 2D environment. It understands the following primitive control commands:

- **fd** – move forward a little bit (usually 20 steps)
- **bk** – move backward a little bit
- **fdlong** – move forward longer distance (usually 100 steps)
- **bklong** – move backwards longer distance
- **lt** – turn left a little bit (different angle values result in different behaviors: 90 deg results in a square grid along which the agent can move, 60 deg results in a hexagonal grid, which is a very efficient navigational environment, other values that are not divisors of 360 result in the ability of the agent to turn to many different directions by repetitive turnings and thus exploit the environment to higher degree – on the cost of more complex navigational sequences).
- **rt** – turn right a little bit
- **done** – task completed, terminate

In addition, the agent is equipped with three binary sensors: short distance wall detection, long distance wall detection, and target-direction sensor. The wall detection sensors indicate



(c) 2005 www.robotika.sk
Send us an e-mail with your experiences, ideas, proposals for new features! virtuallab@robotika.sk

Figure 6: Viewing the progress of simulation in a web browser using a viewer implemented as Java applet.

whether the agent will hit an obstacle with the next `fd`, or `fdlong` command, while the target-direction sensor indicates whether the agent is heading towards the target. The sensor readings are always available to the agent in form of three registers `R1`, `R2`, `R3`.

The task for the robotic agent is starting from an arbitrary starting location in the 2D world to find a path to the target that does not collide with the obstacles. Each collision is penalized. The performance of the agent is evaluated based on its distance from the target location at the moment it stops moving (see below). When the agent arrives at the boundary of the 2D world, it is not penalized, but it slides along the boundary when it tries to move in a direction that is non-perpendicular to the boundary. Since the robot should arrive to the destination from different starting locations, it has to develop strategies that are at least somewhat general. For instance a simple linear sequence of left and right turns and forward movements would be a satisfactory solution in case of one starting location, but conditional branching that results in different trajectories is essential when two or more starting locations are used. Eventually, when the agent is trained at many starting locations, it might develop a general strategy applicable to an arbitrary starting point of the 2D world. What is the limiting number of starting locations that leads to general behavior is an interesting question to study. We have performed tests with both types of environments: with and without obstacles. The figure 5 shows three different environments used in our experiments.

The fitness function used in this experiment:

$$fitness = B - s \cdot q_s - \sum_{i=1}^{n_{starts}} (d^2(T, P_i) + r_i \cdot q_r + h_i \cdot q_h)$$

where n_{starts} is the number of starting locations, $d(A, B)$ is a function returning the distance of two points A and B, T is the target location, P_i are the locations where the robot stopped moving, s is the size of the genotype, r_i is the number of execution steps – either state transitions, or evaluated nodes in GP-tree, and h_i is the number of times the agent crosses the edge of the pond. The weight coefficients q_s, q_r, q_h are chosen to comply with strict significance relation: hits $\gg$ squared distance $\gg$ size $\gg$ number of steps. B is a sufficiently large number to keep the fitness positive – and maximizing.

4.5 Experiment "dock"

In this experiment, we utilize the ability of the Imagine software environment to connect to simulated and real robots. A circular robot is placed in a rectangular environment. It understands the same set of primitive operations as in the task "find_target" above. In addition, the robot is equipped with bottom light sensors in the front and in the back that can distinguish white and black color on the floor. The primitive operations `stopON` and `stopOFF` turn on the sensitivity to the black color: whenever the front or rear part of the robot – depending whether it is moving forward or backwards – enters or remains above the black surface, the movement commands are cancelled. The rotation commands `lt` and `rt` are not suppressed. The feedback to the program is passed through the register `R1`, which is set to 0, if the robot stopped moving because it entered black surface, and it is reset to 1 otherwise.

The task for the robot is to navigate to a target location, which is marked by a black rectangle. A pair of "horizontal" parallel lines that are extensions of two opposite sides of the rectangle pass though the whole width of the environment. The starting locations of the robot are in a quadrant "under" these two parallel lines and "to the left" of the rectangle, see figure 6, which shows the simulated environment as viewed by an applet in a web browser.

The performance of the program is measured as the distance of the centre of the robot from the centre of the rectangle when it stopped moving.

$$fitness = B - s \cdot q_s - \sum_{i=1}^{n_{starts}} (d^2(T, P_i) + r_i \cdot q_r)$$

5 Results

Various flavors of the "find_target" task with GP-tree representation were used to build and test the software engine, and find starting feasible set of parameters. The GP-trees were successful in quickly encoding specific trajectories. While the number of the starting locations remained low, only a couple of sensor conditions in a resulting program were sufficient to generate different and correct[8] trajectories. Utilizing the `seq` non-terminal, the trajectories represented as GP-trees are easy to extend with preceding or succeeding trajectory segments. Encoding trajectories, which have random and non-repetitive shape using FSA representations is far less suitable. Almost each new trajectory segment requires assembling a new fragile state with a very specific structure (number of transitions, their destinations, and actions on the transitions). The new state is, during the continuation of the evolution, subject to further disruptive changes. Chart in figure 7 shows the progress of the evolution – best individuals for both GP-tree and FSA representations in an environment with two starting locations and 16 obstacles arranged in a row. We required that the agent arrives to the target location (in a distance less than one short step). The state-representations evolved solution in 21, 70, 77, 105, 120, 148, 157, 182, 197, 211, 486, 520, and 558 generations, while the GP-trees evolved in 19, 21, 21, 24, 32, 33, 43, 43,

[8]By correct we mean that the agent both successfully avoids collisions with obstacles and navigates from the assigned starting location to the target.

```
[seq ()
  [seq ()
    [seq ()
      [if (R1 > 1)
        [rt]
        [repeat (1)
          [seq ()
            [seq ()
              [repeat (4)
                [longfd]
                [lt]]
              [lt]]
            [bk]]
          [fd]]]
      [repeat (6)
        [fd]
        [repeat (2)
          [fd]
          [lt]]]]
    [if (R1 > 1)
      [nop]
      [repeat (1)
        [repeat (10)
          [fd]
          [bk]]
        [seq ()
          [rt]
          [fd]]]]]
  [fd]]
```

```
6 states
—state 1 with 2 transitions
[R3 < 0] 2 [fd]
[R3 < 1] 2 [bk]
—state 2 with 1 transitions
[R3 < 1] 3 [bk]
—state 3 with 1 transitions
[R2 < 1] 4 [bk]
—state 4 with 1 transitions
[R3 < 1] 5 [bk]
—state 5 with 2 transitions
[R3 > 1] 6 [rt]
[R1 > 0] 5 [fd]
—state 6 with 2 transitions
[R1 > 0] 2 [rt]
[R1 < 1] 5 [longfd]
```

Table 1: A final evolved GP-tree and FSA (after trimming) for the environment `experiment_fence` from one evolutionary run.

43, 46, 47, 50, 73, 73, 73, 73, 75, and 98 generations. The fastest-evolved GP-tree and FSA are shown in table 1.

The figure 8 shows the trajectories of best individuals from all generations for both representations. Characteristic features of the GP-tree trajectories are branching, and easy extensions, while the FSA trajectories are good at making loops and traveling long distances where the sensor conditions do not change.

Both representations evolved solutions for `complex_environment` – an environment with two starting locations, and 40 obstacles. Tree representation used 57 generations, while FSA representation used 109 generations (both with population size 250, probability of mutation 0.7, probability of crossover 0.5, strict brooding crossover with brood size 4, tournament selection with size 4 and prob. 0.8, 15 non-duplicit elite individuals). Figure 9 shows the trajectories of the best individuals in all generations for both representations. We can observe from the figure that FSA representation evolved first an individual that was avoiding the obstacles from the left, and only later preferred the direction towards the center. In comparison, the tree representation approached the target location in gradual approximation, coming somewhat closer each time. The tree representation encoded the trajectories more directly, extending them slowly by successfully appended segments. The FSA representation encoded *strategies*, which either

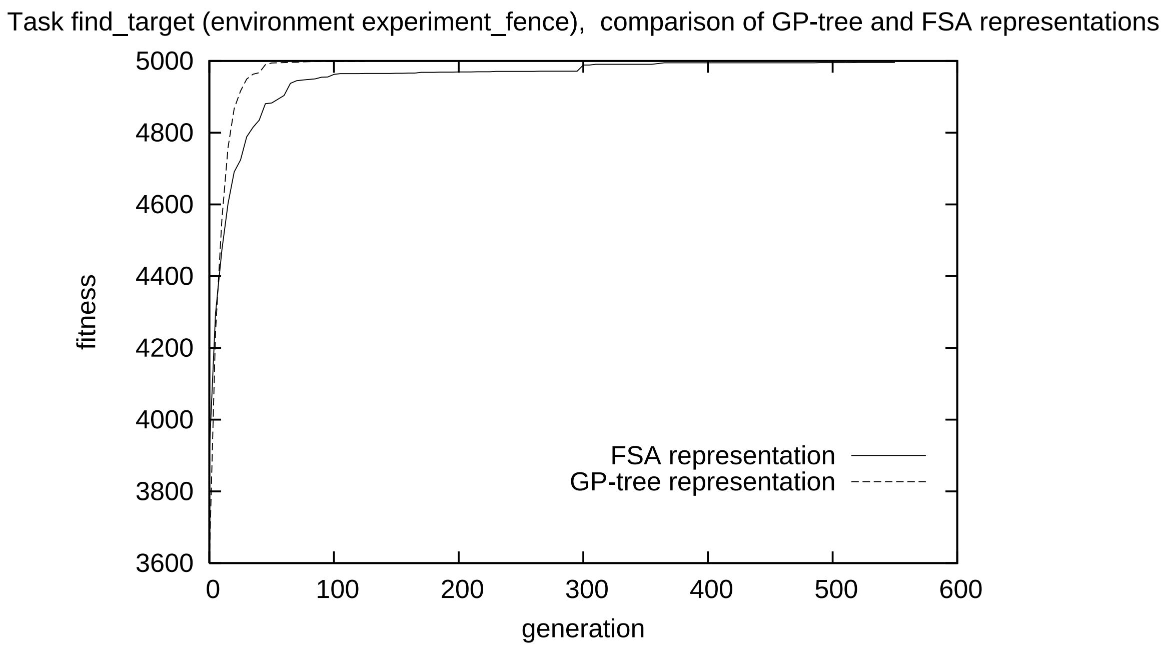


Figure 7: Average of the fitness of the best individuals in the “find_target” task, environment `experiment_fence`, comparison from 13 FSA and 17 GP-tree runs. The maximum possible fitness is 5000.

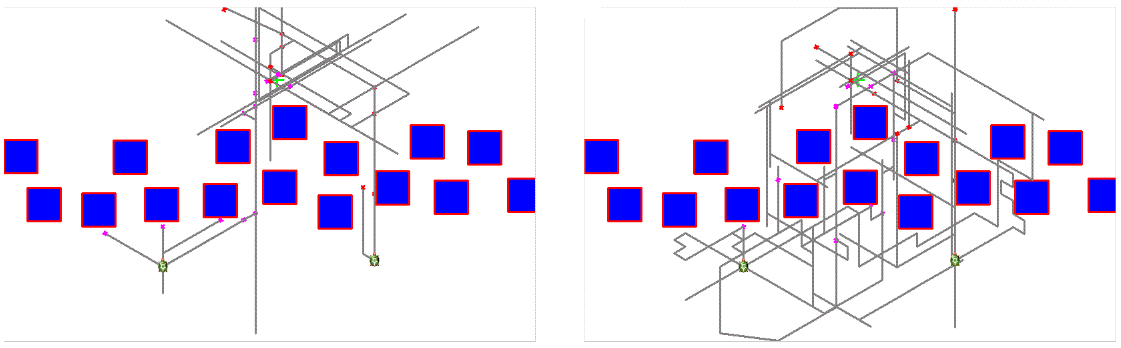


Figure 8: Example trajectories of evolved individuals using the GP-tree (left) and the FSA (right) representations, task “find_target”, environment `experiment_fence`. The set of trajectories resembles the internals of the representations: loops are easier to be formed in FSA representations, the GP-tree representation that is subject of crossover and structural mutation often takes a part of a solution and extends it with further trajectory segments.

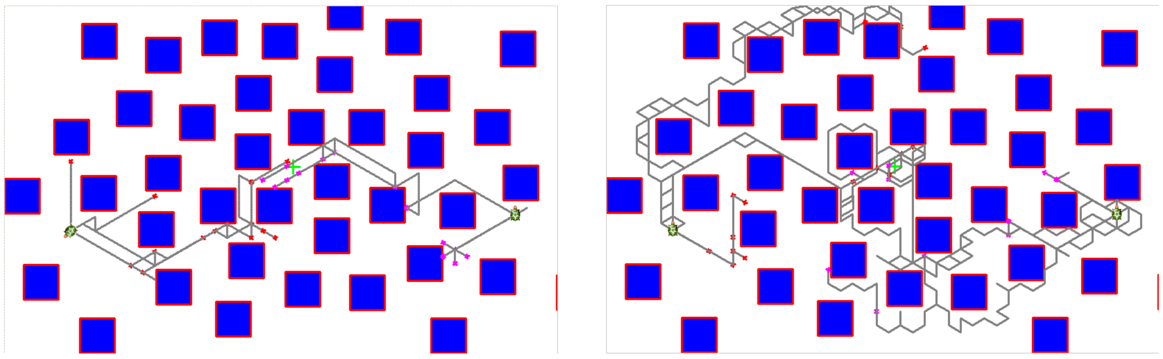


Figure 9: Trajectories of the best individuals from all generations in one evolutionary run, task “find_target” with complex environment. GP-tree representation is on the left-hand side, FSA representation is on the right-hand side.

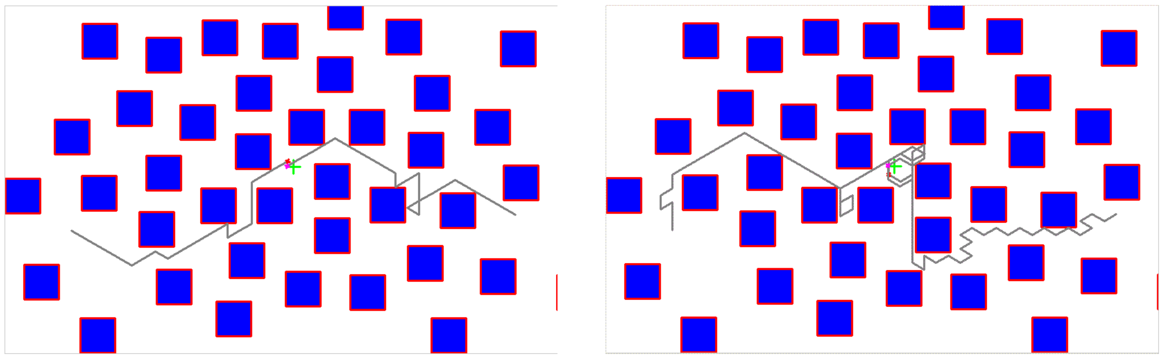


Figure 10: Trajectories of the best individuals from final generation in one evolutionary run, task “find_target” with complex environment. GP-tree representation is on the left-hand side, FSA representation is on the right-hand side.

performed well – in a lucky case, or took the individual astray the route to the target. This is supported also by the trajectories of the evolved best individuals in both representations (figure 10): the FSA individual arrives to the target and remains circling around it, while the GP-tree individual arrives to the target and terminates. However, both runs failed to evolve a general solution, the figure 11 shows their performance when started from 7 random starting locations. It yet remains to see under what circumstances a general strategy could be evolved. We tried evolution using all starting locations shown in figure 11, but we terminated the experiment after several days of no progress.

In `ten_around` environment with 10 starting locations and no obstacles, we expected a general navigational strategy to arise. Both representations evolved solutions very quickly, examples are shown in table 2. Most of the time, FSA representation reached a correct solution faster, but the difference is not significant (see figure 12).

Figure 14 shows the performance of the evolved individuals from randomly selected runs for 132 different uniformly distributed starting locations. The agents were restricted with 50 execution steps (same parameter was used during the evolution). The size of the circle corresponds to the performance from the given starting location – the smaller the final distance of the agent from the target the larger the circle. The lines correspond to the trajectories of the agents. We can see that FSA performs better, because the execution steps are more powerful: each execution step corresponds to a single state transition, when the agent either performs a

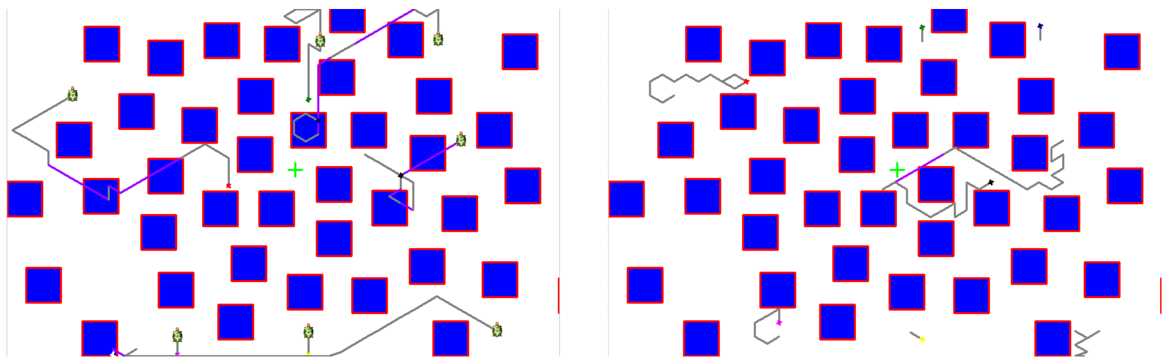


Figure 11: Performance of the best individuals from final generation in one evolutionary run when started from 7 other starting locations, task "find_target" with complex environment. Small crosses depict the final location the individual achieved. GP-tree representation is on the left-hand side, FSA representation is on the right-hand side.

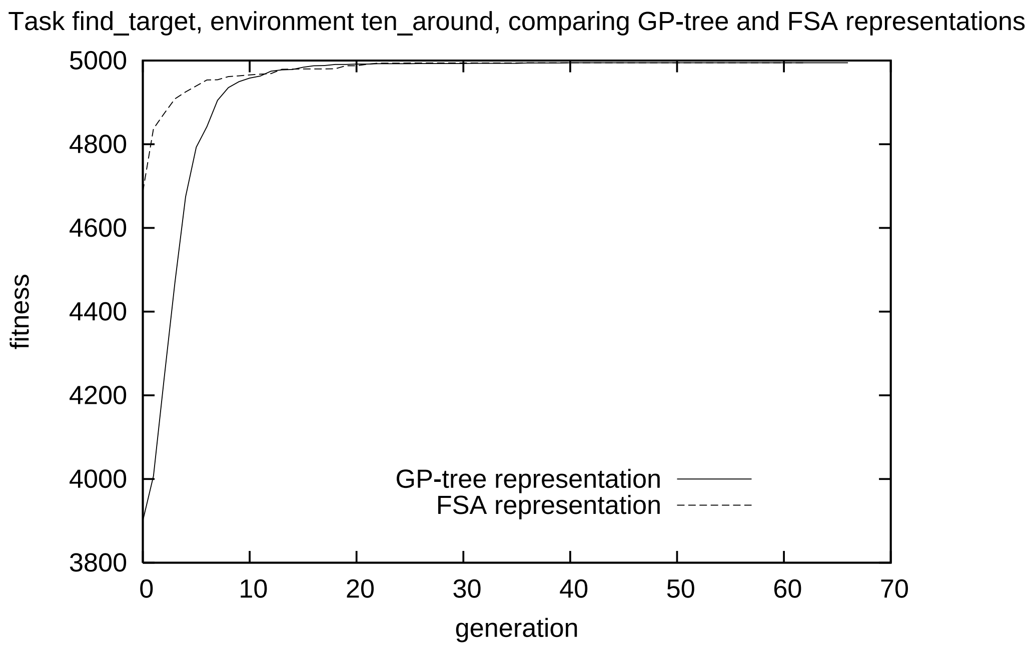


Figure 12: Performance of the FSA and GP-tree representations on task "find_target", environment `ten_around`. Average of 25 (GP-tree) and 23 (FSA) runs. FSA individuals quickly learn to arrive close to the target, but take longer time to fine-tune the solution to arrive exactly to the target location than GP-tree individuals.

```
2 states
—state 1 with 2 transitions
[R3 < 0] 2 [longbk]
[R3 > 1] 2 [fd]
—state 2 with 2 transitions
[R3 > 1] 2 [fd]
[R2 > 0] 2 [lt]
```

```
[repeat (4)
  [while (R2 < 1)
    [if (R2 < 0)
      [nop]
      [while (R3 < 1)
        [rt]
        [fd]]]
    [nop]]
  [lt]]
```

Table 2: A final evolved GP-tree and FSA (after trimming) for the environment `ten_around` from one run of simple "bit_collect" task.

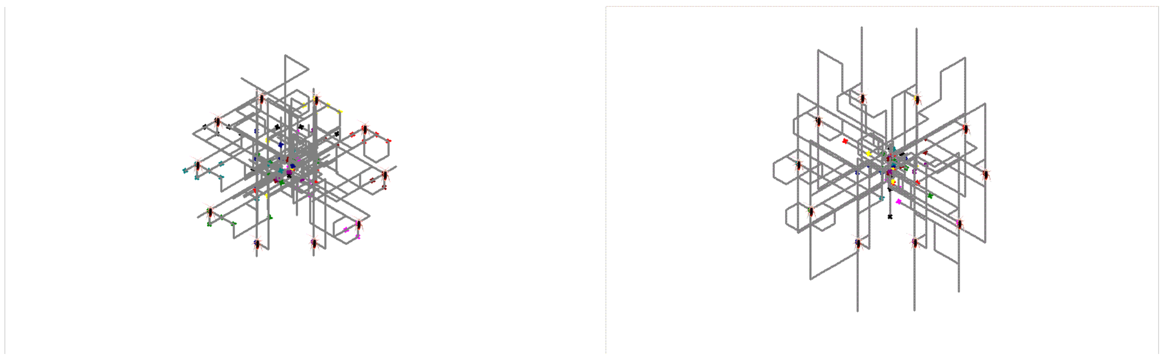


Figure 13: Trajectories of the best individuals from final generation in one evolutionary run, task "find_target" with `ten_around` environment. GP-tree representation is on the left-hand side, FSA representation is on the right-hand side.

move, or a turn. In the GP-tree representations execution steps correspond both to executing the nodes containing terminals and non-terminals. If the number of execution steps was not restricted, both representations reached target in an optimal way from any location.

We have experimented with both flavors of the "bit_collect" task: an easy version where the holes (zeros) in the input word need to be filled with ones, and a much more complex one, where the holes need to be "moved away". In the easy version, both representations evolved correct solutions quickly, although the FSA representation was quicker on average, see figure 15.

The difference of performance is larger in the more complex version of the task, see figure 16. None of the runs evolved correct solution in 600 generations. A correct solution required the general strategy to be acquired – that is repeatedly search the input word for a hole, then move or propagate it to the end or the start of the input word. This strategy is difficult to discover in approximating steps. Evolved solutions were thus typically only estimating an average number of holes to be compensated for. Thus, we have still not sufficiently answered the question whether the GP-tree or FSA representation is more successful in acquiring [this kind of] algorithmic solutions, this remains for future investigations, the task was either easy or too difficult for both.

The task $(abcd)^n$, proved to be a challenging one for both representations. The programs replacing the whole input word with the same symbol quickly appeared scoring high as they filled 25% of tape slots correctly. Gradual modification of this dominant strategy appeared to be non-trivial, because our computational model requires the program to first write the symbol and then move to the next tape slot with a separate instruction. Thus producing an individual

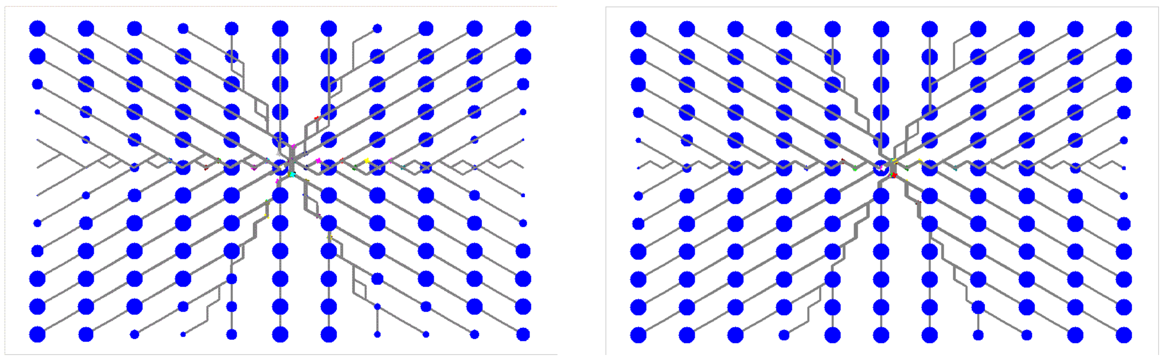


Figure 14: Trajectories and performance of best individuals from final generation in one evolutionary run, task “find_target” Trajectories of the best individuals from final generation in one evolutionary run, task “find_target” with `ten_around` environment, starting locations that were not used in training. GP-tree representation is on the left-hand side, FSA representation is on the right-hand side.

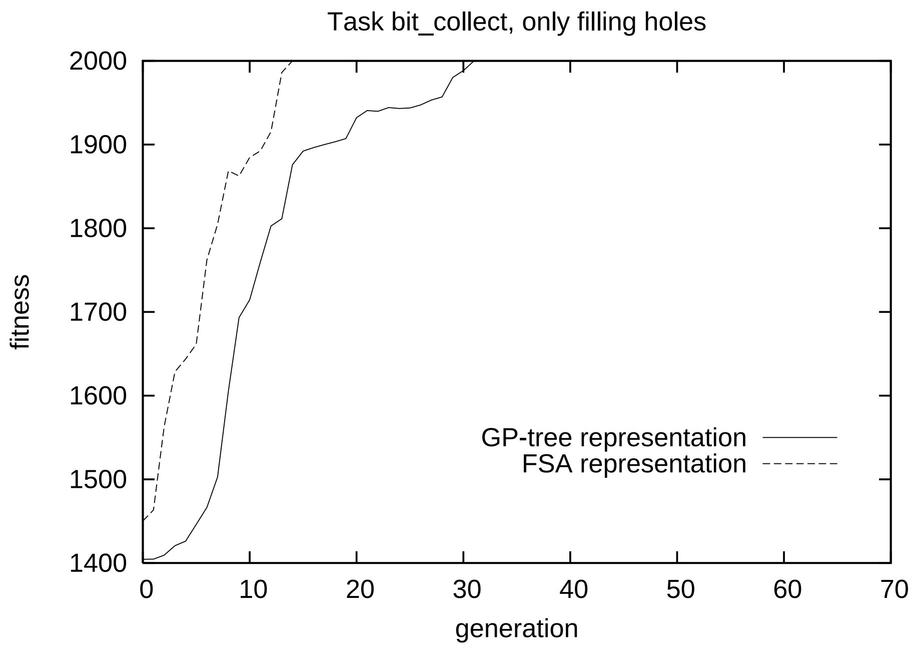


Figure 15: Performance of the FSA and GP-tree representations on simple version of task “bit_collect”. Average of 19 (GP-tree) and 22 (FSA) runs. The programs were presented 20 input words of length 5 to 30, containing about 75% of ones, and were allowed to make at most 200 execution steps. Other parameters: population size: 250, prob. crossover: 0.5, brooding crossover (number of non-strict broods 3, 30% of training samples used for brooding), combining crossover (GP-trees): 0.25, prob. mutation: 0.7, 7 elite individuals, tournament selection (tournament size 4, probability 0.8), max. GP-tree depth: 12, max. number of FSA states/transitions: 22/10, FSA shuffle mutation: 0.4.

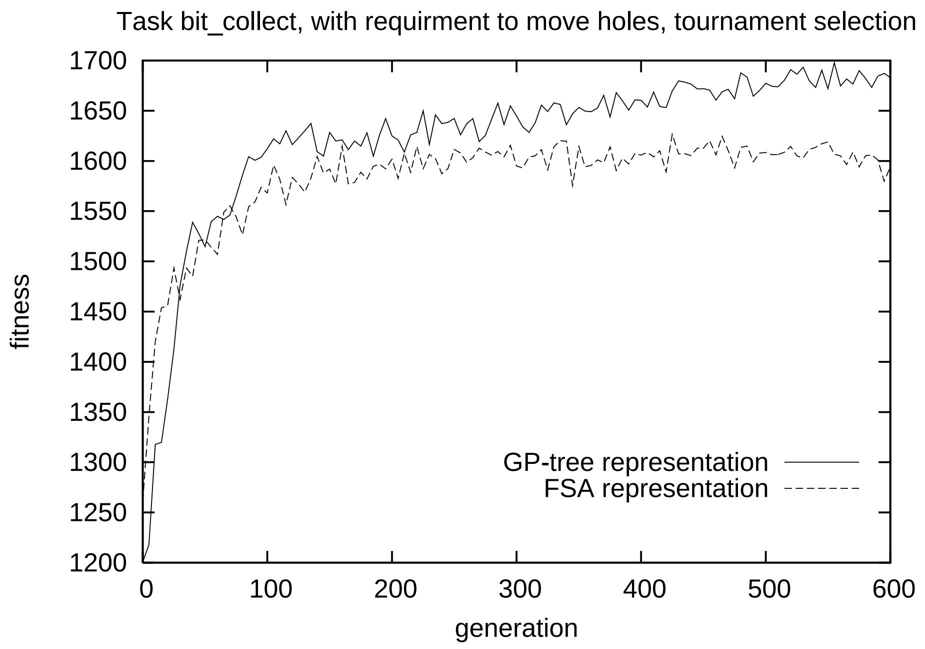


Figure 16: Performance of the FSA and GP-tree representations on more complex version of task "bit_collect". Average of best individuals in each generation from 9 (GP-tree) and 13 (FSA) runs. Other parameters were the same as in the simple version of the task, we terminated the runs after 600 generations if the solution did not evolve.

that alternates two symbols already requires 4 correctly-tuned transitions, while filling with one symbol needs only two. GP-tree representation performed somewhat better on this task, see figure 17. The `while` non-terminal used in GP-trees is very powerful in this task. The programs need to keep writing the repeated sequence and moving right while there is a non-zero symbol in the input. Only 17 out of 23 FSA runs (74%) evolved a correct solution, while 23 out of 25 GP-tree runs (92%) succeeded within 2000 generations. The remaining 2 GP-tree runs placed only 2 symbols incorrectly, while the incorrect FSA erred on 7–14 symbols. The table 5 shows the shortest evolved GP-tree and FSA, they are both easy to trace and understand.

The task *switch* is an example, where state representation outperforms the GP-tree representation, and here we have performed several experiments. We started with experiments with four symbols `A,B,C,D`, however, we found the task to be too difficult – none of the GP-tree runs found anything better than "doing nothing" solution, and only 3 out of 10 runs with FSA representation found a complete solution within 2000 generations, see figure 18. Thus we reverted to a simpler version of the task with 3 symbols `A,B,C`. Figure 19 shows the best fitness progress for both representations. The convergence varied very much – the fastest run found solution after 77 generations, the slowest after 1887 generations, and on average the solution was found after 594 generations (median 383). One possible explanation of this local-optimum traps could be our using of the fast-converging tournament selection (we used tournament size 2, and probability of selecting winning individual 0.8), however, since fitness-proportionate selection seems to perform worse, and since all runs eventually evolved a target solution, we did not try to replace it with different selection mechanism [yet]. How to escape these local optima remains for future studies.

With one exception, all runs with the FSA representation evolved a correct solution[9], while no runs with the GP-tree representation found a correct solution, both within 2000 generations.

[9]In one run, the evolved solution produced usually no errors, but still failed on some input strings.

```
6 states                          [while (R1 == 1)
—state 1 with 5 transitions         [while (R1 == 1)
[R1 == 1] 2 [write2]                  [seq ()
[R1 == 5] 2 [right]                     [while (R1 == 1)
[R1 == 3] 3 [write2]                      [seq ()
[R1 == 2] 2 [right]                         [write2]
[R1 == 4] 2 [right]                         [seq ()
—state 2 with 3 transitions                   [right]
[R1 == 5] 1 [write4]                          [write3]]]
[R1 == 1] 4 [write4]                      [seq ()
[R1 == 2] 4 [right]                         [right]
—state 3 with 6 transitions                 [write4]]]
[R1 == 0] 3 [left]                      [seq ()
[R1 == 4] 2 [write3]                      [right]
[R1 == 2] 2 [write2]                      [write5]]]
[R1 == 5] 2 [left]                    [right]]
[R1 == 3] 2 [right]                 [right]]
[R1 == 1] 3 [write5]
—state 4 with 5 transitions
[R1 == 3] 1 [done]
[R1 == 5] 1 [right]
[R1 == 2] 5 [right]
[R1 == 1] 6 [write4]
[R1 == 4] 5 [right]
—state 5 with 1 transitions
[R1 == 1] 4 [write5]
—state 6 with 3 transitions
[R1 == 2] 1 [write2]
[R1 == 4] 3 [write3]
[R1 == 0] 6 [right]
```

Table 3: A final evolved GP-tree and FSA (after trimming) for the environment `ten_around` from one run of "abcd^n" task.

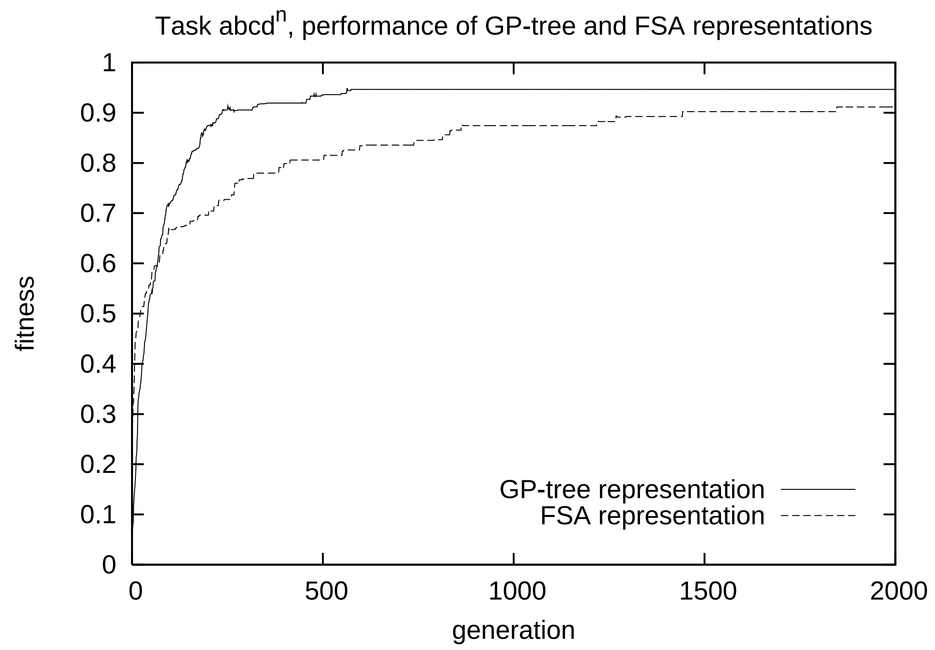


Figure 17: Performance of the FSA and GP-tree representations on task "abcd^n". Average of 25 (GP-tree) and 23 (FSA) runs. The programs were presented with input word containing 32 ones, and had 150 execution steps for writing the output word. Population size 300, prob. crossover 0.6, 15 strict-elite individuals. Other parameters were the same as in the "bit_collect" task, we terminated the runs after 2000 generations if the solution did not evolve.

All parameters common for both representations were the same. The smallest evolved FSA contained four states, but used only three, and it is shown at the figure 20 after pruning redundant transitions and state. The distributions of the sizes of the best individuals in the final generations and their useful parts are shown in table 4. In our runs, we did not prune the FSA during the evolution in order to keep the possibly reusable genetic material in the states that are not reachable from the starting state.

Even though this task has been designed with having the FSA representation in mind, we believe that many tasks in various domains, including autonomous robot control, may have similar structure. We believe and our results suggest that the GP-tree and FSA representations are to high degree complementing each other. Tasks where FSA perform well may be difficult for GP-tree representation. This hypothesis, however, needs to be studied in more depth, and verified on more cases.

A win/win compromise could be hybrid representations – either GP-trees with state machines in the nodes, or state machines with GP-tree code on the state transitions, or inside of the states. The choice between the two should again depend on the task structure.

We observe and conclude that the tasks where FSA representation is suitable deal with processing streams of data, where chunks of data of the same type repeat in many instances, and where the sequence of interactions of the evolved program with the input contains specific patterns and reactions. In this context, it would be very interesting to compare the performance against other representations. For instance, Benson [5] for the purposes of automatic target detection classification problem developed EMMA representation, which is a FSM that contains GP-trees in each state. Also of a high relevance is the work of Koza [19] on automatically defined functions.

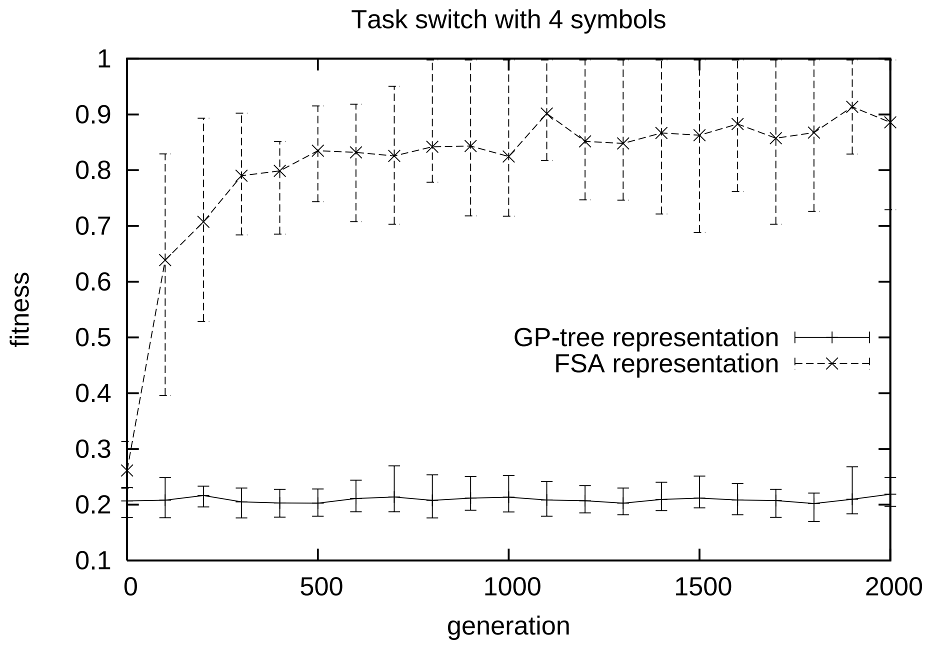


Figure 18: Average of the best fitness from 14 (GP), or 10 runs (FSA) on a switch task with tournament (FSA), or fitness proportionate (GP-trees) selection, population size 300, prob. of crossover 0.5, crossover brooding of size 3, with 0.3 starting locations used to evaluate brooding individuals, probability of mutation 0.9, 15 elites, each individual evaluated on 10 random strings, input word length randomly varying from 10 to 60 with maximum 10 continuous 0-symbols, maximum number of GP-tree or FSA execution steps 300, FSA: pshuffle=0.4, number of states 1–15, number of transitions: 1–15, GP: pcross_combine=0.25, maximum tree depth=15. The number of evaluations is proportional to the generation number. The error bars show the range of fitness progress in all runs.

total number of states	FSA count
4	1
6	1
7	1
8	4
10	6
11	1
12	4
13	2
14	3
15	11

number of reachable states	FSA count
3	2
4	1
5	2
6	5
7	6
8	6
9	5
10	3
11	2
12	1
13	1

Table 4: The number of states in the evolved FSA (left) and the number of states that are reachable (right) in the 34 runs of the switch task with three symbols.

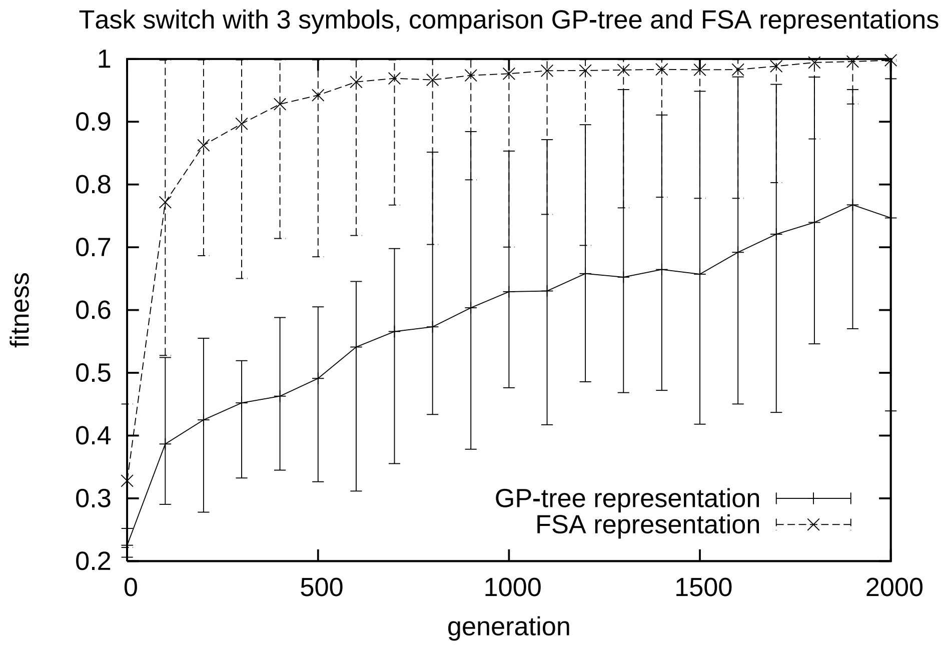


Figure 19: Average of the best fitness from 15 (GP), or 34 runs (FSA) on a switch task with tournament selection, population size 300, prob. of crossover 0.5, crossover brooding of size 3, with 0.3 starting locations used to evaluate brooding individuals, probability of mutation 0.9, 15 elites, each individual evaluated on 10 random strings, input word length randomly varying from 10 to 60 with maximum 10 continuous 0-symbols, maximum number of GP-tree or FSA execution steps 300, FSA: pshuffle=0.4, number of states 1–15, number of transitions: 1–15, GP: pcross_combine=0.25, maximum tree depth=15. The number of evaluations is proportional to the generation number. Also notice that due to the randomness of the testing input strings, the performance in the succeeding generation can decrease, even though the quality of the individual remains or even increases. The error bars show the range of fitness progress in all runs, notice that the partial overlap is only due to the randomness of strings, but the evolved individuals in all FSA runs outperform those with GP-trees.

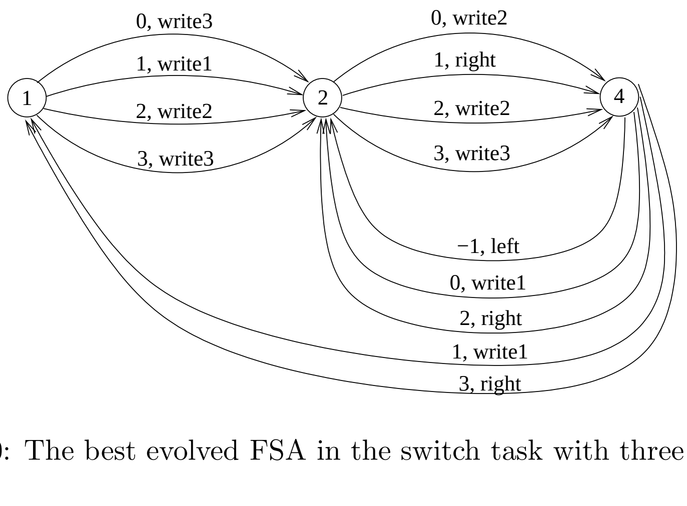


Figure 20: The best evolved FSA in the switch task with three symbols.

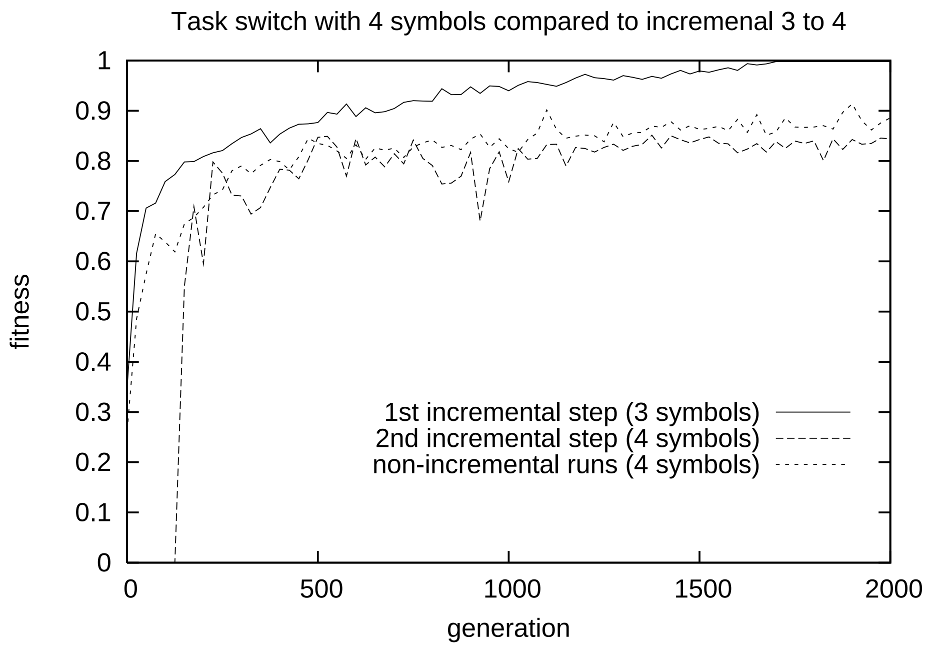


Figure 21: Average of the best fitness from 21 (incremental) and 10 (non-incremental) runs on a 4-symbol switch task with FSA representation and tournament selection, population size 300, prob. of crossover 0.5, crossover non-strict brooding of size 3, with 0.3 starting locations used to evaluate brooding individuals, probability of mutation 0.9, 15 elites, each individual evaluated on 10 random strings, input word length randomly varying from 10 to 60 with maximum 10 continuous 0-symbols, maximum number of execution steps 300, pshuffle=0.4, number of states 1–15, number of transitions: 1–15, The number of evaluations is proportional to the generation number.

Incremental evolutionary experiments

We were not satisfied with the low performance on the 4-symbol version of the "switch" task, and studied if it could be evolved incrementally – starting with simpler task and when solved, increasing the task difficulty, and optionally modifying the set of terminals.

We started with a simple idea of first evolving "switch3" (using the `write1--3` terminals) and then proceeding to "switch4" by adding the `write4` terminal, and modifying the fitness function and input words generator. We wanted to verify if the number of evaluations required for evolving the complete solution will be less than in a non-incremental "switch4" task. We let the evolution proceed in the first step for 30 extra generations after the solution has been found in order to optimize it and spread more in the population. Figure 21 compares the incremental and non-incremental runs: the line for the first incremental step plots the average from all runs – if the run proceeded to the 2^{nd} step, we assume the final fitness from that run in subsequent generations; the line for the second incremental step averages in each generation all the runs that already proceeded to the second step. The evolution progressed to the second incremental step in generations 1228, 563, 797, 172, 307, 1749, 616, 1192, 229, 924, 918, 1071, 681, 126, 728, 1476, 1679, 852, 327, 556, 788.

From the chart, we can read that the incremental runs did not perform better in this case, and in fact the correct solution was found only in 4 out of 21 runs within 2000 generations. Our analysis attempts to explain this as follows: in the incremental runs, we forced the evolution to progress in one particular direction (evolve "switch3" first). However also the fitness function in the non-incremental case rewarded the partial "switch3" solutions. Thus the selection pressure

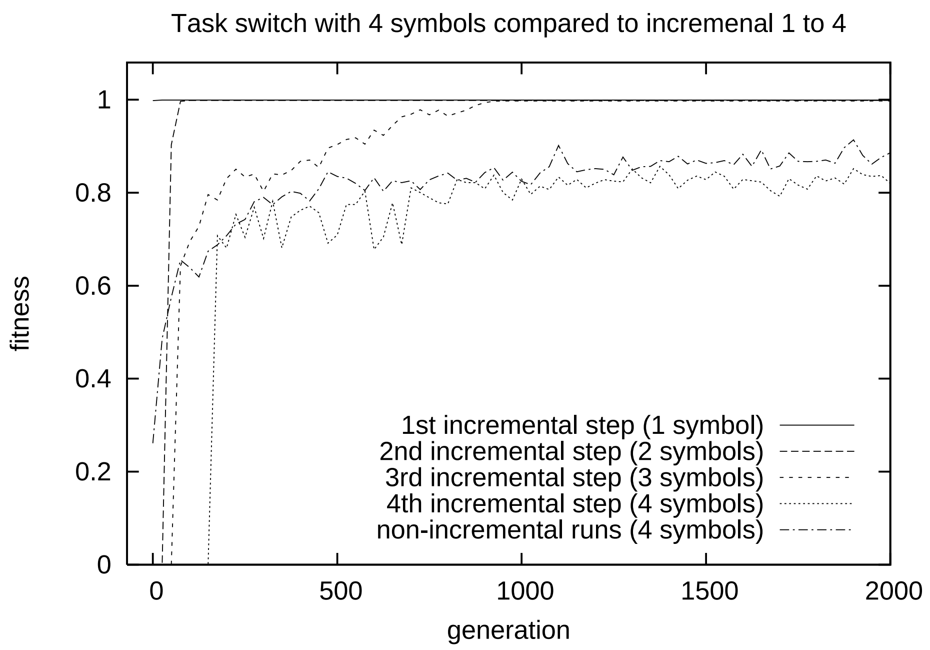


Figure 22: Average of the best fitness from 11 (incremental) and 10 (non-incremental) runs on a 4-symbol switch task with FSA representation and tournament selection. Same parameters as in 3 to 4 incremental task.

in the direction of such partial solutions was similar in both cases. However, non-incremental cases allowed and rewarded also other partial solutions that solved other 75% of the complete task, and these partial solutions might lay on shorter or simply different path to the complete solution, omitting to pass through the complete “switch3” solution “gateway”. This disadvantage exceeded the advantage of dealing only with `write1--3` terminals in the first incremental step – most of the difficulty lay in the final step, where all four symbols were involved.

In the following experiment, we organized the evolution into four incremental steps – requiring first evolution of task dealing with one, then two, three, and finally four symbols. From the above analysis, we could expect that the runs would not outperform the non-incremental ones. Figure 22 confirms this. The transitions to the next incremental step occurred in 31^{st} generation after first step, i.e. solution was found in the first generation, 71^{th}–113^{th} generation after second step, and 183^{rd}–958^{th} (avg. 517) generation after third incremental step.

In the last two experiments, we considered other options for helping the evolutionary process in order to make the incremental method more efficient. After each incremental step, we have frozen the evolved best individual that was a complete and correct solution to the task in that step, and continued the evolution. In later steps, more states and transitions were added, keeping the frozen part unmodified, and dominant. By dominant we mean that the frozen transitions in each state were placed on top, and were chosen first. As a consequence, the later evolutionary step could not change the frozen evolved behavior, only add transitions and states that reacted to new input – novel symbols that were not part of input word or terminal set in earlier steps.

On the other hand, in the first of the two experiments, once a new symbol appears on the input, the FSA could enter another state and react to the old symbols in a different way, and thus disturb the frozen behavior strongly. In other words, the terminal set in later incremental steps still included the terminals to write the old symbols, for instance, the terminals `write1, write2` in the third step.

In the second of the two experiments, the terminal sets in respective incremental steps contained only the new symbol – thus `write1` was only possible during the first step, `write2`

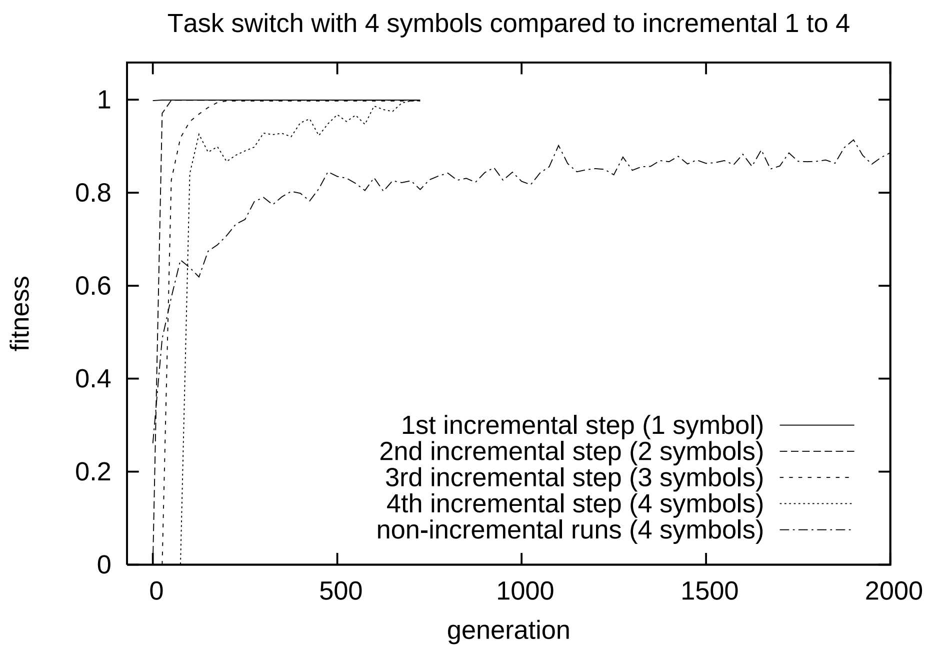


Figure 23: Average of the best fitness from 10 runs on a 4-symbol switch task with FSA representation and tournament selection. Individuals were frozen at the end of each step, and full set of write-terminals was available. Same parameters as in 3 to 4 incremental task.

during the second step, `write3` in the third, and `write4` in the last.

In both experiments, before freezing the FSA, we have removed all unused states and transitions, and we always increased the number of allowed states when proceeding to the next incremental step. We have also changed the number of generations used to fix the solution after it has been evolved in each step to be more gradual, in particular $e = (10 \cdot s)$, where e is the number of extra generations added in step s.

Figure 23 shows an evolutionary progress from the first of the two experiments. The transitions to the next incremental step occurred in 11^{th}, 36^{th}–54^{th} and 82^{nd}–211^{th} (avg. 132) generation. Table 5 shows the evolved frozen individuals from the run that evolved after lowest number of generations.

As expected, the second of the two experiments evolved faster, the performance of the best individuals in each generation is shown in figure 24. The transitions to the next incremental step occurred in 11^{th}, 32^{nd}–46^{th}, and 65^{th}–298^{th} (avg. 132) generation. Both of the last two incremental experiments performed significantly better than the non-incremental experiment.

Finally, we run the "dock" experiment, which was at the very start of the motivation for this work. We tried several different angles for one turning step, and different moving steps. Finally, we attempted to evolve a solution with `turning_angle=90`^{o}, `short_moving_step=20`, and `long_moving_step=400`. Since we have only one installation of the simulator of the remotely-operated robotics laboratory, and the simulator performs only couple of times faster than the real robot, one evolutionary run takes several days. We therefore set to implement a simulator of a simulator directly as part of EI, speeding up the runs by several orders of magnitude. We ran the algorithm with both representations for 2000 generations, with population size 300, tournament selection (4, 0.8), 10 different elite individuals, strict brooding crossover with 2 broods, 0.5 crossover probability, 0.7 mutation probability. Figure 25 shows that the GP-tree representation performed better than the FSA representation.

Here, another important difference between our implementations of the GP-tree and FSA

```
step1: 1 states
—state 1 with 2 transitions
[R1 == 1] 1 [right]
[R1 == 0] 1 [write1]
```

```
step 2: 3 states
—state 1 with 4 transitions
[R1 == 1] 1 [right]
[R1 == 0] 1 [write1]
[R1 == 2] 2 [right]
[R1 == -1] 2 [left]
—state 2 with 4 transitions
[R1 == 1] 3 [done]
[R1 == -1] 1 [left]
[R1 == 2] 3 [write2]
[R1 == 0] 3 [write2]
—state 3 with 3 transitions
[R1 == 2] 3 [right]
[R1 == 0] 1 [left]
[R1 == 1] 3 [right]
```

```
step3: 4 states
—state 1 with 5 transitions
[R1 == 1] 1 [right]
[R1 == 0] 1 [write1]
[R1 == 2] 2 [right]
[R1 == -1] 2 [left]
[R1 == 3] 3 [right]
—state 2 with 5 transitions
[R1 == 1] 4 [done]
[R1 == -1] 1 [left]
[R1 == 2] 4 [write2]
[R1 == 0] 4 [write2]
[R1 == 3] 4 [right]
—state 3 with 4 transitions
[R1 == 2] 2 [left]
[R1 == 3] 4 [right]
[R1 == 0] 4 [write3]
[R1 == 1] 2 [write3]
—state 4 with 4 transitions
[R1 == 2] 4 [right]
[R1 == 0] 1 [left]
[R1 == 1] 4 [right]
[R1 == 3] 4 [right]
```

```
step4: 5 states
—state 1 with 6 transitions
[R1 == 1] 1 [right]
[R1 == 0] 1 [write1]
[R1 == 2] 2 [right]
[R1 == -1] 2 [left]
[R1 == 3] 3 [right]
[R1 == 4] 4 [right]
—state 2 with 5 transitions
[R1 == 1] 5 [done]
[R1 == -1] 1 [left]
[R1 == 2] 5 [write2]
[R1 == 0] 5 [write2]
[R1 == 3] 5 [right]
—state 3 with 5 transitions
[R1 == 2] 2 [left]
[R1 == 3] 5 [right]
[R1 == 0] 5 [write3]
[R1 == 1] 2 [write3]
[R1 == -1] 2 [right]
—state 4 with 5 transitions
[R1 == 4] 2 [write1]
[R1 == 1] 2 [done]
[R1 == -1] 2 [write1]
[R1 == 3] 2 [write4]
[R1 == 0] 2 [write4]
—state 5 with 5 transitions
[R1 == 2] 5 [right]
[R1 == 0] 1 [left]
[R1 == 1] 5 [right]
[R1 == 3] 5 [right]
[R1 == 4] 4 [right]
```

Table 5: Evolved frozen individuals in the incremental steps 1–4.

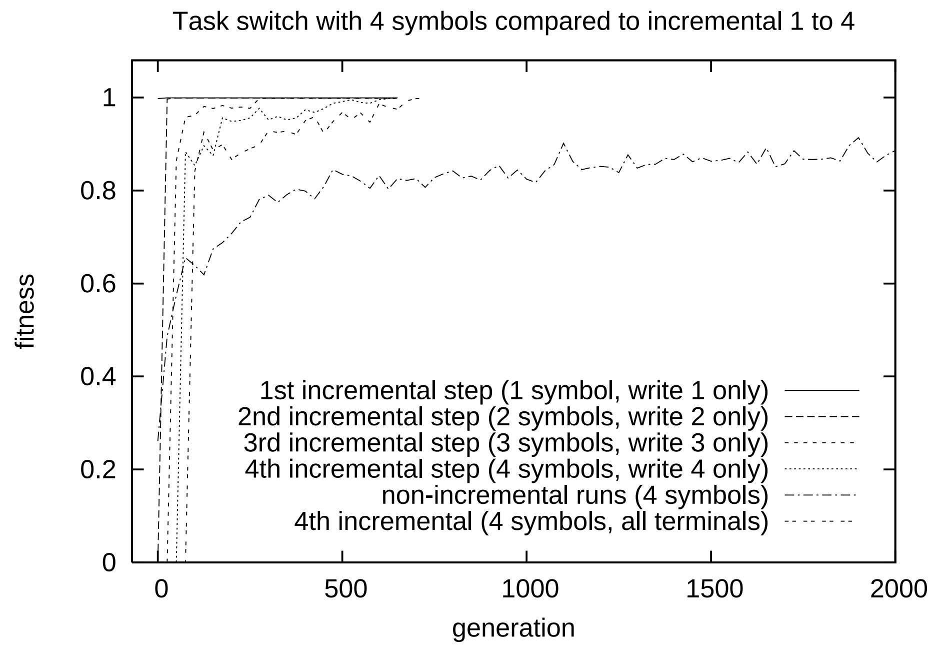


Figure 24: Average of the best fitness from 10 runs on a 4-symbol switch task with FSA representation and tournament selection. The incremental runs had restricted set of terminals. The curve from the previous experiment (full set of terminals) is also plotted for comparison. Same parameters as in 3 to 4 incremental task.

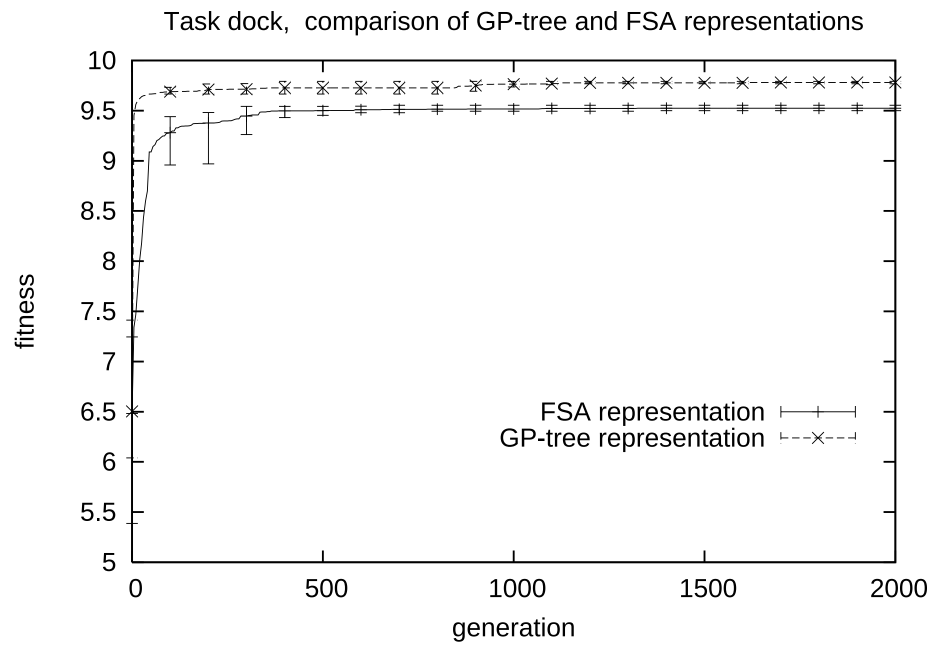


Figure 25: Performance of the GP-tree and FSA representations on simulated "dock" task. Best individual fitness from each generation, average from 10 runs. Errorbars show range.

```
[seq
  [repeat (6)
    [longbk]
    [repeat (10)
      [longfd]
      [rt]]]
  [repeat (7)
    [longfd]
    [repeat (8)
      [longfd]
      [repeat (3)
        [longbk]
        [bk]]]]]
```

Table 6: The best evolved GP-tree solution in the “dock” task when the robots are allowed to push towards wall without penalty.

representations comes to importance. The GP-tree representation can execute commands in a predefined sequence ignoring all sensor readings. On the contrary, each state transition in the FSA representation is triggered only when its transition condition is satisfied – i.e. a particular register contains the required value, i.e. the sensor reading gives that required value. This has two consequences.

First, the FSA representation is more likely to evolve solutions, where the sensitivity to environmental events is important. If in addition there is a possibility of approximating the solution to a large degree using a predefined deterministic sequence of commands, the FSA solutions may achieve a better quality (utilize the sensors instead of deterministic sequences). In this particular experiment, one of the GP-tree representation evolved the solution shown in table 6.

This solution is exploiting the feature of robots being allowed to push against the wall without punishment, which helps them in aligning. The robot aligns itself at the bottom edge of the rectangular area first: it travels backwards, more than the available space allows. Thus, regardless of the y-coordinate of its starting location, it always becomes "horizontally" aligned with other runs started from other starting locations. Next, the robot travels forward to acquire the correct y-coordinate, and then turns right, where again, it travels all the way forward, until it pushes against the right edge, and becomes aligned also "vertically", having the same x-coordinate regardless the x-coordinate of its starting location. Finally, the robot travels back to acquire the requested target location.

The second consequence regards the set of registers available for the FSA representation in a particular experiment. In cases, when the task might require deterministic sequences that do not depend on the sensory input, the set of registers that are used in FSA transitions should include constant registers in addition to those mapped to sensor values. In that way a set of states can be connected by transitions, which are always satisfied and a deterministic piece of behavior can be evolved. However, it is likely that even if the constant registers are available, the GP-tree representation is more suitable for evolving deterministic sequences of commands thanks to the `seq` non-terminal.

Finally, we aimed at finding solutions that are not utilizing the "aligning on the border" feature, because this may contribute to a mechanical damage of the wheels and engines of real robots. We therefore ran the experiment again, giving a penalty ($q_h = 3$) for each movement, which collided with one of the edges of the rectangular area. The overall performance of individuals evolved in 2000 generations dropped slightly, and typical solutions were unable to

15 states (unused states not shown)	[repeat (1)
—state 1 with 1 transitions	[repeat (3)
[R1 == 1] 3 [longfd]	[longfd]
—state 2 with 1 transitions	[seq ()
[R1 == 1] 11 [longfd]	[lt]
—state 3 with 1 transitions	[repeat (6)
[R1 == 1] 6 [stopON]	[longbk]
—state 5 with 2 transitions	[stopON]]]]
[R1 == 0] 2 [longfd]	[repeat (1)
[R1 == 1] 7 [longfd]	[repeat (2)
—state 6 with 1 transitions	[seq ()
[R1 == 1] 12 [bk]	[rt]
—state 7 with 2 transitions	[repeat (4)
[R1 == 1] 8 [longbk]	[longfd]
[R1 == 0] 5 [stopOFF]	[longfd]]]
—state 8 with 1 transitions	[if (R1 == 0)
[R1 == 1] 5 [longfd]	[done]
—state 10 with 1 transitions	[stopOFF]]]
[R1 == 1] 3 [rt]	[seq ()
—state 11 with 1 transitions	[seq ()
[R1 == 1] 10 [longfd]	[lt]
—state 12 with 1 transitions	[longfd]]
[R1 == 1] 5 [fd]	[longfd]]]]

Table 7: Selected evolved individuals with the best performance for the "dock" task. The FSA individual is also shown in the figure 26.

utilize sensors, only moving the robot somewhere close to the target, see figure 27. However, in few cases, the solutions did utilize the sensors and successfully navigated inside of the target square, FSA representation finding a better solution than the GP-tree representation, see figure 28. Individuals for both representations are shown in table 7 (GP-tree representation individual achieved fitness 9.67399 and FSA representation individual achieved fitness 9.77089). At the very end of the experiments, we have performed an experiment with a real robot running in the remotely-controlled laboratory. The evolved FSA individual from table 7 has been run on real robot using the replay feature of the EI software. Robot successfully navigated to target in all of the 10 performed runs when the initial heading was correct. However, when the power of the battery dropped below critical level, the sensors were returning incorrect values, and the performance was compromised (the sensors operate well only with fresh battery – in the future versions of the robot, we therefore recommend to separate the power source for the motors and the sensors. A video of screen recording is available at the remotely-operated laboratory homepage [25]. The initial and target locations from one run are shown in figure 29.

Role of the crossover operator and selection methods

We were curious about the contribution of the crossover operator to the evolutionary progress. We repeated the "switch3" experiment with the probability of crossover equal to zero, thus relying only on the structural mutation operators. Figure 30 plots the average of the best fitness for both types of runs, with and without the use of crossover. The performance is approximately the same when plotted against the generation number, however, the runs with crossover used extra evaluations due to the use of brooding crossover ($num\_evaluations$ =

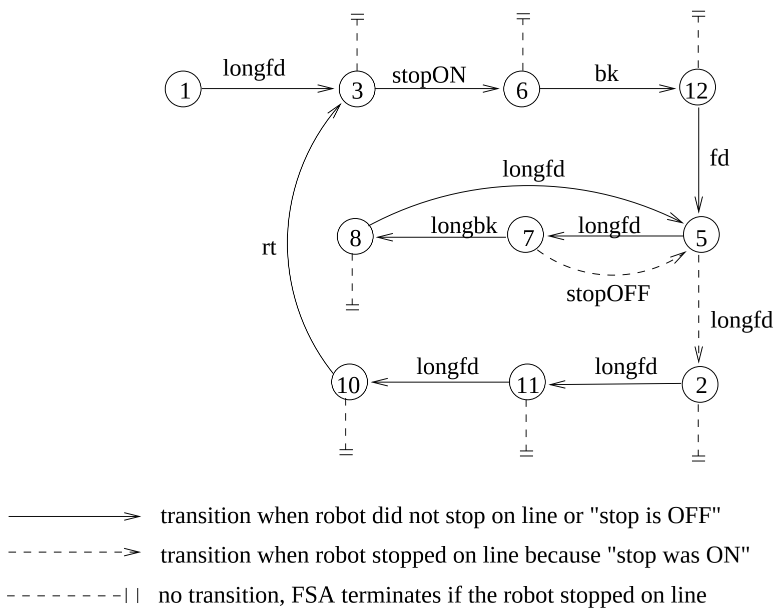

1
longfd
3
stopON
6
bk
12
fd
longfd
8
longbk
7
longfd
5
rt
stopOFF
longfd
10
longfd
11
longfd
2
transition when robot did not stop on line or "stop is OFF"
transition when robot stopped on line because "stop was ON"
no transition, FSA terminates if the robot stopped on line

Figure 26: A FSA that evolved in the "dock" task. The robot first moves forward in the loop of states 5–7–8 until it arrives to line, then it moves forward three more times (states 2–11–10), turns right, and proceeds again until it stops at line (the left line of the target square). Next, it moves into the square (states 2–11–10 again), turns right facing now down, and finally the FSA terminates when the robot attempts to move back in the state 6 when it encounters a line (top line of the target square as the robot is facing down).

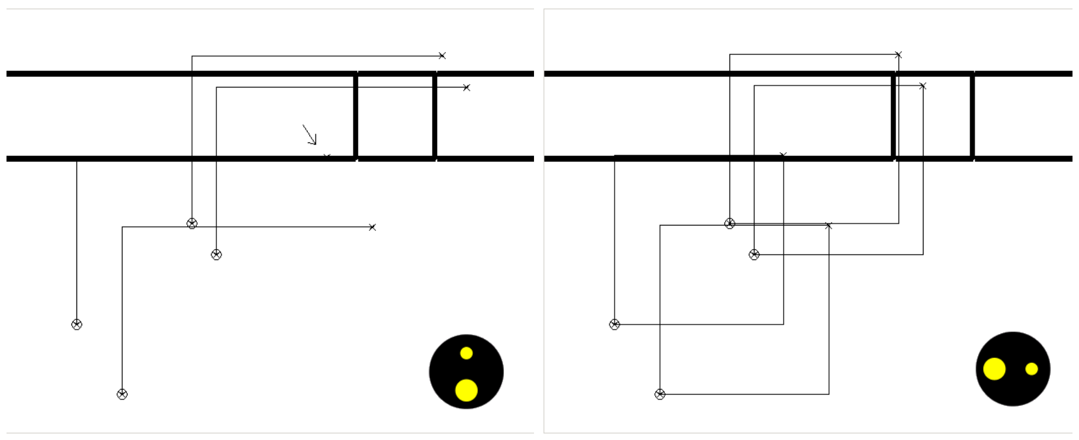


Figure 27: Trajectories for the evolved individuals in *typical* runs: GP-tree representation on the left-hand side, FSA representation on the right-hand side. Starting locations are marked by a small cross in a circle, and final positions are shown by a small cross. Trajectories for 4 starting locations are shown. The GP-tree individual simply moves forward and right the same distance regardless its starting location. The FSA individual loops several times in a square and terminates in the corner closest to the target, thus exploiting the feature of terminating the individual after certain number (80) of steps – i.e. the individual correctly times its loop that consists of several forward and backward movements (only the resulting trajectory can be seen in the figure, not the individual movements). The robot on the bottom-right is shown for illustration: it is using a downwards-oriented light sensor that detects black line. The sensors are placed both at the very front and the very back of the robot.

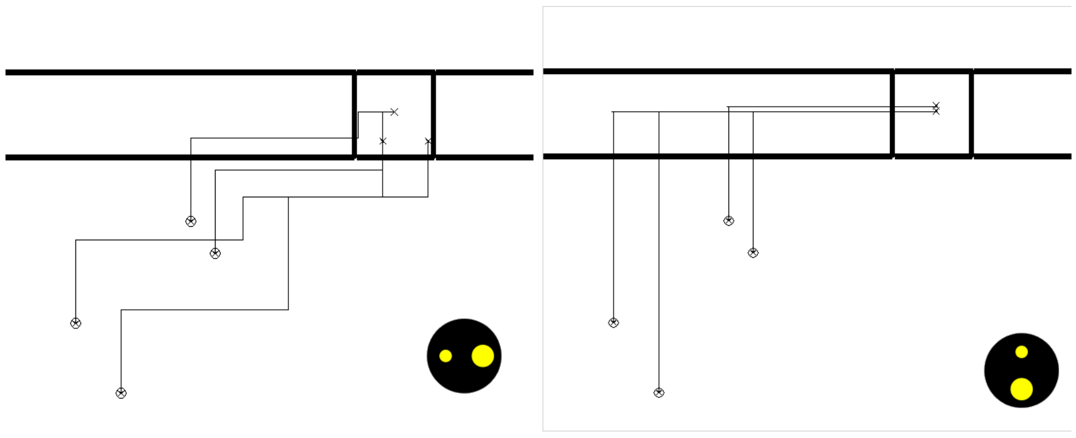


Figure 28: Trajectories for the evolved individuals in *selected* runs for both representations. The 4 starting locations are marked by a small cross in a circle and the final positions are marked by a small cross. Individuals in both representations utilize the light sensor, however the FSA solution is more clean – moving straight in the middle between the two horizontal lines, turning right and finding the target square. The GP-tree representation is approaching the target in a stair-like movement and faces difficulties to align with the target location correctly when the sensoric experiences in the final segments of the trajectories vary.

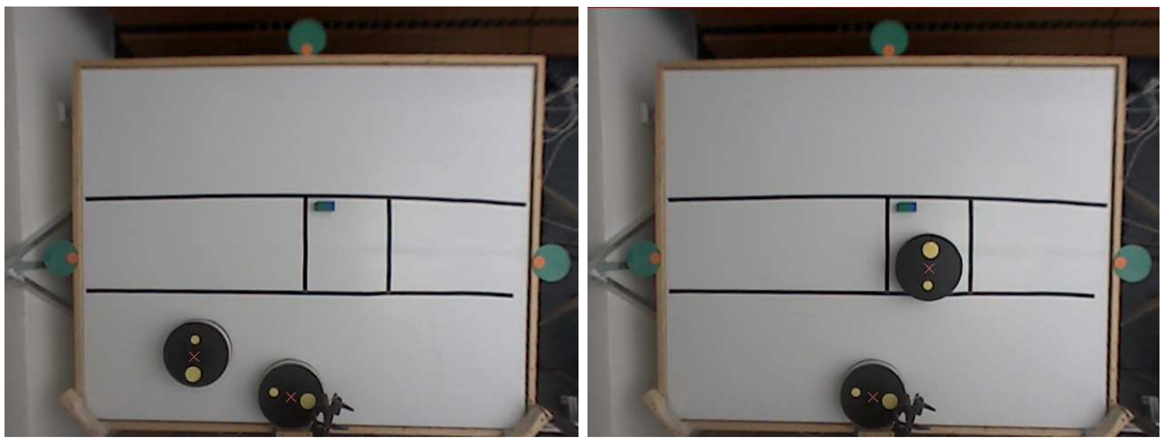


Figure 29: Initial (left) and target (right) situation for an experiment performed with real robot using the best evolved FSA shown in table 7.

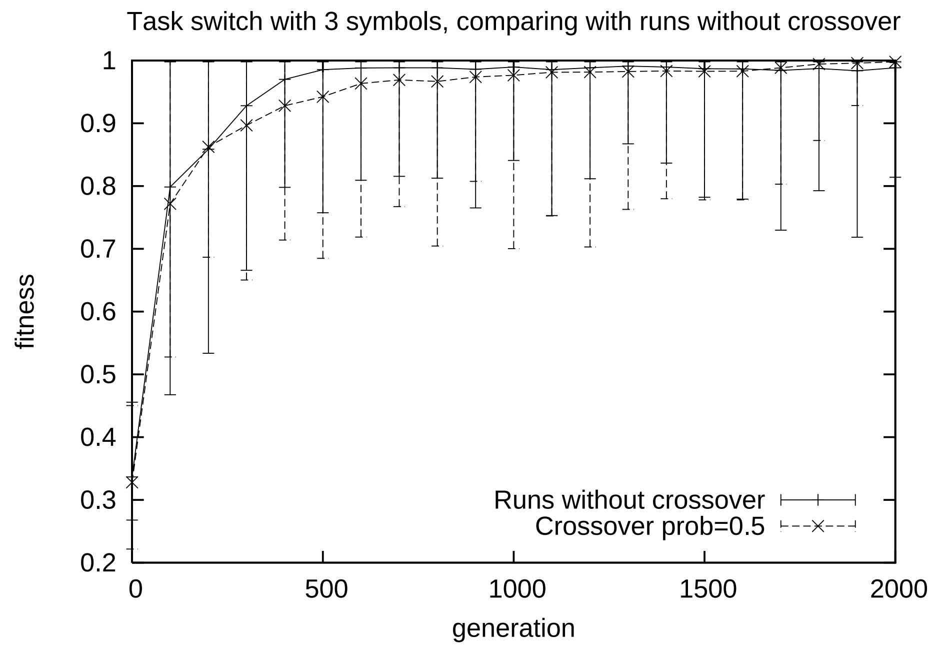


Figure 30: Role of the crossover operator for the FSA representation, experiment "switch" with 3 symbols. Average from 20 (no crossover) and 34 (switch3) runs.

$population\_size \cdot pcross \cdot 2 \cdot crossover\_brooding \cdot num\_starts \cdot cross\_brood\_num\_starts\_q)$. This suggests that the mutation operators are sufficient for evolutionary progress, and/or that too few reusable and easy-to-combine modules emerged throughout the evolution. This, however, could be the case in other tasks or in general, and therefore we have used the crossover operator in our experiments, believing that a richer set of operators should lead to higher potential of the algorithm even at the cost of slower convergence rate.

In the following experiment, we compared the performance of two different evolutionary selection methods: tournament selection and fitness-proportionate selection. Figures 31 and 32 show the performance on two different tasks and two different representations. In all comparisons we performed, the tournament selection converges faster and leads to either best or better final evolved solution.

We have also performed several experimental runs with the HMM representation on the "abcd^n" experiment, however we either did not find a correct set of parameters, or the representation was not capable of better performance than the FSA representation. This remains for the further study.

Transferring the experiments to real robot

The main purpose of our research is to study methods for creating programs for real robots. We have therefore evaluated the evolved individuals on real robots with the same morphology and sensoric equipment. Obviously, the nature of experiments is simple in this case, however, it is still an important start and test of this research and educational platform.

6 Conclusions and Future Work

This work touches several important issues of artificial evolution with direct program representations, in particular GP-trees and FSA. While the GP-tree programs tend to have a relatively

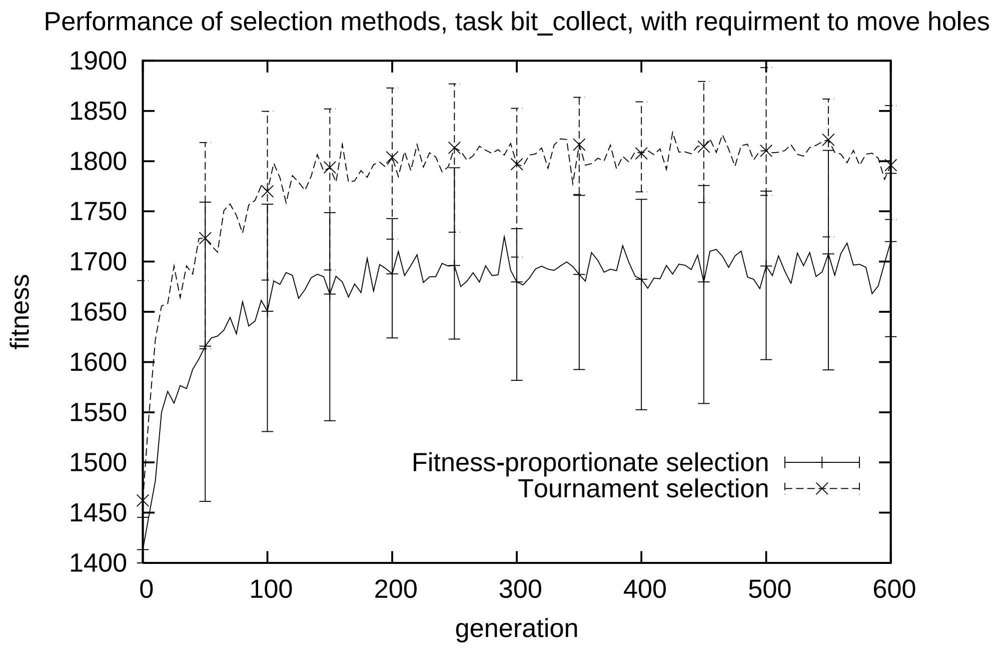


Figure 31: Comparison of two selection methods, difficult version of the experiment "bit_collect" and FSA representation. Average from 13 runs (both). The tournament selection used tournament size of 4 individuals, and probability of choosing the best individual 0.8, the individuals were not removed from the population after being selected. Errorbars show the range.

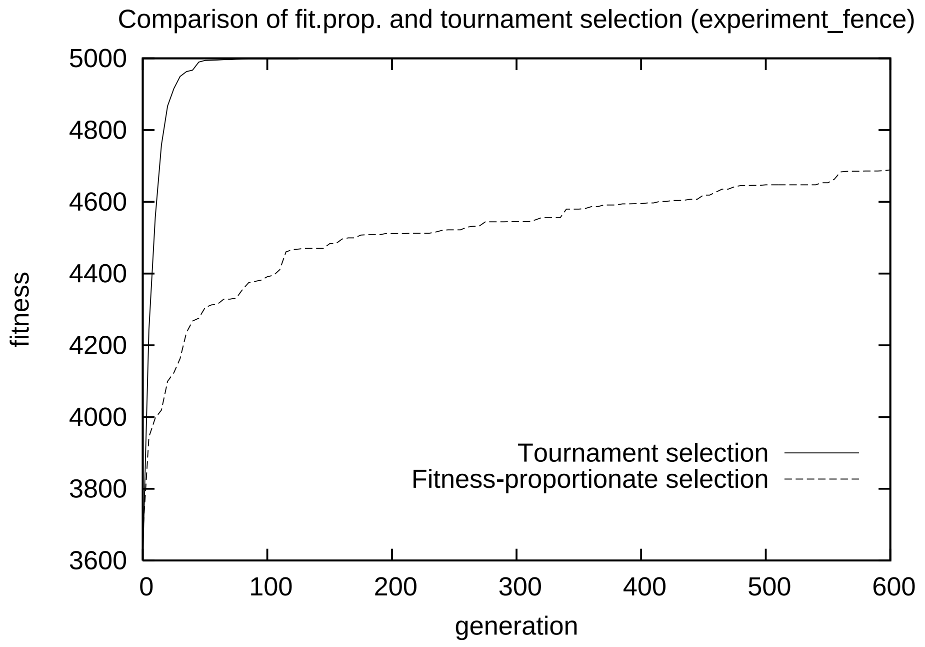


Figure 32: Comparison of two selection methods on experiment "find_target" with environment `experiment_fence` and the GP-tree representation. Average from 14 (fit-prop) and 17 (tournament) runs. Same parameters as in the comparison in experiment "bit_collect".

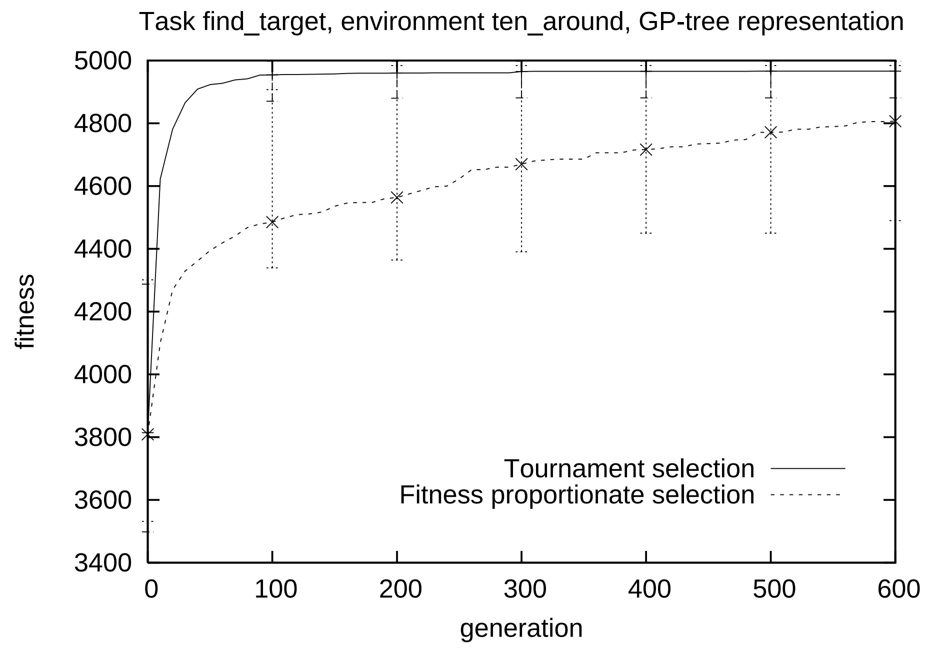


Figure 33: Comparison of two selection methods on experiment "find_target" with environment `ten_around` and the GP-tree representation. Average from 20 (fit-prop) and 12 (tournament) runs. Same parameters as in the comparison in experiment "bit_collect".

linear path of execution, the FSA are powerful in representing repeated and possibly irregular patterns and behaviors that react to percepts and possibly launch different mode (or state) of operation. The internal topology of the representations corresponds 1) to the topology of interactions of the evolved solution with its environment – as demonstrated on the "switch" task and 2) to the topology of the space searched by the evolutionary algorithm – as demonstrated by the "find_target" task. On several sample tasks, we study the performance of both representations, and confirm that both representations can outperform one another, depending on the structure of the interactions the program is to perform in the particular task. Further comparisons may involve automatically defined functions of Koza [19], and hybrid approaches that combine both FSA and GP, for instance [5].

Another important difference of the GP-tree and FSA representations is that the FSA individuals require some condition to be satisfied between performing any two actions. This is not the case in the GP-tree representation, especially when a `seq` non-terminal can be used. FSA representation can be compensated by introducing tautology transition conditions, however, it still remains more sensitive to sensoric inputs (values of the registers).

Some tasks prove to be too difficult for an evolutionary algorithm, and further guidance might improve the chances for discovering a correct solution. Incremental evolution is a possible method for such guidance. In this work, we show that using incremental evolution requires careful preparation and understanding of the evolutionary process. Our experimental runs confirm that making the evolutionary algorithm incremental can both help and hinder the success rate of the evolutionary algorithm. Usually, incremental evolution introduces an extra bias, requiring the evolution to pass through stages that could possibly be avoided in a single run. This *incremental bias* must be compensated by larger benefits resulting from evolving incrementally, in favor of faster and easier progress of the evolution, otherwise the incremental method performs worse.

The future work could focus on further elaboration on the abilities of FSA-based representations, and task-characterization guidelines that may suggest suitable genotype representation.

This would be especially interesting in the context of incremental evolution. In the related work section, we mention several possible modifications to the set of evolutionary operators and setup of the FSA evolution, comparing those to our implementation would also be valuable. Deeper investigations into less deterministic representations, such as probabilistic state automatons remain for future work.

7 Acknowledgments

The simulated experiments have been performed on the departmental cluster of student computers with kind support of the technical group at the Department. I am thankful to a valuable support from my advisor professor Keith Downing. Practical experiments were performed using the remotely-operated robotics laboratory of Robotika.SK with kind support of Ing. Richard Balogh.

References

[1] Ricardo Nastas Acras and Silvia Regina Vergilio. Splinter: A generic framework for evolving modular finite state machines. In *SBIA 2004*, pages 356–365. Springer-Verlag, 2004.

[2] Peter J. Angeline and Jordan Pollack. Evolutionary module acquisition. In *Proceedings of The Second Annual Conference on Evolutionary Programming*, 1993.

[3] Daniel Ashlock, Andrew Wittrock, and Tsui-Jung Wen. Training finite state machines to improve pcr primer design. In *Proceedings of the Congress on Evolutionary Computation CEC '02*, pages 13–18, 2002.

[4] Daniel A. Ashlock, Scott J. Emrich, Kenneth M. Bryden, Steve M. Corns, Tsui-Jung Wen, and Patrick S. Schnable. A comparison of evolved finite state classifiers and interpolated markov models for improving pcr primer design. In *Proceedings of the 2004 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, CIBCB '04*, pages 190–197, 2004.

[5] Karl A Benson. Evolving automatic target detection algorithms that logically combine decision spaces. In *Proceedings of the 11th British Machine Vision Conference*, pages 685–694, Bristol, UK, 2000.

[6] Kumar Chellapilla and David Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC'99*, volume 2, 1999.

[7] Charlie H. Clelland and Douglas A. Newlands. Pfsa modelling of behavioural sequences by evolutionary programming. In *Complex '94 - Second Australian Conference on Complex Systems*, pages 165–72. IOS Press, 1994.

[8] David B. Fogel. Evolving behaviors in the iterated prisoners dilemma. *Evolutionary Computation*, 1(1):77–97, 1993.

[9] Lawrence J. Fogel. *On the Organization of Intellect.* PhD thesis, UCLA, 1964.

[10] Lawrence J. Fogel, Peter J. Angeline, and David B. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In *Proceedings of the 4th Annual Conference on Evolutionary Programming.* MIT Press, 1995.

[11] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley.

[12] L.J. Fogel. Autonomous automata. *Industrial Research*, 4(2):14–19, 1962.

[13] Clemens Frey and Gnter Leugering. Evolving strategies for global optimization - a finite state machine approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 27–33. Morgan Kaufmann, 2001.

[14] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Adison-Wesley, Reading, Mass., 1979.

[15] Jason W. Horihan and Yung-Hsiang Lu. Improving fsm evolution with progressive fitness functions. In *GLSVLSI04*, 2004.

[16] Michael S. Hsiao. *Sequential circuit test generation using genetic techniques.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[17] David Jefferson, Robert J. Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles E. Taylor, and Alan Wang. Evolution as a theme in artificial life: The genesys/tracker system. *SFI Studies in the Sciences of Complexity*, 10:549–578.

[18] Ivan Kalas and Andrea Hrusecka. *The Great Big Imagine Logo Project book.* Logotron, 2004.

[19] John R. Koza. *Genetic Programming II.* MIT Press, 1994.

[20] Simon M. Lucas. Evolving finite state transducers: Some initial explorations. In *European Conference on Genetic Programming, EuroGP'2003*, pages 130–141, 2003.

[21] Pavel Petrovic. Evolving automatons for distributed behavior arbitration. Technical Report IDI 05/05, Norwegian University of Science and Technology, 2005.

[22] Pavel Petrovic, Andrej Lucny, Richard Balogh, and Dusan Durina. Remotely-accessible robotics laboratory. In *Robtep'2006*, 2006.

[23] William M. Spears and Diana F. Gordon. Evolving finite-state machine strategies for protecting resources. In *Proceedings of the 12th International Symposium on Foundations of Intelligent Systems*, pages 166–175. Springer-Verlag, 2000.

[24] Evolve with imagine cvs.
`http://webcvs.robotika.sk/cgi-bin/cvsweb/robotika/src/imagine/gp/.`

[25] `Remotely-operated robotics laboratory project page.`
`http://www.robotika.sk/projects/virtuallab/.`

8 Appendix A – List of EI parameters

Universal parameters:

representation:	genotype representation, one of tree, fsa, hmm
terminals:	list of terminals
terminals_p:	list of terminals weights (proportional to their probability)
terminals_args:	definition of terminals arguments (list)
conditions:	list of predicates
num_registers:	number of registers
max_constant:	global constant range (1-max_constant)
max_eval_steps:	maximum number of execution steps

Evolutionary parameters:

num_elitism:	number of directly copied best individuals
elite_allow_dups:	if set to false, the directly copied individuals will be chosen to be different
num_generations:	number of generations
pcross:	probability of crossover
crossover_brooding:	number of crossover broods
cross_brood_num_starts_q:	portion of the sample from the test cases that is used to evaluate broods
pbrooding_crossover:	probability of using the brooding crossover
strict_brooding:	whether the outcome of brooding should provide different fitness than both parent genotypes
pmutation:	probability of mutation
population_size:	number of individuals in the population
selection:	either `tournament_selection` or `fitness_proportionate_selection`
remove_after_select:	if set to true, the individuals will be removed from old population after they are selected
normalize_fitness:	whether to scale the fitness to 0–1 interval and square it in each generation
tournament_size:	size of the tournament if `tournament_selection` is used
tournament_selection_p:	probability of taking the winning individual in the `tournament_selection`
num_starts:	number of testing samples used by the objective function
fitness_size_q:	penalty for the size of the genotype (to encourage shorter genotypes)
evalsteps_q:	penalty for the number of execution steps (to encourage faster-running programs)
log_file_name:	string used for naming log file (together with time stamp)

GP-tree representation:

nonterminals:	list of non-terminals
nonterminals_p:	list of non-terminals weights (proportional to their probability)
nonterminals_args:	definition of non-terminals arguments (list)
max_genotype_depth:	maximum depth of GP-tree
pcross_combine:	probability of combining crossover
phomologic_crossover:	probability of homologic crossover

FSA/HMM representation:

transition_condition:	definition of transition arguments
pshuffle:	probability of shuffle change-mutation (changes order of transitions in a state)
min_fsa_states:	minimum number of states
max_fsa_states:	maximum number of states
min_fsa_trans:	minimum number of transitions within one state
max_fsa_trans:	maximum number of transitions within one state

HMM representation:

palterprob:	probability of changing the weight of transition in change-mutation

All with tape machine:

infinite_tape:	true or false
num_ones_min:	minimum length of the input word
num_ones_max:	maximum length of the input word

Experiment switch:

switch_num_symbols:	whether we use symbols A,B,C, or A,B,C,D (3 or 4)
increment3to4:	when set to true, the experiment will continue with 4 symbols after reaching full performance for 3 symbols
max_switch_sequence_len:	maximum number of consecutive zeros between other symbols on the input tape

Experiment find_target:

fitness_hits_q	penalty for hitting an obstacle
find_target_fitness_type	how to compute fitness (1 – fraction, 2 – subtract from high value)
target	the target location to be reached
turning_step	number of degrees to turn left or right at once
moving_step	number of steps to move on short move commands
long_moving_step	number of steps to move on long move commands
num_obstacles	number of obstacles
obstacles	list of obstacles
starts	list of starting locations
target_shape	shape of the target for visualization only
turtle_shape	shape of the robot for visualization only

Experiment dock:

turning_step	number of degrees to turn left or right at once
moving_step	number of steps to move on short move commands
long_moving_step	number of steps to move on long move commands
lab_images	background images for all starting locations
targets	target locations for all starting locations
starts	list of starting locations
dock_ip	IP address of the simulator
dock_port	network port of the simulator

Experiment bit_collect:

bit_collect_fill_only	version of task (true for simple version)
prob_one	probability of symbol 1 in the input word
max_zeros	maximum number of symbols 0 in the input word
holes_q	fitness penalty for remaining symbols 0
ones_q	fitness penalty for extra or missing symbols 1