BlindBuilder : a new encoding to evolve Lego-like structures

Alexandre Devert, Nicolas Bredeche, Marc Schoenauer

Blindbuilder : A new encoding to evolve Lego-like structures

Alexandre Devert, Nicolas Bredeche, and Marc Schoenauer

TAO team - INRIA Futurs - LRI, Bat 490 - Université Paris-Sud - France

Abstract. This paper introduces a new representation for assemblies of small Lego®-like elements: structures are indirectly encoded as construction plans. This representation shows some interesting properties such as hierarchy, modularity and easy constructibility checking by definition. Together with this representation, efficient GP operators are introduced that allow efficient and fast evolution, as witnessed by the results on two construction problems that demonstrate that the proposed approach is able to achieve both compactness and reusability of evolved components.

1 Introduction

In recent years, there have been many important achievements in the domain of Evolutionary Design using Evolutionary Computation in general [2], and Genetic Programming in particular [9]. These works range from evolving robots [10] to the design of satellite antennas [11]. Among these works, there is a strong interest for evolving constructions or robots using simple elements such as Lego bricks [13, 8].

The work described in this paper aims at evolving complete structures from small atomic elements (such as Lego® bricks or Kapla® elements) in order to obtain walls, bridges and so on. Many representations have been proposed for such constructions (see section 3), but many of them easily lead to either non-physical structures (overlapping elements), or structures that are impossible to actually construct (even though no element overlap).

One way to overcome this difficulty is to indirectly represent a structure through a construction plan. Indeed, construction plans provide a rather expressive representation formalism, and Evolutionary Computation provides an efficient way to evolve a plan (a genotype) such that the structure resulting from the application of this plan (a phenotype) is optimal for given objectives. One of the critical issues is then to provide evolution with efficient variation operators (crossover, and mutation) that explore some relevant part of the search space.

This paper proposes *BlindBuilder*, a representation for indirect encoding of structures that uses a direct representation for construction plans, described as Directed Acyclic Graphs (DAG). The variation operators borrow from the Embryogenic approaches of GP [6]. The paper is organized this following way:

Section 2 describes the framework of this work. Section 3 briefly reviews some important contributions in the field of Evolutionary Design of assemblies

of small elements, in both structural design (walls, bridges, tables and chairs, ...) and robotics (evolving both the morphology and the controller of robots). Sections 4 and 5 present the contributions of this work regarding *BlindBuilder*, respectively describing the language used to represent plans as DAGs, and the associated variation operators that have been defined to bias the exploration of such a search space. Section 6 shows the results on several classical problems of structural design. As usual, the final section discusses the results and sketches some directions for future work.

2 Problem setting

The framework of this paper is the automatic building of constructions made of small elements such as (but not limited to) Lego-like bricks. Such set of elements gives to the user a huge expressivity (endless possible constructions) with very few biases (such elements are not targeted toward building any specific constructions). The basic objective of this work is to provide an efficient encoding language as well as the corresponding relevant variation operators to evolve constructions that are optimal with respect to given objective functions (e.g. filling space, building high-and-wide bridges, ...). An other longer-term goal is to evolve element-based morphologies for mobile robots.

In the context of Evolutionary Design, the following three issues must be addressed: (1) **Representation**: what is the search space to explore? should direct or indirect encoding be used? Can a given coding achieve generality, modularity, robustness ; (2) **Variation operators**: How to design relevant crossover and mutation operators to enable efficient evolution? (3) **Evaluation and Simulation**: how to evaluate structures regarding some given objective function(s)? Should the resulting structures be built and tested in the real world, though this usually is far too time consuming? And if going for a simulated evaluation, how to tackle the trade-off between precise but costly physical simulations and faster but inaccurate heuristic computations?

The next section will survey how these issues have been addressed in the literature for similar Evolutionary Design problems, focusing on the representations used to encode the structures.

3 Related Work

Representations for structures made of small elements can broadly be broken in two categories: *direct encoding* representations encode the position of elements in the environment; *indirect encoding* representations rely on a language that specifies *how to assemble the elements*.

Indirect encoding is largely favored in the literature, be it in the field of robotics or Structural Design, because it provides an easy and efficient way to bias evolution towards relevant structures. A remarkable exception is that of the GOLEM project [10] where real world implementation of evolved robots is achieved through a direct encoding that specifies anchor points that are linked

by rigid sticks. But most other works rely on some indirect encoding: Karl Sims [15] evolves the building process of simulated robots through graph-based flow machines; more recently, the TinkerBots project [8] relies on L-systems to indirectly encode the construction process to build virtual and real world creatures. In both cases, the use of grammar- or L-system-based encoding makes it possible to obtain highly modular representations. In the field of Structural Design, [4] describe the evolution of a construction process that successfully builds 2D cantilever bridges, and [13] introduces a DAG-based representation to represent *construction plans* that are used to build small constructions such as pillars, walls and staircases.

Some previous works [1, 7] have shown that such indirect encoding representations are indeed much more efficient than direct encoding representations. The efficiency of an indirect encoding seems to have two main causes : *compactness* and *bias*. Indirect encoding is more expressive than direct encoding thanks to the possibility of reusing portions of the code; thus, appropriate factorisation in the representation may occur, that makes it possible to have more expressive code with shorter length, and, as a direct consequence, to speed up evolution. Indirect encoding also makes it possible to potentially represent only part of all possible structures, i.e. only a specific class of physical structures can be expressed; with the appropriate choice of implementation this enables the introduction of relevant domain knowledge.

Thus, several important properties should be considered[1] : *modularity* (the ability to reuse a part of the construction plan. Modularity may or may not be recursive) *hierarchy* (the ability to consider as one single element what has already been built as opposed to having to target specific sub-elements for any new operations) *generality* (the property according to which the representation can be easily extended to accept new kind of elements), and *3D representation* (some representations only consider 2-D structures, or don't scale-up well to 3-D structures).

The main drawbacks of indirect encoding is that they usually achieve some trade-off between language expressivity and constructibility. As a result, there is a clear separation between works that rely on direct encoding approach and that are actually implemented in the real world and works that exploit the power of the indirect encoding approach but are usually limited to simulation. A noteworthy exception is [8], where an indirect encoding approach is used to build real-world robots; but the approach is limited to a rather small number of elements.

Yet, it is possible to avoid, or at least limit, the problem of non constructibility by relying on an indirect encoding approach that works in the space of *construction plans* as proposed in [13]. A *construction plan* is evaluated to build a physical structure through a sequence of construction operations. In [13], construction plans are represented as Directed Acyclic Graphs where nodes are physical Lego elements and arcs are connection operators. With such a rep-

[1] Note that in [7], some of these terms are used in the context of programs rather than graphs, with different meaning.

resentation, it is possible to iteratively check at each construction step if the physical structure is buildable rather than evaluating the whole structure only at the end of the construction procedure.

4 BlindBuilder : a new indirect encoding language

This section introduces *BlindBuilder*, an indirect encoding language for the description of construction plans. Basically, a *BlindBuilder* individual is a Directed Acyclic Graph (DAG) where nodes can be either atomic elements (e.g. Lego-elements, Kapla-Elements, Joints, Sticks, Tubes) offering *connectors* to other elements, or construction operators of a given arity (e.g. SNAP, CONNECTWITHHINGEJOINT, CONNECTWITHBALLJOINT) parameterized by the connections they achieve between their arguments. More precisely :

- **Atomic element** are terminals of the DAG (i.e. they don't have any argument since their arity is zero). However, they are not considered as *physical* elements but rather as element templates that may be instantiated when needed. Each element template is defined with a given geometry and a set of connectors. Examples of atomic elements are Lego-elements, Kapla-elements (that have 0 connectors), tubes, wheels, artificial muscles, servomotors.
- **Construction operators** are functional nodes with a fixed number (*arity*) of arguments, i.e. targeted sub-nodes in the DAG, that specify what the defined function should be applied on (either other construction nodes or atomic elements). Moreover, each construction operator has internal parameter that specify how to connect its arguments together and that are subject to evolution. An example of a simple operator used in the following is the SNAP operator, that takes as arguments two elements to connect (e.g. *elements 1 and 2*) as well as parameters that define the anchor points and orientation. SNAP is formally written as : SNAP *[element1 target connector, element1 orientation connector, element2 target connector, element2 orientation connector] (element1 , element2)* . The *connector* arguments are used to pick up one actual connector from each argument-element (modulo the number of connectors of the actual argument), and the orientation of the connection is determined according to the *orientation* parameters (the number of parameters is thus independent of the size of bricks).

A well-formed *BlindBuilder* individual is hence a DAG such that the atomic elements are terminal nodes while the construction operators have as many sub-nodes as their arity. Moreover, there is a unique special node called the *top-level operator* (i.e. the entry point), so as to generate a single construction. The program run when using such a DAG to build a structure starts from the top-level operator and iteratively builds the structure by evaluating every operators until all terminal elements have been reached.

Figure 1 gives a very simple example of a construction plan with such properties together with the resulting structure. As a matter of fact, this example also illustrates some useful properties of this representation: **hierarchy**, when the snap operator at the top (right) reuses the results of its subgraph (at his left); and **modularity**, as four physical elements are built from the same element template. Moreover, as already mentioned, the ability to work in the construction space makes it possible to check for constructibility at each steps of the DAG evaluation, thus reducing the chances to obtain a non-constructible structure.

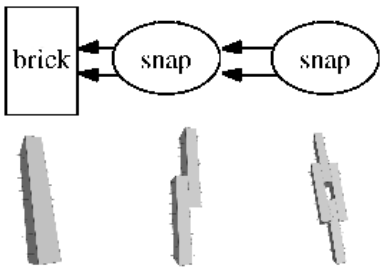

Fig. 1. A simple construction plan example and the resulting structure. The parameters of the *S*nap operators are not shown.

BlindBuilder is somewhat related to the graph-based approach described in [13]. Both languages are represented by a DAG and rely on similar construction operators (e.g. the *snap* operator). However, *BlindBuilder* considers both elements **and** construction operators as possible nodes of the graph. Moreover, element nodes are considered as templates and instantiated into physical elements, which makes it possible to endow hierarchy as well as modularity [2].

To summarize, *BlindBuilder* implements a language that is hierarchical, modular, and general while focusing on buildable plans in 3 dimensions, thus departing from previous work in the literature. Moreover, no a priori assumption is given for the definition of operators. It is hence possible to define a wide range of operators, as will be described in next section. As a consequence, it should be highlighted that all works described in section 3 can easily be expressed within *BlindBuilder* framework – from Karl Sims' creatures to Pollack's GOLEM robots and Lego-like constructions – as soon as the appropriate elements are properly designed.

5 Variation operators

In order to evolve *BlindBuilder* individuals, it is necessary to design variation operators. Some examples of GP-based evolution of graphs exist in the literature [16]. However, the DAGs resulting from the variation operators for *BlindBuilder* must comply with the definition of a well-formed *BlindBuilder* individuals, as stated in the previous section.

Classical operators in graph-based GP such as *crossover* (creating a construction plan from two existing plans) and *mutation* (altering a construction plan) may be used to evolve a *BlindBuilder* DAG. However, two main problems arise.

[2] While recursivity is possible, it is not yet implemented in the present work.

First, performing even a syntactically correct crossover upon two DAGs may result in very different structures in the end because of the very structure of a DAG (i.e. semantic) is ignored in the blind process of standard crossover. Because the topology of *BlindBuilder* DAGs is of utter importance, no useful crossover operator could be designed, and the evolutionary process described hereafter only relies on mutation. Second, simple random mutation operator (i.e. replacing a (group of) node(s) by randomly generated nodes) is confronted to the difficulty of matching arities between deleted and inserted (group of) node(s).

However, though the proposed approach is definitely not an embryogenic approach, the operators used as nodes in the seminal work in embryogeny-inspired GP [6] were used as inspiration for the present work and led to introducing the following five mutation operators:

1. *GrowForward* (fig. 2-b): a new non-terminal node B is added downstream from the target non-terminal node A. All arcs outgoing from A are connected with B. B is randomly chosen among all operators with the same arity than A, and its parameters are uniformly initialized;
2. *GrowBackward* : a new non-terminal node B is added upstream from the target non-terminal node A. The ingoing arcs to A become ingoing arcs to B. B is randomly chosen among all non-terminal node and its parameters are uniformly initialized;
3. *Split* : target node A is split into B and C, two nodes at the same level. Ingoing arcs to A are randomly assigned to B or C, while outgoing arcs are duplicated. These new nodes are randomly chosen among the set of arity-compatible nodes. This operator cannot be applied to the upper level node;
4. *Permute* : outgoing arcs of the target non-terminal node are randomly permuted *and* parameters are uniformly reset;
5. *Replace* : the target node is replaced by a randomly chosen node with the same arity *and* its parameters are uniformly reset.

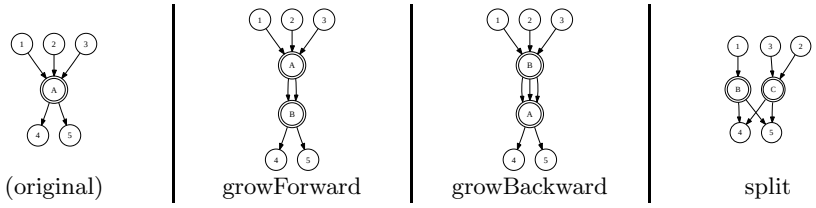

Fig. 2. Effects of *growForward, growBackward* and *split* operators

The only restriction in those operators is that the *s*plit operator cannot be applied to the top level node, to avoid conflicting entry points for the evaluation process. This feature ensures that all plans can be generated (in theory) from DAG made of a single terminal: this is what will be used in the initial population.

6 Experimental Results

This section presents the experimental setup (software and evolution parameters) along with two experiments. Each experiment relies on the use of one specific species of construction elements: Lego-like and Kapla-like[3]. Due to the type of elements involved, *BlindBuilder* construction operator list is limited to the SNAP operator described in section 4. For Lego-like elements, the SNAP operator results in establishing a real physical connection while for Kapla-like elements, it is only used to position the various elements (i.e. the resulting construction may be destroyed because of gravity).

6.1 Experimental setup

As said above, all individuals in the initial population are single-node DAGs, for which the unique terminal is uniformly chosen among the set of element templates.

The selection is a tournament selection (typically of size 7) based on a hierarchical multi-criterion comparison operator that incorporates both the target objective(s) and some parsimony pressure in a lexicographic way similar to that proposed by [12]. Note that this is **not** a Pareto-based optimization (e.g., two individuals are always comparable).

- Define the relative distance between two values a and b as $\frac{|a-b|}{max(a,b)}$;
- Order the list of objectives from most to less important;
- Two individuals are said to be equivalent for a given objective if the relative distance between their values for this objective is less than a given threshold (typically 0.1);
- The comparison of two individuals is then lexicographic, i.e. individual x is better than individual y if, for some objective rank i, x and y are equivalent for objectives $1, \ldots, i-1$, they are not equivalent for objective i, and the value of x for objective i is larger than that of y.

In the following experiments, tournament size is set to 7 and population size to 1000. The threshold for the comparison of objective values is set to 0.1. All experiments were run 13 to 20 times. Each experiment took about 16 hours on a PC with Intel Pentium 4 running at 3.6 GHz under Linux.

A few preliminary experiments (not shown here) showed that a Pareto approach (relying on NSGA-2 algorithm [3]) was slower the hierarchical approach described above. Moreover, a standard generational GA evolution (i.e. 1000 offspring are generated at each generation and replace all parents) using tournament selection was observed to be more efficient than both $(\mu, lambda)$-ES and $(\mu + lambda)$-ES, with $\mu = 15$ or $\mu = 30$ and $\lambda = 7\mu$, the latter giving better result than the former. Finally, a maximal size of 50 for a construction plan was set to avoid uncontrolled code growth – but the limit was hardly ever reached.

[3] http://www.kapla.com

The preliminary experiments also showed the relative importance of the variation operator *replace*: Indeed, this operator is much more conservative than the others, and is mandatory to fine tune existing structures, while all other operators result in important changes in the resulting structure. As a consequence, the rate for the *replace* operator is set to 0.7 while all other operators have a rate of 0.075 in the following experiments.

The *BlindBuilder* approach was implemented within *Open-BEAGLE*, a framework for artificial evolution written in $C++$ [5]. *Newton Game Dynamics*[4], was used in order to simulate and evaluate the resulting structure in a physical environment. All the experiments are in three dimensions.

6.2 The Pillar experiment

The goal is to build the biggest possible structure using Lego-like elements (1x2, 2x2, 2x3, 2x4 and 2x6 bricks). Lego-like elements are characterized by physical connections that hold them together. The objective functions to maximise are, ordered by priority:

1. The *volume*: $V = \sum V_i$, where V_i is the volume of ith atomic element i.
2. The *compacity*: $\mathrm{C} = \frac{V}{V_{full}}$ where V_{full} is the volume of the convex hull of the whole structure.
3. The *parsimony*: $P = 50 - S$ where S is the number of nodes of the construction plan (max. 50 elements).

Figures 3 and 4 shows evolution results and an example of obtained structure when using only a 2x2 element. Results show that for this simple constrained problem, maximum compacity is achieved very quickly. Moreover, optimal individuals are found with the smallest possible construction plan. Figures 5 and 6 shows the same experiment but with all 5-elements templates possible. The bigger and thus most appropriate element (2x6) is always used, even though the optimal plan is not yet reached at the end of evolution (it may be reached if evolution is carried on further). The two examples shown on Figure 6 are very different construction plans, the latter being larger, but leading to a more compact construction. In all experiments, reusability has been heavily exploited, as can be observed in the sample plans of Figure 6.

6.3 The Bridge experiment

Kapla-like elements can be defined as Lego-like elements with no connections. Thus, Kapla construction are much more unstable. Moreover, by changing the set of possible values of the *orientation* parameter in the `Snap` construction operator, the user can decide to go from "flat" structure (allowing a single value 0) to square structures (allowing 0, 90, 180 and 270 degree orientations) to complex 3D structure (allowing any floating-point value). The goal of this experiment is to build the longest horizontal structure with as few elements on the floor using Kapla-like elements. The objective functions to maximise are:

[4] Freeware but not open-source, see `http://www.newtondynamics.com/`

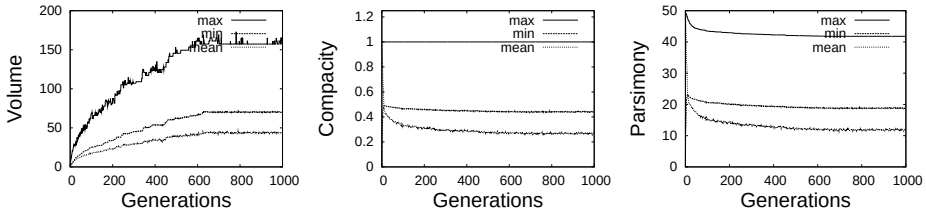

Fig. 3. Averge results for Pillar experiment using only the 2x2 Lego-element template

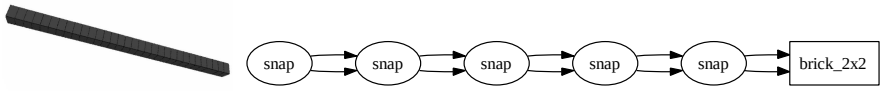

Fig. 4. Example of best solution for the Pillar experiment using only the 2x2 Lego-element template. Snap parameters not shown.

1. The *length*, the horizontal length of the structure.
2. The *grounding*, $n - f$ where n is the number of atomic elements of the construction, and f the number of atomic elements that are in direct contact with the floor.
3. The *parsimony* defined as in the Pillar experiment above.

Figure 7 show results of obtained individuals. Every runs succeeded in generating quite successful individuals, either by deeply optimizing one of the objective function or making a compromise between the three objective functions. The most striking results is that evolution has been able to build *cantilever* bridges with *arches*, for which various examples are shown in figure 8. Each example represent the best individual for a given run, as a matter of fact, there is a great variability between runs.

6.4 Discussion

The results shown here are clearly competitive to that of the literature of evolutionary design using Lego-like elements. The approach of [13] and *BlindBuilder* both use a DAG-based representation of construction plans. However, the former lacks properties of modularity due to the intrinsic nature of the graph (node are physical elements only, arcs are functions that connect these elements and the language is limited to Lego elements). As a consequence, experiments are limited to simple constructions (walls, pillars) and evolution is slower than what has been shown here - for instance, construction size can only grow by adding one element after another while a *BlindBuilder* construction can double in size thanks to the addition of a single SNAP operator at the top of a graph.

The experiments presented in [4] also demonstrated that bridges made of Lego elements can be evolved according to the cantilever principle. However, evolution took place in a two-dimensional environment (even though "flat" 3D

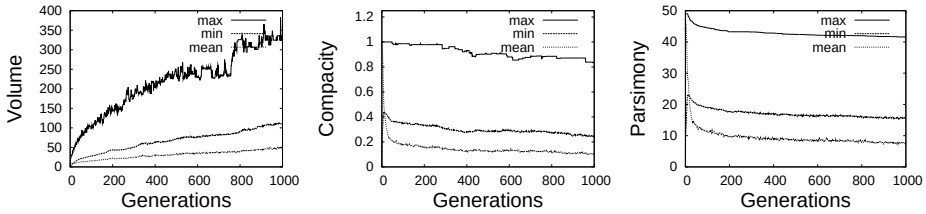

Fig. 5. Average results for Pillar experiment using five possible Lego-element templates

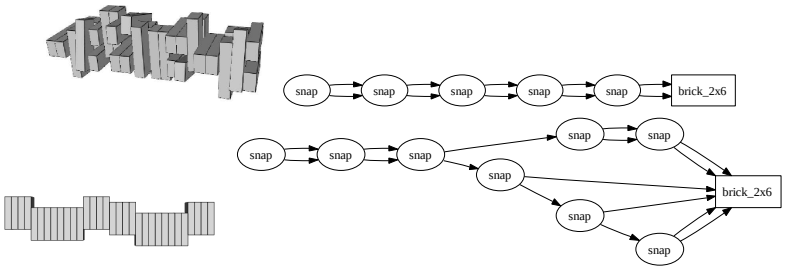

Fig. 6. Examples of best solutions found for the Pillar experiment using five possible Lego-element templates. Snap parameters not shown.

models were shown). Moreover, the language used in [4] lacks reusability. On the opposite, the *BlindBuilder* approach leads to comparable results in a true 3D environment, with a more compact representation, thanks again to modularity (here : ability to reuse arches and cantilever principle as soon as their definitions are evolved in a construction plan).

One current limitation of our work is that for every run, the evolution process failed to maintain diversity. As a results, best individuals are very different from one run to another, but very similar within one given run. A current track under investigation is that of introducing island models so as to maintain candidate solutions with similar performance but different structures within one single run.

7 Conclusion and Perspectives

This paper has introduced a new indirect encoding language for structures made of small elements called *BlindBuilder*. It is designed to represent construction plans, i.e. plans to iteratively build structures such as bridges or robots from atomic elements. *BlindBuilder* shows interesting features such as compactness and reusability thanks to hierarchy and modularity. Moreover, construction plans make it possible to check for constructibility at each time steps of the evaluation instead of having to evaluate the whole structure. To our knowledge, this language is the first to endow all these properties in a single framework.

Alongside, a set of mutation operators have been defined to perform efficient evolution. These kind of mutation operators explore the space of construction

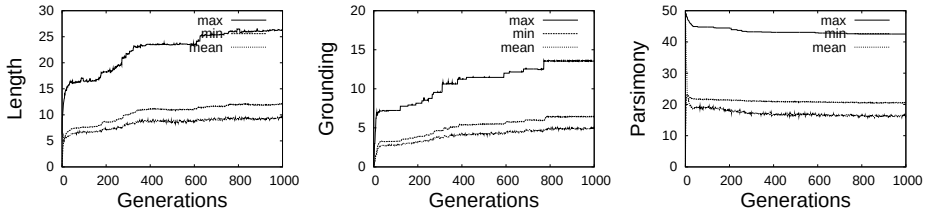

Fig. 7. Results for the Bridge experiment

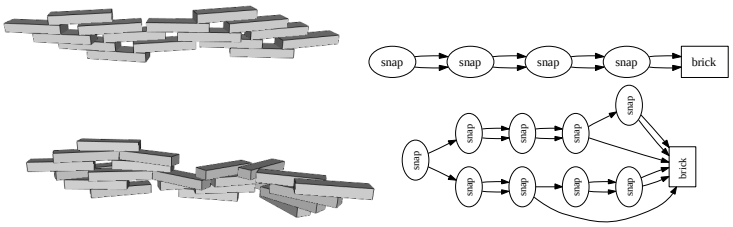

Fig. 8. Examples of best solutions. Snap parameters not shown.

plans and alter existing construction plans in such a way that resulting individuals are well-formed *BlindBuilder* graphs.

The experiments showed that *BlindBuilder* features are exploited by the evolution process and do achieve compact representation with reusable components. Interesting results were achieved when building bridges with Kapla-like elements, such as the rediscovery of arches and cantilever principle so as to minimise contact points while maximising bridge length.

Future works on *BlindBuilder* include adding recursivity, though there is no way to easily specify a terminating condition within a graph. We also intend to refine the variation operators, especially the *permute* and *replace* operators, with respect to the modification of the parameters: parameters are at the moment modified uniformly, while more real-value-oriented mutations, such as Gaussian mutation for the orientation in the case of Kapla elements, should be more appropriate and should allow both more variety in the results and better fine-tuning of the final solution. As for constructibility issues, recent works [14] have shown that a promising way is to evaluate candidates as they are built, and not just the resulting structure, which can be easily implemented using *BlindBuilder* – but this will have some computational cost ...

References

1. Peter Bentley and Sanjeev Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 35–43, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

2. Peter J. Bentley. *Evolutionary Design by Computers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
3. K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In M. Schoenauer et al., editor, *Proceedings of the* 6^{th} *Conference on Parallel Problems Solving from Nature*, pages 849–858. Springer-Verlag, LNCS 1917, 2000.
4. Pablo J. Funes and Jordan B. Pollack. Computer evolution of buildable objects for evolutionary design by computers, 1998.
5. Christian Gagné and Marc Parizeau. Open BEAGLE: A new C++ evolutionary computation framework. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, page 888, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
6. F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Suprieure de Lyon, France, 1994.
7. Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1729–1736, Washington DC, USA, 25-29 June 2005. ACM Press.
8. Gregory S. Hornby, Hod Lipson, and Jordan B. Pollack. Generative representations for the automated design of modular physical robots. *IEEE transactions on Robotics and Automation*, 19(4):709–713, August 2003.
9. John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.
10. Hod Lipson and Jordan B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.
11. Jason Lohn, Gregory Hornby, and Derek Linden. Evolutionary antenna design for a NASA spacecraft. In *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 13-15 May 2004.
12. Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In *GECCO-2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. Morgan Kaufmann, 2002.
13. Maxim Peysakhov, Vlada Galinskaya, and William C. Regli. Representation and evolution of lego-based assemblies. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, page 1089. AAAI Press / The MIT Press, 2000.
14. John Rieffel and Jordan Pollack. Automated assembly as situated development: using artificial ontogenies to evolve buildable 3-d objects. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 99–106, New York, NY, USA, 2005. ACM Press.
15. Karl Sims. Evolving 3d morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.
16. Astro Teller and Manuela Veloso. PADO: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.