Solving LEGO brick layout problem using Evolutionary Algorithms

Pavel Petrovic

Division of Intelligent Systems,
Department of Computer and Information Science,
Norwegian University of Science and Technology, 7491 Trondheim, Norway,
pavel.petrovic@idi.ntnu.no

Abstract. LEGO® presented the following problem at the SCAI'01 conference in February 2001: Given any 3D body, how can it be built from LEGO bricks? We apply methods of Evolutionary Algorithms (EA) to solve this optimization problem. For this purpose, several specific operators are defined, applied, and their use is compared. In addition, mutation operators with dynamic rate of mutation updated based on their contribution to progress of evolution are proposed. Different population organization strategies are compared. Early results indicate that EA is suitable for solving this hard optimization problem.

Keywords: LEGO brick layout, evolutionary algorithms, dynamic mutation rate, multiple-deme GA, genome representation.

1. Introduction

Evolutionary Algorithms are among standard methods for solving optimization problems. Their basic theory is well based on mathematical proofs, and their practical application showed many successful results, see for example [Dasgupta, Michalewicz, 1997]. In this work, we apply a version of the Genetic Algorithm [Holland, 1975] to a hard combinatorial optimization problem of LEGO brick layout. The problem was proposed by engineers from LEGO Company presenting a poster at Scandinavian Conference on Artificial Intelligence in Odense in February 2001. According to them, this problem has not been solved yet. It has been tackled by a group of mathematicians on a one-week workshop [Gower et.al, 1998] who devised a cost function for Simulated Annealing [Kirpatrick et.al. 1983, Laarhoven & Aarts, 1987] general search method, but no practical implementation and study has been performed. The problem has typical features of problems suitable for solving using Evolutionary Algorithms (EAs): hard combinatorial optimization problem, solutions are relatively easy to represent, the optimal (best) solution is not required, the search process should generate only a "good enough" solution. EAs are therefore a natural tool for solving this problem. Recently, [Bentley, 1996] studied a general problem of design of 3D objects using EAs, and [Funes & Pollack, 1997] designed 2D structures built from LEGO bricks with the help of physical model and parallel asynchronous GA. The problem of brick layout was adopted and discussed by our group at IDI NTNU, and we started to work towards a solution for the brick layout problem. In this article, we start with definition of the problem, summary of the work of the mathematician group of Gower et al., and description of our approach. We implemented a first experimental system that successfully generates brick layouts using a parallel genetic algorithm. In the following

sections, we describe the chosen representation and the designed and implemented genetic operators. We will present a short comparison of early results obtained by runs of the implemented package. Further sections will introduce you to the idea of adaptive reinforced mutation and show how it can speed up the search process. Finally, we will discuss an indirect representation, which promise delivery of higher-quality results, but requires implementation of specially tailored operators.

2. Problem description

In order to build complicated structures from LEGO bricks efficiently, LEGO engineers need a program that will generate brick layouts for all layers of models they want to build. The input to the system is a 3D model partitioned into multiple layers, each of which is divided into rectangular grid. Each unit square of this grid is supposed to be either covered by some brick or left empty. For the purposes of this work, we use only standard bricks shown on figure 1, and we consider only monochromatic models. For the real application purposes, we would have to consider bricks with triple height and DUPLO bricks with units of double size in addition. The extension to color can be in the first round achieved easily by splitting longer bricks at places where the color is changing into two or more shorter bricks. In the next stage, the optimization algorithm should take the color information into account already while searching for a stable solution.

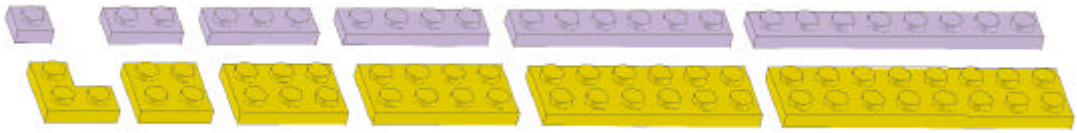

Figure 1. Standard LEGO bricks

The goal is to find such a layout, which will provide good stability – both per brick and as a whole. Ideally, one could apply the laws of statics to compute the stability of a model. However, this is computationally complex procedure and therefore the search algorithm should try to follow heuristic hints loosely corresponding to a stability of the model.

3. Previous Work

Gower et al. identified the following heuristics for stable solutions:

1. A high percentage of each brick's area should be covered, above and below, by other bricks. This means that a brick hanging in the air without any support is not very stable and each brick should be supported from as much of its area as possible.
2. Larger bricks should be preferred for the smaller bricks; they give better overall stability to the model.
3. Bricks placed on top of each other in consecutive layers should have alternating directionality. If for example long bricks are placed perpendicularly instead of parallel to each other, they provide a strong support for overall stability.
4. Bricks in the layers above and below should cover a high percentage of each brick's vertical boundaries (ideally, bricks above and below should be centered on these boundaries). This means that bricks in the consecutive layers should not have boundaries at the same place thus creating a possible unstable cut for the model.

5. The same holds within the same layer: neighboring bricks should be placed so that their boundaries are as far from each other as possible (i.e. ideally the brick should start in the middle of the neighboring brick).

The group recommended using a cost function of a form:

$$P = C_1P_1 + C_2P_2 + C_3P_3$$

Where C_i are weight constants and:

P_1 is perpendicularity, i.e. penalty given to bricks, which do not fit with factor 3 above

P_2 represents vertical boundaries; i.e. penalizes the placement of bricks, which does not fit with factor 4 above

P_3 is horizontal alignment; i.e. penalty to bricks placed in a way that goes against factor 5 above

Other hints coming from this work were to use a library for certain patterns, pattern filling and translating, and to use a local search or Simulated Annealing. A feasible pattern library seems to be problematic: the number of possibilities and ways how micro-patterns could be combined is too high and the size of such library would grow rapidly with the size of patterns.

4. Evolutionary approach

In order to solve a problem with EAs, uniform representation of solutions to the searched problem has to be found (that is how the solution – phenotype – is represented by genotype). Mutation and recombination operators have to be devised, objective function written, the type of an algorithm chosen, and finally the parameters have to be setup and tuned.

There are two classes of approaches to genotype representations: direct and indirect representations. In direct representations, the phenotype (the solution) is identical to the genotype and individual genes in the genotype correspond directly to features in the phenotype. In indirect representations, phenotype is constructed from genotype through a series of transformations. Indirect representations have often the disadvantage of being vulnerable to the use of standard genetic operators. Crossover and mutation operators applied on an individual can lead to an offspring with features completely unlike its parents. The search thus becomes only blind random search, and such a representation combined with the search operators (that in fact define the neighborhood of each search location and thus the whole search space ordering) might result in a very rough fitness landscape. On the other hand, indirect representations, when chosen properly, might significantly reduce and focus the search spaces, and sometimes might be necessary to avoid searching through number of unfeasible solutions.

Our approach is to converge to a smart indirect representation, but we argue that this cannot be easily achieved without studying direct representations first, at least in this case. The transition to indirect representation can then be smooth: direct representation can be augmented with indirect features and the evolution might decide how much direct or indirect the genotype features should be.

The experiments described in this paper were performed with the following direct representation:

A solution to the problem is an unordered list of triples (row column brick-type), where [row; column] are coordinates of a reference point of a brick (upper-left corner if available), and brick-type is an index of one of the bricks shown on figure 1. Brick-type contains also information about the brick orientation. Clearly, each brick layout can be represented as such a list. On the other hand, many lists don't correspond to a feasible layout. We can either ignore this problem with the hope that evolution will search its way through the mass of infeasible solutions, or define special operators. Early results with representation allowing unfeasible solutions showed that it is difficult for evolution to find one feasible solution, not even an efficient one. Therefore we introduced the requirement that all of the individuals present in the population should correspond to a feasible layout, i.e. layout where bricks don't overlap or extend beyond the given input pattern. At the moment, there are 4 different initialization operators available: `CarefulInitializer` processes the input pattern from top row to bottom and each row from left to right (or from bottom to top and from right to left, the chances of both possibilities are 50%). Each time it finds a unit that is not yet covered by some brick, it stochastically selects one of the bricks, which fit at this location, with larger bricks having higher chance to be selected (the chance is proportional to the area of the respective brick). `EdgeFirstInitializer` is inspired by the hint provided by LEGO engineers – first the edges of the model are covered and only afterwards comes the "inside" of the model. This initializer first collects those units that lie on the border of the model and fills them in a random order with random bricks (using larger bricks with higher probability) and thereafter fills the rest in a random direction. Figure 2 shows comparison of performance of these two initializers.

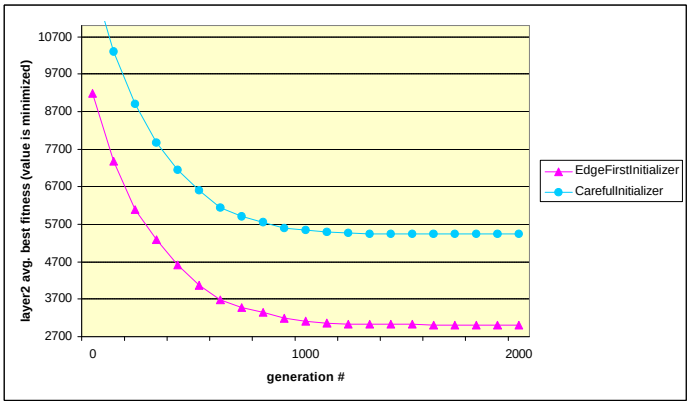

Figure 2. The fitness of the best individual, average from 3 different models, 10 runs for each. Demetic GA with 5 populations of size 500 was used in this experiment.

The initializers – `MultiCarefulInitializer` and `MultiEdgeFirstInitializer` first call the respective initializer several times (5 rounds were used in the experiments), and then select the solution that has the best fitness among them. Thanks to this initialization strategy, already the solutions in generation zero are relatively good solutions. In the experiments described below, the `MultiEdgeFirstInitializer` was used.

Clearly, standard one-point crossover would lead into overlaps. Instead, our search uses `RectCrossover` operator, which chooses random rectangular area inside of the layer grid. The offspring is composed of all bricks from one parent that lie inside of this rectangle, and all bricks from the other parent that don't conflict with already placed bricks. In this way, some of the areas of the input pattern might become uncovered. However, they are covered later in the objective function, see below.

Similarly, random mutation operator would generate overlaps. Instead, each individual, which is selected for mutation, is randomly mutated in one of the following ways:

- single brick is replaced by other random brick (larger bricks are used with higher probability); bricks that are in the way are removed
- single brick is added at random empty location, larger bricks are used with higher probability; bricks in the way are removed
- single brick is shifted by 1 unit square in 1 of the possible 4 directions (with respect to keeping the borders); all bricks that are in the way are removed
- single brick is eliminated from the layout
- single brick is extended by 1 unit in 1 of the possible 4 directions; all bricks that are in the way are removed
- all bricks that are in a random rectangle are removed and the area is filled with random bricks (larger bricks are used with higher probability)
- the whole layout is generated anew using `MultiEdgeFirstInitializer`.

Each mutation type was assigned a probability: 0.1 for new individual and brick remove, 0.15 for remove in rectangle, replace by random brick, and extend brick, and 0.2 for creating a new brick.

The objective function is inspired by the previous work mentioned above. The following function has been used in the experiments:

$$Fitness = c_{numbricks} * numbricks + c_{perpend} * perpend + c_{edge} * edge + c_{uncovered} * uncovered + c_{otherbricks} * otherbricks + c_{neighbor} * neighbor$$

where each of the following variables is scaled by its weight constant:

`numbricks` – number of bricks in the layout. This corresponds indirectly to use of large bricks, this objective is minimized;

`perpend` – total sum of ratios through all bricks describing how well (0 – 1) each of them covers the previous layer perpendicularly: for all unit squares that the brick covers, perpend looks whether the brick at the corresponding unit square in the previous layer contains perpendicularly placed brick; 2x2 brick, L brick, and 1x1 brick don't contribute to this variable, this objective is maximized;

`edge` – total sum of ratios through all bricks describing how much (0 – 1) each of them has edges at the same locations as the edges in the previous layer, this objective is minimized;

`uncovered` – total sum of squared ratios through all bricks describing how large area of the brick is uncovered in the previous and following layers, this objective is minimized;

`otherbricks` – total sum of ratios `(1 – 1 / x)` through all bricks, where `x` is the number of bricks from the previous layer that the brick covers. The more bricks in a previous layer a brick covers, the better is the overall stability of the model, this objective is maximized;

`neighbor` – total sum of ratios through all bricks describing how far from the brick's edge is an edge of a neighboring brick (in the direction towards brick's center). This relates to factor 5 described in previous work section. This objective is maximized.

Thus the fitness function corresponds directly to points 1-5 from Gower et al. and we introduced a new heuristics 'otherbricks'.

It has already been mentioned that even though the initializers generate complete and feasible layouts, some operators can lead to layouts with uncovered gaps. To compensate for this, the objective function, before actual computation of fitness, first finishes the layout by randomly filling all of the empty places with fitting bricks (larger bricks are used with higher probability). Thanks to this feature, it was possible to disable the use of 1x1 brick in initializers and other operators. Empty gaps 1x1 are always filled in the objective function. In other cases, 1x1 bricks served only as an obstacle in the way of larger bricks. In addition, it seems that more transformations when generating phenotype from genotype can be useful. For example, the evolution often generates solutions, where two neighboring bricks can be directly merged (i.e. they can be replaced by a larger brick). The merging operation has been carefully implemented into genotype to phenotype transformation before computing the fitness of each individual. More transformations, perhaps based on some library of transformations, might be possible.

The weight constant parameters were tuned empirically and the following values were used in the experiments: $c_{numbricks}$ = 200, $c_{perpend}$ = –200, c_{edge} = 300, $c_{uncovered}$ = 5000, $c_{otherbricks}$ = –200, $c_{neighbor}$ = –300.

4.1 Results

A software package was developed to test the evolutionary approach to this problem. It consists of a program that generates random continuous 3D layered model, implementation of the Genetic Algorithm with operators as described in the previous section, and a simple viewer, which allows to display an evolved solution layer-by-layer either when the evolution is finished or during the progress of evolution. Early runs were used to tune the GA parameters, it seems that the values *pmut*=0.5, *pcross*=0.5, *popsize*=500, and *numgen*=4000 provided a reasonable performance.

Separate runs on the same data showed that based on the part of the search space, which is exploited by random initialization, the resulting solutions can significantly differ for the same input pattern in different runs. However, no clear relation between fitness in layer1 and overall fitness has been noticed, as shown on figure 3. This lead into an idea of running several parallel populations at the same time and occasionally exchanging several genotypes in order to give a chance to the crossover operator to generate good hybrids from individuals evolved in different sub-populations. This is a standard strategy referred to as multiple-deme GA, see [Cantú-Paz, 1998] for parallel GA survey.

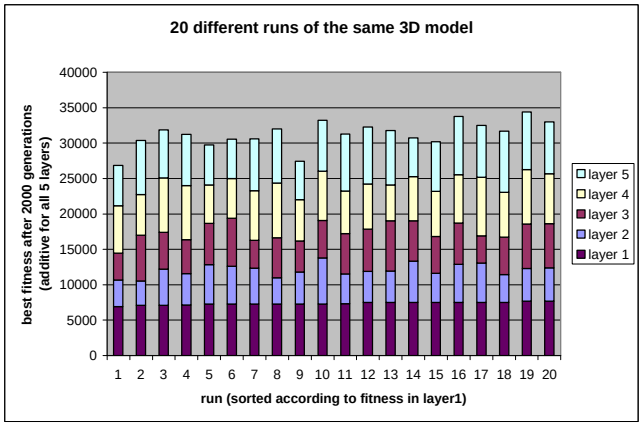

Figure 3. Dependence of the total fitness on the first layer fitness. The overall final fitness varies significantly, its average is 31505.87, and standard deviation is 1869.257.

In addition, the selection method used in the GA can significantly influence the speed of convergence. We experimented with standard roulette-wheel selection and steady state GA – the two basic implementations that are available in the GA library we used for running experiments.

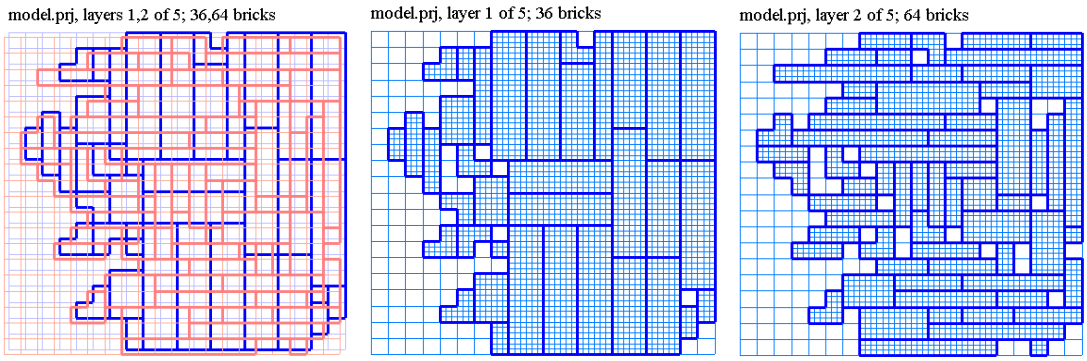

Figure 4. First and second layers of a model evolved with roulette-wheel selection. On the left, the layers are shown together. Notice the low number of edges common in both layers – this makes the layout more stable.

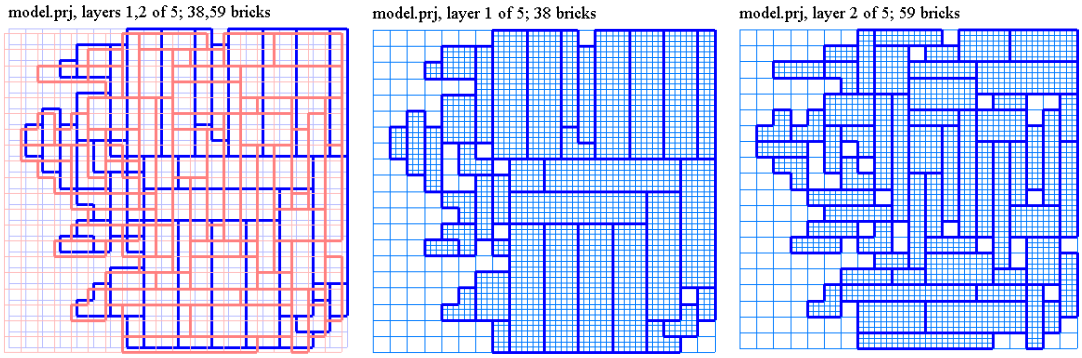

Figure 5. First and second layers of a model evolved with steady-state GA. These input patterns generated by rndmodel tool are somewhat unrealistic and the algorithm should be tested on real LEGO models.

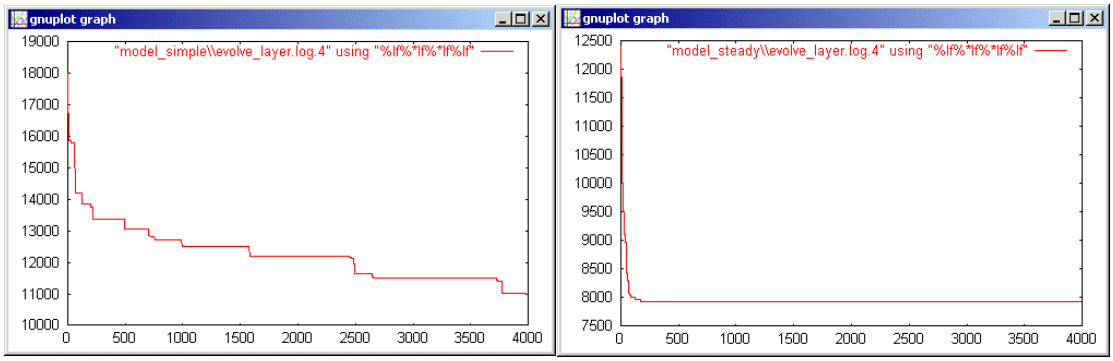

Figure 6. Comparison of evolution progress for roulette-wheel selection (left) and steady-state GA (right). The charts display the fitness of the best individual against the generation number for an example run. The actual computational time spent on one generation of demetic GA is much shorter than on one generation of single-population GA. The total times for 4000 generations were 41311 seconds for roulette-wheel selection and 17391 seconds for steady-state selection. The charts show that the progress is more continuous, smoother, faster, and better converging in case of steady-state selection. See also fig.7.

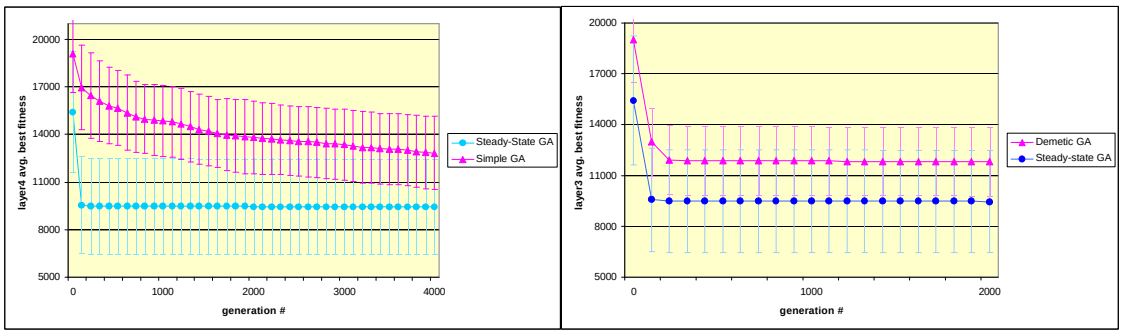

Figure 7. Best individual fitness with standard deviations (average of 20 different random 3D models 20x20x5) for layer 4. Steady-state selection is compared to both roulette-wheel selection (left) and multiple-deme with 5 populations of 1/5 size of population size of steady-state GA and 10 individuals migrating after each generation (right). Both comparisons show that the steady-state GA performs better.

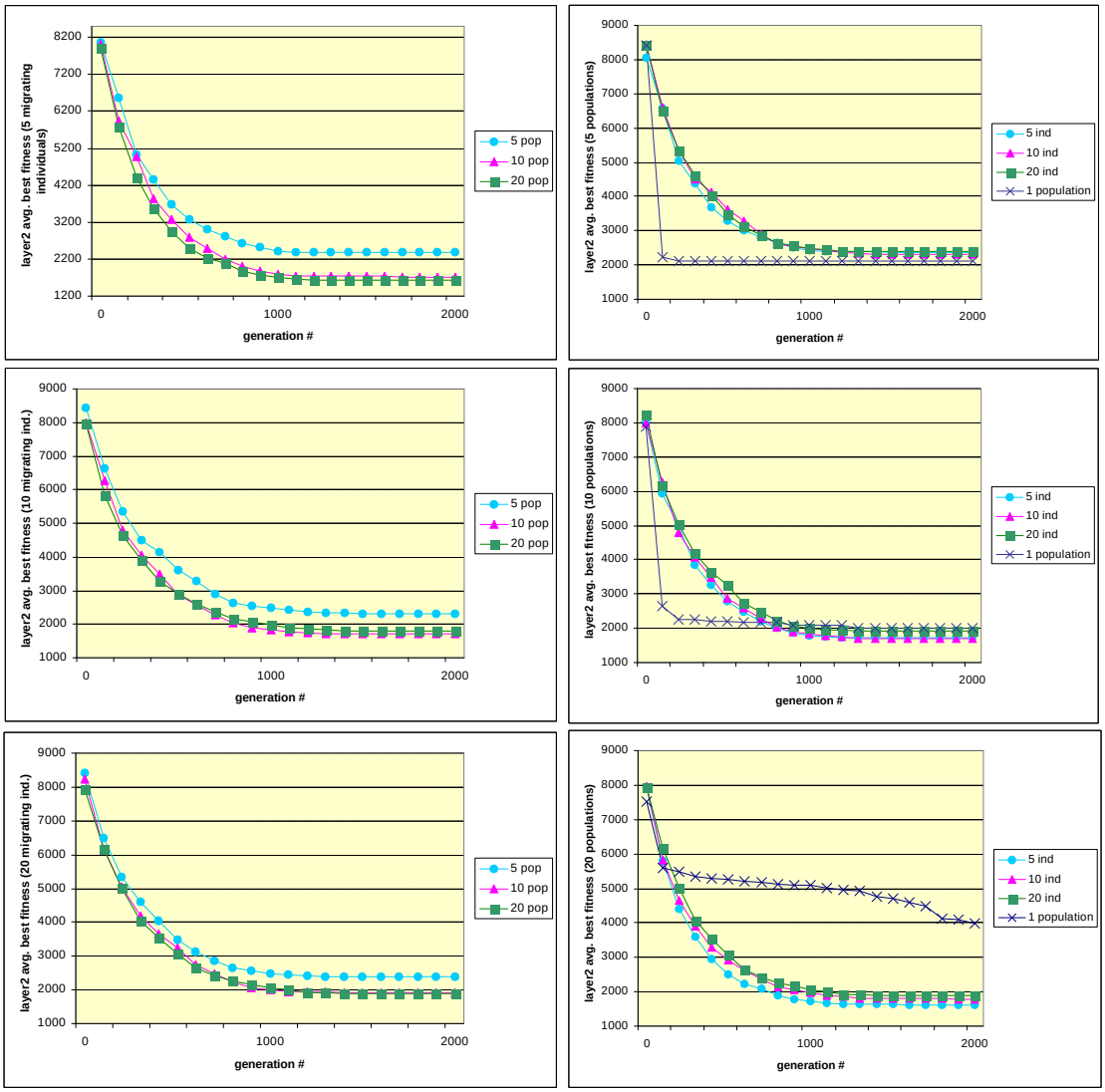

Figure 8. Best individual fitness with standard deviations (average of 20 different random 3D models 20x20x5) for layer 4. Steady-state selection is compared to both roulette-wheel selection (left) and multiple-deme with 5 populations of 1/5 size of population size of steady-state GA and 10 individuals migrating after each generation (right). Both comparisons show that the steady-state GA performs better.

The first part of the results compares these two selection methods. The second part compares single-population and multiple-population GAs with the better selection mechanism. Figures 4 and 5 show two evolved layers for the same example input pattern evolved with both roulette-wheel selection and with steady-state GA. The quality of the final solution is comparable, but the computational effort for the steady-state GA is significantly lower.

Figure 6 displays the fitness of the best individual for the second layer of the same example model for both roulette-wheel selection and steady-state GA. Notice that the required time for the same number of populations in steady-state GA is much smaller.

Figure 7 shows the average of fitness of best individual over 20 different runs on different random models. In the experiments, we used 5 sub-populations and 10 individuals migrated after each generation. A stepping-stone migration scheme was used (i.e. individuals migrate only to a neighboring population arranged in a cycle).

Finally, we studied the dependence of the final fitness on the number of migrating individuals and number of populations used in the multiple-deme GA. We tried all combinations of 5, 10, and 20 populations with migration rates 5, 10, and 20 individuals. The population sizes of all sub-populations were always 500 individuals. Figure 8 shows the averages obtained from runs on 10 different models (2^{nd} layer). For comparison, we evolved the layouts for the same models also using a steady-state GA with the population size equal to the sum of all population sizes for all 3 different population numbers. In case of 20 populations, the population size of the steady state was too large for the selection mechanism to work, so the multiple-deme GA reached better final fitness. In other cases, the single-population GA usually performed better. The total best result was obtained in case of 20 populations and 5 migrating individuals, but took the longest time. 5 populations always performed worse than 10 or 20 populations, which performed very similarly. No clear tendency was observed with respect to the number of migrating individuals.

The software package with which these results were obtained including the data files from the described experiments is available for download at [Project homepage] to ensure the repeatability of results.

5. Future work

5.1 Adaptive Reinforced Mutation Rate

The mutation operator we used is complex – it uses 7 different mutation strategies. The probability of each strategy was established empirically, which is obviously not ideal. Furthermore, for different models, different mutation operators might perform with different success rate. We therefor propose the following improvement to the search algorithm. The rate for each type of mutation will consist of a sum of two components: fixed and variable part. The variable parts will be initialized to a uniform value in the beginning of the evolution. During the evolution, each individual will contain information about how (by which operator) it was created. If its fitness will be over average, the variable part of corresponding operator will be increased by a small step, if the fitness of individual will be below average, the variable part of the corresponding operator will be decreased by a small step. In this way, the successful operators will be invoked with a higher probability and thus the progress of evolution will be faster.

5.2 Indirect representation

The current status of the direct representation allows having individuals consisting of only one brick – the remaining bricks would be inserted by genotype to phenotype conversion in the objective function before computing individual's fitness. This allows easy integration of indirect features in this representation. Particularly, the following extension is proposed.

Each brick placement in the genome list will contain an optional pseudocode. This pseudocode will allow generating more bricks based on this brick placement. The pseudocode will work with a cursor and relative cursor movement commands. Among other, they will include: move cursor 1 unit in any direction; move cursor in a distance of (half of) width/height of current brick; rotate cursor 90º, place a copy of a brick, place complementary brick (rotated 90º); repeat the block several times. In addition, certain standard pattern-filling and tiling algorithms will be provided instead of forcing the evolution to discover them. Possibly, the pseudocode might be evolved in a separate population and each brick placement might stochastically select from this pool of prepared and evolved algorithms. Empirical tests are needed to see whether this strategy could lead to successful solutions.

Alternately, one could completely omit the direct part in the representation and rely for example on genetic programs to generate a complete tiling algorithm. However, it is likely that this approach would need some local search or post-processing applied after the genetic program generates layout in order to achieve full and efficient coverage.

5.3 Interactive evolution

We are also very interested in adding interactivity to the evolutionary process. While the algorithm is searching for a good solution, a human observer can often provide valuable hints about the direction of the search. In case of the brick layout problem, this could be achieved by interactive interface, where the user could see the current representative solution of the population – for example the best individual (this part is already implemented). In this window, a user could edit the layout – remove, rotate, or add new bricks. The new layout would be inserted into the current population in several exemplars, or perhaps it could spawn another parallel population - deme.

6. Discussion

This paper presents an up to date status of a research project directed towards developing an efficient method for generating brick layouts for 3D models built from LEGO bricks. The approach relies on Genetic Algorithm with special initialization, recombination and mutation operators. The genotypes and phenotypes differ, and the representation and objective function are designed in such a way as to allow an easy augmentation of representation with indirect features.

Results demonstrate that the implemented algorithm successfully generates brick layouts for 3D models. In addition, different selection mechanisms and population organization strategies are applied and compared. Steady-state GA performed better and faster than standard roulette-wheel selection and multiple-deme GA.

Adaptive mutation rate is proposed for tuning probabilities of individual operators. By obtaining a global progress measure for different mutation operators, the evolutionary engine can speed up the evolution by using more successful operators with higher probability. Indirect representation for the brick layout problem is discussed. The augmentation of direct representation with indirect features is proposed as a future

research topic. An idea of interactive evolution where a human user can direct the search is proposed and left for future research. More studies and both quantitative and qualitative analysis should be performed in the future. At the moment a Master-level student at University of Southern Denmark adapted the project recently and thus there is a chance for future progress.

7. Acknowledgments

The software for this work used a modified version of GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.

Author is grateful for several useful discussions with members of the EVAL group, particularly Dr. Keith Downing, Magne Syrstad, and Boye Annfelt Høverstad and to anonymous reviewers of the earlier draft of this paper.

8. References

Bentley P.J., (1996), Generic Evolutionary Design of Solid Objects using a Genetic Algorithm, PhD thesis, University of Huddersfield, UK.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. Calculateurs Paralleles. Vol. 10, No. 2. Paris: Hermes.

Dasgupta, Dipankar, Michalewicz, Zbigniew (1997), Evolutionary Algorithms in Engineering applications, Springer-Verlag.

Funes, P., and Pollack, J.(1997), Computer Evolution of Buildable Objects, Fourth European Conference on Artificial Life, P. Husbands and I. Harvey, eds., MIT Press 1997. pp 358-367.

Gower, Rebecca A.H., Heydtmann, Agnes E., Petersen, Henrik G.(1998), LEGO Automated Model Construction, 32nd European Study Group with Industry, Final Report, pp. 81-94, Technical University of Denmark, 30 August – 4 September 1998.

Holland, J.H. (1975). Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press.

Kirpatrick S., Gelatt C.D., Vecchi M.P. (1983), Optimization by simulated annealing, Science 220, pp 671-680.

Laarhoven P. J. M., Aarts E. H. L. (1987), Simulated Annealing: Theory and Applications, D. Reidel Publishing.

Project homepage: http://www.idi.ntnu.no/grupper/ai/eval/brick/.