Automatisation and optimisation of a heuristic design task

Martyna Jakubowska, s173176 - F21

Introduction

Sustainability is becoming an inevitable theme in almost all aspects of life, the building industry being one of them. Wooden construction has gotten a lot of attention in the past couple of years. Of all wooden products, cross laminated timber must be one of the most talked about and praised for its properties. As a result it is no wonder that European CLT firms nearly can't keep up with the demand.

One of such firms is KLH, an Austrian CLT manufacturer, that also provides services for production of work plans for the construction of a solid timber structure. Especially this part of the process - the drawing of work plans, has in many cases become a bottleneck.

In this process configuration of the project, the manufacturer most often receives the plans in late stages of the project, when it is too late to complete major changes. Product-specific restrictions that could have been incorporated more seamlessly into the design earlier, become major road-blocks that harm the cost-effectiveness, as well as sustainability, labor and timeframe.

Another aspect of this problem is that large scale projects can sometimes introduce so much work plan complexity that it can be difficult to survey all rational and possible solutions, and compare them with each other.

A method that could enable architects to become aware of product-specific restrictions and testing different configurations early in the process, could prevent costly last-minute changes. A computerised method, could also have a wide range of unconventional layouts (search space) available, thus possibly produce an optimised solution that would have been difficult for a human to find. This could open up new ways of both using existing standard CLT slab sizes in the most optimal ways, but also using unconventionally sized recycled or residual pieces. Solid wood, if cared for properly, can have a surprisingly long lifetime, so with time, putting emphasis on recycling existing pieces, would be even more in line with sustainable thinking than manufacturing new building materials.

A computational tool that connects with commonly used BIM software would support and speed up the slab layout creation process. Of primary concern is integrating into known working environments and creating an optimal, but complete, workflow that would allow for rapid production of layouts with transparent optimisation metrics.

Theory

The first step of solving such a complex problem as automatic generation of CLT slab layouts, is to define and formalise the task. Key questions to ask for a better understanding of the problem are:

- Which individual components or actions make up the heuristic that experienced engineers use when they produce slab plans?
- Can they be translated and formalised to be understood by a computer?
- What objectives are the engineers more or less subconsciously trying to optimise?
- What could make this heuristic better, thus the solution more optimal in some way? In which ways?

The first outlined question is closely related to a branch of Artificial Intelligence called knowledge-based expert systems (KBESs). KBESs consist generally of two parts; a knowledge database and an inference engine. The knowledge database is a collection of facts about the world, while the job of the engine is to infer new knowledge (reason) based on the known facts.

A related field of AI, case based reasoning (CBR) involves finding solutions to problems based on past good solutions to similar problems. In the past years many researches have employed both methods to the beam-slab layout task. [23-30]

The drawbacks of both methods are that they need a large, good quality database of past projects and complex adaptation techniques for adapting methods from past examples or inferring new knowledge. The two methods can also tend to not be able to incorporate large search spaces of probable solutions, as they are limited by the heuristics used in input data. Especially in situations where one wants to examine the possibilities beyond the typical solutions found by humans, these methods may not be the best choices.

The above mentioned reasons, and the fact that this project doesn't have access to expert opinions and large scale data for the slab layout problem isn't freely available (or exists), has been the deciding factor in moving away from constructing a solution based on human expertise.

The next obvious idea would be if a mathematical solution to such an issue exists. The task of placing rectangular objects inside a polygon, while minimising some parameter, could after all seem trivial.

In computational geometry, many related problems do exist with countless variations each, and have been the subject of research for at least a century. Problems stated in terms of polygons often allow for more efficient solution algorithms when restricted to rectilinear polygons, that is, polygons all of whose edge intersections are at right angles. Therefore this restriction has also been introduced to the slab layout problem, with possibility of expanding the solution to all polygons eg. with custom slab pieces afterwards.

The category of problems nearest to the slab layout task would be packing problems. A popular variant is the bin packing problem, which deals with either packing as many objects into a bin as possible or packing all objects using as few bins as possible. Even though it is a combinatorial NP-hard problem, approximation algorithms exist and optimal solutions to many instances can be produced with sophisticated algorithms.

In a version of the bin packing problem, the inverse or reverse bin packing problem both the number of bins and their sizes are fixed, but the item sizes can be changed. The goal is to change the sizes of as few items as possible. This version of the problem, although more relevant for the CLT slab layout, is not as popular or widely studied and seems to be a harder problem to find even approximate solutions to. In the paper "Notes on inverse bin-packing problems" [41] , authors write

"In this paper, we have shown that the inverse binpacking number problem (IBPN) is NP-hard in the strong sense, regardless of the objective function measure, and intrinsically hard to approximate."

The bin packing problem can also be seen as a special case of the cutting stock problem. The cutting stock problem deals with cutting standardised pieces of stock material, while minimising waste. On the other hand, if the number of bins is restricted to one and each item has both a volume and a value, the problem of maximising the value of items that can fit in the bin is known as the knapsack problem.

The slab layout problem, although not completely, could be described as a form of unbound, geometric knapsack problem. In the unbound knapsack problem (UKP) there is no restriction on how many of each of each item should be used, thus items can be repeated or not used at all. The slabs could be thought of having values in terms of price per m2 or some other measure that would make choosing one slab over another desirable.

Other areas of related studies are polygon partitioning, decomposition and covering. They all have advantageous elements, but don't overlap with the features/goal of the problem completely.

The slab layout problem could be formalised as:

> *Let a set S contain rectangles representing the manufacturer's various CLT slabs.*
>
> *Let a representation of the floorplan P be a rectilinear polygon.*
>
> *Each rectangle has an associated weight, that is some measure of utility, for example cost/m2.*
>
> *Let C be the set for containing the constructed slab layout.*
>
> *Allowing 90 degrees rotation, pick elements of S and pack P completely, such that the elements must be disjoint and their union must be equal to the target polygon (P).*
>
> *Stop, when either the union of the elements of S are equivalent to the plan P,*
> *or when it become impossible to insert disjunct rectangles.*
>
> *If a complete packing is not possible, try to generate novel polygons, that can be contained in some element of S, in order to fill the plane.*
>
> *Overall goal is to minimise the sum of weight per P.*

The proposed expression of weight as price/m2 comes from an idea that the more "processing" such as cutting is needed at the CLT factory, the more expensive a slab would be. Following this logic, custom pieces that fall outside of the standard assortment would be even more expensive. These assumptions aren't supported by any industry information, and could very well be false. It can easily be imagined that in some cases one would want to use as many small pieces as possible to utilise discards from other projects or use recycled pieces. These assumptions have been chosen as a starting point, and changes to the objective wouldn't be difficult to implement.

Price/m2 is also only an example of a weight, and is actually substituted for perimeter of the slabs (rectangles) in the implementation. The bias coming from choosing arbitrary prices with no real life basis, could introduce violability or unwanted behavior in the algorithm. The perimeter is an objective measure that decreases pr m2 with size and is logically correlated between the rectangles. But if actual price data was available, it could possibly better represent the degree to which the elements should be prioritised.

On top of these requirements, there are all of the other conditions that differ this case from a theoretical exercise in computational geometry. For example, the fact that each slab must be supported by walls and that in an arbitrary design would possibly be areas that by default could not be covered only using the existing walls as support. Even if an optimal algorithm that could satisfy all the before mentioned features, existed, the physical constraints would be very difficult to incorporate into an existing algorithm. The expandability of such a solution

would also be poor, as each change would mean total top-down redesign of the algorithm, most likely hurting its runtime significantly.

Thus, no simple solution for the slab layout problem becomes apparent after examining the literature. The topic is very complex, with many variants of each problem. For all of them I studied, applies that none can be solved in polynomial time, and it varies widely, with seemingly small details in the setup, if good approximation algorithms exist. None of the classically defined problems overlap with the necessary features of the slab layout problem, with maybe the exception of a variant of the inverse bin packing problem, for which none relevant literature could be found.

This suggests that an alternative, more flexible solution, that could accommodate the wide range of requirements and be easily expanded upon in the future, would be preferable.

In a series of papers; "Automated layout design of beam-slab floors using a genetic algorithm" [1] and "A genetic algorithm for beam–slab layout design of rectilinear floors" [2] from 2009 and 2010, authors propose a new way of solving the beam-slab layout design using a genetic algorithm (GA) as a design task heuristic.

In the papers the use case is based on concrete beam slab systems, so the optimal placement of beams is an integrated part of the task. Also, not all walls are considered load-bearing.
A sophisticated penalty scheme developed in "An adaptive penalty function in genetic algorithms for structural design optimization" [3] research paper was used. The slab score (later translating to the fitness score) is based on how many corners of the slab are supported by a column. The slab score is given as 1, 0.75, 0.5, 0.25, or 0 if the slab has 4, 3, 2, 1, or 0 corner columns, respectively. The GA described in the paper obtains very good results, and even matches the solution found by engineers for a real life case, as seen on Fig. 1.

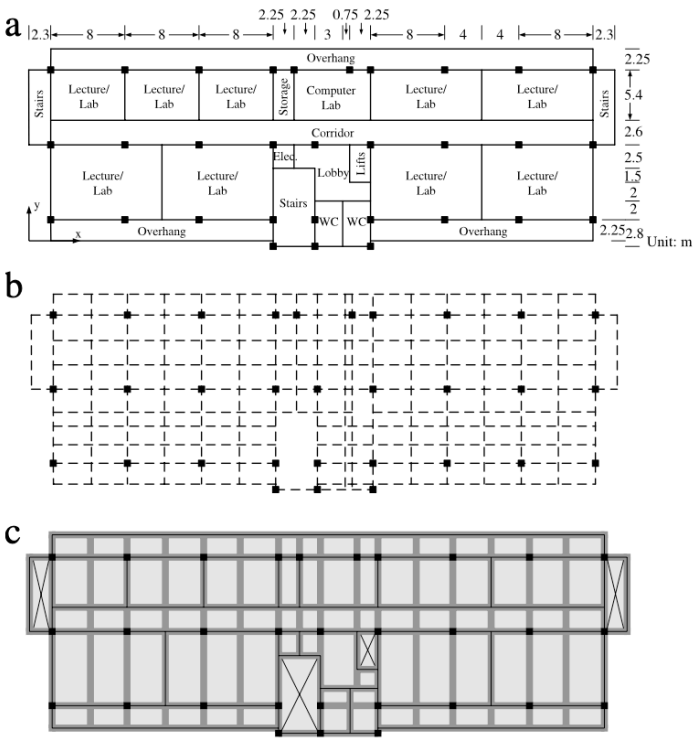

Fig. 19. Problem 3. (a) The simplified architectural floor plan. (b) The grid. (c) The best solution.

Figure 1

The main difference between the case described by Nimtawat and Nanakorn and this project, is that they don't work with predefined sizes of slabs. The slabs are, even though they do have a maximum length, are secondary and adjust to the beam placement. The goal of this project is to optimise the use and waste of prefabricated elements. But the Nimtawat and Nanakorn papers have served as an inspiration and proof that a genetic algorithm is capable of obtaining good results for a complex design task.

Genetic algorithm

A genetic algorithm is part of a larger group of algorithms called evolutionary algorithms, which use mechanisms from biological evolution. Genetic algorithms are inspired by natural selection and borrow concepts as selection, mating, mutation and crossover. I won't go into great detail with the description of GA, there are many good quality resources about it. The main structure is outlined in Fig. 2. The algorithm begins by initialising a number of random solutions (individuals) called a population. It is an iterative process, and the population in each iteration is called a generation. Then each individual of the generation is scored according to the fitness function. The best individuals from each generation have a higher probability of being chosen to be a parent. The selection probability scales with the fitness score, thus individuals with low fitness score still have a chance of being selected. It ensures variation and prevents the algorithm from converting too quickly, thus finding only the local maxima.

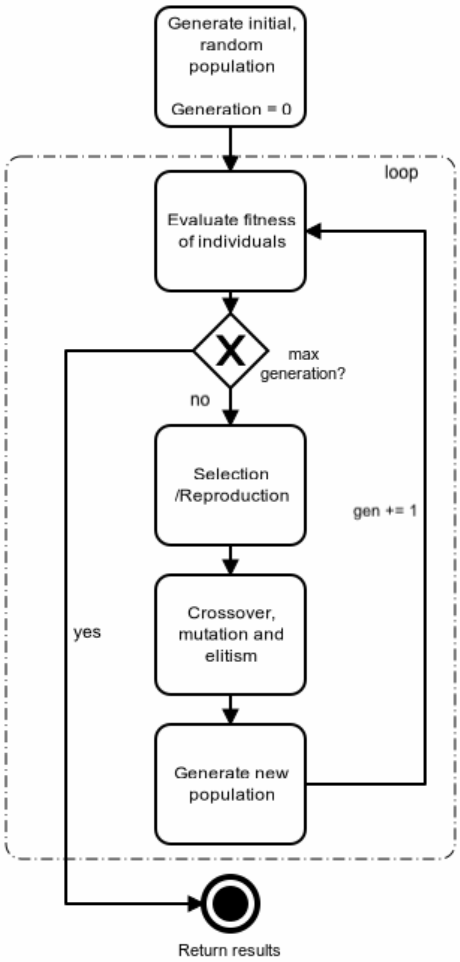

Figure 2

Then the parents "mate" and produce offspring that carry genes from both parents. This can be done in a number of ways, traditionally with binary gene encodings the offspring simply inherits half of genes from parent A and half from parent B. The offspring have a predefined probability of mutating, where a random gene is being changed/flipped. This step also ensures variability.
An extra step can be added to guarantee that the best individuals are not lost between generations. The process is

called elitism and simply carries the best *n* individuals over to the next generation without change.
The remaining "spots" are simply filled with random solutions, consequently making a new generation. All is repeated until the number of generations is equal to a predefined number or the fitness score matches a target.

Method

Code and other relevant files, together with a video of how the program runs can be found on https://github.com/mdjska/slab-place-GA

The target user environment of the tool was from the beginning specified to be Autodesk Revit, since it is the most popular tool used in the later stages of a project by architects and engineers. For clarity and simplicity's sake, there was a wish that the tool had a GUI, that didn't require the user to be familiar with any kind of traditional or visual programming, nor have an extensive knowledge about CLT in particular.

Newer versions of Revit do come with a visual scripting environment called Dynamo, but since it is still in many areas an underdeveloped tool, and among others lack a good library for scripting a user interface, I decided against using it in this project. Instead, Grasshopper, a visual scripting tool for Rhinoceros 3D was utilised as the main link between Revit geometry and the algorithm. Grasshopper can be used with Revit through a new feature called Rhino.Inside where Rhinoceros is emulated inside Revit. Rhino.Inside also has a library that translates native Revit geometry to NURBS, which can further be translated to numeric values and matrices.
The logic of the tool is shown in Fig. 3. Grasshopper has a built-in python component that can run code, but there are several disadvantages to writing code directly inside of the environment. The first is that Grasshopper runs on a non-standard distribution of python called IronPython that is targeted to the .NET framework in which a large part of Rhino geometry is written (RhinoCommon SDK). The IronPython project unfortunately hasn't been actively developed in almost 10 years, and thus hasn't been updated from the python 2.7 release. This means that all syntax and features from the 3.4 release onwards, together with almost all external libraries are not available in the Grasshopper python component.

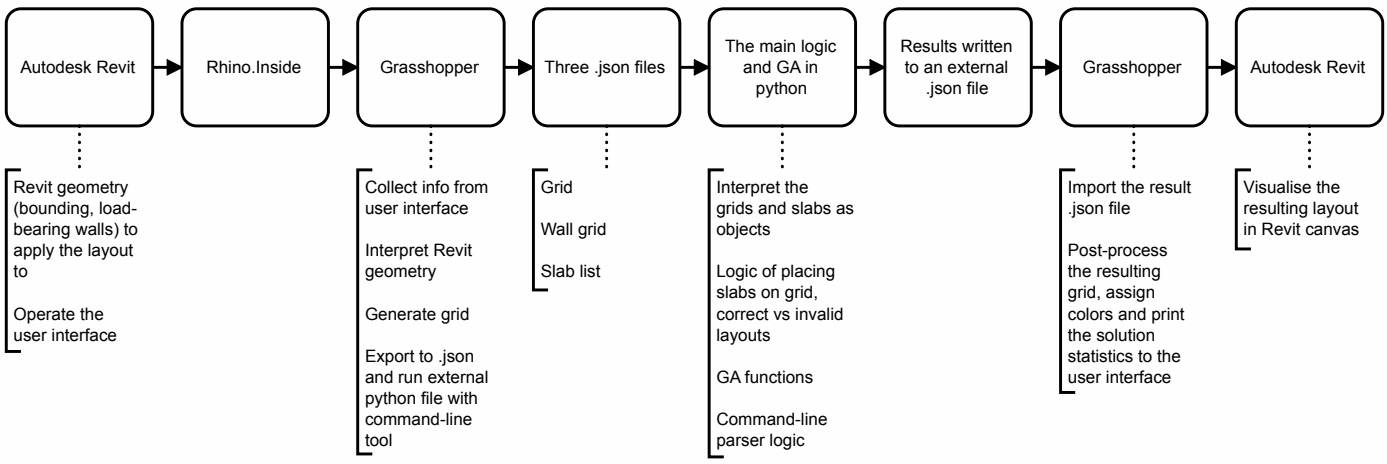

Figure 3

To improve the runtime of the program and have the freedom of writing code in an external editor with all libraries available, the processes have been split by exporting and then importing data to/from .json files. Users can trigger a python component in Grasshopper that runs the main file through a command-line function. This is a fast and stable solution that improved the responsiveness of Grasshopper significantly. When the program is done, another .json file containing a grid (matrix) with the placed slabs is written, and can then be fetched by Grasshopper for post-processing.

On Fig. 4 the whole Grasshopper script can be seen. Fig. 5 shows how native Grasshopper data trees are converted to lists of lists, written to files and the main program is run.

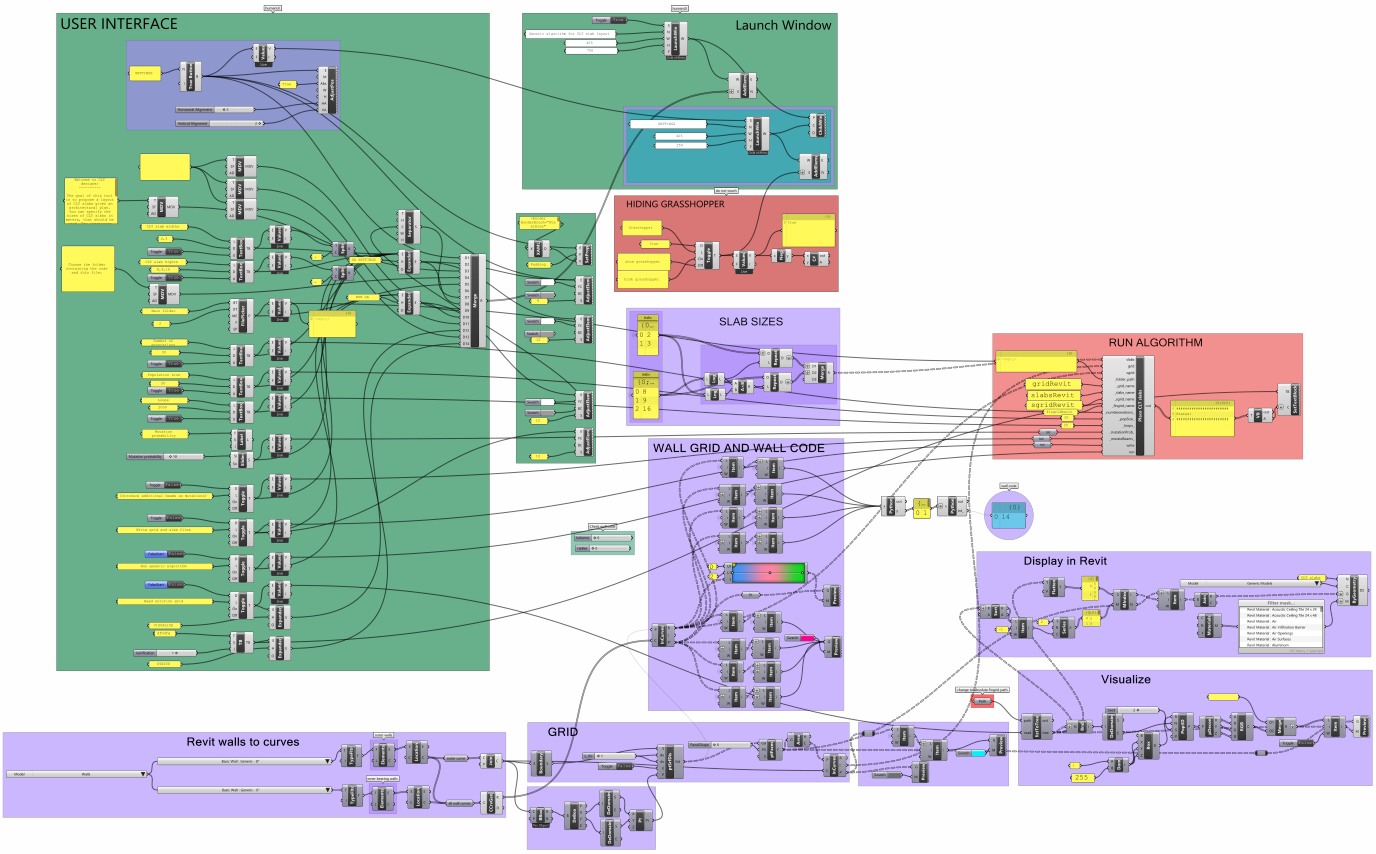

Figure 4

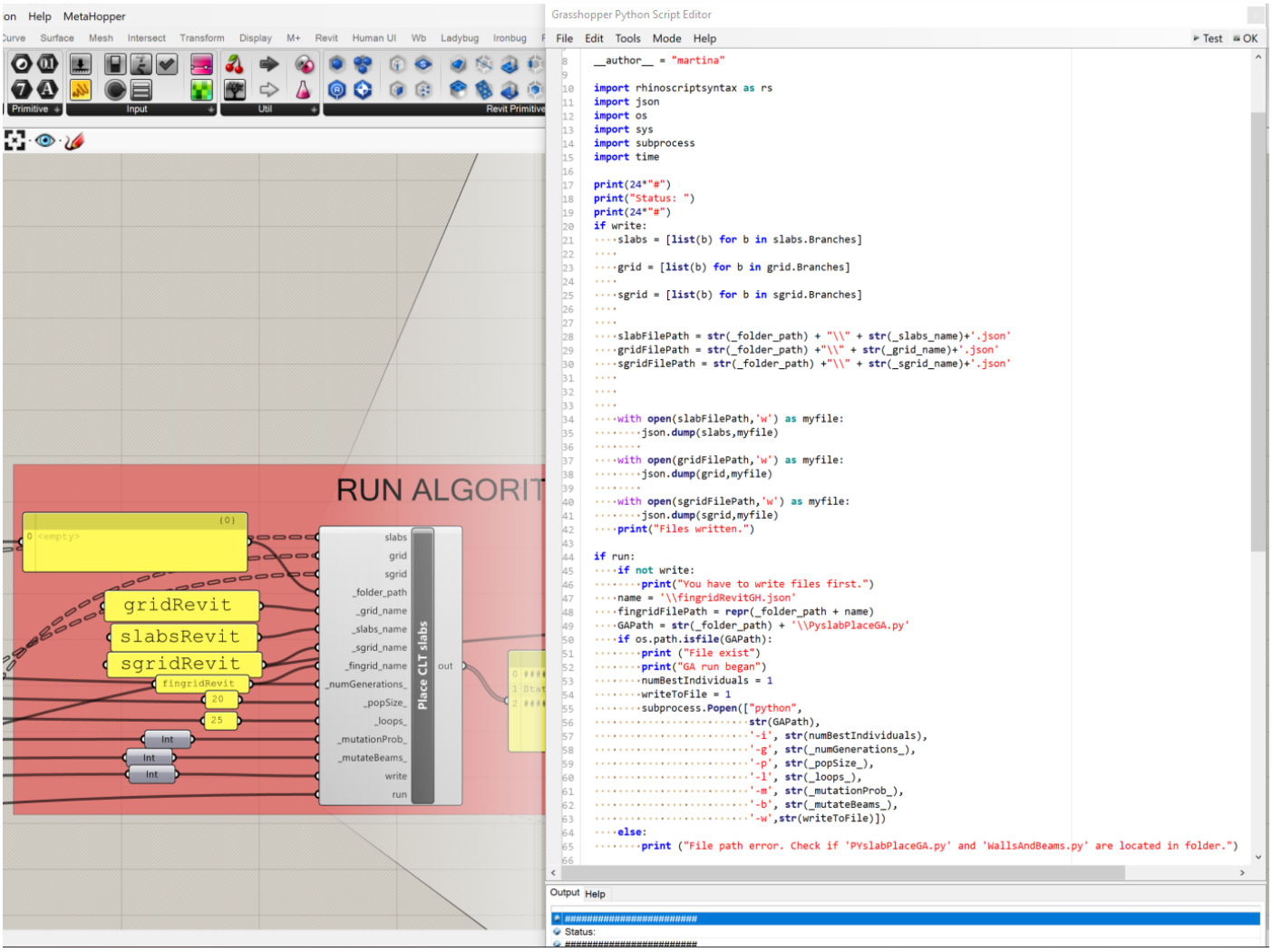

Figure 5

The user interface is kept simple, with only the necessary parts included; a short explanation, settings, run toggles and a status window.

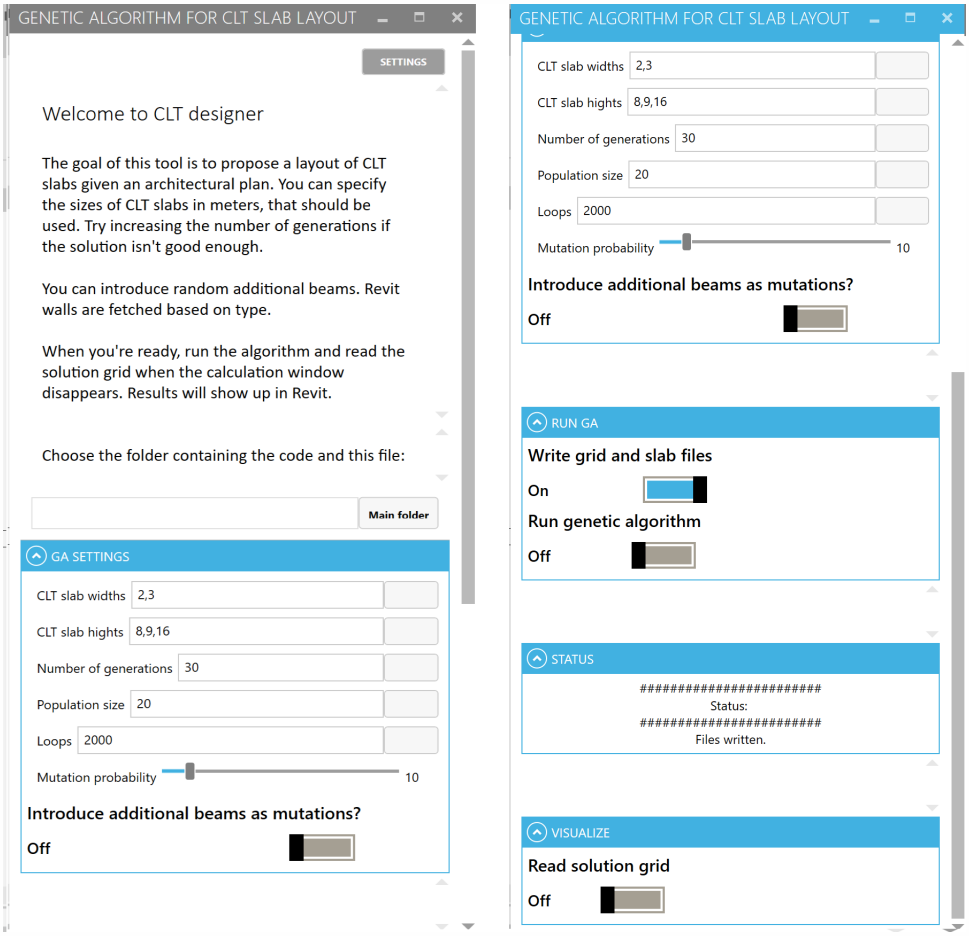

Figure 6

The main code

The main code with documentation can be found on ***github link***. It consists of two files, WallsAndBeams.py and GAmain.py. WallsAndBeams.py contains the logic of the physical characteristics of the project, while the main file contains the GA itself.

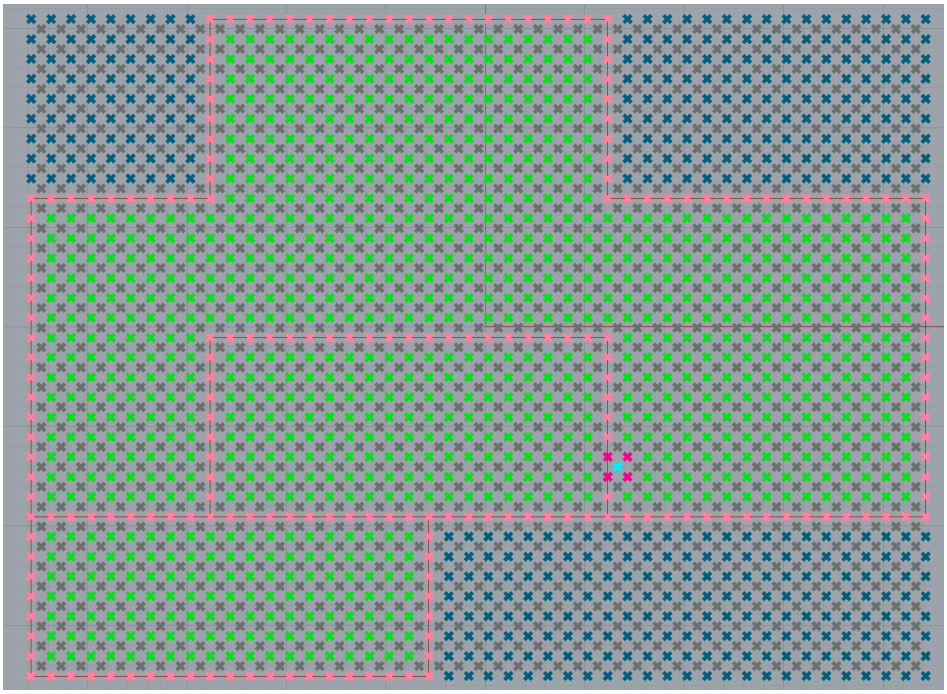

Figure 7

Two grids are imported from the .json files. They are staggered in relation to each other. On Fig. 7 the grey crosses represent the main grid, that slabs are later added to (grid A), while the colored crosses represent the grid that overlaps with, and thus has information about, the walls (grid B). Grid A starts by having two possible values, -1 if grid cell is outside the architectural plan, and 0 if it is inside. Grid B can have three values, -1 for outside the plan, 0 for “empty” and 1 if the grid point overlaps with a wall. On Fig. 7, -1, 0 and 1 values for grid B are marked with dark blue, green and pink respectively.

Since all slabs need to be supported from at least two opposite sides, the grid A grid cells need to retrieve information about adjacent walls. To contain the information in only one number, an original system inspired by the binary code is used. As seen on Fig. 8, for one grid cell A there are 4 grid points B associated, ordered from 1 to 4. The sequence of associated points corresponds to four digits that can either have value 0 or 1. 0 and 1 values correspond to the values from grid B representing either wall or no wall. Thus for example a grid cell that is adjacent to a wall to the left, would have the code 1010. These digits can then be interpreted as binary numbers and converted to arabic numerals. The cell marked with bright blue on Fig. 7, would have the wall code 1010 = 10. Fig. 8 shows the rest of the used wall codes. This system makes it easy for a given slab to check if it can be supported with either from “North” and “South” (up and down) or “East” and “West” (right and left) at any given location.

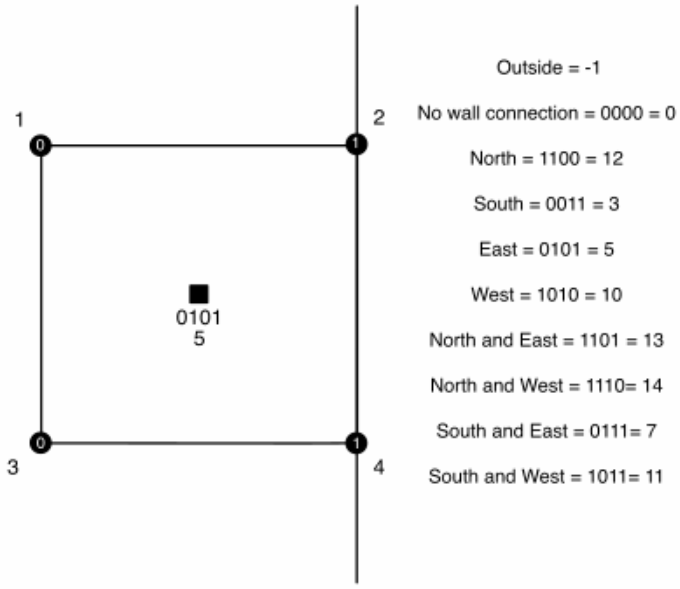

Figure 8

Another interesting implemented feature is the possibility of introducing one or more beams as an additional mutation in the algorithm. Since it is not guaranteed that the available slabs will be able to cover the whole plan, an additional beam can be necessary at times. To follow common practice, the beams can be laid as an extension to existing walls, starting at a corner and following in one of two possible directions shown in Fig. 9.

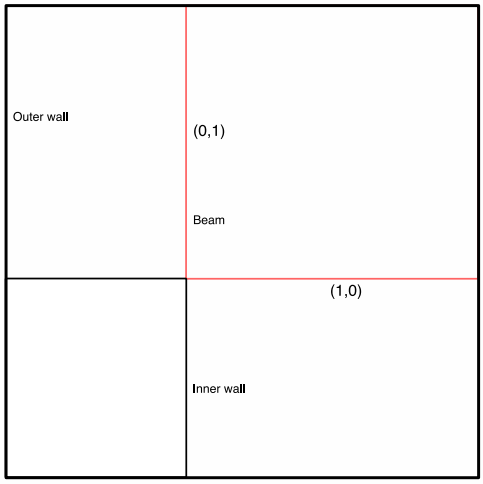

Figure 9

A price-based penalty could be introduced in the fitness function, that would help determine the beneficial effect of an additional beam.

As mentioned in the theory section, the scoring factor for each slab has been changed from a (fictional) price, to the perimeter in the fitness function. This creates a bias towards bigger slabs (as they cover more area with less perimeter), as they are assumed to be cheaper per m2. The number of grid cells that are not covered by any slabs, should of course also be discouraged, and is arguably more important than the size of the slabs. Therefore in the fitness function it is being emphasised by squaring the value. The reciprocal of the expression is used to get a positive, growing fitness score for better layouts (so the fitness function can be maximised). The fitness function looks like:

$$fitness = 1/(total\ perimeter + empty\ grid\ cells^2) \tag{1}$$

Since the algorithm could be run with a couple of hundred generations at a time, it was important to keep the runtime manageable. Before the optimisation, a 20 generation run for a test building with around 1200 grid cells, would take around 25 seconds. A strategy to improve the runtime was implementation of a just-in-time compiler called Numba. Numba works best for NumPy arrays and functions, and has some peculiarities around how the code is written, but after implementation, the same 20 generation run took consistently 3.6 seconds. That is an almost 7-fold improvement, that guaranties good performance even for larger projects and many generations.

Results and discussion

A simple architectural plan, that has been tested, can be found on Fig. 10. On Fig. 11, a run of the algorithm with population size of 30 and 200 generations is shown.

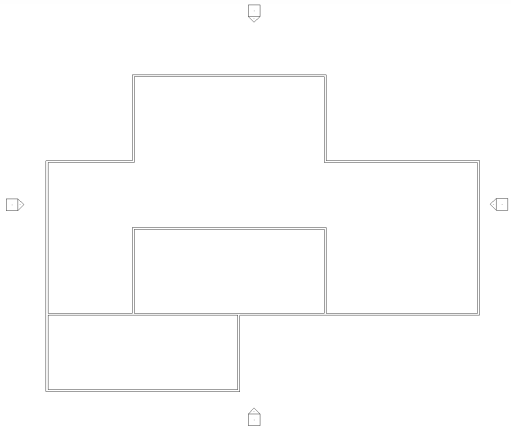

Figure 10

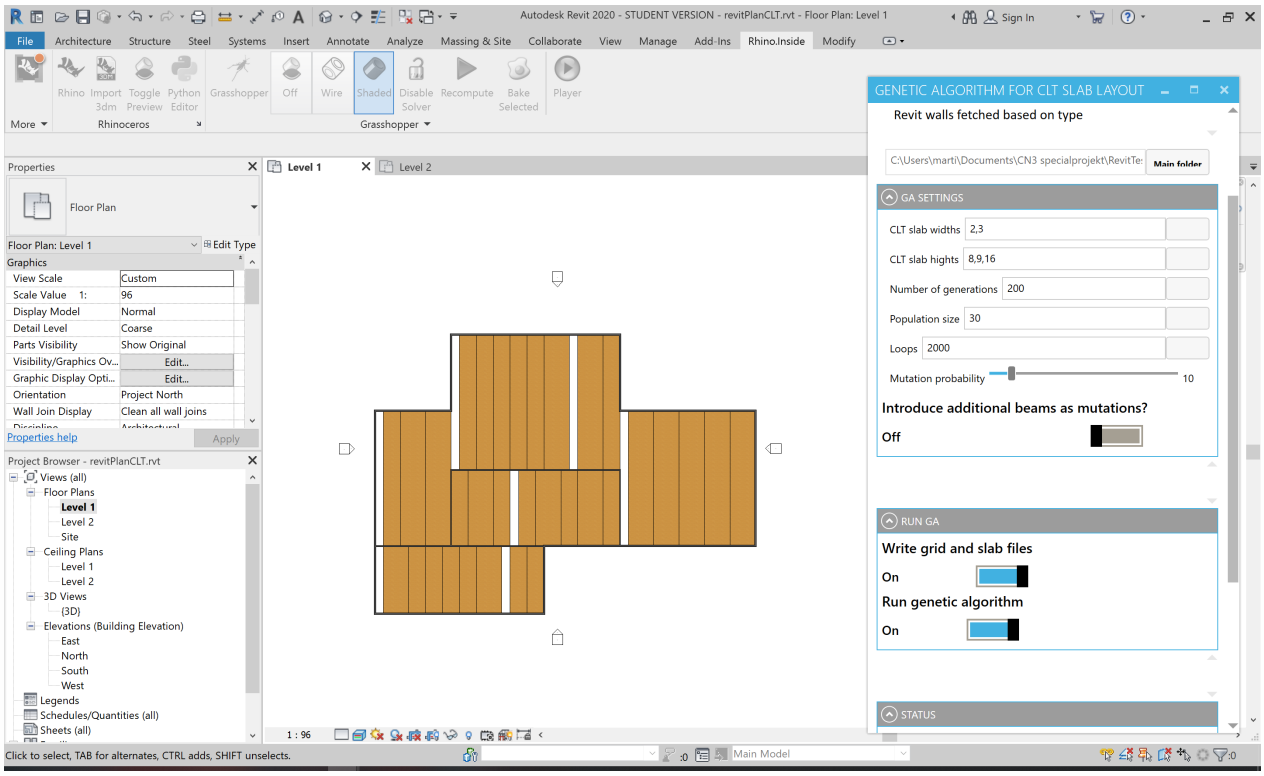

Figure 11

At 200 generations, the algorithm doesn't find a full covering of the plan, but is pretty close. At the point of tinkering with the parameters and trying different versions of fitness function, the genetic algorithm becomes an art. It often takes quite a bit of experience and/or time to get it right. A lot of research has been done on the subject of how to define good fitness functions and mating mechanisms. The methods for both fitness function, mating and mutation probably aren't the best or most optimal. They were tries, as not much literature is comparable to this unique case.

For example, if the fitness function prioritised (gave higher score to) slabs that had direct contact with either the walls or other slabs, it would possibly solve the problem of narrow gaps. In this configuration, it is possible that there is simply too little difference in the scores in the beginning iterations, when most of the grid cells are still empty. The algorithm can easily fall into a nonoptimal design, simply because it has covered more grid cells.

For this sample plan, the GA seems to converge around 200 - 400 generations. Fig. 12 shows the results for 300 generations, and the differences are not big.

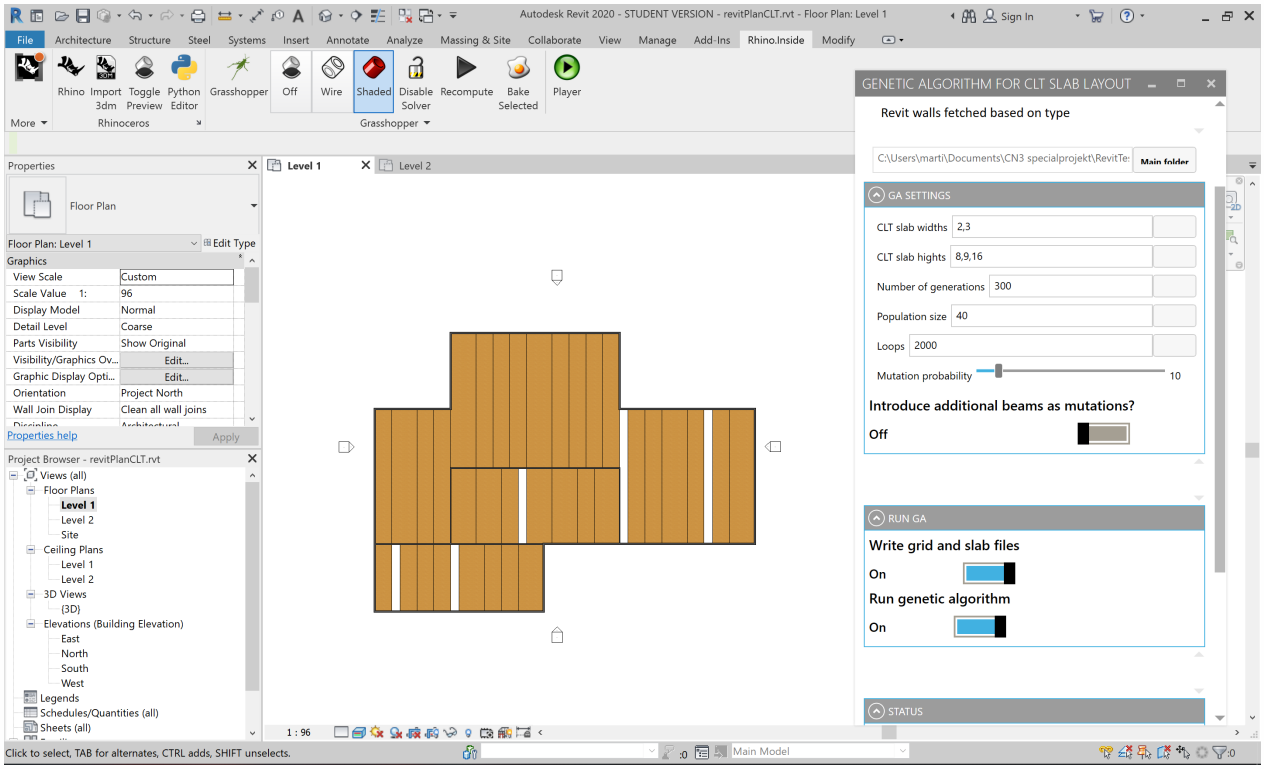

Figure 12

A result with 50 generations but with the use of additional beams can be seen on Fig. 13. The new beam is marked with red.

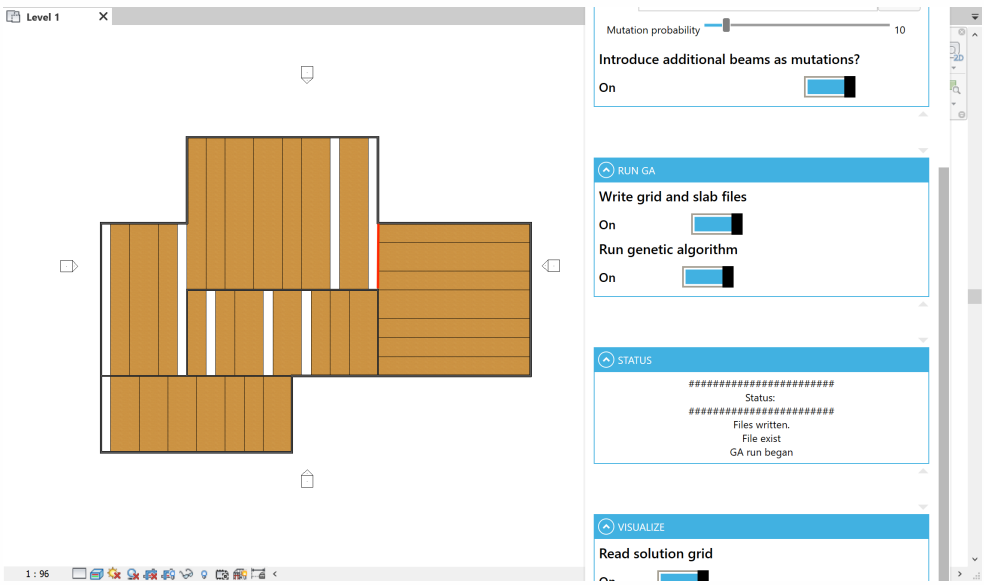

Figure 11

Conclusions

The work of many engineers is filled with repetitive tasks that may also be quite easy for them to complete. That is easy — but not necessarily simple. These tasks are simple for humans because they are a heuristic developed through many years of experience with problem solving and snap judgments in everyday life. Humans do this by identifying and using intrinsic properties of a problem together with conceptual knowledge about the world and past experiences.

Those are not metrics that a computer can (simply) understand. Thus a problem that is relatively easy to solve for a human, for example filling a polygon with rectangles, can be very complex and difficult to delegate to a computer. If no exact algorithm is known, an evolutionary algorithm can act as a kind of shortcut that mimics human heuristics.
This approach has a big potential in the category of easy but not simple design tasks.

Hopefully incorporating, and perfecting more of such methods can leave more time for tasks that are neither easy nor simple.

References

[1] Nimtawat A, Nanakorn P. Automated layout design of beam–slab floors using a genetic algorithm. Comput Struct 2009;87:1308–30.

[2] Nimtawat A, Nanakorn P. A genetic algorithm for beam–slab layout design of rectilinear floors. Comput Struct 2010;32:3488–00.

[3] Nanakorn P, Meesomklin K. An adaptive penalty function in genetic algorithms for structural design optimization. Comput Struct 2001;79:2527–39.

[4] Handbook of computational geometry, chapter 11: "Polygon Decomposition"*

[5] Graph-Theoretic Solutions to Computational Geometry Problems

[6] Jukka Jylänki. (2010). A thousand ways to pack the bin – a practical approach to two-dimensional rectangle bin packing .

[7] Mathews JD. Optimisation and decision support during the conceptual stage of building design – new techniques based on the genetic algorithm. PhD thesis, School of Civil and Sructural Engineering, Faculty of Technology, University of
Plymouth; 2000.

[8] Rafiq MY, Mathews JD, Bullock GN. Conceptual building design – evolutionary approach. J Comput Civ Eng 2003;17:150–8.

[9] Sisk GM, Miles JC, Moore CJ. Designer centered development of GA-based DSS for conceptual design of buildings. J Comput Civ Eng 2003;17:159–66

[10] Shaw D, Miles J, Gray A. Determining the structural layout of orthogonal framed buildings. Comput Struct 2008;86:1856–64.

[11] Bernard, C., 1983. The bottom-left bin packing heuristic: an efficient implementation. IEEE Trans. Comput. 32 (8), 697–707.

[12] Goldberg DE. Genetic algorithms in search, optimization, and machine learning. Massachusetts: Addison-Wesley; 1989.

[13] Chu, P.C., and J.E. Beasley. 1998. "A Genetic Algorithm for the Multidimensional Knapsack Problem." *Journal of Heuristics* 4: 63-86. DOI: 10.1023/A:1009642405419.

[14] Yao B, Chen H, Cheng CK, Graham R. Floorplan representations: complexity and connections. ACM Trans Des Autom Electron Syst 2003;8:55–80

[15] Murata H, Fujiyoshi K, Nakatake S, Kajitani Y. VLSI module placement based on rectangle-packing by the sequence-pair. IEEE Trans Comput-Aided Des Integr Circuits Syst 1996;15:1518–24.

[16] Maßberg, J., & Schneider, J. (2011). Rectangle packing with additional restrictions. Theoretical Computer Science, 412(50), 6948–6958. https://doi.org/10.1016/j.tcs.2011.09.007

[17] Balasa F. Modeling non-slicing floorplans with binary trees. In: IEEE/ACM international conference on computer-aided design, digest of technical papers. 2000. p. 13–6.

[18] Bausys R, Pankrasovaite I. Optimization of architectural layout by the improved genetic algorithm. J Civil Eng Manag 2005;11:13–21.

[19] Mathews JD, Rafiq MY. Adaptive search to assist in the conceptual design of concrete buildings. In: Developments in neural networks and evolutionary computing for civil and structural engineering. 1995. p. 179–87.

[20] Wu, L., Tian, X., Zhang, J., Liu, Q., Xiao, W., & Yang, Y. (2017). An improved heuristic algorithm for 2D rectangle packing area minimization problems with central rectangles. Engineering Applications of Artificial Intelligence, 66, 1–16. https://doi.org/10.1016/j.engappai.2017.08.012

[21] Rémila E. (2004) Tiling a Polygon with Two Kinds of Rectangles. In: Albers S., Radzik T. (eds) Algorithms – ESA 2004. ESA 2004. Lecture Notes in Computer Science, vol 3221. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30140-0_51

[22] Huang, W., Chen, D., & Xu, R. (2007). A new heuristic algorithm for rectangle packing. Computers & Operations Research, 34(11), 3270–3280. https://doi.org/10.1016/j.cor.2005.12.005

KBE and CBR:

[23] Tsakalias GE. KTISMA: a blackboard system for structural model synthesis of asymmetrical skeletal reinforced concrete buildings. Artif Intell Object Oriented Approaches Struct Eng 1994:15–21

[24] Syrmakezis CA, Mikroudis CA, Rouva S. Development of the VK.Expert system for computer-aided preliminary design of reinforced concrete buildings. Inf Process Civ Struct Eng Des 1996:45–51.

[25] Syrmakezis CA, Mikroudis GK. ERDES – an expert system for the aseismic design of buildings. Comput Struct 1997;63:669–84

[26] Sacks R, Warszawski A. A project model for an automated building system: design and planning phases. Autom Construct 1997;7:21–34.

[27] Sacks R, Warszawski A, Kirsch U. Structural design in an automated building system. Autom Construct 2000;10:181–97

[28] Bailley SF, Smith IFC. Case-based preliminary building design. J Comput Civ Eng 1994;8:455–68.

[29] Kumar B, Raphael B. CADREM: a case-based system for conceptual structural design. Eng Comput (Hist Arch) 1997;13:153–64

[30] Fenves SJ, Rivard H, Gomez N. SEED-Config: a tool for conceptual structural design in a collaborative building design environment. Artif Intell Eng 2000;14:233–47.