PrismArch

Deliverable No D2.3

Final revised version of parametric space of design, algorithms for AI assisted editing/design in VR, and algorithms for designer modelling

Project Title:	PrismArch - Virtual reality aided design blending cross-disciplinary aspects of architecture in a multi-simulation environment
Contract No:	952002 - PrismArch
Instrument:	Innovation Action
Thematic Priority:	H2020 ICT-55-2020
Start of project:	1 November 2020
Duration:	27 months
Due date of deliverable:	September 31st, 2022
Actual submission date:	October 18th, 2022
Version:	1.0
Main Authors:	Konstantinos Sfikas (UoM), Antonios Liapis (UoM), Georgios N. Yannakakis (UoM), Joel Hilmersson (AKT II)

Project funded by the European Community under the H2020 Programme for Research and Innovation.

Deliverable title	Integration-ready version of AI algorithms to traverse the parametric solution space
Deliverable number	D2.3
Deliverable version	1.0
Contractual date of delivery	September 31st, 2022
Actual date of delivery	October 18th, 2022
Deliverable filename	Deliverable_D2.3_v1.0.pdf
Type of deliverable	Other
Dissemination level	Public
Number of pages	94
Work Package	WP2
Task(s)	T2.3
Partner responsible	UoM
Author(s)	Konstantinos Sfikas (UoM), Antonios Liapis (UoM), Georgios N. Yannakakis (UoM), Joel Hilmersson (AKT II)
Editor	Konstantinos Sfikas (UoM)
Reviewer(s)	Jeg Dudley (AKT II), Spiros Nikolopoulos (CERTH)

Abstract	Deliverable D2.3 describes the algorithmic library for exploration of the design-space of all three involved AEC disciplines: Architecture, Structural Engineering and MEP Engineering. It presents the final application of Quality Diversity and Designer Modelling to all three disciplines, following the design principles that were introduced in D2.1 and D2.2 and have been finalised with the feedback of AEC partners through a large number of dedicated workshops and technical meetings.
Keywords	Artificial Intelligence, Parametric Design, Possibility Space, Design Intelligence, Spatial Analytics, Fitness Functions,

	Constraints, Evolutionary Algorithms, Diversity Measures, Designer Modeling, User Modeling, Structural Engineering, MEP Engineering, Architecture

Copyright

Deliverable history

Version	Date	Reason	Revised by
0.1	15/09/2022	Table of Contents	UoM, CERTH
0.2	22/09/2022	Beta version, including description of algorithmic approaches and AI tools for each discipline	UoM
0.3	29/09/2022	Additional material from AKT (alternative approach to applying Quality Diversity to Structural Engineering problems). Additional section describing aspects of cross-disciplinary cooperation based on the AI-Tools.	UoM, AKT
0.4	05/10/2022	Finalisation of all content, addressing minor issues with internal reviewers.	UoM, AKT
1.0	10/10/2022	Final version, ready for external review	UoM

List of abbreviations and Acronyms

Abbreviation	Meaning
AEC	Architecture, Engineering and Construction
AKT	AKT II Limited
AR	Augmented Reality
BIM	Building Information Modelling
CAD/CAM	Computer-Aided Design & Computer-Aided Manufacturing
CERTH	Center for Research & Technology Hellas (ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS)
DoA	Description of Action
EA	Evolutionary Algorithm
EC	Evolutionary Computation
IP	Intellectual Property
MAP-Elites	Multidimensional Archive of Phenotypic Elites [4]
NDA	Non-Disclosure Agreement
QD	Quality-Diversity
ToC	Table of Contents
UoM	University of Malta (UNIVERSITA TA MALTA)
VR	Virtual Reality
WP	Work Package

Executive Summary

Deliverable D2.3 describes the algorithmic library for exploration of the design-space of all three involved AEC disciplines: Architecture, Structural Engineering and MEP Engineering. It presents the final application of Quality Diversity and Designer Modelling to all three disciplines, following the design principles that were introduced in D2.1 and D2.2 and have been finalised with the feedback of AEC partners through a large number of dedicated workshops and technical meetings.

Chapter 1 (Introduction) includes the description of action (DoA) of this deliverable as well as its relation to other deliverables. Furthermore it explains how the specific approach of this deliverable has been shaped, in cooperation with the AEC industry partners, so as to facilitate the different perspectives, as well as the designers' subjective criteria. Finally, it presents and justifies the slight deviations from the DoW.
Chapter 2 (Source-Code Description) describes the basic structure of the developed computational framework.
Chapter 3 (General purpose AI algorithms) describes the novel algorithmic approaches that have been developed by the University of Malta in the context of PrismArch. Those algorithms are combining aspects of (Interactive) Quality-Diversity and Designer Modelling into a suite of potential approaches and combinations that offers a very flexible toolset for addressing a broad range of problems.
Chapter 4 (Integration of Algorithms into an AI assisted tool, per discipline) describes the specific way in which the developed algorithmic approaches have been applied to each of the three AEC disciplines.
Chapter 4.1 (AI-Tool for Architecture) describes the application of the *DM-based Interactive FI-MAP-Elites* algorithm to design problems of the Architecture discipline.
Chapter 4.2 (AI-Tool for Structural Engineering) describes the application of the *DM-Based FI-MAP-Elites* algorithmic approach to design problems of the Structural Engineering discipline.
Chapter 4.3 (AI-Tool for MEP Engineering) describes the application of the *DM-Based FI-MAP-Elites* algorithmic approach to design problems of the MEP Engineering discipline.
Chapter 5 (Cross-discipline integration) describes the ways in which the aforementioned AI-tools are aligned with the cross-discipline cooperation between the three involved AEC disciplines, taking into consideration broader aspects of the PrismArch platform.
Chapter 6 (Conclusions) Offers a general summary of the progress made so far, a number of ideas for future research and development, and general conclusions.
Chapter 7 (References) Includes a number of references to scientific publications and other deliverables of PrismArch.

Table of Contents:

1. Introduction

1.1. Description of Actions and relevant deliverable documents

As prescribed in the DoA and, more specifically, the description of T2.3 and D2.3, this software deliverable “includes the updated AI algorithms of D2.2, following feedback from tests in WP6 (D6.3), as well as the trained designer models for adjusting the generated suggestions to match the target design discipline and personal preferences of the designers”.

During the second year of development, a large emphasis has been placed on expanding the application of developed algorithms to an equal degree, across all three involved AEC disciplines: Architecture, Structural Engineering and MEP Engineering. The currently developed computational framework now includes a number of case-studies that showcase its applicability to key-aspects of each discipline’s design problems. The framework’s modular implementation, which was planned since the beginning and reported extensively in D2.2, has already paid off, as it allowed for the algorithms’ application to all three disciplines and in multiple ways for the Structural Engineering discipline.

Furthermore, the set of novel algorithmic approaches - and their accompanying user interfaces - has been implemented and documented herein. Consultation with AEC partners and user experience partners has already informed the interaction paradigms and modelling algorithms, and preliminary tests have shown their potential, while more usage tests will be conducted until the conclusion of PrismArch as part of WP6.

1.2. Continuous cooperation with AEC partners

Following collaboration throughout the course of WP2, evidenced by end-user questionnaires in [D2.1], the final development of algorithms and –importantly– of interaction paradigms and designer models was in close cooperation with AEC partners. A series of workshops, offline discussions, and virtual meetings was conducted between the submission of D2.2 and the preparation of D2.3. Such meetings were often bilateral, attempting to dig deeper on each discipline’s needs and the most impactful case-study where AI suggestions and models would be useful. Through this process, additional concerns were raised regarding portability of the algorithms in more problems and addressed through a non-VR application in Rhino/ Grasshopper (see Section 1.3). Throughout the project, the potential uses of the developed AI algorithms were presented to AEC partners for their feedback. Some requirements specific to the AI algorithms (e.g. the definition of Behavioural Characterizations / BCs) were made clear to AEC partners; in collaboration with experts in their respective discipline, specific BCs were designed.

Beyond AEC partners, user experience experts of WP3 were consulted regarding best ways of presenting AI output in a way that minimizes cognitive load. Finally, the integration of the AI algorithms in a cross-disciplinary fashion was designed with close collaboration with developers (CERTH), usability experts (ETH) and AEC end-users.

1.3. Deviations from the Description of Actions

The main deviation from the Description of Actions has to do with the User-Interface through which the designers interact with the algorithmic approaches. The initially promised goal was to incorporate all algorithmic operations within the VR-based PrismArch platform. However, implementing all the necessary ui elements and providing the necessary degree of flexibility was deemed to be infeasible, at this stage of the project, especially for the engineering disciplines that need a much finer degree of control. The Rhino/Grasshopper design environment was utilised to mitigate this deviation and to provide a much faster way to offer the necessary degree of control to all disciplines. The following paragraphs describe in full detail the ways in which the UI of the AI-Tools was implemented.

The User-Interface for the Architectural Discipline has been implemented directly within the PrismArch platform's VR application, for the most part. The initial definition of a Design Specification takes place within the Rhino/Grasshopper environment, while the core-part where the designer interacts with the algorithm to co-explore the solution space takes place within the VR-application.

For the Structural Engineering and MEP Engineering disciplines, a significant number of discipline-specific controls are needed to create input geometries and assign initial values within the PrismArch VR platform. Given the project's level of development at that time in the project, it was decided by all consortium members to not attempt to create these PrismArch-internalised components, and instead to take advantage of the functionality already provided by Rhino/ Grasshopper. The specific platform was chosen for a number of reasons: First, Rhino / Grasshopper is a broadly used software solution, across many AEC disciplines. Second, it offers extensive 3D modelling functionality, as well as a computational design environment, both of which can be utilised and extended when developing plug-ins within that platform. Third, the platform supports an integration with the C# programming language, further facilitating the easy integration of our already developed and general-purpose algorithmic approaches.

This decision gave us the flexibility to quickly develop a complex user interface that offers the degree of flexibility and access to detail that engineers are accustomed to. The adjustment of Rhino/ Grasshopper as our front-end has not diminished the underlying development of our computational framework, as the AI codebase is a generic C# based library and is in no way "tied" to the Rhino/Grasshopper environment. The underlying algorithmic source-code can easily be transformed so as to apply to another User-Interface that is completely integrated with the VR application.

While not strictly a deviation, discussion and collaboration with PrismArch partners has also allowed for a more crystallised concept regarding the designer modelling algorithms and their use. Therefore, Designer Modelling was treated per discipline, in order to respect the

boundaries posed by legal issues (IP, etc.). Moreover, due to the differences in interfaces (VR versus Rhino/ Grasshopper) and the different priorities between disciplines, each discipline has its own type of designer model. That said, the underlying AI algorithms developed are the same for all disciplines and can be used beyond AEC concerns as general-purpose libraries. Each discipline's model has different parameters to be considered for ascertaining a user's preference. Moreover, each model is used differently, focusing either on the preference as quality (Architecture) or focusing on constraint satisfaction as the main quality dimension and preference as a dimension for diversification (Structural and MEP Engineering). As another emergent pattern, it was deemed that the designer models should be persistent, stored as part of the data structure of PrismArch; these models are stored on a per-user basis, thus respecting privacy and IP issues (see Section 5).

2. Source-Code Repository Description

The source code for all AI-Tools has been developed in the C# programming language, using the Visual Studio Community Edition 2022 IDE, v17.1.3 (https://visualstudio.microsoft.com/). It is organised into a number of interdependent Projects, all of which can be found within a Visual Studio Solution named "Evolutionary_Design_Framework" (EDF).

The project is split into the following sub-projects:

- **Back-End**
  - **Common_Tools:** A utility-project, containing various reusable methods that are utilised by most parts of the solution.
  - **EDF__Core:** Includes the implementation of all developed Algorithms, as well as a number of Data-Structures and their relevant generation, mutation and evaluation methods. The Data-Structure for the Architectural Discipline can be found in this project.
  - **EDF__Designer_Modeling:** Includes the main algorithms and data-structures for Designer Modelling.
  - **EDF__Serialization:** A utility project that offers serialisation functionality for the data-structures included in the EDF__Core project.
  - **EDF__Visualization:** A utility project that offers visualisation functionality for the data-structures included in the EDF__Core project.
  - **EDF__Experiment_Runners:** A utility project that enables the easy set-up and execution of algorithmic experiments which can be used to evaluate the algorithms' performance.
- **Discipline-Specific Projects**
  - **EDF__Rhino_Architecture:** Implementation of the first part of the AI-Tool for Architecture, as a plugin for the Rhino/ Grasshopper design environment. This part includes the process of defining a Design Specification.
  - **EDF__Unreal_Backend:** Implementation of the second part of the AI-Tool for Architecture. This part includes the process of Interactive, Designer-Model-Driven QD for Architecture.
  - **EDF__Rhino_Structural:** Implementation of the AI-Tool for Structural Engineering, as a plugin for the Rhino/ Grasshopper design environment.
  - **EDF__Rhino_MEP:** Implementation of the AI-Tool for MEP Engineering, as a plugin for the Rhino/ Grasshopper design environment.

3. General purpose AI algorithms

The computational framework developed by the University of Malta (UM) for PrismArch includes a variety of algorithms as initially reported in [D2.2]. Those include implementations of existing approaches, such as variants of classic *Genetic Algorithms* and *MAP-Elites* [9], while others are novel algorithmic approaches that have been specifically developed for the needs of PrismArch, such as *FI-MAP-Elites*, *Interactive FI-MAP-Elites*, as well as their Designer-Model-based variants.

As reported in [D2.2], the framework includes a number of Benchmarks that have been used in order to test the implemented algorithms, as well as three realistic case-studies, that have been defined in cooperation with the Architecture, Engineering and Construction (AEC) industry partners, each of which corresponds to one of the involved disciplines: Architecture, Structural Engineering and MEP Engineering. The three case-studies are described in full detail in the next chapter.

Note that all algorithms are modular and they can apply to any Benchmark or Case Study (see also Deliverable D2.2).

3.1. Final algorithms for traversing the solution space

The following sections focus on the two main algorithmic contributions of the UM in PrismArch: *FI-MAP-Elites* and *Interactive FI-MAP-Elites*. The former is aimed mainly towards the engineering disciplines, where the fitness criteria are clearer, and the degree of subjectivity around a solution is smaller, while the latter is aimed towards the Architecture discipline, where the degree of subjectivity is larger, and thus the designer is directly involved in the generative process through interaction.

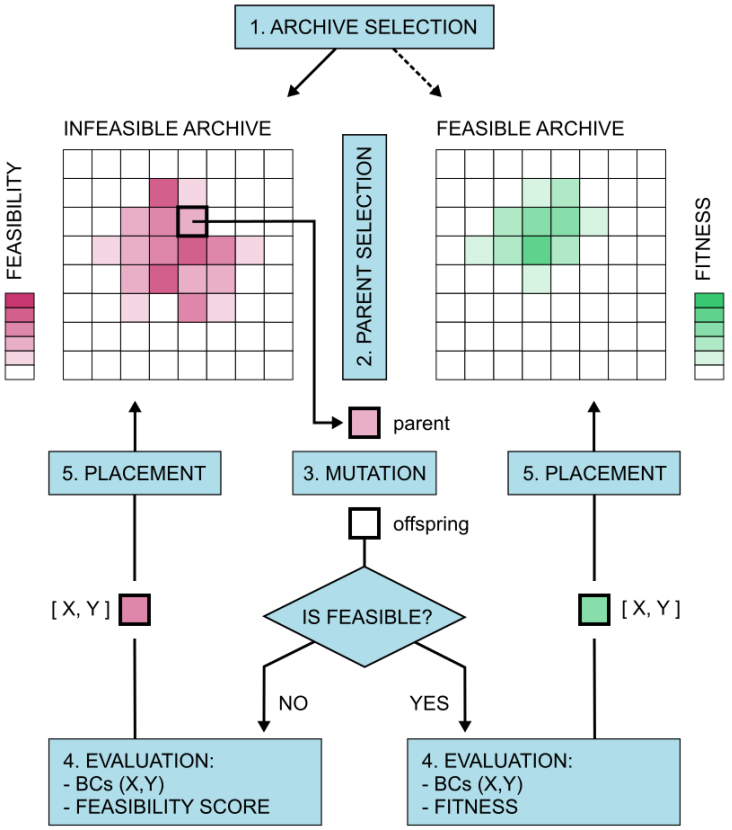
3.1.1. FI-MAP-Elites

Feasible-Infeasible Multidimensional Archives of Phenotypic Elites (*FI-MAP-Elites*) is a hybrid of the *MAP-Elites* [9] and *FI-2Pop GA* [10] algorithms. Its main goal is to combine the illumination capabilities of the former with the constraint-solving capabilities of the latter. It is strongly inspired by the *Constrained MAP-Elites* [11] algorithm but operates in a slightly different way which can be beneficial for reasons explained later on. The following paragraphs briefly describe all relevant algorithms and conclude with an in-depth description of *FI-MAP-Elites*.

MAP-Elites is an illumination algorithm which explores all areas of a behavioural space with a selection pressure towards locally higher fitness [26]. Its operation is based on a discretization of the behavioural space into cells of equal size, each of which can store a single individual whose Behavioural Characterizations (BCs) lie within that cell. In every step (evaluation), the algorithm randomly selects an individual from the archive, mutates it and evaluates the offspring's fitness and BCs. It then places the offspring back in the archive at its corresponding coordinates: if that cell is already occupied the fitter individual of the two survives in that cell. By repeating this operation, the algorithm's archive of solutions is gradually expanded (coverage) and optimised (fitness).

The *Feasible-Infeasible two-population genetic algorithm* (*FI-2Pop GA*) [10] handles constrained optimization problems by retaining two populations. The first one contains only feasible individuals and its selection pressure is the objective. The second one contains only infeasible individuals and its selection pressure is a feasibility score, calculated as the percentage of satisfied constraints. Individuals are selected and evolved from both populations, with offspring placed on the corresponding population based on their feasibility. The *FI-2Pop GA* has been extensively used for level generation [12, 13] as it is inherently a constrained optimization problem.

The *FI-MAP-Elites* algorithm operates by retaining two distinct archives of solutions, one for the feasible and one for the infeasible population. During its operation, the algorithm will select a random individual either from the feasible or the infeasible population (in alteration), mutate it, evaluate the offspring and place it in the proper archive, based on its feasibility. The replacement criterion for the feasible population is based on the objective function (fitness), while for the infeasible population it is based on the degree of feasibility. Through many repeated iterations of this process, the algorithm will gradually fill both archives with solutions that tend to approach their maximum fitness (objective function or degree of feasibility). After a number of iterations, the algorithm typically manages to provide a diverse set of fit and feasible solutions for the end - users.

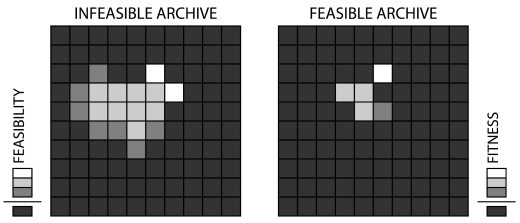


Functional diagram showcasing the operation of the FI-MAP-Elites algorithm.

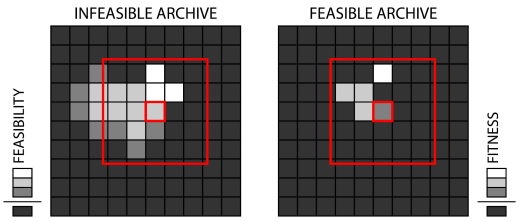
3.1.2. Interactive FI-MAP-Elites

The *Interactive FI-MAP-Elites* algorithm extends the operation of *FI-MAP-Elites* in a way that allows the user's subjective criteria to directly affect its operation. Its operation is summarised in the following steps:

- **Algorithm Initialization:**
  The following three steps are executed once, during the algorithm's initialization process.
  - **Initial Population Generation:**
    Using random initialization, the algorithm generates 100 individuals and places them in the feasible or infeasible archives, accordingly.
  - **Archive Expansion:**
    The underlying *FI-MAP-Elites* algorithm operates (as explained in the previous section) until at least 5% of the feasible archive's cells are populated.

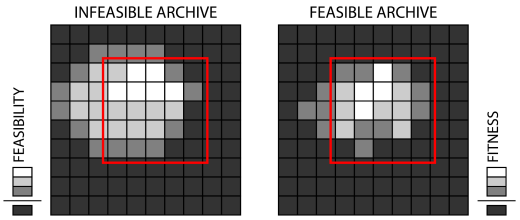


  - **Initial Selection Window Placement:**
    An occupied cell of the feasible archive is selected, at random, and a selection window is centred at that cell. That selection window is a special feature of the *Interactive FI-MAP-Elites* that restricts the algorithm's parent selection in a specific subregion of the feasible & infeasible archives.

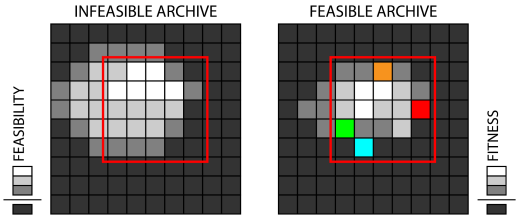


- **Interactive Mode:**
  Upon entering the interactive mode, the algorithm executes the following steps indefinitely, until the Designer decides to terminate the process. During those steps, the designer's input is indirectly guiding the algorithm's resource allocation towards specific regions of the Behavioural Space.
  - **1. Windowed Archive Extension:**
    Initially, the underlying *FI-MAP-Elites* operates for 1000 iterations, with the selection window restricting the parent selection process. I.e. parent individuals can only be selected only from within that window, in both the

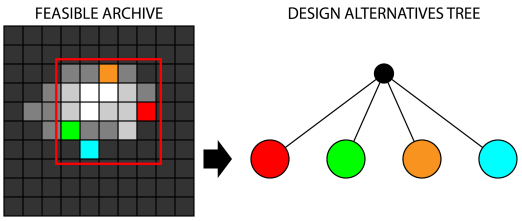
feasible and infeasible archives.



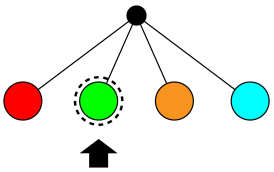
- **2. Design Alternatives Selection:**
  Four feasible individuals are selected from within the selection window, to be presented to the designer, as alternatives.



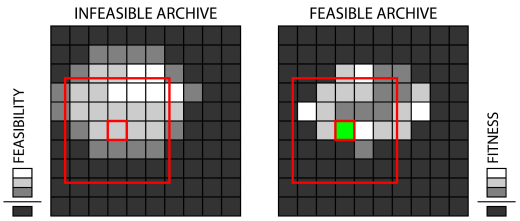
- **3. Design Alternatives Presentation:**
  The four selected individuals are presented to the designer.



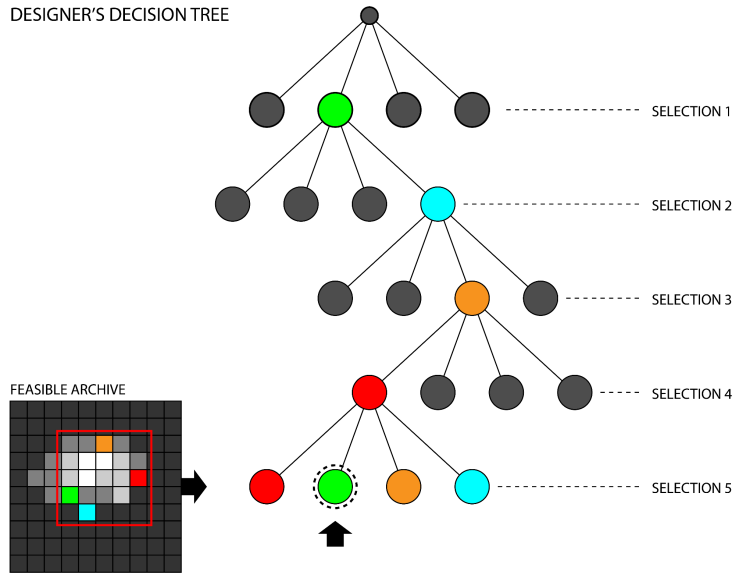
- **4. Designer Input:**
  At this stage, the designer may select one of the presented solutions and let the algorithm's operation proceed. Alternatively, they may choose to terminate the algorithmic process.



- **5. Selection Window Placement:**
  The selection window is moved to a new position, namely being centred at the cell of the last selected solution (by the designer). Then, the process continues at the 1st step of the interactive mode.



As the algorithm operates, the decision-tree is stored in memory for future use / analysis purposes. This tree stores all options that the designer was presented with, during the interactive evolution process.



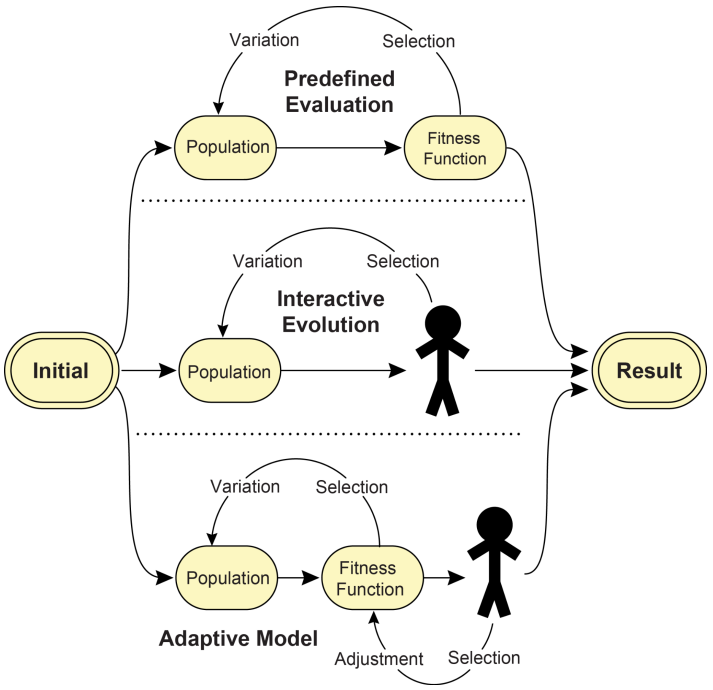
3.2. Final Algorithms for Designer Modelling

We approach Designer Modelling (DM) [7,8] in the context of a closed-loop system, one that integrates content generation through (interactive) Quality Diversity with DM, resulting in a designer-tailored generative system that adjusts to the designers' subjective preferences.

We follow the Adaptive Model paradigm of Search-Based content generation (illustrated in the following figure) introduced by Liapis et al. [1], where the user's subjective criteria adapt the evaluation function (designer model) and then the generated content is optimised based on the personalised evaluation function.

We employ a Preference Learning method to update the Designer Models, following the approach of "Rank-based Interactive Evolution (RIE)" introduced by Liapis et al. in [2]. In order to simplify the users' involvement in the data-generation process, we omit the ranking step and directly capture preference-pairs[1] either indirectly, during the user's continuous interaction with the system (see section of Architecture) or directly, in between algorithmic sessions (see Structural Engineering and MEP Engineering).

[1] A preference-pair is a pair of feature vectors (for example a set of numeric properties of a design), one of which has been evaluated as better than the other, by a user. The preference-pair does not hold any other information about the feature-vectors, such as their exact fitness or ranking.



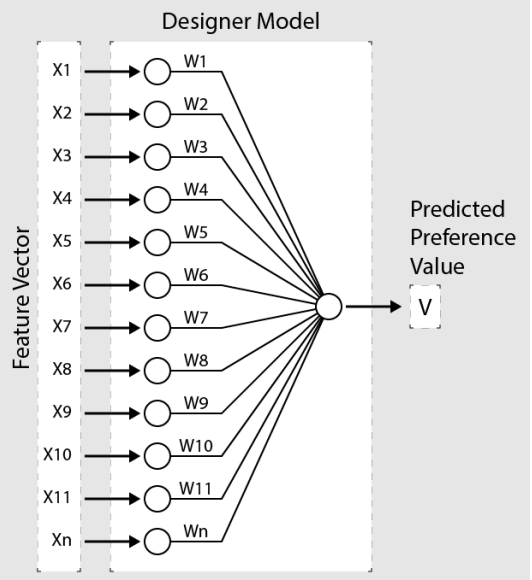
Functional diagram showcasing three approaches to search-based procedural content generation: 1) Predefined Evaluation, 2) Interactive Evolution, where the user replaces the evaluation method and (3) *Adaptive Model* where the user's input adjusts the fitness function which, in turn, ranks content by a personalized measure of quality. Source: [1]

3.2.1. Designer Modelling Implementation

The Designer Modelling implementation that we developed is based on a Linear Model (Single-Layer-Perceptron), similar to the work of Liapis et al. [2]. The model's inputs include all of the measurable properties of a given solution (feature vector), while the model's output is a real number, which is essentially a preference score for that feature vector. The model can either be used to compare two solutions and predict the designer's preference, or used as a scalar in the context of QD, essentially replacing the fitness function or one of the Behavioural Characteristics. A visual representation of the selected architecture is shown in the following figure, while an example of a serialised Designer Model is shown in the following table.

Before starting the model-training process, a data-set is needed. This dataset must be in the form of preference-pairs. Each one of those pairs includes a preferred feature-vector and a non-preferred feature-vector.

The modelling task at hand can be formalised as finding a function F(x) that satisfies: $\forall(x_i, x_j), r_{x_i} > r_{x_j} : F(x_i) > F(x_j)$, where x_j is an object and r_{x_j} is its corresponding rank. As mentioned above, we define our function as a weighted sum of a solution's feature-vector which is similar to a Single Layer Perceptron [3] employing a linear activation function. Then we can train those weights by combining a standard gradient descent algorithm [4] with a cost function defined in terms of pairwise preferences [5].



Visualisation of an abstract Designer Model, which predicts the designer's preference based on an N-Dimensional Feature Vector.

```
{
  "weights": [
    0.10252656838974294,
    -0.5571194065535066,
    0.9893482709253898,
    0.4951349941525305,
    0.038948392979311075,
    0.37489226151951227,
    -0.6305013315894181,
    0.038948392979311075,
    -0.659382659473292047
  ]
}
```

An example of a serialised, 8-dimensional Designer Model.

3.2.2. DM-Based FI-MAP-Elites

The *DM-Based FI-MAP-Elites* is an extension of the *FI-MAP-Elites algorithm*. Namely, it personalises its operation by using a Designer Model as either its fitness function or one of its Behavioural Characterizations. The Designer Model, which is initially generated as equal preferences towards all solutions, is updated in between sessions in the following way:

- As soon as the algorithm's operation is finished, the Designer is asked to select the solutions that they prefer, among the ones included in the feasible archive.
- Based on the designer's selections, a data-set of pairwise preferences is generated. Namely, every solution that was selected is treated as being better than every

solution that was not selected. For example, if the designer selects 5 solutions out of 10 available ones, then a dataset of 25 pairwise preferences will be generated (5 selected solutions are better than the 5 unselected ones).

- The generated dataset of pairwise preferences is used to update the Designer Model's weights, as described in Section 3.2.1.

The updated Designer Model can either be used directly for running a new session, or stored for later use.

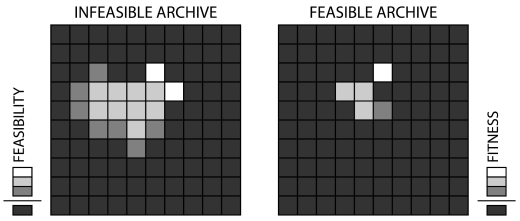
In our computational framework, the *DM-Based FI-MAP-Elites* algorithm is used for the Structural Engineering and MEP Engineering principles. More details about its specific application to those disciplines can be found in the relevant sections.

3.2.3. DM-Based Interactive FI-MAP-Elites

The *Designer-Model-Based Interactive FI-MAP-Elites* algorithm is an extension of the *Interactive FI-MAP-Elites* algorithm. Namely, it enhances the algorithm's operation via a Designer Model that is being trained on-line, capturing the subjective preferences of the designer. The way that this Designer Model is used during an interactive session is an attempt to capture and exploit the designer's personal preferences, as they are expressed through their design-choices.

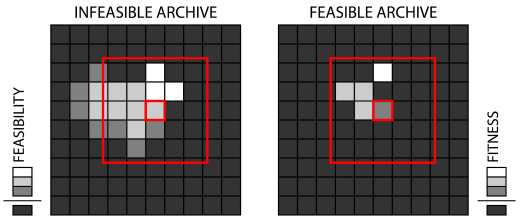
The algorithm's operation is almost identical to that of *Interactive FI-MAP-Elites*, except for a few extra steps that supersede the designer's input.

- **Algorithm Initialization:** The following three steps are executed once, during the algorithm's initialization process.
    - **Designer Model Initialization:** An initial designer model can either be loaded from disk, or generated at random. This Designer Model will be used as the fitness function of the feasible population.
    - **Initial Population Generation:** Using random initialization, the algorithm generates 100 individuals, from scratch, and places them in the feasible or infeasible archives, accordingly.
    - **Archive Expansion:** The underlying *FI-MAP-Elites* algorithm operates (as explained in the previous section) until at least 5% of the feasible archive's cells are populated.

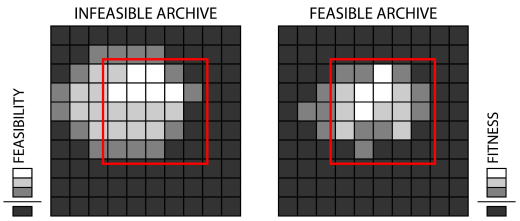


    - **Initial Selection Window Placement:** An occupied cell of the feasible archive is selected, at random, and the selection window is centred at that cell. The Selection Window is a special feature of *Interactive FI-MAP-Elites* that can constrain the algorithm's parent selection in a specific subregion of

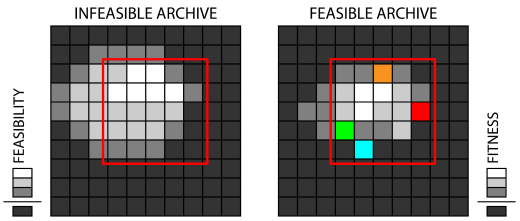
the (feasible & infeasible) archive.



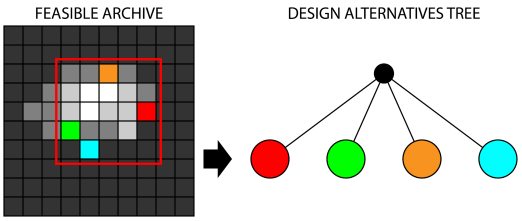
- **Interactive Mode:** Upon entering the interactive mode, the algorithm executes the following steps indefinitely, until the Designer decides to terminate the process. During those steps, the designer's input is essentially guiding the algorithm's resource allocation towards the regions of the Behavioural Space that the designer prefers to explore.
    - **1. Windowed Archive Extension:** The underlying *FI-MAP-Elites* operates for 1000 iterations, with the selection window constraining the parent selection process. I.e. parent individuals can only be selected from within that window, in both the feasible and infeasible archives.



    - **2. Design Alternatives Selection:** Four feasible individuals are selected from within the selection window, to be presented to the designer, as alternatives.

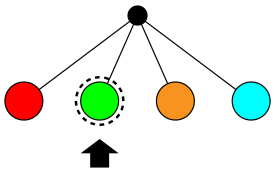


    - **3. Design Alternatives Presentation:** The four selected individuals are presented to the designer.

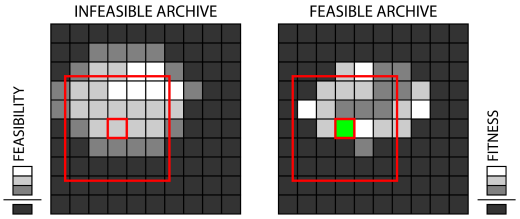


    - **4. Designer Input:** At this stage, the designer may choose to terminate the process. Otherwise, they may select one of the presented solutions and let

the algorithm's operation proceed.



- **5. Designer Model Update:** The designer's selection generates a small dataset of three pairwise comparisons. The selected individual is treated as being better than the three unselected ones. This small dataset is used to update the existing Designer Model.
- **6. Feasible State re-evaluation:** Based on the updated designer model, all individuals that are currently stored in the feasible archive are re-evaluated (fitness).
- **7. Selection Window Placement:** The selection window is moved, being centred on the cell of the solution that the designer selected. Then, the process continues at the 1st step of the interactive mode.



4. Integration of Algorithms into an AI assisted tool, per discipline

4.1. AI-Tool for Architecture:

The AI-Tool for Architecture focuses on the generation of architectural layouts, based on a user-specified set of topological and other constraints. This approach coincides with the initial phases of architectural design, during which architectural design values the discovery of diverse designs within the set constraints. Thus the chosen QD algorithms are highly suited to this problem and can help the architects to explore a variety of potential layouts.

This section describes how the QD algorithms are integrated with the PrismArch platform. That is: how the user interacts with the algorithm, in order to generate designs, through the VR-based UI, within the PrismArch platform. Based on these prerequisites, an interactive evolution system has been developed. This system is based on the *Interactive FI-MAP-Elites* algorithm, whose operation requires the intervention of an external operator. From a technical perspective, this external operator can either be an artificial decision-making agent which can evaluate a set of solutions according to its own fitness function, or a computational model of the designer.

4.1.1. Final Parametric Space of Design

4.1.1.1. Data-Structure

The architectural layouts generated by our approach form different regions that we identify as “space units". Those can refer to clearly defined interior spaces, such as rooms of any type, as well as types of exterior spaces such as verandas or gardens that are in direct contact with the plan’s boundary. Different space units are separated by walls, and two adjacent ones can be connected with a door. Importantly, the generator produces 2D layouts that adhere to a high-level specification. This dual representation, from the high-level Design Specification to the geometric Design Implementation is described below.

Design Specification

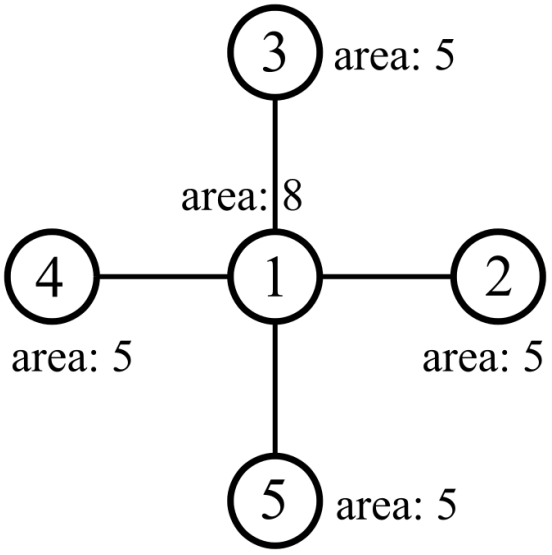
A Design Specification (DS) is an abstract description of the architectural layout that is to be generated. It contains an undirected graph whose vertices represent a set of space-units and whose edges represent a direct connection between a pair of rooms through a door. Each vertex (room) is also assigned with an area that this room should occupy. A DS is a critical part of the design constraints and is utilised in various parts of our algorithmic approach, including the initialization, mutation and evaluation methods.
An example of a Design Specification is shown in Fig. 1.a.

Design Implementation

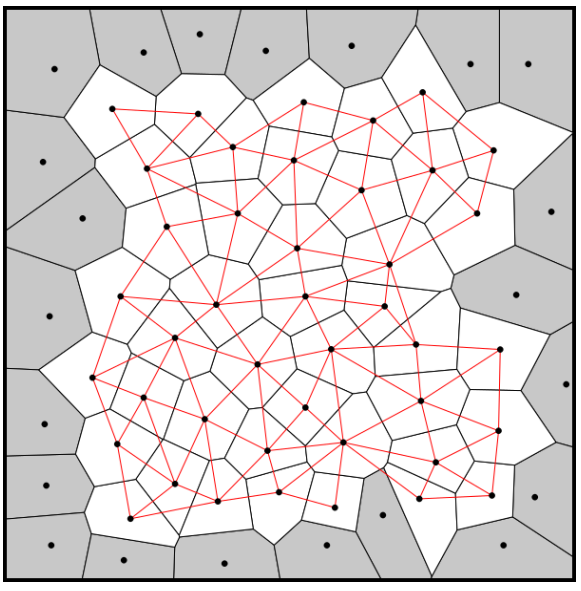
A Design Implementation (DI) is the geometry that matches the provided DS, produced by the generator. It includes the specific geometric boundaries of each space unit, as well as the exact placement of doors and other openings. A generated DI can be feasible or

infeasible, based on whether it satisfies the constraints posed by the DS and a few more, described in Sec. 4.1.1.4.
In our proposed algorithmic approach, a DI consists of three hierarchically dependent layers of information (shown in Fig. 1):

- **Layer 1 (L1):**
  L1 describes the Voronoi tessellation of a 2D rectangle, whose cells become the building blocks for the shape of rooms and levels (as shown in Fig. 1c). Square and Hex grids are simply special cases of the Voronoi diagram, where points have been pre-arranged accordingly (see Fig. 1.b). The L1 genotype consists only of the points' coordinates, while the calculated L1 phenotype includes the generated Voronoi diagram and the resulting adjacency between active cells. The L1 genotype can be mutated by changing coordinates of existing points, resulting in a different Voronoi diagram. Adding or removing points has been disabled in this study, for the sake of simplicity.
- **Layer 2 (L2):**
  L2 describes the rooms' placement on the 2D plane and their exact shape (as shown in Fig. 1.c). The L2 genotype is a dictionary that assigns the use of specific rooms of the DS to specific cells of the Voronoi diagram. The L2 phenotype consists of the rooms' boundaries. Those are calculated by merging the regions of cells that belong to the same room.
- **Layer 3 (L3):**
  L3 describes the walls and doors, as well as the resulting connectivity graph of the interior space (as shown in Fig. 1.d). The L3 genotype represents the doors' locations as a pair of neighbouring cells. The exact coordinates of each door are part of the L3 phenotype and lie in the mid-point of the line-segment shared between the two cells. The L3 phenotype also includes a computed connectivity graph that accounts for boundaries (walls) between rooms and between a room and the exterior, and doors that allow connections between adjacent rooms.



(a) Example of a Design Specification with


(b) L1

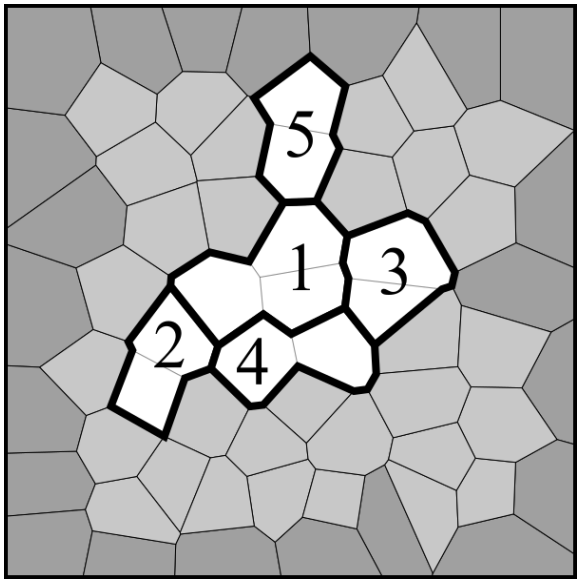
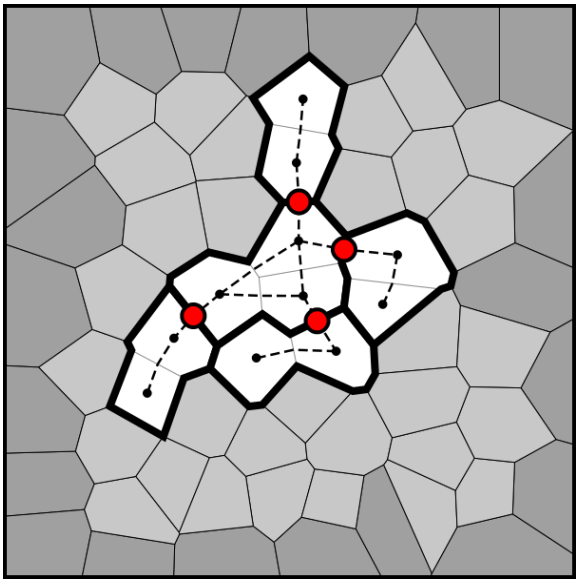
5 space units and 4 connections.	
	
(c) L2	(d) L3

Figure 1: A Design Specification and the three layers of information (L1, L2, L3) of a feasible Design Implementation.

4.1.1.2. Random Initialization

Random initialization is a critical part of the algorithm's operation, as it generates the initial population which will be extended and optimised through mutation operators described in the next section. The random initialization process consists of a sequence of semi-stochastic operations that attempt to generate a feasible DI based on the DS, as described in the following steps:

- **Tessellation Definition (L1):** First, a set of points that lie within the specified bounding rectangle is assigned, and the resulting tessellation and connectivity graph is calculated. The points' coordinates can be random, (forming a generic Voronoi diagram), or structured to form a square or hex grid.
- **Space Units' Placement (L2):** The following steps are repeated until all rooms have been placed on the map. First, the algorithm selects a room in the DS that is not yet in the DI, prioritising rooms with more connections. Then the room is assigned to a single cell that is near already placed rooms that it connects to (according to the DS). If no such cell exists (e.g. for the first room), a free cell will be selected at random. Starting from this first cell, the room's area is expanded using free adjacent cells chosen randomly, until the room's area falls within the specified error margin.
- **Openings Placement (L3):** Finally, the algorithm finds all potential locations for door placement per connection in the DS and places a door at one of them, at random. Proper locations include the lines of the shared boundary between connected rooms, whose length is at least 0.5 units. This threshold is adjustable depending on the size of the door meshes and the player's avatar.

4.1.1.3. Mutation

Mutation occurs in two stages: First, the destruction method randomly alters parts of the layout without taking constraints into account. Then, the repair method attempts to bring the layout back to a feasible state. The repair operations are semi-stochastic and do not guarantee feasibility, but greatly increase the chances for it.

Destruction

The destruction method selects between 1 and 3 operations from the following list (chosen at random) and applies them on the selected DI, generating an altered, usually infeasible, offspring. The destruction operators target specific DI layers, noted in parentheses. Note that L1 destruction operations are only applied on Voronoi grids.

1. **“Move all Voronoi points”**
    a. Implementation name: MM__Destruction__L1__Move_All_Voronoi_Points
    b. Description: Alters the underlying Voronoi tessellation (Level 1) by moving all generator points at random, within a specified range relative to their current position.
2. **“Move some Voronoi points”**
    a. Implementation name: MM__Destruction__L1__Move_Voronoi_Points
    b. Description: Alters the underlying Voronoi tessellation (Level 1) by moving a percentage of the generator points at random, within a specified range relative to their current position.
3. **“Blow-up a space unit”**
    a. Implementation name: MM__Destruction__L2__Blow_Up_Space_Unit
    b. Description: Expands a space unit in such a way that it occupies all of its previously surrounding cells, including ones that may have been occupied by other space units.
4. **“Blow-up a space unit safely”**
    a. Implementation name: MM__Destruction__L2__Blow_Up_Space_Unit__Safe
    b. Description: Expands a space unit in such a way that it occupies all of its previously surrounding cells, excluding ones that are occupied by other space units.
5. **“Delete a space unit”**
    a. Implementation name: MM__Destruction__L2__Delete_Space_Unit
    b. Description: Completely removes a space unit from the DI.
6. **“Erode a space unit safely”**
    a. Implementation name: MM__Destruction__L2__Erode_Space_Unit__Safe
    b. Description: Randomly selects a space unit and removes as many cells from it as possible, without breaking its coherence.
7. **“Delete random openings”**
    a. Implementation name: MM__Destruction__L3__Delete_Random_Openings
    b. Description: Randomly selects and deletes a percentage of openings.

Repair

The repair method attempts to bring a DI back to a feasible state by sequentially applying all operations of the following list (denoting the DI layer on which they operate).

1. **“Add missing space units”**
    - Implementation name: MM__Repair__L2__Add_Missing_Space_Units
    - Description: If there are any space units in the DS that do not exist in the DI, then this operation places the missing units in the DI, in the following steps: (1) Select a missing space unit, prioritising the ones of higher order (more connections). (2) Place the first cell of this space unit on the plan, so that it is adjacent to as many of its specified connections as possible. (3) Expand the space unit, until its area is within an error margin of the prescribed area.
2. **“Repair connections”**
    - Implementation name: MM__Repair__L2__Fix_Connections
    - Description: If there are any prescribed connections in the DS that are not included in the DI, this operation will attempt to implement them. For every pair of space units that must be connected, this operation will find the shortest path between them and give half of its cells to each space unit.
3. **“Repair space units’ coherence”**
    - Implementation name: MM__Repair__L2__Fix_Space_Units_Coherence
    - Description: If any space unit is incoherent (i.e. exists in two or more disconnected regions), then this method deletes all but one of those regions, at random.
4. **“Repair space units’ area”**
    - Implantation name: MM__Repair__L2__Space_Unit_Area__Increase_Decrease_Threshold
    - Description: If any space unit’s area is smaller than or greater than an acceptable range, this operation will attempt to bring it back to an acceptable size.
5. **“Repair openings”**
    - Implementation name: MM__Repair__L3__Openings
    - Description: This operation will remove any improperly placed openings and then attempt to properly place the missing ones.

4.1.1.4. Constraints

The distinction between Feasible and Infeasible solutions is a crucial part of the algorithmic operation. A set of Hard Constraints (boolean functions) have been implemented for that purpose, each of which tests a critical aspect of a Design Implementation. In order for an individual to be considered as feasible it must satisfy all of these hard constraints.
The following list enumerates and explains all hard constraints that a Design Implementation must satisfy in order to be considered as feasible. Constraints 1 and 2 are low-level, technical constraints implemented after experimentation with the algorithm’s operation, while constraints 3 to 8 are higher-level constraints that check whether the Design Implementation satisfies the Design Specification.

1. **“Voronoi Graph is Connected”**
    - Implementation name: CEM__L1__Voronoi_Connected
    - Description: Returns true if the connectivity graph of the underlying Voronoi tessellation is connected.
2. **“Minimum Percent Active Cells”**
    - Implementation name: CEM__L1__Voronoi_Min_Percent_Active_Cells

    - Description: Returns true if at least 50% of the Voronoi cells are active (i.e. do not touch the border.
3. **“All Prescribed Connections Exist”**
    - Implementation name: CEM__L2__Prescribed_Connections_Exist
    - Description: Returns true if all prescribed connections of the DS are implemented in the DI.
4. **“All space units are coherent”**
    - Implementation name: CEM__L2__Space_Units_Are_Coherent
    - Description: Returns true if all space units exist and are coherent. A coherent space unit is one whose cells form a single region.
5. **“Space units’ areas are within an error margin”**
    - Implementation name: CEM__L2__Space_Units_Areas_Within_Margin
    - Description: Returns true if the areas of all space units are within their specified acceptable error margin.
6. **“All space units exist”**
    - Implementation name: CEM__L2__Space_Units_Exist
    - Description: Returns true if all prescribed space units exist in the DI.
7. **“No Narrow Passages”**
    - Implementation name: CEM__L3__Not_Narrow_Passages
    - Description: Returns true if there are no narrow passages in the plan. The minimum width for a passage can be set by the designer and is typically set to 0.8 metres.
8. **“Prescribed Openings Exist”**
    - Implementation name: CM__L3__Prescribed_Openings
    - Description: Returns true if all prescribed openings exist properly in the DI.

4.1.1.5. Feasibility Score

If any of the constraints of the DI is not satisfied, the DI is infeasible and its feasibility score is calculated. The feasibility score is a way of calculating the proximity of a DI to being feasible, and is used to determine elites in the infeasible population. In *FI-2Pop GA* [10], this proximity is calculated as the percentage of satisfied constraints. Given the complexity of the problem at hand, however, and based on preliminary tests, we further provide a gradient for each constraint based on partial satisfaction in order to assist the search for feasible DIs. The following list describes the conversion of the constraints to fine-grained scores that capture the degree of satisfaction for each constraint. The feasibility score is calculated as the average of all those scores. Note that if a constraint is satisfied, its score is 1.

1. **Connected Graph Score (L1):**
    - Implementation name: EM__CEM__L1__Voronoi_Connected
    - Description: Is calculated as $S = 1/n$ where n is the number of islands found in the Voronoi graph.
2. **Active Cells Score (L1):**
    - Implementation name: EM__CEM__L1__Voronoi_Min_Percent_Active_Cells
    - Description: Returns 1 if more than 50% of Voronoi cells are active. Otherwise it returns a value in the range of [0…1] for percentage of active cells in the range of 0% to 50%.
3. **Prescribed Connections Score:**

- Implementation name: EM__CEM__L2__Prescribed_Connections_Exist
- Description: Is calculated as $S = x/n$ where x is the number of connections that exist in the DI and n is the total number of connections that are prescribed in the DS.

4. **Space Units Coherence Score:**
   - Implementation name: EM__CEM__L2__Space_Units_Are_Coherent
   - Description: Is calculated as $S = x/n$ where x is the number of space units that exist and are coherent and n is the total number of prescribed space units.
5. **Space Units Proper Area Score:**
   - Implementation name: EM__CEM__L2__Space_Units_Areas_Within_Margin
   - Description: Is calculated as the average Area Score of all space units. If a space-unit's area-error is smaller than a specified value, then its area score is 1. Otherwise, its Area Score is calculated as $S = x/x_{min}$, where x is the current fractional similarity between the space unit and its prescribed area, while x_{min} is the minimum permitted fractional similarity between the space unit's area and its prescribed area.
6. **Space Units Existence Score:**
   - Implementation name: EM__CEM__L2__Space_Units_Exist
   - Description: Is calculated as $S = x/n$ where x is the number of space units that exist in the DI and n is the total number of space units that are prescribed in the DS.
7. **Not Narrow Passages Score:**
   - Implementation name: EM__CEM__L3__Not_Narrow_Passages
   - Description: Is calculated as the average not-narrow-score of all bridge-edges. Bridge edges are connections between cells that do not have any other common neighbours. The not-narrow-score of a bridge edge is calculated based on the length (L) of the shared boundary between the connected cells. If L is greater than the smallest permitted value, then the not-narrow-score of that edges is 1. Otherwise, it is calculated as $S = L/L_{min}$.
8. **Prescribed Openings Score:**
   - Implementation Name: EM__CM__L3__Prescribed_Openings
   - Description: Is calculated as the percentage of correctly placed openings in the Design Implementation, according to the constraints imposed by the Design Specification.

4.1.1.6. Fitness

Fitness is implemented as a measure of the average area precision of all rooms. For room i its area precision is $P_A(i) = 1 - E_A(i)$, where $E_A(i)$ is the room's area error. A room's area error is calculated as $E_A = 1 - \frac{min(A(i), A_p(i))}{max(A(i), A_p(i))}$, where A_i is the area of room i and $A_p(i)$ is the prescribed area of room i.

4.1.1.7. Diversity

After extended discussions with the architectural partners, it was deduced that in the field of architecture it is not easy to pin-point specific measures that objectively characterise a design. In other words, the architects' criteria are too complex to be quantified. Taking this as a given, we implemented a number of abstract Behavioural Characterizations (explained in the list below) that characterise a design's geometry and / or topology, borrowing ideas and implementation by relevant literature. The idea is that none of these measures is a sufficient qualitative description of a solution, but that perhaps a computational Designer Model can sufficiently approximate a designer's preference, to a satisfactory degree, as a function of all these measures. The following list is by no means exhaustive, however other BCs can easily be added in the future, to extend the framework's functionality.

1. **"Plan compactness"**
    - Implementation name: EM__L2__Compactness__Plan
    - Description: Is calculated as $C = 2\pi A/\Pi$ where A is the plan's total area and Π is the plan's perimeter.
2. **"Average compactness of space units"**
    - Implementation name: EM__L2__Compactness__Space_Units
    - Description: Is calculated as the average compactness of all space units in the plan. The compactness of each space unit is calculated as $C = 2\pi A/\Pi$, where A is the room's area and Π is the room's perimeter.
3. **"Average compactness of used cells"**
    - Implementation name: EM__L2__Compactness__Used_Cells
    - Description: Returns the average compactness of all used cells (i.e. the cells of the Voronoi tessellation that are part of any space unit). The compactness of each cell is calculated as $C = 2\pi A/\Pi$, where A is the cell's area and Π is the cell's perimeter.
4. **"Angles' orthogonality - average"**
    - Implementation name: EM__L2__Angles_Orthogonality
    - Description: Returns the average orthogonality of all angles between connected lines in the plan. Overlapping lines are considered as one. An angle's orthogonality is calculated as $F(a) = -1 + (4a/\pi)\ mod\ 2$ where α is the angle in radians.
5. **"Non-Acute Angles Score"**
    - Implementation name: EM__L2__Angles_Non_Acute
    - Description: Returns the average non-acute score of all angles between connected lines in the plan. An angle's non-acute score is $S = 1$ when $a \geq \pi/2$ and calculated as $F(a) = 2a/\pi$ when $a < \pi/2$.
6. **"Lines' orthogonality - average"**
    - Implementation name: EM__L2__Lines_Orthogonality
    - Description: Returns the average orthogonality of all line-segments that are part of any space unit's boundary. Overlapping line segments are treated as one. A line-segment's orthogonality is calculated as $S = -1 + (4a/\pi)$ where a is the line-segment's angle, in radians.
7. **"Lines' orthogonality - weighted average"**
    - Implementation name: EM__L2__Lines_Orthogonality_Weighted
    - Description: Returns the weighted average orthogonality of all line-segments that are part of any space unit's boundary. Overlapping line segments are

treated as one. The average, in this case, is weighted based on the line-segments' length, thus prioritising the orthogonality of longer line-segments. A line-segment's orthogonality is calculated as $S = -1 + (4a/\pi)$ where a is the line-segment's angle, in radians.

8. **"Space units' area precision"**
    - Implementation name: EM__L2__Space_Units_Area_Precision
    - Description: Returns the average area precision of all space units. A space unit's area precision is calculated as $P = a/A$ for $a \leq A$, or $P = A/a$ otherwise, where a is the space unit's area in the DI and A is its prescribed area in the DS.
9. **"Average distance between connection doors"**
    - Implementation name: EM__L3__Avg_Dist_Connection_Doors_Normalized
    - Description: Returns the average distance between connection doors, based on the solution's interior connectivity graph.
10. **"Average distance between entrances and connection doors"**
    - Implementation name: EM__L3__Avg_Dist_Entrances_Connection_Doors_Normalized
    - Description: Returns the average distance between entrances and connection doors, based on the interior connectivity graph.
11. **"Average distance between windows"**
    - Implementation name: EM__L3__Avg_Dist_Windows_Normalized
    - Description: Returns the average distance between windows, based on the interior connectivity graph.
12. **"Circulation area percentage"**
    - Implementation name: EM__L3__Percent_Circulation_Area
    - Description: Returns the approximate percentage of interior space that is used for circulation. The average circulation area is calculated based on the shortest paths of movement between all available openings (doors & windows). Those paths are treated as corridors of movement with a width of 1m and summed up to form the approximate circulation area.

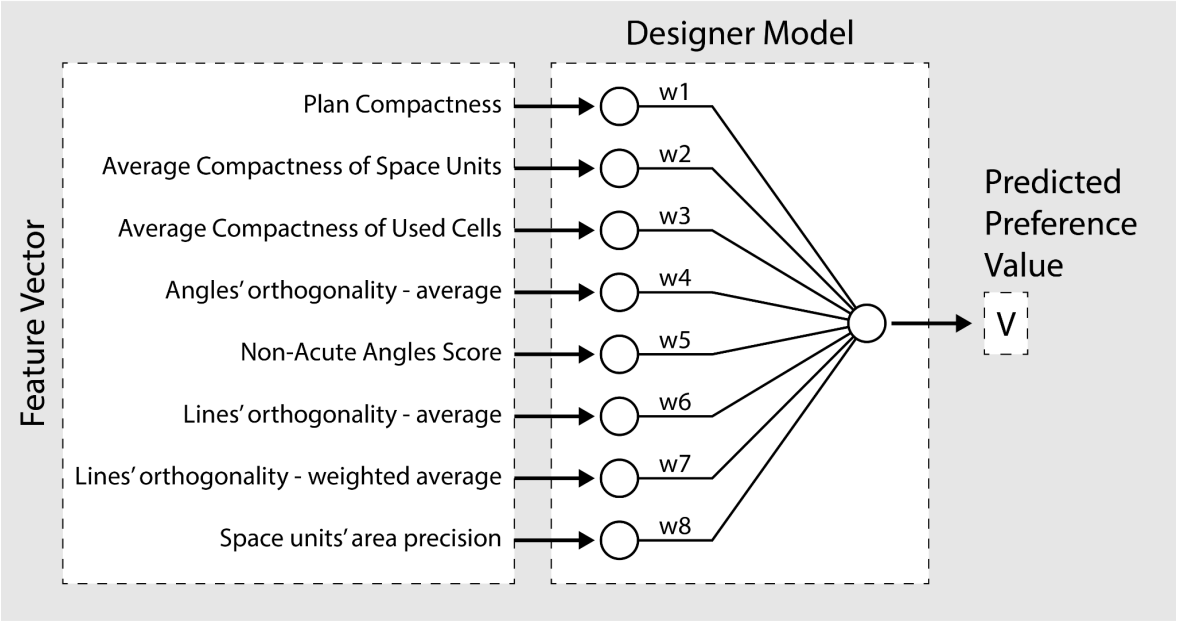
4.1.1.8. Designer Model

For the architectural discipline, the Designer Model's input is an 8-dimensional feature-vector, that characterises an architectural solution, while its output is a single value that approximates the designer's preference for that specific solution. The model's architecture is illustrated in the following figure, while a serialised example is shown in the following table. Importantly, the model's feature space consists of the following 8 Behavioural Characterizations, the detailed description and definition of which can be found in the previous section:

1. Plan Compactness
2. Average Compactness of Space Units
3. Average Compactness of Used Cells
4. Angles' Orthogonality - Average
5. Non-Acute Angles Score
6. Lines' Orthogonality - Average
7. Lines' Orthogonality - Weighted Average

8. Space Units' Area Precision

Designer Model for the Architectural Disipline



A visualisation of the Designer Model for the Architectural Discipline. The model's input is an 8-dimensional feature vector that characterizes an architectural layout, while the model's output is a single, numeric value that attempts to capture the subjective preference of the designer towards that solution.


```
{
  "weights": [
    0.10252656838974294,
    -0.5571194065535066,
    0.9893482709253898,
    0.4951349941525305,
    0.038948392979311075,
    0.37489226151951227,
    -0.6305013315894181,
    0.038948392979311075,
    0.4951349941525305
  ]
}
```

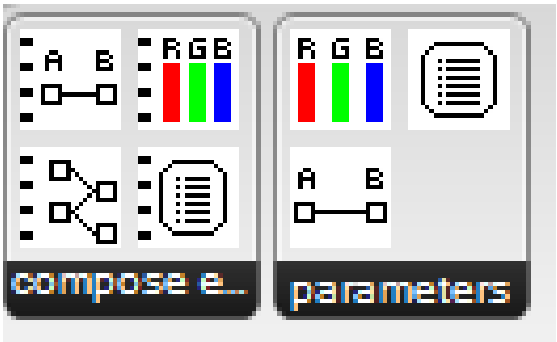

An example of a serialised Designer Model for Architecture. The model is represented as a list of 8 numbers, each of which corresponds to one of the Model's weights.

4.1.2. Interactive, DM-based QD for Architecture

4.1.2.1. Defining a Design Specification

The following sections describe how a Design Specification can be defined, using a set of custom tools that have been developed as plugins in the Rhino / Grasshopper environment. The following custom components have been developed, all of which can be found under the “PrismArch - Architecture” tab of the grasshopper environment:

- **GH - Components**
  - **Compose connection:** facilitates the definition of a Connection
  - **Compose space unit:** facilitates the definition of a Space Unit
  - **Compose rgb colour:** facilitates the definition of an RGB colour that can be used as the colour of a space unit.
  - **Compose design specification:** facilitates the definition of a Design Specification
- **GH - Parameters**
  - **Connection:** can hold a single Connection or al list of Connections
  - **Space Unit:** can hold a single Space Unit or a list of Space Units
  - **RGB Colour:** can hold a single RGB colour or a list of RGB Colours



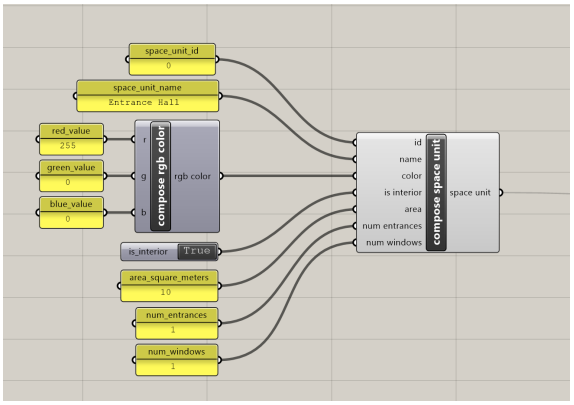
A close-up view of the custom components that can be found in the “PrismArch - Architecture” tab of the grasshopper environment.

Defining Space Units

The user can define a space unit, using the following parameters:

- **ID:** A unique identifier for this space unit
- **Name:** This space unit’s name
- **Colour:** This space unit’s colour
- **Is_Interior:** A boolean parameter specifying whether this space unit is an interior or exterior space.
- **Area:** The desired area of this space unit (in metres).
- **Num Entrances:** The desired number of entrances of this space unit (connecting it to the outer region).
- **Num Windows:** The desired number of windows of this space unit.

The following screenshot showcases how those parameters can be set in the Rhino / Grasshopper environment, using the “compose_space_unit” component.

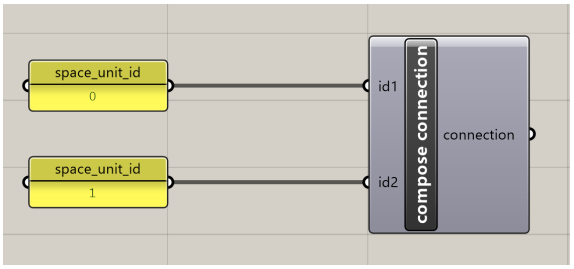
	Close-up view of the definition of a Space Unit, in the grasshopper script.

Defining Connections

The user can define a desired connection between two space units, by defining the following parameters (the ids' order does not matter).

- ID 1: the unique identifier of the first space unit
- ID 2: the unique identifier of the second space unit

The following screenshot showcases how a connection can be defined in the Rhino / Grasshopper environment.

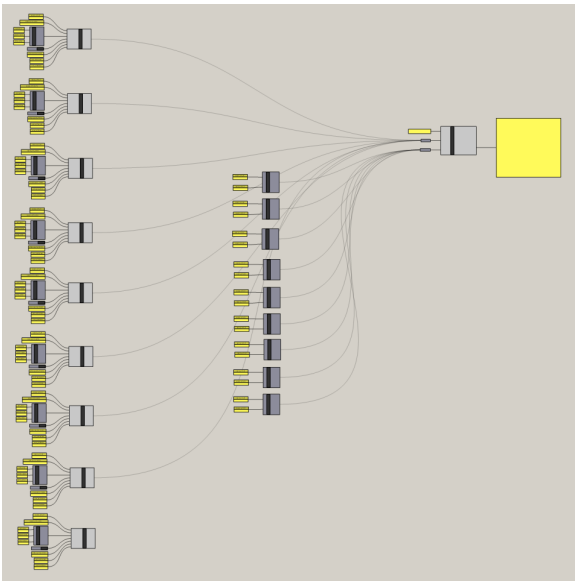
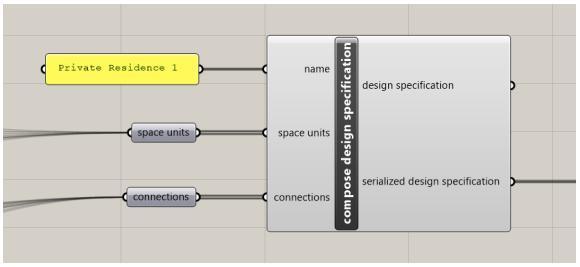
	Close-up view of the definition of a Connection, in the grasshopper script.

Composing a Design Specification

To compose a design specification, the designer can use the "compose design specification" grasshopper component. The definition requires the following input parameters:

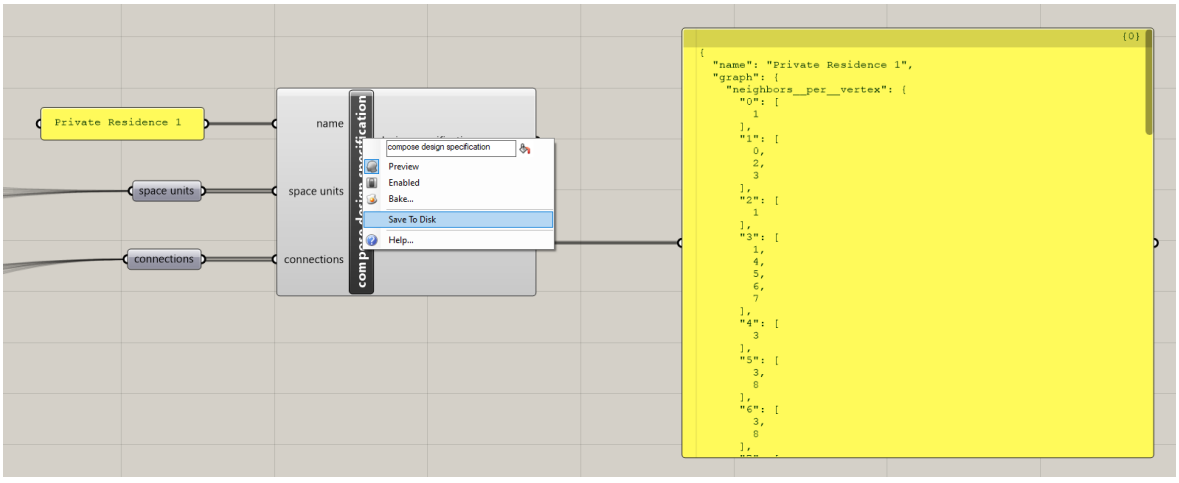
- **Name:** A name assigned to the Design Specification
- **Space units:** The list of space-units, previously defined by the designer.
- **Connections:** The list of connections, previously defined by the designer.

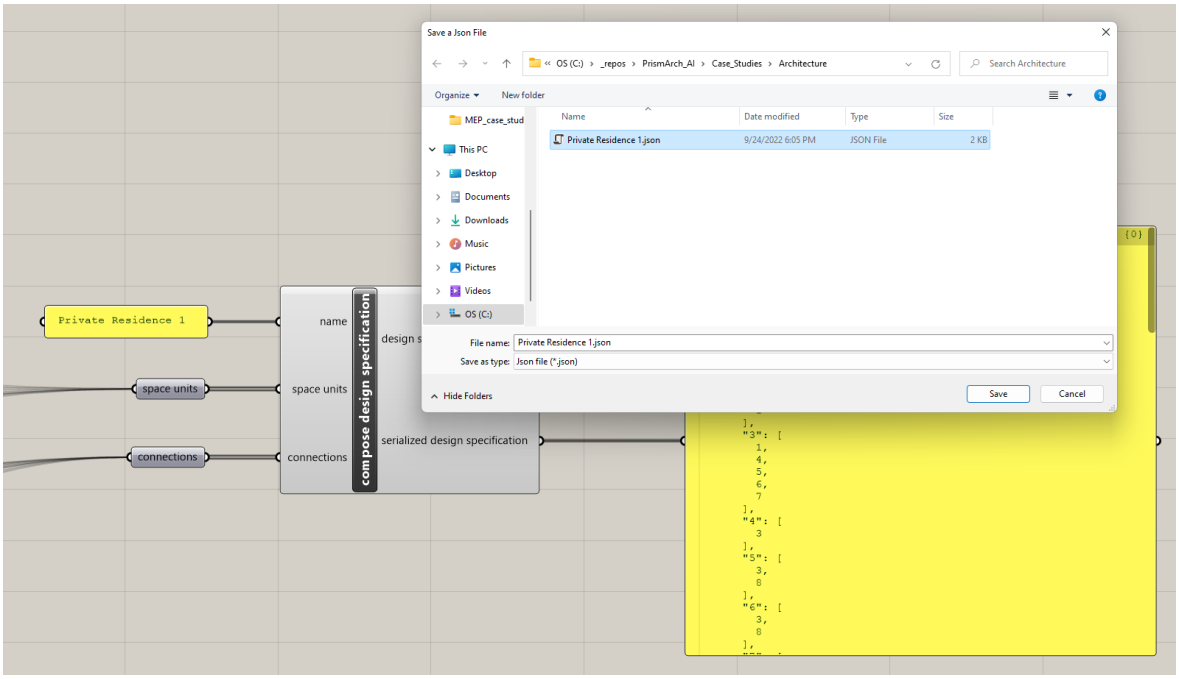
The following screenshots showcase the composition of a Design Specification in the relevant grasshopper script.

	Screenshot showcasing the whole content of the grasshopper script, including the definition of a number of space units, a number of connections and, eventually, the composition of a Design Specification.
	Close-up view of the “compose design specification” component, showcasing its three input variables: a name, a list of space units and a list of connections.

Saving the Design Specification to Disk

As soon as the designer has properly defined a Design Specification, they can save it to disk for later use. In order to do so, the designer must right-click on the “compose connectivity graph” component and select the “save to disk” option. A typical window with folder-navigation capabilities allows them to select their preferred folder and file name. As soon as they select where to save the file, the system essentially serializes the connectivity graph class into a json file, that can be opened later from the main PrismArch VR application and an interactive design session can begin.





Screenshots showcasing how a Design Specification can be saved to disk, using the custom menu element of the “compose design specification” rhino component. Furthermore, as shown in the screenshot at the top, a preview of the output json file can also be inspected from within the grasshopper environment.

4.1.2.2. Interactive Design Session

The Interactive Design Session is the core mode of operation of the AI-Tool for Architecture. During this process, the designer is continuously presented with alternative design solutions to the specified problem. The designer’s selections steer the algorithm’s operation towards certain regions of the problem’s solution space. At the same time, the designer’s selections are used to train an underlying Designer Model, which is also used in order to personalise the generated results, to the personal taste of the designer. In order to support this approach of continuous learning, we employ the rather simple Neural Network topology of a Single-Layer-Perceptron, whose research background can be found in Section 3.2.1. The specific model implementation for the architectural discipline can be found in section 4.1.1.8. Being a rather simple model (it can be represented as a series of floating point numbers), it can be easily serialised to a json file and saved to disk, or uploaded to the Speckle platform.

4.1.2.3. Session Initialization

During the session initialization, the designer must complete the following three steps:

1. **Design specification definition:**
   The user loads a design specification from disk or from speckle. This design specification describes the main constraints that the algorithm is to solve. Those include a connectivity graph that describes the design’s topology, as well as the desired area per space-unit (see section 4.1.1.1).
2. **Designer Model definition:**
   The user loads an existing Designer Model from the disk. Alternatively, a randomised Designer model may be generated by the system.

3. **Algorithm settings:**
   The user selects a number of behavioral characterizations (typically in the order of two or three) that will be used in parallel with their continuously updated designer model to generate diverse solutions adjusted to the designer's preferences.

As soon as those steps are completed, the interactive design session can begin.

4.1.2.4. Session Operation: Continuous Selection and Designer Model Update

The main operation

4. **Initial Population Generation:**
   The algorithm operates in a similar fashion to *FI-MAP-Elites* (see section 3.1.1) until at least 5% of the feasible archive's cells are populated.
5. **Initial Selection Window Placement:**
   An occupied cell of the feasible archive is selected, at random, and the selection window is centred at that cell.
6. **Archive Extension:**
   The *FI-MAP-Elites* operates for 1000 iterations, only selecting individuals from within the selection window.
7. **Design Alternatives Selection:**
   The system randomly selects four feasible individuals from within the selection window.
8. **Design Alternatives Presentation:**
   The four selected individuals are presented to the designer, through the VR-based UI.
9. **Designer Input:**
   The designer selects one of the available design alternatives.
10. **Designer Model Update:**
    Based on the designer's last selection, the Designer Model is updated. More precisely, the designer's selection generates a small dataset of three preference pairs. Those preference pairs are used to update the existing designer model, as described in Section 3.2.1.
11. **Fitness Function Replacement:**
    The fitness function of the feasible population is replaced with the updated Designer Model.
12. **Feasible Population Reevaluation:**
    Based on the updated designer model (which is the fitness function of the feasible population), all individuals of the feasible population are re-evaluated.
13. **Selection Window Placement:**
    The selection window is moved, being centred on the cell of the solution that the designer selected. As soon as this step is over, the process continues at step 6.

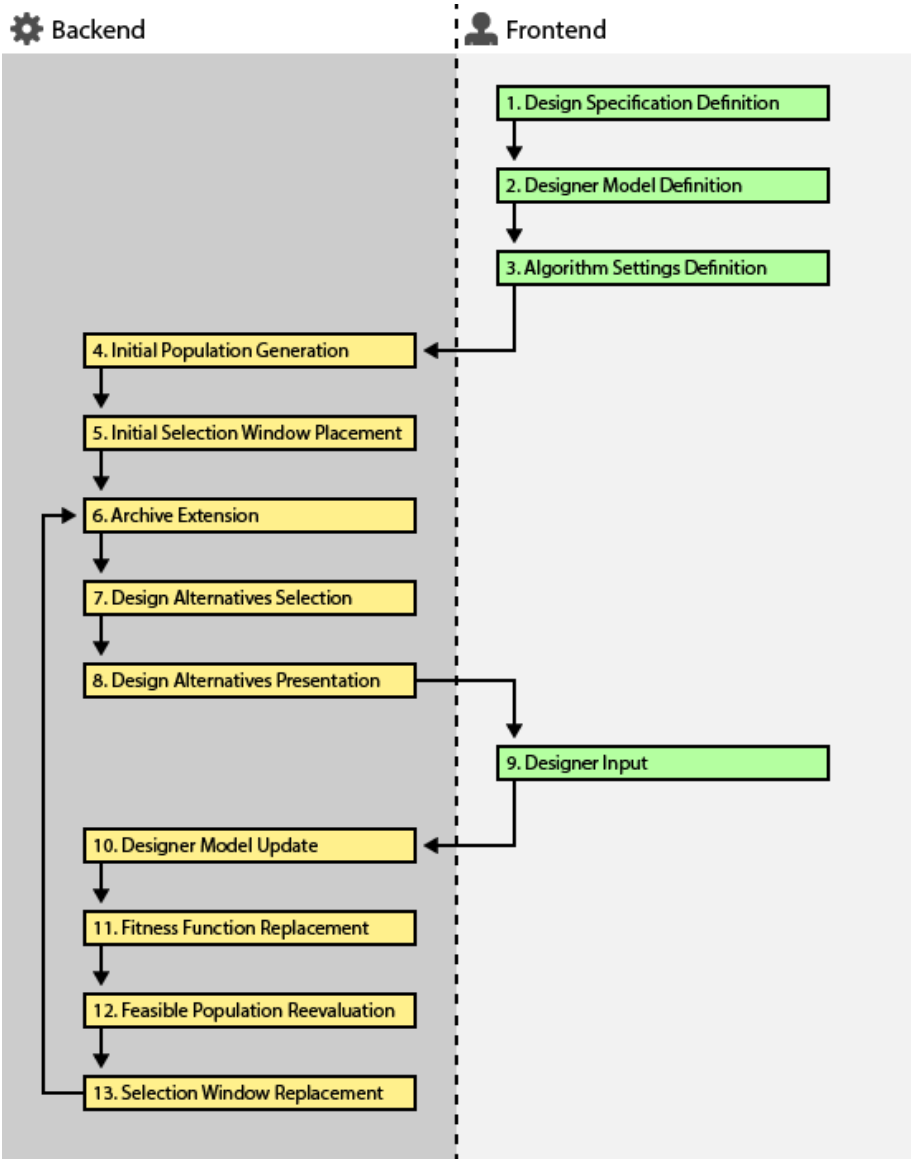


Diagram showcasing the interactive design process, pointing out the steps that occur at the front-end against those that take place at the backend.

Saving a Design Implementation

At any point, during the interactive session, the designer may choose to save a Design Implementation to disk, whether they select it or not. This option is available through the “Save Design Specification” button of the hovering UI.

Saving a Designer Model

At any point, during the interactive session, the designer may choose to save their Designer Model, for later use. This option is available through the “Save Designer Model” button of the AI-Tool’s main UI.

4.2. AI-Tool for Structural Engineering

This section describes the application of the developed algorithmic approaches to a number of families of problems in the Structural Engineering discipline.

4.2.1. Final Parametric Space of Design

4.2.1.1. Data-Structure

The evolved data-structure in this discipline is a stick-model - a model only comprised of line elements and connecting nodes - which is one of the most commonly used initial approaches to solving structural engineering problems of various kinds. The data-structure that describes the stick-model is a structural model defined within the Rhino/ Grasshopper environment, using the Karamba structural optimization plugin (https://www.karamba3d.com/).
The model must consist of linear elements that are tagged as belonging to one of the four categories: 1) Columns, 2) Bracings, 3) Primary Beams, 4) Secondary Beams.
The model must consist of an excessive amount of elements, so that the algorithm can discover feasible, fit and diverse solutions by examining subsets of the provided set of elements. This process is therefore similar to the topological optimisation/ 'soft kill' optimisation strategies that have previously been employed in the AEC domain, albeit in this instance driven by QL algorithms. Undertaking this study therefore enabled us to directly compare results with existing exemplar studies.

4.2.1.2. Random Initialization

Random initialization is executed by randomly activating or deactivating all elements of a karamba stick-model, with a 50% chance of every element being activated. The outcome of this random initialization process is most-probably going to result in an infeasible solution (one that does not satisfy the necessary constraints). However, through iterative mutation and selection, feasible solutions should gradually start appearing and covering the feasible archive.
The random initialization method can be found in the source-code, named as “GM_Random”.

4.2.1.3. Mutation

Mutation takes place in a very similar manner to random initialization. That is, mutation also operates in a fully stochastic manner, without actually attempting to satisfy any constraints or to maximise the probability of finding good solutions. In more detail, the mutation method iterates over all genes of a solution, randomly flipping their value with a rather low probability of 5% (0.05).
This mutation method can be found in the source-code, named as “MM__Mutation_Rate”.

4.2.1.4. Constraints

1. **CEM__Displacement_Limits:**
   Returns True if all displacement limits are smaller than their user-defined maximum acceptable values, or False otherwise.

2. **CEM__Kinematic_Modes:**
   Returns True if the kinematic modes are equal to zero, and False otherwise.

4.2.1.5. Feasibility Score

- **EM__CEM__Displacement_Limits:**
  Returns the average displacement score, across all possible displacements. The displacement score for a single displacement is calculated as follows: If the displacement is smaller than or equal to its acceptable limit, then its displacement score is 1. If the displacement is greater than its acceptable limit, then its score is calculated as $S = \frac{D}{d}$, where D is the maximum acceptable displacement and d is the actual observed displacement.
- **EM__CEM__Kinematic_Modes:**
  Is calculated as $S = \frac{1}{1+K}$ where K is the number of kinematic modes.

4.2.1.6. Fitness

- **"Mass minimization"**
  - Implementation name: EM__Mass_Minimization
  - Description: Returns a value in the range of [0…1]. It is calculated as $S = \frac{M-m}{M}$ where M is the total mass of all potential elements, while m is the mass of the currently active elements.
- **"Elastic energy minimization"**
  - Implementation name: EM__Elastic_Energy_Minimization
  - Description: Returns a value in the range of [0…1]. It is calculated as $S = \frac{1}{1+E}$, where E is the solution's elastic energy.

4.2.1.7. Diversity

The following list enumerates and explains the Behavioural Characterizations that have been implemented. All of them return a value in the range of [0…1] and can be used in any of the developed algorithms.

- **"Percent used elements"**
  - Implementation name: EM__Percent_Used_Elements
  - Description: Returns a value in the range of [0…1] which represents the proportion of used elements in the solution. It is calculated as $S = \frac{n}{N}$ where n is the number of active elements and N is the total number of elements in the stick model.
- **"Percent used columns"**
  - Implementation name: EM__Percent_Used_Columns
  - Description: Returns a value in the range of [0…1] which represents the proportion of used columns in the solution. It is calculated as $S = \frac{n}{N}$ where n is the number of active columns and N is the total number of columns in the stick model.
- **"Percent used bracings"**
  - Implementation name: EM__Percent_Used_Bracings

- Description: Returns a value in the range of [0…1] which represents the proportion of used bracings in the solution. It is calculated as $S = \frac{n}{N}$ where n is the number of active bracings and N is the total number of bracings in the stick model.

- **"Percent used primary beams"**
  - Implementation name: EM__Percent_Used_Primary_Beams
  - Description: Returns a value in the range of [0…1] which represents the proportion of used primary beams in the solution. It is calculated as $S = \frac{n}{N}$ where n is the number of active primary beams and N is the total number of primary beams in the stick model.
- **"Percent used secondary beams"**
  - Implementation name: EM__Percent_Used_Secondary_Beams
  - Description: Returns a value in the range of [0…1] which represents the proportion of used secondary beams in the solution. It is calculated as $S = \frac{n}{N}$ where n is the number of active secondary beams and N is the total number of secondary beams in the stick model.

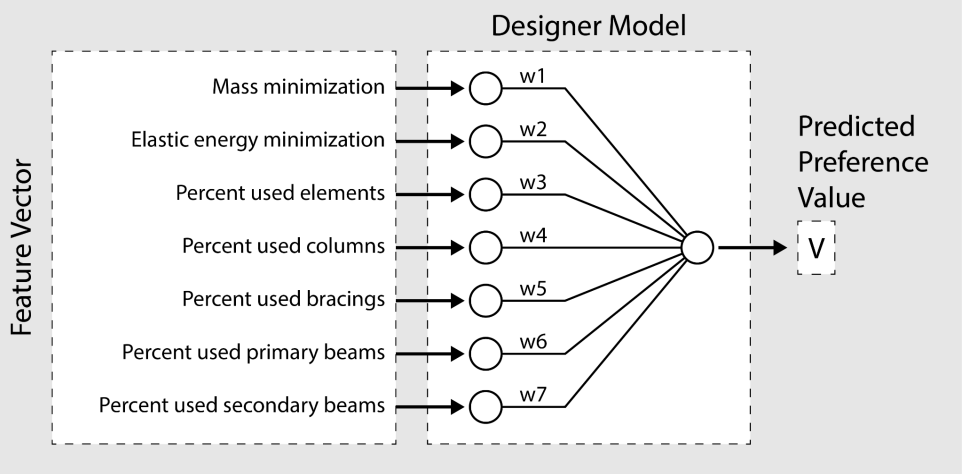
4.2.1.8. Designer Model

For the Structural Engineering discipline, the Designer Model's input is a 7-dimensional feature-vector, that characterises a stick-model, while its output is a single value that attempts to predict the designer's preference for that specific solution. The model's architecture is illustrated in the following figure, and a serialised example is shown in the following table. Importantly, the model's feature space consists of the following 7 Features, including Fitness Functions and Behavioural Characterizations, (the detailed description of which can be found in sections 4.2.1.6 and 4.2.1.7):

1. Mass minimization
2. Elastic energy minimization
3. Percent used elements
4. Percent used columns
5. Percent used bracings
6. Percent used primary beams
7. Percent used secondary beams

For the Structural Engineering discipline, the Designer Model is used as a Behavioural Characterization, which is updated in-between sessions, based on the Designer's recorded preferences at the end of each session. More details can be found in Section 4.2.2.5.

Designer Model for the Structural Engineering Disipline



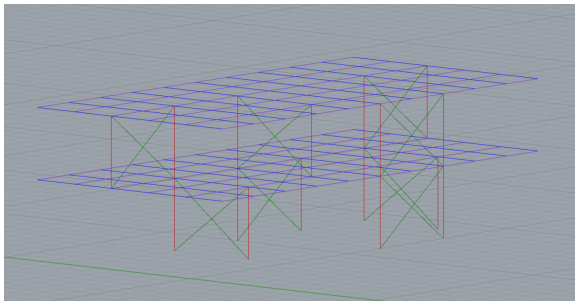
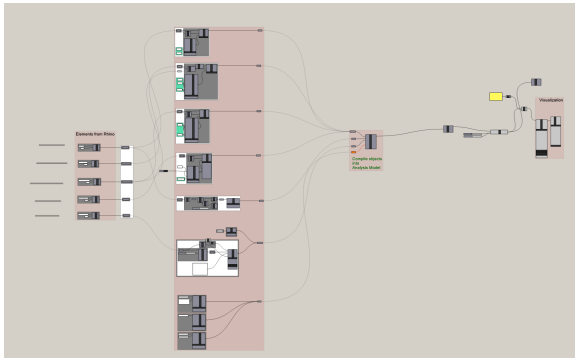
A visualisation of the Designer Model for the Structural Engineering Discipline.


```
{
  "weights": [
    0.10252656838974294,
    -0.5571194065535066,
    0.9893482709253898,
    0.4951349941525305,
    0.038948392979311075,
    0.37489226151951227,
    -0.6305013315894181,
    0.038948392979311075
  ]
}
```


An example of a serialised Designer Model for the Structural Engineering discipline. The model is represented as a list of numbers, each of which corresponds to one of the Network's weights.

4.2.2. Interactive, DM-based QD for Structural Engineering:

This section describes how the QD algorithms are implemented within the Rhino / Grasshopper environment, thus allowing the user to solve their defined problems. The section is based on a case study provided by the Structural Engineering partners (AKT) that has been set-up in the Rhino/ Grasshopper design environment, a preview of which is shown in the following table. The following subsections describe the ways in which a structural engineer can explore the solution space of this problem, using the AI-Tool for Structural Engineering.

Case-study preview	
	Screenshot of the relevant Rhino file.
	Screenshot of the relevant grasshopper script.

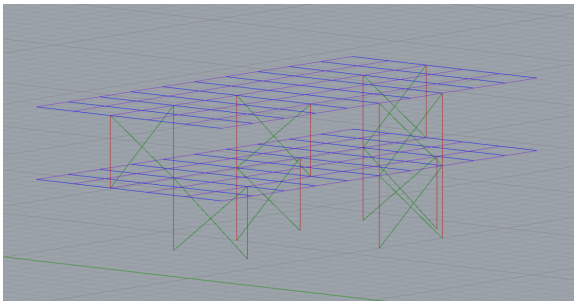
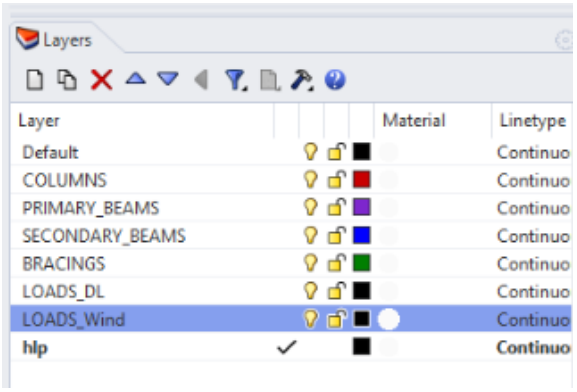
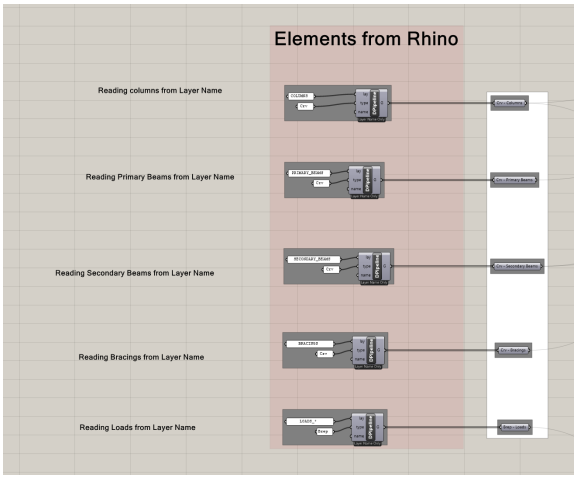
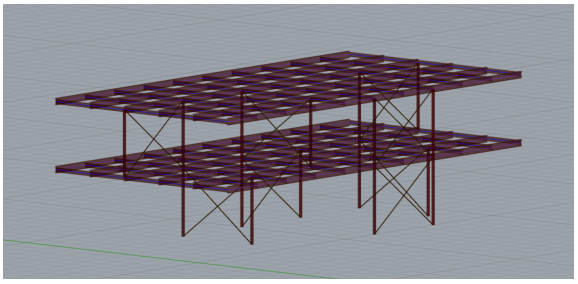
4.2.2.1. Specifying a Design Problem

The designer can specify a Design Problem (in this case a stick-model) by drawing 3D line-segments, using the basic design tools of the Rhino design software. In order for those line-segments to be recognized (and tagged) as columns, beams, bracings, the user must make sure that they belong to their corresponding layer.

As soon as the designer has finished the definition of the input-sketch, the system automatically converts this input data into a Karamba stick-model which can be mutated and evaluated during the algorithm's operation.

The Karamba Model is then used as the initial input of the algorithm, as shown in the following table of figures (User-Input).

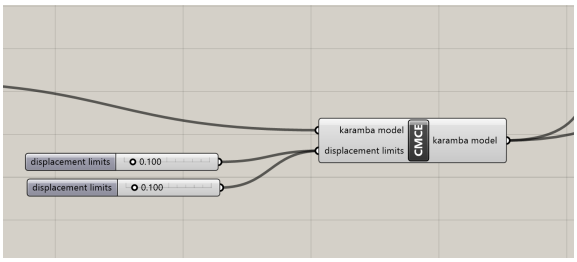
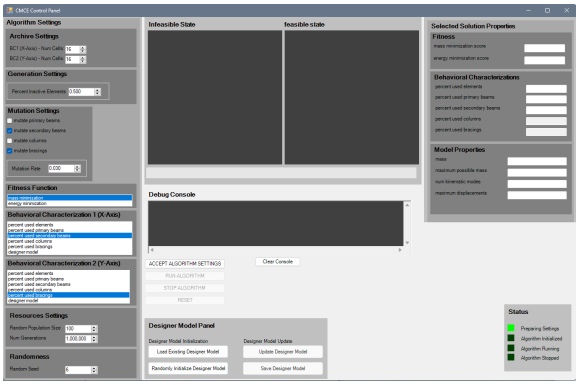
User-Input

	Screenshot showcasing the line-drawing (user input-sketch) that has been set-up as a case-study. All elements are organised in properly-named layers, so that they can be recognized by the grasshopper-script as their proper types.
	The set of used layers (for grouping elements by type).
	The initial part of the grasshopper script that automatically selects elements based on their layer and processes them as parts of a karamba model.
	Isometric view of the initial karamba stick-model that is used as input to the QD algorithm.

4.2.2.2. Algorithm Initialization

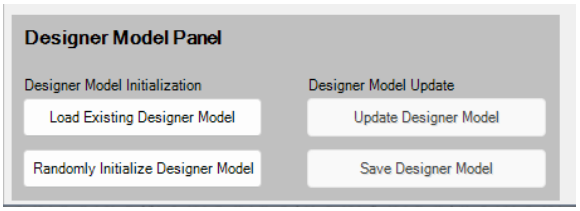
As soon as the input-sketch has been properly specified and processed, the designer can initialise the interactive QD algorithm to start exploring the solution space. All algorithm settings are accessible through a custom-designed user interface accessed through the

“CMCE” grasshopper node. The user can right-click the CMCE node and select “Open Control Panel” to bring the algorithm’s control panel into focus.
The following subsections describe all algorithmic options and parameters that are accessible to the designer, through the algorithm’s control panel.

Accessing the algorithm’s control panel.	
	Screenshot of the CMCE component, which hosts the algorithmic operation of the AI-Tool for Structural Engineering.
	Screenshot of the *FI-MAP-Elites* algorithm’s control panel, at its initial state.

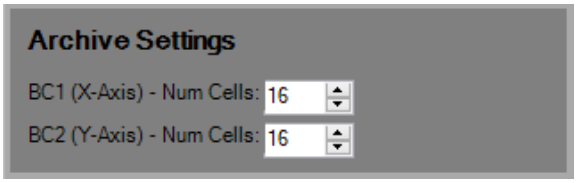
Loading an existing Designer Model

If a Designer Model already exists, based on previous interactions of the designer with the system, the designer may load this model and use it as one of the two Behavioural Characterizations. Loading the model is quite straightforward: The user clicks the “Load Designer Model” button and an open-file dialog appears that enables them to navigate to the serialised model’s location on disk.

	Screenshot of the panel that allows the user to load an existing designer model.

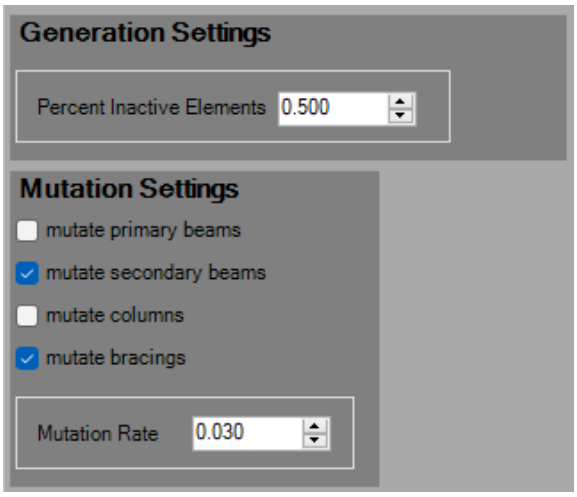
Defining Archive Resolution

The number of subdivisions for each dimension of the Behavioural Space are accessible to the designer. A predefined value of 16 cells is initially set to both dimensions, but the designer is free to experiment with other resolutions.

	Screenshot showcasing the input-fields for defining the archive's resolution along two Behavioural Characterizations.

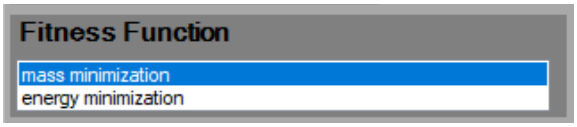
Generation & Mutation Settings

The designer can access a number of settings for the generation and mutation methods. Those include: a) the probability / percentage of active elements during the random initialization process, b) the types of elements that are mutable during the mutation process and c) the mutation rate, i.e. the probability of altering genes during the mutation process.

	Screenshot showcasing the input-fields for adjusting the generation and mutation settings.

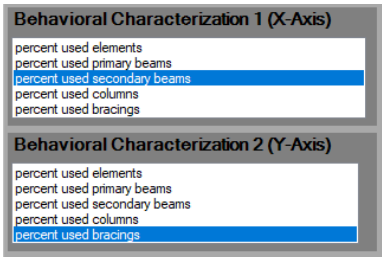
Selecting a Fitness Function

Both implemented fitness functions are available for the designer to select from: 1) Mass Minimization, 2) Elastic Energy Minimization. The user can select one of them, and it will be used as the fitness function of the feasible population.

	Screenshot showcasing the drop-down list for selecting a Fitness Function.

Selecting Behavioural Characterizations

All available Behavioural Characterizations are available for the Designer to choose from, as illustrated in the following figure. The designer can select the existing (loaded or randomised) Designer Model to be used as one of the two Behavioural Characterizations. Should they choose to do so, the one of the two axes of the archive will diversify solutions based on the designer model's evaluation criteria.

	Screenshot showcasing the drop-down lists for selecting a pair of Behavioural Characterizations.

Managing Algorithm Resources

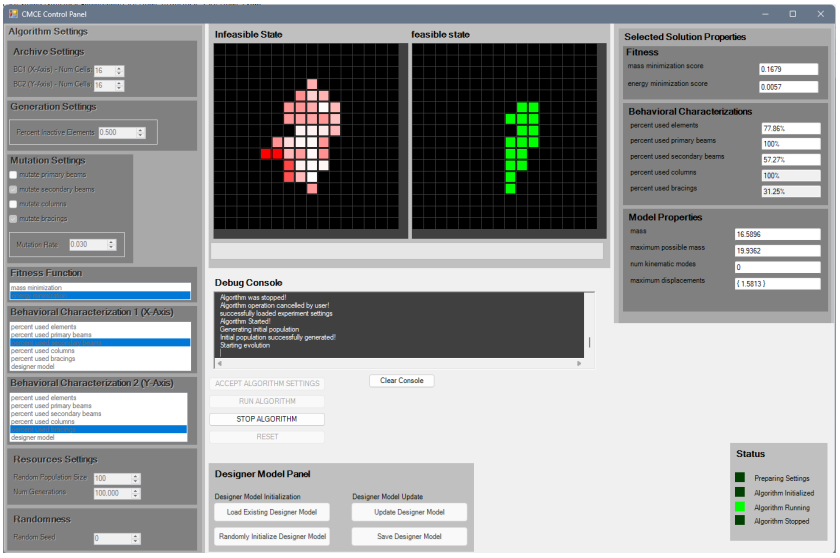
The designer can manage the algorithm's computational resources (as shown in the following figure): 1) the size of the initially generated population and 2) the total number of generations that the algorithm will execute. Furthermore, they can select a random seed, which guarantees repeatability of an experiment, should all other settings be the same.

	Screenshot showcasing the available options for adjusting the algorithm's resources, as well as setting the random-seed for repeatability of the experiment.

4.2.2.3. Algorithm Execution

As soon as the designer has selected their preferred algorithm settings, they must press the "Accept Algorithm Settings" button. This initialises the back-end and prepares the feasible and infeasible archives of the algorithm in the control-panel. Then, the designer can hit the "Run Algorithm" button so that the algorithm's actual operation can actually begin.

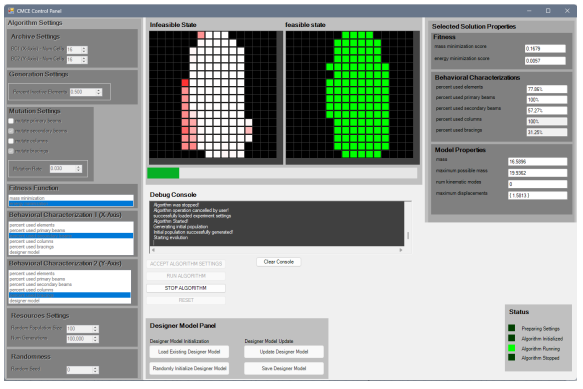
During the algorithm's operation, the archives' contents are visualised in real-time both in the control panel as well as in the viewport. The infeasible solutions are visualised in tones of red, ranging from pure red for the most infeasible solutions to almost pure white for the least infeasible ones. The feasible solutions are visualised in tones of green, ranging from pure green for the least fit solutions to pure white for the most fit ones. The archives' visualisation in the viewport also visualises feasibility and fitness by using the 3rd dimension, for a more legible overview of the archives' contents.

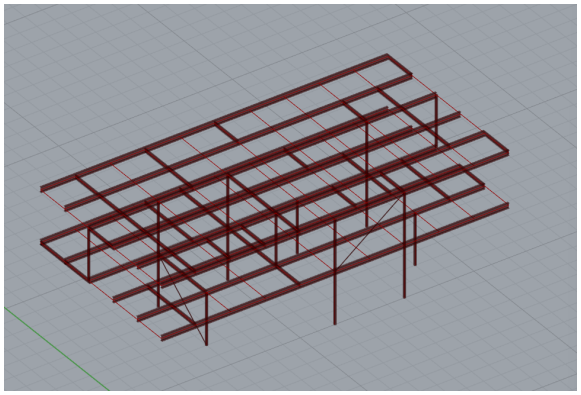
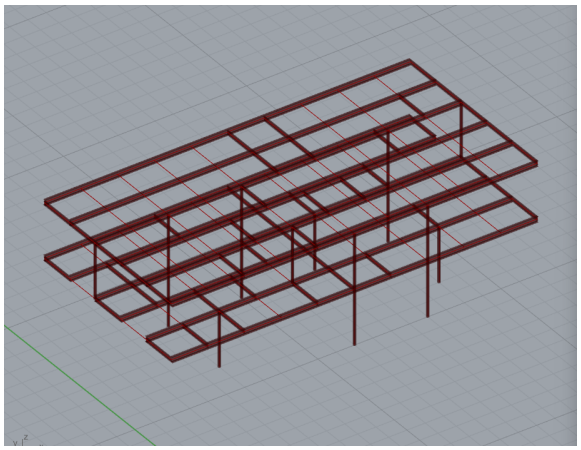


Screenshot showcasing the visualisation of the algorithm's archives, during the algorithm's operation.

4.2.2.4. Inspection of Results

As soon as the algorithm's operation is over, the designer can manually inspect the results in the following way: By clicking on any occupied cell of either the feasible or the infeasible archive, the system will visualise the corresponding solution in the viewport, as shown in the following figure. Furthermore, when a solution is selected, a number of measurable characteristics of that solution are also shown for inspection in the "Selected Solution Properties" panel. This can help the designer to make well-informed decisions about the solution at hand.

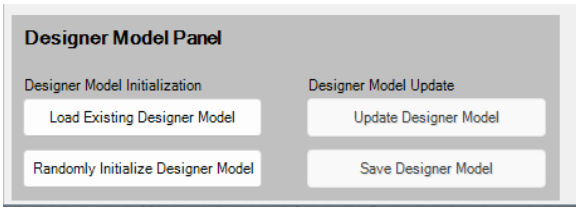
Inspection of results: examples of a feasible and an infeasible solution.	
	Screenshot of the algorithm's control panel, during the final stages of the algorithm's execution, where a fair amount of feasible solutions have been found.

	An example of an infeasible solution, which utilises a few bracings and secondary beams.
	An example of a feasible solution, which utilises a few bracings and secondary beams.

4.2.2.5. Updating and Saving the Designer Model

After the algorithmic process is over, and the designer has taken some time to inspect the generated results, the designer may choose to actively participate in a process to either update an existing model or generate a new one, based on their preferences.

In order to initiate this process, the user must click the “Update Designer Model” button. Once they do so, a special session begins: The user is asked to select their preferred solutions, out of the currently available, feasible ones. The user’s explicitly stated preferences are converted into a set of preference pairs, which are used to update the existing model. Once the training process is complete, a dialog appears that lets the user save the new model, either overwriting the old one or not.

	Screenshot of the panel that allows the user to update and save the designer model.

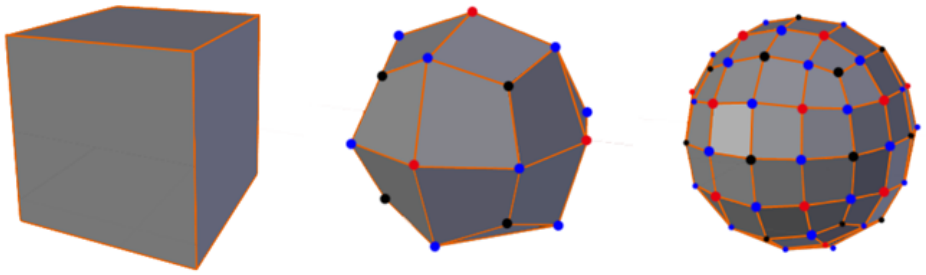
4.2.3. Shape-driven approach to QD for Structural Engineering

4.2.3.1. Overview

The topology driven approach presented in the previous chapter shows one method for working with discrete connectivities and structures made up of primarily linear elements, such as beams and columns. However, structural shapes can also take more continuous forms where the architectural form is inextricably linked to the structural behaviour and performance. Such problems require a close collaboration between architect, structural engineer and other consultants to find a shape that is acceptable to all parties according to their separate demands. An example of a complex curved shape can be seen in, for example the villa case study introduced in deliverable D1.1. For these types of structures, the previously presented approach is not applicable and a shape-driven approach is needed. It should be noted that the final shape in many cases might be built from discrete linear elements, but the surface will still dictate the overall performance and serve as the main point of negotiation between the disciplines. This process will therefore be considered to target early stage design and as design factors converge, the discretization problem is considered in latter stages.

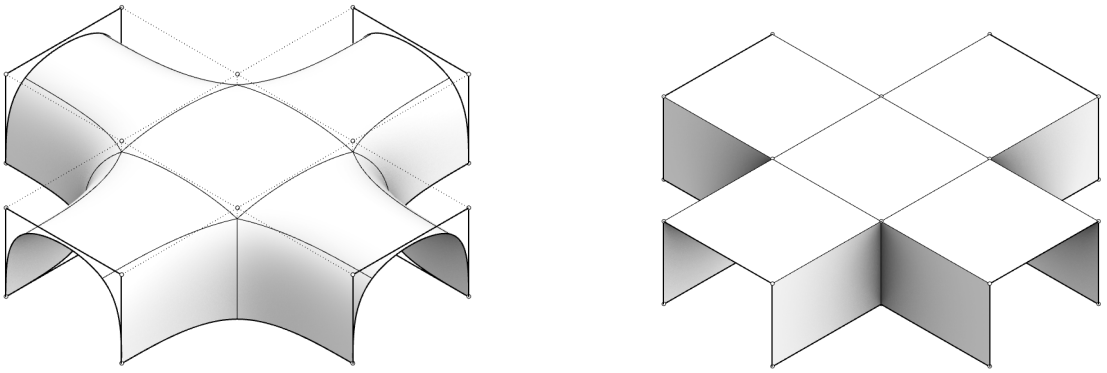
(L) Heydar Aliyev Centre (Zaha Hadid & AKT II) © Hufton & Crow (R) Shell Models in the office of Heinz Isler. Reproduced from [https://www.schwartz.arch.ethz.ch/Publikationen/Dokumente/Isler.pdf]

A shape-driven approach would not turn binary elements on and off, but instead work with a set of discrete control variables to shape or sculpt a complex continuous surface. The framework would therefore need to be flexible in dealing with a wide range of base shapes that can be mutated and evolved using the discrete controls. A common way of dealing with geometry in this fashion is subdivision surface modelling; a method used to translate low poly models, that are easy to sculpt, into smooth representations. One of the most common methods to achieve this is Catmull-Clark [16] subdivision, which recursively splits mesh quads into smaller mesh quads with vertex smoothing at each iteration. Through this technique complex geometries can be controlled with a very limited set of “control points”. An example of Catmull Clark subdivision is shown below.



Subdivision of a coarse mesh (Left) to a smooth mesh (right)

Rhino provides a complex object based on this approach called SubD which will be used to mutate and control complex shapes as part of the QD process. An example of this object is shown below, with the control polygon on the right.

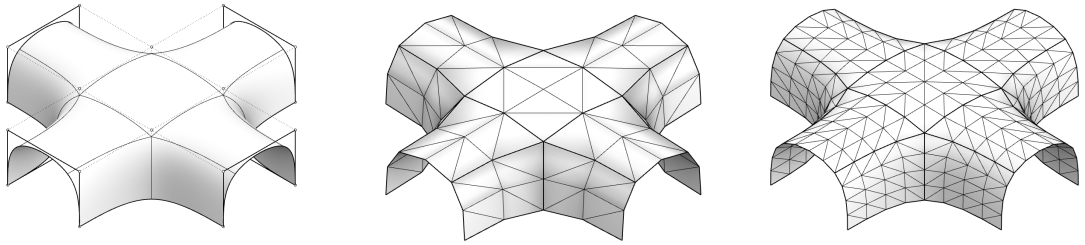


Example of a Rhino SubD object

4.2.3.2. Approach

Data Structure Approach

The data structure differs from the topological approach in two fundamental ways. Firstly, where the topological model has intrinsic variability in the multitude of elements that enable different configurations, the SubD object is singular. Therefore the data structure is two-fold as it needs to embed both the base representation of the geometry described, and also provide information about how the shape can be adapted or mutated. Secondly, where the topological approach can work directly with the structural model the shape optimization must regenerate the structural model each time, as changes to the shape also change the stiffness, and therefore have to be recomputed. Here a benefit of using a subdivision approach is that a refined mesh can be quickly generated using the subdivision logic, thus, no complex meshing algorithm is needed, and the computation time for each iteration is decreased. An example of the fast structural meshing process is shown below:

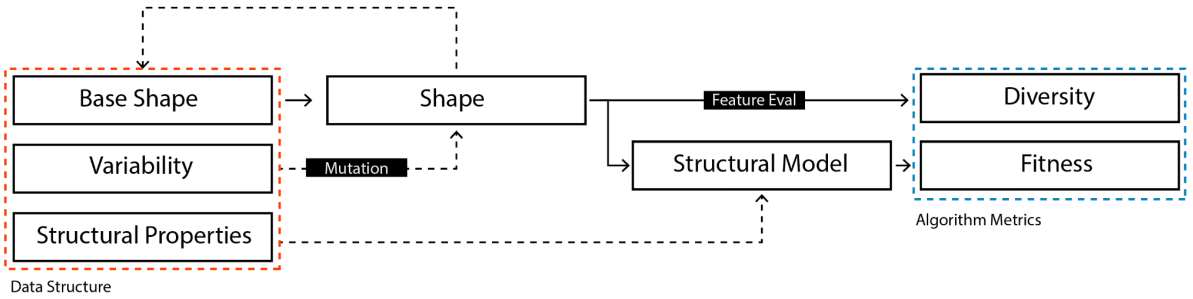


Left: Initial SubD Middle: Subdivision level one (triangulated) Right: Subdivision level 2 (triangulated)

To properly run a shape optimisation process, the following information is needed:

- A set of structural properties to define the analysis problem
- An embedded parameterisation or variability

The process should embed the parameterisation in the object so it can be distributed with the geometry and also potentially be configured and executed in a VR environment, with the presence of either a standard FE engine or a surrogate model. To achieve this, the data structure has been set up according to the following diagram.



The data structure stores the information on the left to enable shape manipulation and automated generation of a structural model for simulation. The shape provides the basis for feature evaluation and the structural properties include loads, materials and support conditions. At each mutation a new object with an updated base shape is generated.

Process overview

A SubD model, created either using manual modelling in Rhino or through some parametric approach in the Grasshopper environment serves as a starting point. From there, a set of control points are assigned a certain variability according to constraints and preference decided between the disciplines. The engineer specifies the structural properties and the features to evaluate. Once the model has been established, these inputs are passed to the QD algorithm along with the desired settings. The solver runs for the desired number of iterations, and from there solutions can be extracted and filtered based on their metrics.

4.2.3.3. Implementation

The implementation uses the framework presented in chapter 3 as the core engine with a set of bespoke additions.

Main Data Structure

The main data structure holds a SubD object to represent the geometry along with a set of additional objects which are describing the constraints, variability and structural properties. All of these are necessary for the model to be complete and runnable. The data structure then provides the methods to generate a structural model (using karamba) and to solve the structural model, which populates the fitness fields of the model. When the instances are evolved the base shape is copied along with the structural properties and the arrays of constraints and variations. After the new instance is mutated the new structural model is generated and solved.

Sub Structures for model control

The additional data structures added to handle sub components include objects for describing the variability of the model, the support conditions and the necessary structural properties. The objects defined are:

- **“Variable Point”**

The first is the variable point definition. This object is used to explicitly define a domain within which a point can vary, and is provided by a point, a vector along which the point can move and a domain which determines the bounds. The point variation comes in two forms, one coordinate based public version, and one index based internalised version. The coordinate based type is used for user input and, through being coordinate based, can be input without knowledge about the index numbering of the SubD. During assembly this is then resolved together with the SubD and stored internally as an index based point, as when the mutation is underway the coordinates of the points are no longer constant.

- Implementation name: SubD_VariablePointXYZ

Description: Used as a builder object within the grasshopper environment. Stores the point as a Point3d object. To define the variability, it holds a 3d vector to define the direction in which the point can move and a domain to define the bounds on how far it can move each direction from the initial location.

- Implementation name: SubD_VariablePointIndex

Description: When the model is assembled the points are defined as references to the SubD vertex list. The variability is now recreated as a geometric line which is used to sample the parameter value for the mutation process.

- **“Constrained Point”**

The logic of the constrained points follow the same logic as the variable point with a public coordinate based builder object and an internal index based version which is resolved together with the SubD. In addition the constrained point stores six boolean values to describe which modes of rotation and translation are fixed. Any point on the SubD control polygon is made creased to ensure that the point that is constrained actually exists on the smooth shape.

- Implementation name: SubD_ConstrainedPointXYZ

Description: The logic of this object directly mirrors its equivalent for the variation. The difference is that this object, in addition to the 3d point, holds 6 boolean values which describe the restraint conditions at the point.

- Implementation name: SubD_ConstrainedPointIndex

Description: The index based support is used to feed the correct support conditions and positions to the karamba model as the location of the point may again vary.

- **“Structural Properties”**
  - Implementation name: SubD_Structural_Properties

Description: The structural properties are all contained within this object and not as individual fields. This is largely to allow the properties to be created in isolation from the model and passed to the model constructor as a bundle. This object holds information about shell thickness, material properties, loads and meshing resolution.

Diversity

To categorise the generated shapes, a set of methods that are different to the topology driven approach are needed. Here, the methods need to capture some aspect of the sculpted form, such as symmetry, curvature or height. The current implementations of feature evaluations focus on properties at predescribed points on the subd as single values, average values or difference between a set of values. Due to the difficulty of normalising these values, each feature also holds a remapping domain which is provided by the user.

- **“Edge Point Coordinate”**
  - Implementation name: EM__Edge_Point_Coordinates

Description: Returns the coordinate value at a specific parameter of one edge in the subd geometry. Whether the X, Y or Z component is returned is chosen by the user. The feature is defined by an edge ID and a parametric value *[0<= t <= 1]*.

- **“Edge Point Coordinates Average”**
  - Implementation name: EM__Edge_Point_Coordinates_Average

Description: Returns the average coordinate value of a set of parametric points on a series of edges in the SubD geometry. Whether the value is based on the X, Y or Z component is chosen by the user. The average is simply calculated as the sum of all point values divided by the number of points. The feature is defined by a list of edge IDs and a list of corresponding parametric locations *[0<= t <= 1]*.

- **“Edge Point Coordinates Deviation”**
  - Implementation name: EM__Edge_Point_Coordinates_Deviation

Description: Returns the deviation of coordinate values from a set of parametric points on a series of edges in the SubD geometry. Whether the value is based on the X, Y or Z component is chosen by the user. The deviation is calculated as the difference between the average value and the value per point divided by the number of points. The feature is defined by a list of edge IDs and a list of corresponding parametric locations *[0<= t <= 1]*.

- **“Edge Point Curvature”**
  - Implementation name: EM__Edge_Point_Curvature

Description: Returns the absolute curvature at a specific parameter of one edge in the subd geometry. The feature is defined by an edge ID and a parametric value *[0<= t <= 1]*.

- **“Edge Point Curvatures Average”**
    - Implementation name: EM__Edge_Point_Curvature_Average

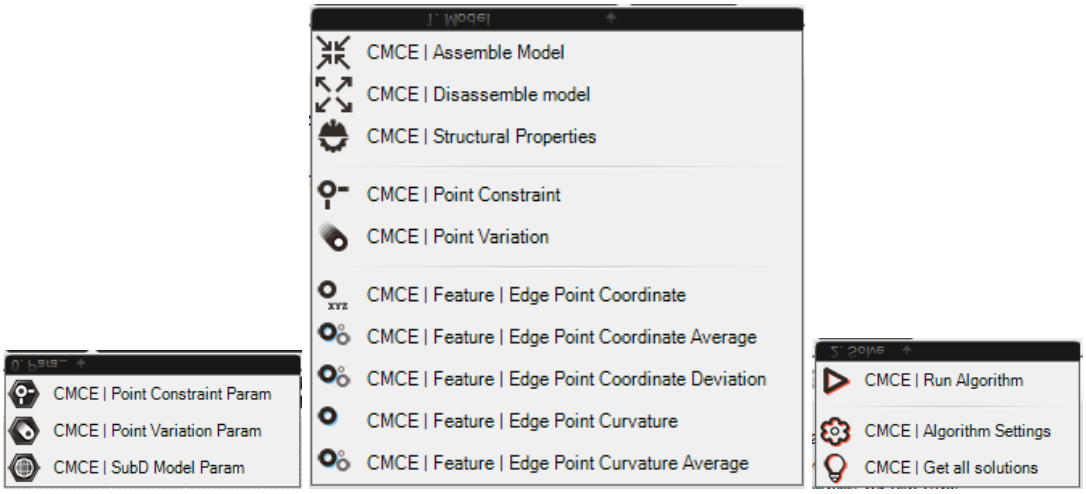
Description: Returns the average curvature from a set of parametric points on a series of edges in the subd geometry. The average is simply calculated as the sum of all curvature values divided by the number of locations. The feature is defined by a list of edge IDs and a list of corresponding parametric locations *[0<= t <= 1]*.

Mutation

- **“Mutate Shape”**
    - Implementation name: MM__MutateShape

Description: The mutation method iterates over all the point variations defined in the model and moves the point along the defined vectors according to two parameters, which are ***Max Step Size*** and ***Mutation Rate***. The maximum step size dictates how large the step in the parameterization can be. The step is generated as a randomised value between ±max step. This means that if the step size is capped at a low value, the mutated shape will stay close to the previous shape and the evolution will evolve in more detail, whereas if it’s set to a large value the mutated shape may be completely different to the previous one. The mutation rate simply sets the probability that a point variation is executed, so all variations might not be called at each mutation.

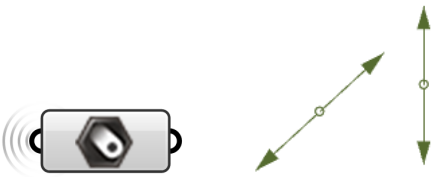
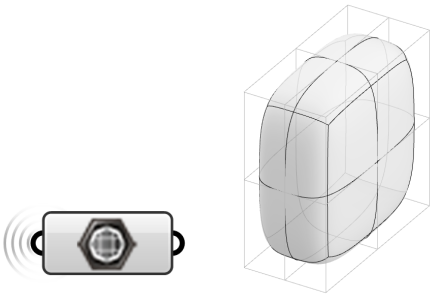
4.2.3.4. Toolset

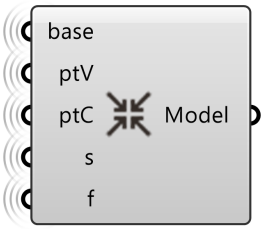


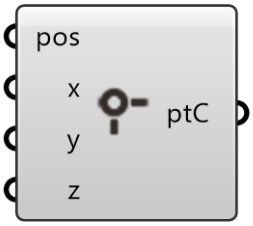
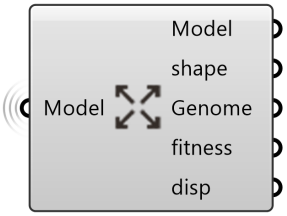
Overview of the grasshopper toolbar

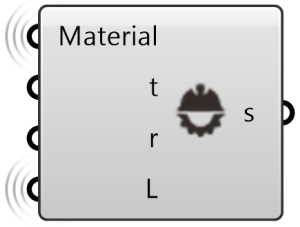
To enable the above workflow a set of grasshopper components was developed. The approach here differs from the one previously presented, as the design space has to be explicitly defined and, to make the tools as generic as possible, this was done through following the grasshopper paradigm. This means encapsulating different parts of the process into a range of components, which can be used to parametrically assemble a bespoke logic. The toolset is divided into two major sub groups, namely ***model,*** which provide tools for defining both the parametric space, features and the structural aspects of the shape to

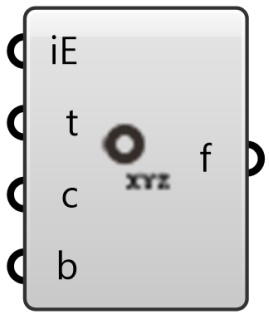
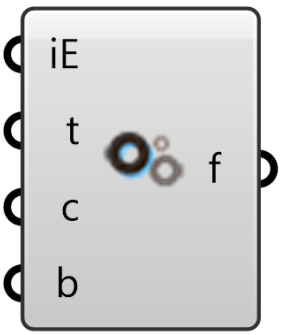
optimise, and ***solve***, which provide the settings and components to run the process, along with tools to retrieve results. In addition the toolset also provides a set of params to store and preview the custom objects presented in the previous chapter.

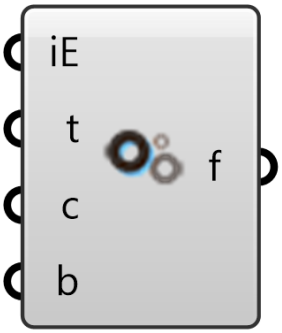
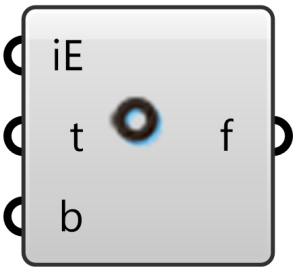
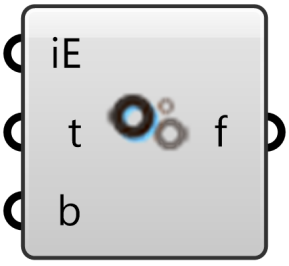
Params	
	**CMCE | Point Constraint Param** **Description:** The point constraint param is used to hold point constraint objects before they are assembled and allows the display of the support conditions in the Rhino Viewport.
	**CMCE | Point Variation Param** **Description:** The point variation param is used to hold point variation objects before they are assembled and allows the display of the variability domains in the Rhino Viewport as arrows.
	**CMCE | SubD Model Param** **Description:** The SubD Model param holds an assembled model and displays the shape of the smooth geometry.

2. Model	
	**CMCE | Assemble Model** **Description:** This component assembles the base geometry, constraints, variations etc. into a complete shape optimisation model that is ready to be executed. **Inputs:** Base Geometry (base): A SubD object representing the base geometry. Point Variations (ptV): A list of point variations for the mutation process. Point Constraints (ptC): A list of constrained support points for the model. Structural Properties (s): The structural properties for the model.

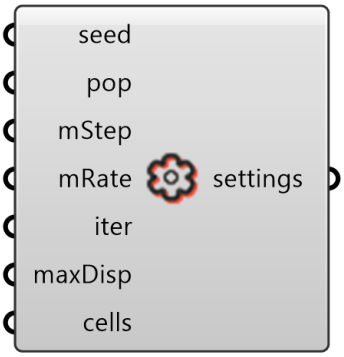
	Feature Functions (f): A list of two or three feature functions for the diversity classification process. **Outputs:** Model: A SubD Model object
	**CMCE | Point Constraint** **Description**: Creates a point constraint object from a 3D Point and a set of restraint booleans. **Inputs:** Position (pos): Point location of constraint. x: Restraint in x direction y: Restraint in y direction z: Restraint in z direction **Outputs:** Point Constraint (ptC): The generated point constraint object.
	**CMCE | Point Variation** **Description**: This component is used to define the necessary structural properties for the geometry. **Inputs:** Position (pos): Original location of variation. Direction (dir): Vector along which to move point. Domain (dom): Distance bounds from initial position. **Outputs:** Point Variation (ptV): The generated point variation object.
	**CMCE | Disassemble Model** **Description**: Disassembles the solved model to expose fitness and shape of a certain model **Inputs:** Model: The SubD Model to disassemble **Outputs:** Karamba Model(Model): The structural model of the solution Shape(shape): The geometric SubD representation of the model. Genome: Array of parameter values for each point variation. Fitness: Fitness value per loadcase.

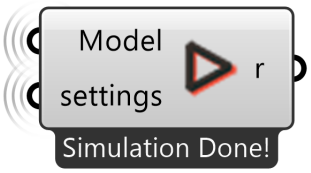
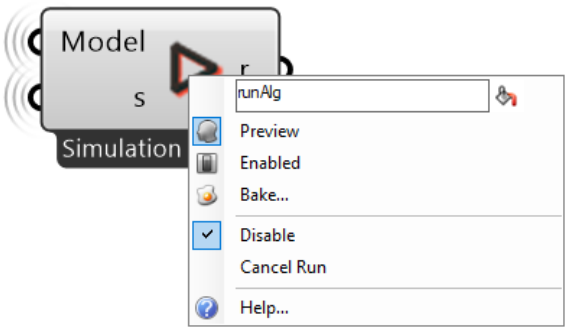
	Displacement (disp): Max displacement per loadcase
	**CMCE | Structural Properties** **Description:** This component is used to define the necessary structural properties for the geometry. **Inputs:** Material: A Karamba FEM Material Thickness (t): Shell thickness in metres Subdivision Level (r): Number of recursive subdivisions to get structural mesh. Loads (L): List of loads. Either as vectors, which will be distributed on the whole shape, or custom karamba mesh loads. **Outputs:** Structural Properties (s): An object that stores the properties to be passed to the model assembly.

	**CMCE | Feature | Edge Point Coordinate** **Description:** Create an evaluation based on the coordinate value of a single point. **Inputs:** SubD Edge (iE): Index of the subd edge Parameter (t): Evaluation parameter on edge Component (c): Component to evaluate (x, y or z) Bounds (b): Domain for remapping to feature map. **Outputs:** Evaluation Method (f): The generated feature evaluation method
	**CMCE | Feature | Edge Point Coordinates Average** **CMCE | Feature | Edge Point Coordinate** **Description:** Create an evaluation based on the average coordinate value of a set of points. **Inputs:** SubD Edge (iE): List of subd edge indexes Parameter (t): List of parameters at which to evaluate edges.

	Component (c): Component to evaluate (x, y or z) Bounds (b): Domain for remapping to feature map. **Outputs:** Evaluation Method (f): The generated feature evaluation method
	**CMCE** | **Feature** | **Edge Point Coordinates Average Deviation** **Description**: Create an evaluation based on the deviation between coordinate values of a set of points. **Inputs:** SubD Edge (iE): List of subd edge indexes Parameter (t): List of parameters at which to evaluate edges. Component (c): Component to evaluate (x, y or z) Bounds (b): Domain for remapping to feature map. **Outputs:** Evaluation Method (f): The generated feature evaluation method
	**CMCE** | **Feature** | **Edge Point Curvature** **Description**: Create an evaluation based on the curvature value at a point on an edge.. **Inputs:** SubD Edge (iE): List of subd edge indexes Parameter (t): List of parameters at which to evaluate edges. Bounds (b): Domain for remapping to feature map. **Outputs:** Evaluation Method (f): The generated feature evaluation method
	**CMCE** | **Feature** | **Edge Point Curvatures Average** **Description**: Create an evaluation based on the average curvature value of a set of points on a series of edges. **Inputs:** SubD Edge (iE): List of subd edge indexes

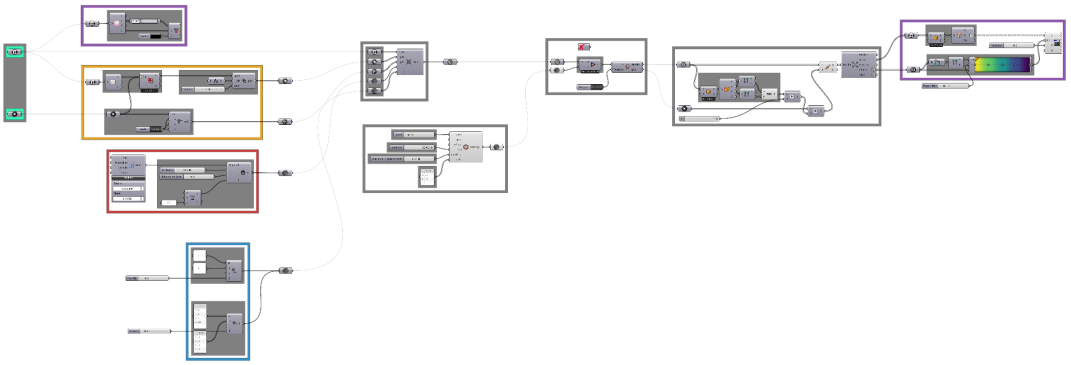
	Parameter (t): List of parameters at which to evaluate edges. Bounds (b): Domain for remapping to feature map. **Outputs:** Evaluation Method (f): The generated feature evaluation method

3. Solve	
	**CMCE** | **Algorithm Settings** **Description**: The basic settings needed for the QD process. **Inputs:** Seed (seed): List of subd edge indexes Initial Population (pop): List of parameters at which to evaluate edges. Maximum Step (mStep): Maximum step in the parametric domain for the mutation of each gene. Mutation Rate (mRate): Probability that a gene changes during mutation. Iterations (iter): Domain for remapping to feature map. Maximum Displacement (maxDisp): List of maximum allowed displacements per loadcase. Tessellation (cells): List of number of cells in each direction (2 or 3 depending on amount of features.) **Outputs:** Settings (f): The settings for the process
	**CMCE** | **Get All Solutions** **Description**: The basic settings needed for the QD process. **Inputs:** Results (r): Result object returned from the solver component. Feasible: Boolean toggle to select whether to show the feasible or infeasible solution map. **Outputs:** Model: List of all solutions

	Map Location (pos): List of the models corresponding positions in the feature map.
 	**CMCE | Run Algorithm** **Description**: Runs the quality diversity algorithm on the supplied model. The solver runs asynchronously and provides progress reporting during the solution. After solution the solver is disabled and has to be re-enabled to run again. This is done by right-clicking on the component. **Inputs:** Mode: The SubD Model to evaluate. settings: The settings for the algorithm. **Outputs:** Result: Object containing the results from the process.

4.2.3.5. Case Study 1: Simple shells

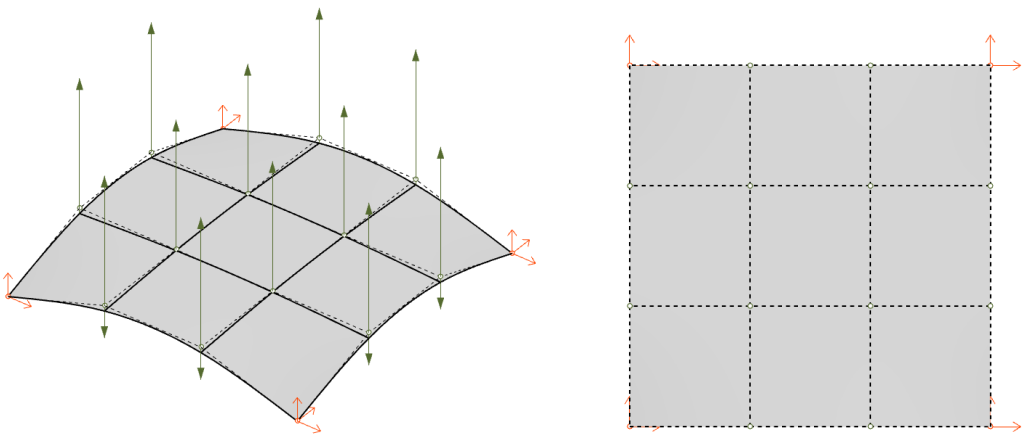
The set of tools was initially tested on a set of simple shell problems to examine the viability of the approach and to understand if the map could produce a useful image of the design space. Here the high performing solutions are largely known beforehand so they act as benchmarks to ensure that the solutions provided in the map are as expected. A series of slightly different geometries and properties are tested to see how the set of solutions in the space adapts. To run the test cases a simple base script was set up which is shown below. The script can be found in the *ShapeOptimisation* folder under the *EDF__Rhino_Structural project*.



To run an analysis the base shape and supports need to be input in the containers on the left, the rest of the point will be assigned as variable.

Square base

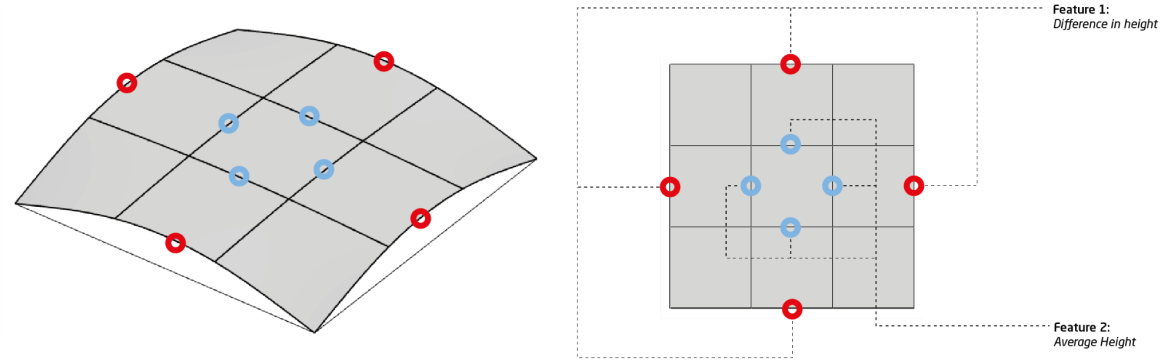
The first case study is a square base subdivided into 9 patches and 16 points and can be seen in the image below. The algorithm will then vary the height of each point in order to optimise the shape, which is inspired from the form finding examples of, for example, Heinz Isler shown in chapter 4.2.3.1. Each point is given a variability in Z-direction from 0 to 5, which is represented by the green arrows at each control point.



The structure is analysed as a structural shell with pinned supports located at each corner. The structural analysis process is set up with the following properties:

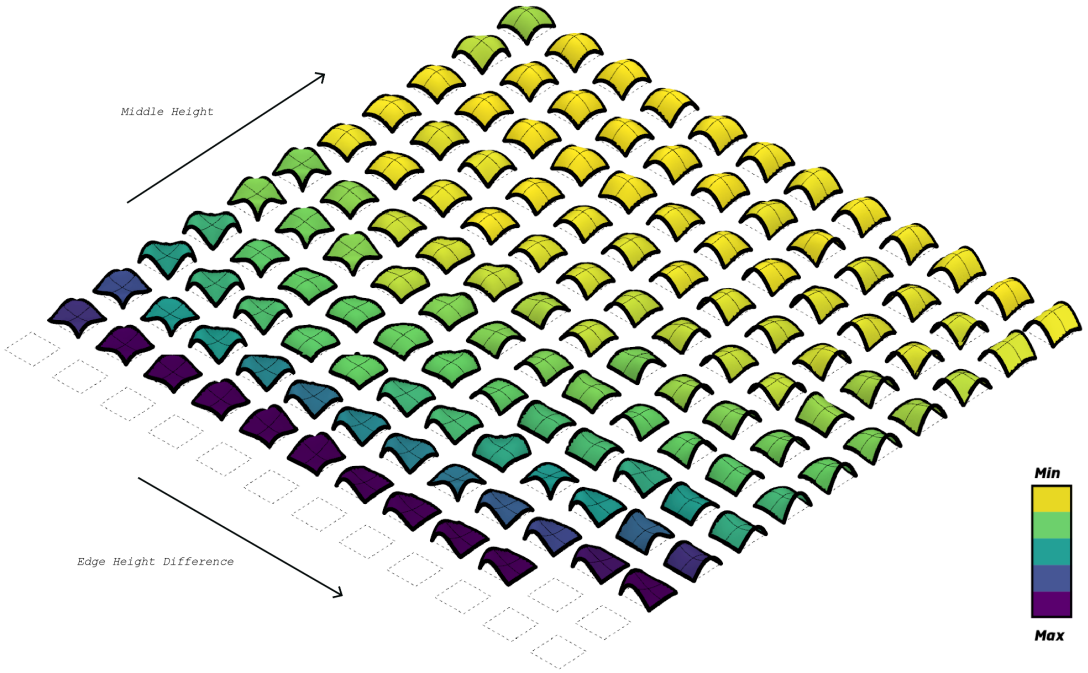
Material	Thickness [cm]	Mesh Resolution	Loads
C16/20	7	2	2kN/m2

The classification of diversity is set up to focus on two target areas, namely the centre point and the arches. The first feature method is taken as the average of the edges of the central face, to capture the height at the middle of the shell, and the second is the difference in the height between the midpoints of the side arches, in order to capture asymmetric aspects of an otherwise symmetrical problem.



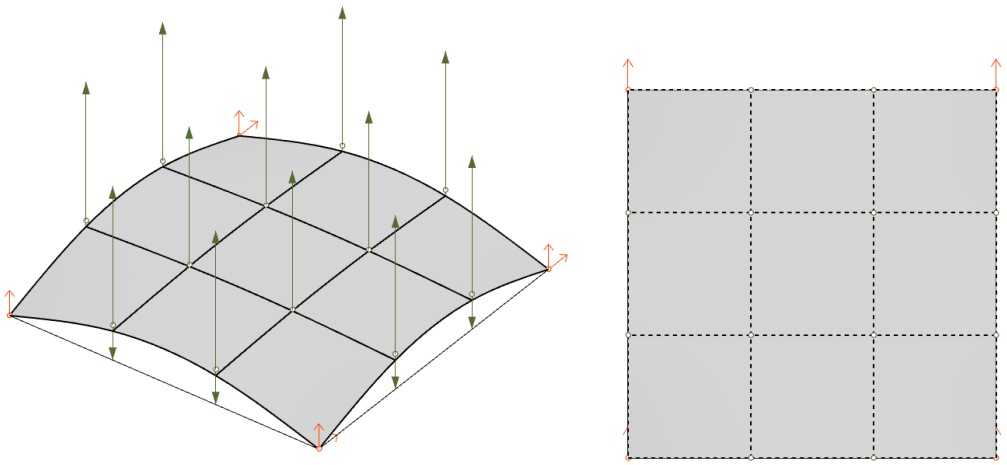
The algorithm settings are defined according to the table below and the optimisation is using the energy minimisation fitness criteria and displacement feasibility constraint as described in chapter 4.2.1.6.

Initial Population	Maximum Step	Mutation Rate	Iterations	Max Displacement	Map Tessellation
500	0.20	0.50	100,000	0.05 [m]	12x12

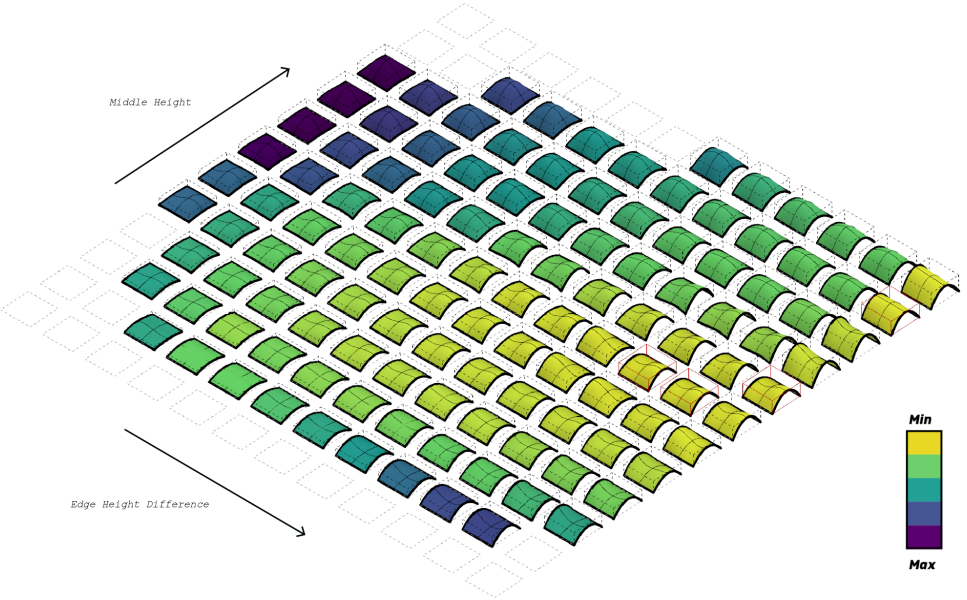


The figure above shows the map of feasible solutions, with a colour map showing the fitness (elastic energy) with yellow being the “fitter” solutions as these show a higher stiffness (low elastic energy). The map shows the importance of height and the best performing solutions are as expected found in the top left corner where the features correspond to the solution being high and symmetric. However, it can be seen that asymmetric vaulted solutions can be used with little trade off in stiffness.

To see how the map can be used to illuminate the impact of design conditions, the same process was run with slightly modified support conditions. Instead of fully pinned supports, horizontal thrust was now only allowed in one direction, as shown in the image below:

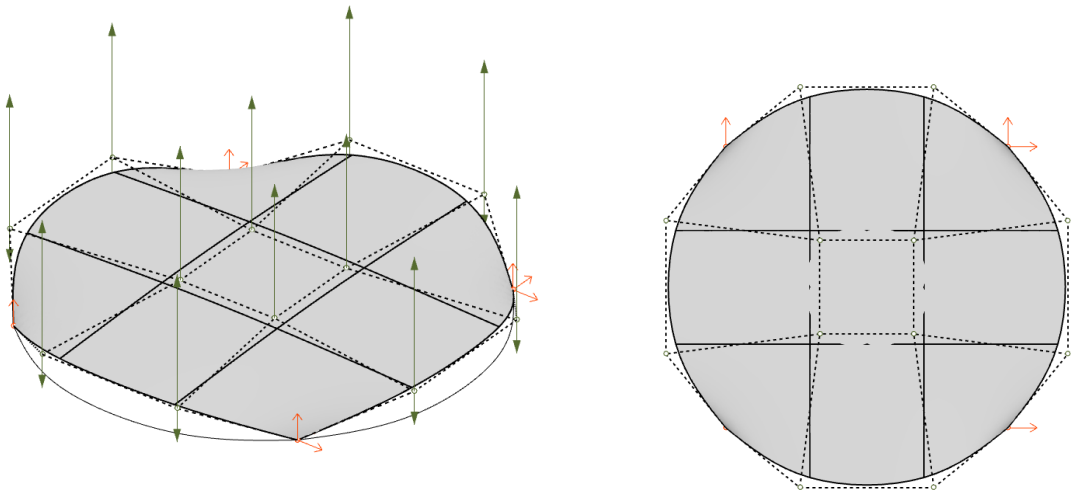


This yields the following map, illustrating how the symmetric compression shells are no longer viable solutions and the high performing spots have shifted to the asymmetric areas of the map, where single span vaulted solutions are found.

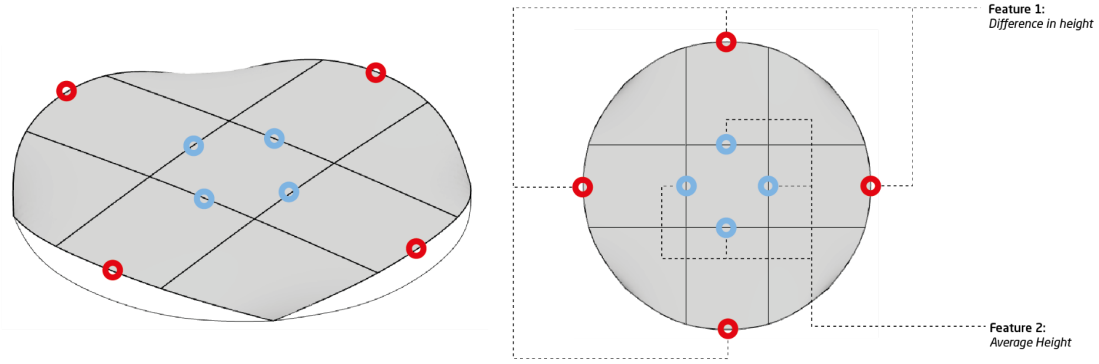


Circular Base

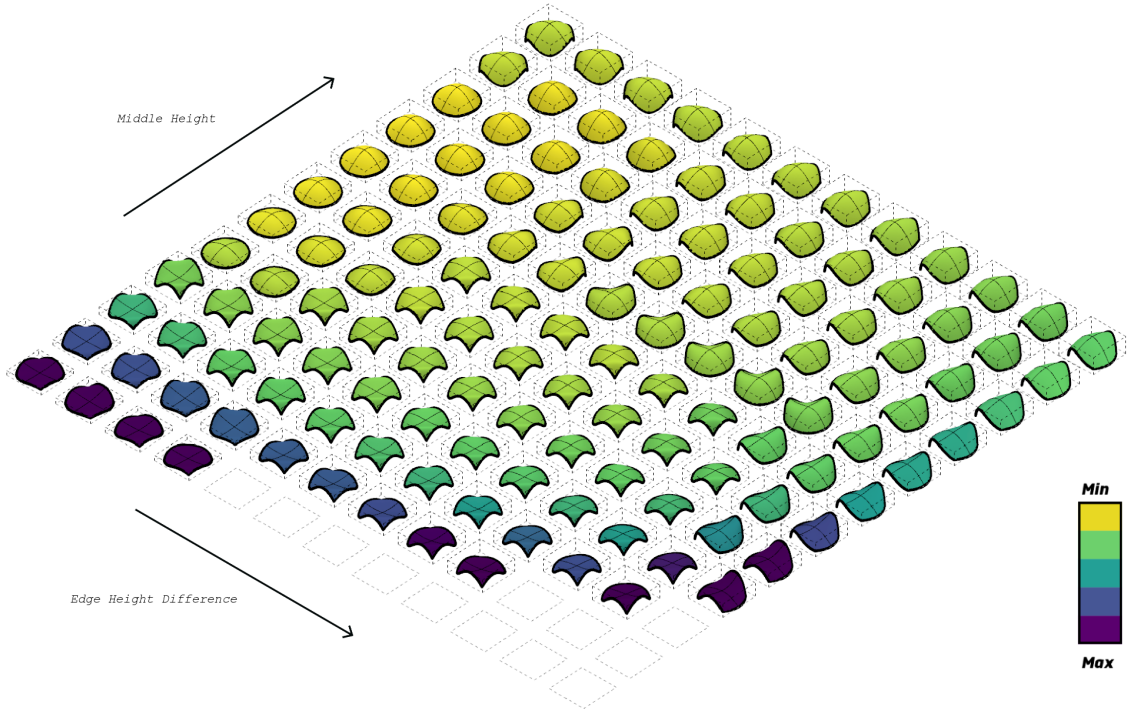
To understand how the map would adjust in the case of changes in the base geometry, the square patch was tweaked by curving the edges to near circular segments, to give the base shape a more continuous boundary. The support conditions and parameters remained the same. The problem is illustrated in the figures below.



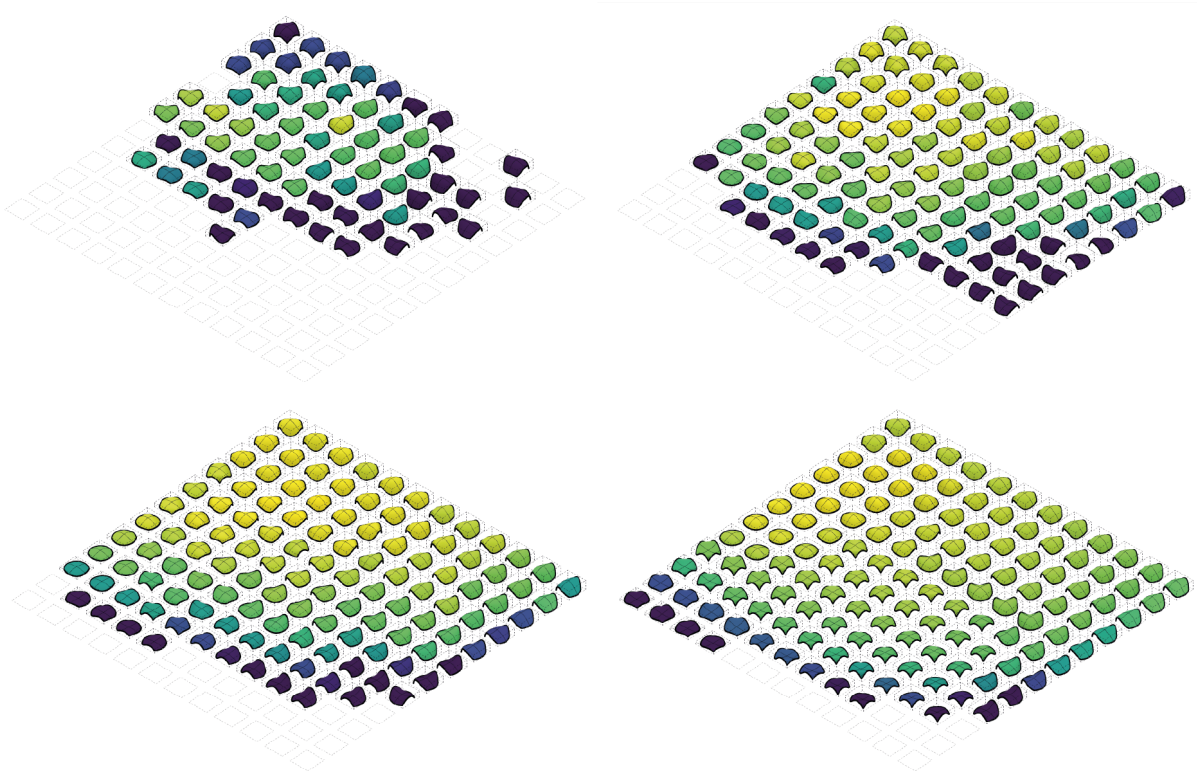
The features used for the diversity are also identical to the ones used for the square base, with the two aspects considered being the middle height and the height at the middle of the spanning circular segments.



The algorithm is run using the same settings as the previous one and after 100,000 iterations yields the following map.



As expected the fittest solution is high and symmetric, however, in this case the arches are low to form a dome, and the fittest region has moved back to being high, but only so high that it can have flat side arches.. The colour map again shows the relative fitness. To see how the solutions evolved over time, the image below shows the map after 1000, 5000, 10,000 and 100,000 iterations.

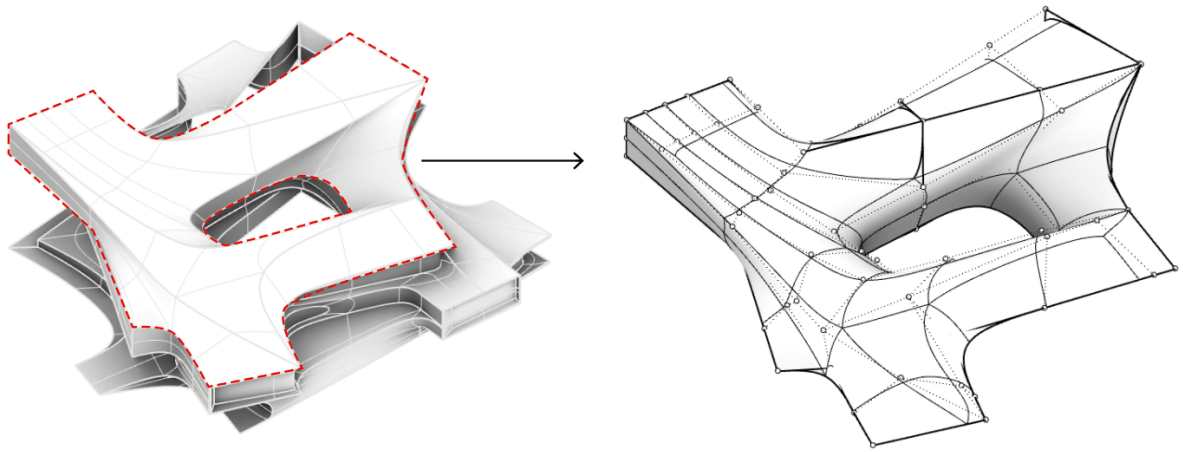


4.2.3.6. Case Study 2: Villa Roof

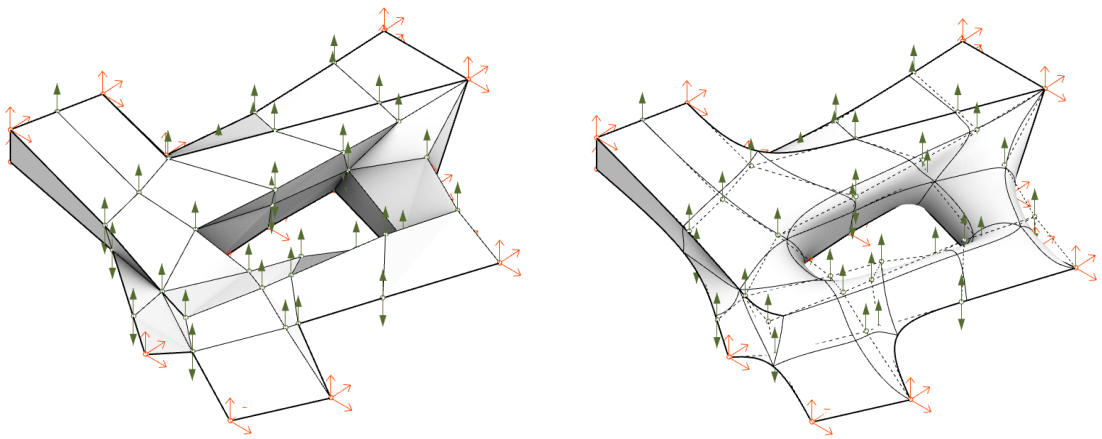
Base Problem

Following on from the two previous structural case studies, that were intentionally carried out on basic forms to verify the underlying accuracy of the algorithm, a third case study was initiated based on the more complex form of the Villa Project, that has been referenced in several previous deliverables.

As the whole building will not be a single structural shell the following case focuses on the roof of the villa and imagines it as a curved shell structure. This geometry was provided by the architects as a SubD, and a portion was selected, as shown in the image below, to serve as the base geometry for the process.

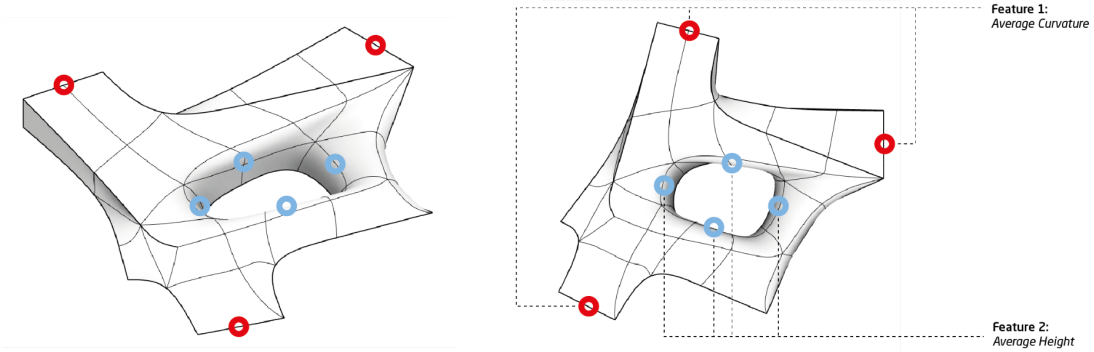


To work in a reasonable way within the QD process the geometry was simplified in order to reduce the complexity of the problem. Some additional control points were added to ensure the desired variability is included from a structural point of view, and in addition some redundant control points were removed. The final model with added point constraints and variations is shown in the images below. On the left the control mesh is shown and on the right the high resolution (smooth) geometry.



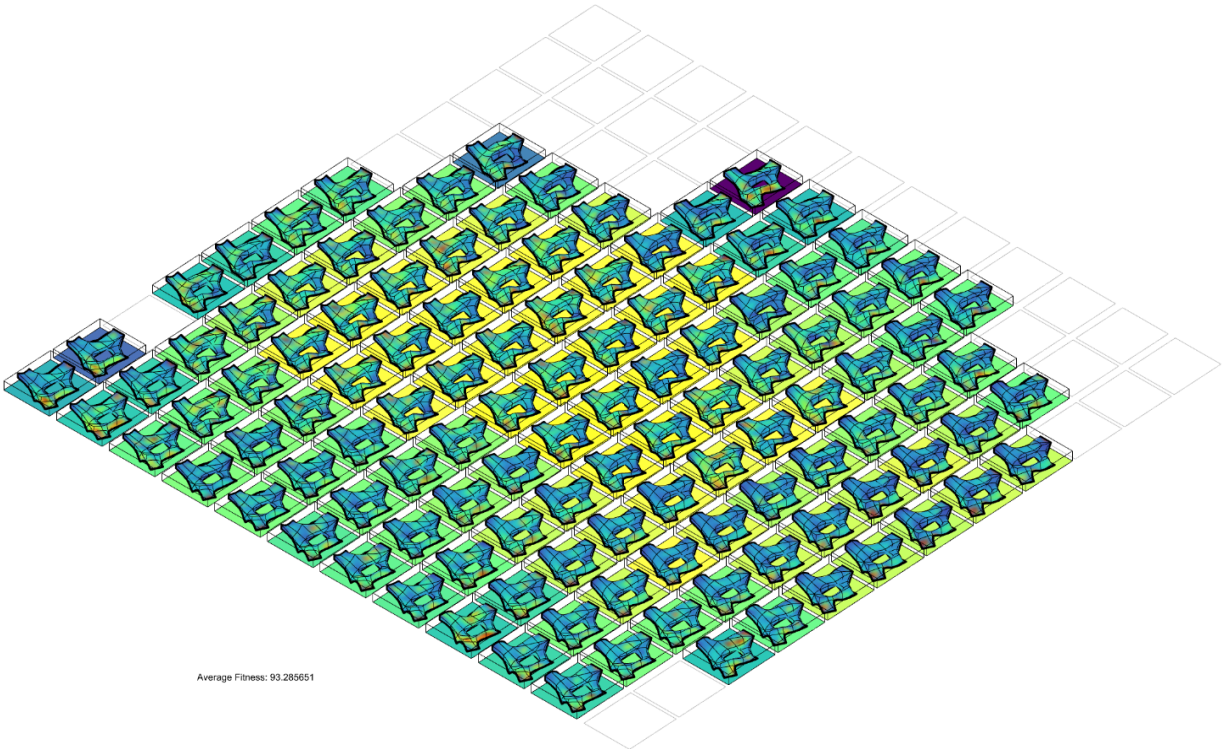
Features

Defining the features for the villa model is less obvious than the previous examples, as the shape and design of the villa is more refined and should not fundamentally change during the process. Therefore the features will be more subtle as the process will focus on a refinement of the design as opposed to the full generation of a shape or form. This is more akin to how a real world collaboration between architect and structural engineer would occur, as the engineer would likely work in response to an initial design.



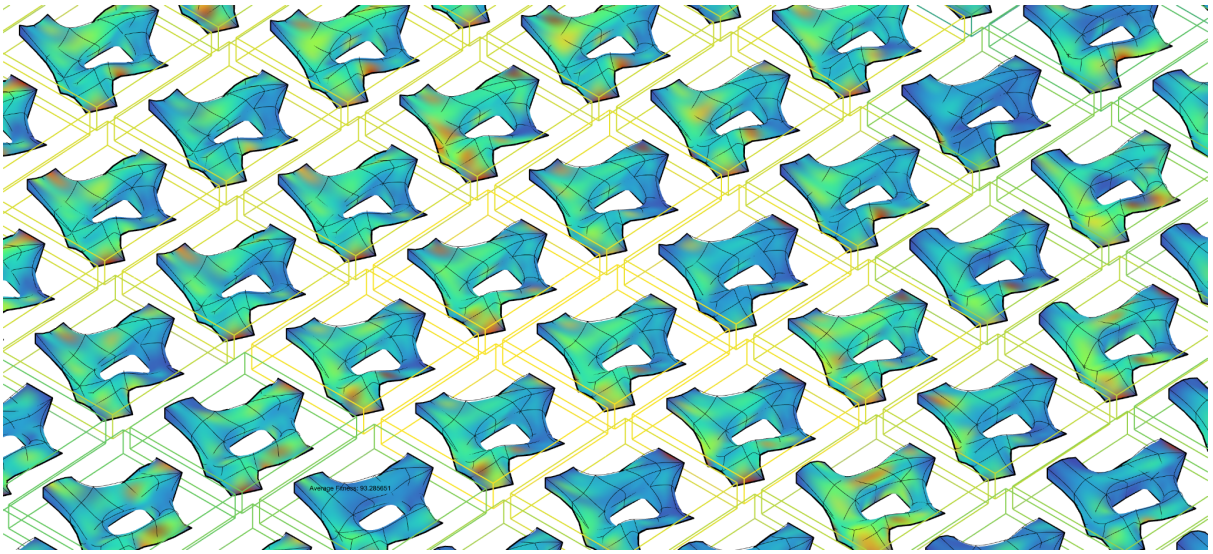
To capture the features of the villa model, the focus was therefore put on two areas which could have significant impact on both the architectural design aspects and the structural performance, which are the curvature of the framed window portions and the height of the middle “funnel”.

Results



The plot above shows the design map for the problem. It can be seen that the variations in each option is less distinct, however, zooming in and looking at the displacement plots for a

smaller selection of the shapes, we can see that variations are still noticeable across the different options. This is shown in the figure below, where the relative displacements of the high performing options are plotted. Red zones represent locations with the highest displacement for each option, and these zones clearly change location across different solutions - these studies therefore provide the engineer with a guide to refine the shape and improve structural performance.



Reflection

It should be noted that this exercise is not fully refined and its purpose is to serve as a prototype for exploring how the process could work in a real world scenario. Through this study a set of insights have been made, which provide direction for further research. The design insight generated within these solution maps is clear, as they can be used as a framework for negotiation between the various disciplines.
However, a geometry of the complexity found in the villa example is difficult to capture in two or three numerical parameters. The point-based evaluations have proven quite limited, and there is scope and need for further development of this area. To unlock the full potential, the QD model should ideally be able to incorporate additional aspects - for example, constructability - as part of the feature categorisation.

Furthermore, a geometry like the villa might require another level of complexity and may need to work with a greater number of features (3 or potentially even more) in various combinations, as all of the complex geometric aspects of a design might not be captured in their entirety in a single 2D map.

4.3. AI-Tool for MEP Engineering

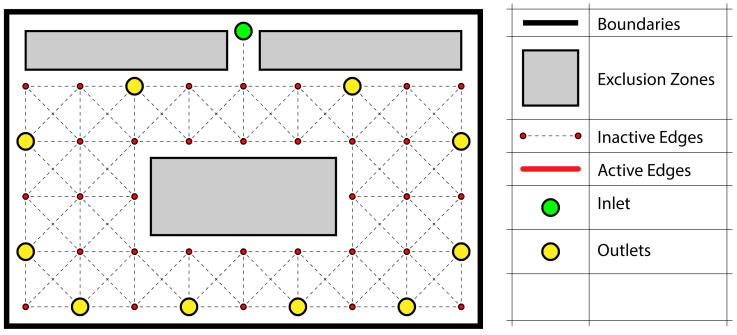
The AI-Tool for MEP Engineering focuses on a commonly addressed problem in MEP Engineering: the design of Ventilation Systems. This problem can be addressed as a problem of graph-traversal, in which case the main optimization aspects can be easily addressed by classical methods such as the Breadth-First-Search [16], Minimum Spanning Tree [17] or A* [18] algorithms. However, in this case study we are applying a constrained Quality-Diversity approach to this problem, thus allowing the MEP engineers to explore the distribution of fitness (which is the minimisation of cost / energy consumption of the system) along certain geometrical / topological characteristics of the system.

Section 4.3.1 presents all the technical details of the AI-Tool's implementation, while section 4.3.2 presents the application of this method to a case-study provided by the MEP engineers, using the custom-developed UI within the Rhino/Grasshopper environment.

4.3.1. Final Parametric Space of Design

4.3.1.1. Data-Structure

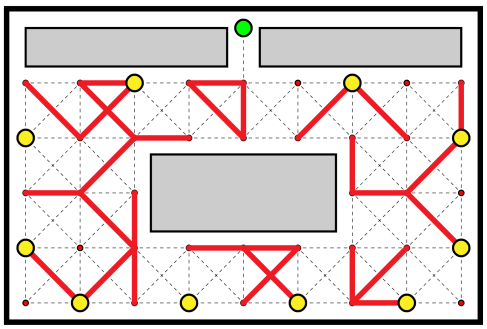
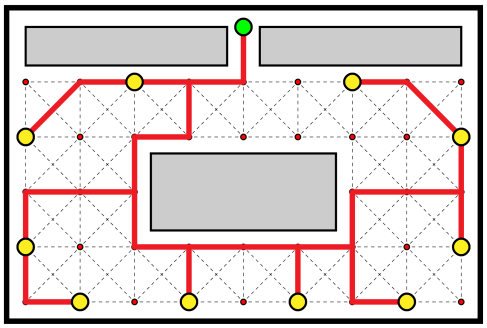
The DS__Mutable_Ventilation_System (MVS) data-structure includes the definition of a densely populated graph of potential connections (as shown in the following figure). The MVS's genotype is simply a boolean array that specifies which connections should be activated or deactivated. A generated instance of an MVS includes all the potential connections (edges) between points of the grid, as well as the positions of the inlet and outlets. The instance's genotype is simply a boolean array that activates or deactivates edges.



Problem specification of an MVS, including the outer boundary, the exclusion zones, a set of potential connections and the positions of the inlet and outlets.

4.3.1.2. Random Initialization

Random initialization is executed by randomly activating or deactivating all elements of a given MVS, with a 50% chance of every element being activated. The outcome of this random initialization process is most-probably going to result in an infeasible solution (one that does not satisfy the necessary constraints). However, through iterative mutation and selection, feasible solutions should gradually start appearing and covering the feasible archive.

	A random configuration of connections, resulting in a solution that does not satisfy any of the necessary constraints.
	A proper configuration of active edges, resulting in a feasible ventilation system.

4.3.1.3. Mutation

Mutation occurs in two stages: First, the destruction method randomly alters parts of the level without taking constraints into account. Then, the repair method attempts to bring the level back to a feasible state. The repair operations are semi-stochastic and do not guarantee feasibility, but increase the chances for it.

Destruction

The destruction method selects between 1 and 3 operations from the following list (chosen at random) and applies them on the selected DI, generating an altered, usually infeasible, offspring. The destruction operators target specific DI layers, noted in parentheses. Note that L1 destruction operations are only applied on Voronoi grids.

- **MM__Random_Edge_Flip:**
  Iterates through the MVS's edges and flips some of them at random, based on a user-specified probability (mutation rate).

Repair

The repair method attempts to bring a DI back to a feasible state by applying some or all of the following operations. The repair method does not

- **MM__Repair__Remove_Empty_Islands:**
  Iterates over the MVS's islands and eliminates the ones that do not include an inlet or outlet.
- **MM__Repair__Remove_Empty_Leaf_Nodes:**
  Removes all empty leaf-nodes. This operation is not recursive. I.e. after applying it, there may still be leaf-nodes remaining.

- **MM__Repair__Connect_Isolated_Inlets_Outlets:**
  Attempts to connect any isolated inlets or outlets, using the shortest possible path.

4.3.1.4. Constraints

The constraints imposed on the generative system are summarised, at a high-level, by the following rules: First, the solution must be a tree-structure, where the inlet is connected to all outlets without any loops and without any unused leaf-nodes. Second, any free junction can be (a) right angle, (b) obtuse angle, (c) straight line or (d) a T-junction. Third, any inlet or outlet junction can be (a) a straight line or (b) a leaf-node. Finally, the solution must contain no intersecting routes. Those constraints are the result of relevant technical consultation with the expert MEP Engineers (partners from Sweco).

The following list describes all constraints, as they are implemented in the computational framework.

- **CEM__Single_Island:**
  Returns true if the generated solution is a single island, containing all inlet and outlet nodes.
- **CEM__No_Cycles:**
  Returns true if the generated solution contains no cycles.
- **CEM__Proper_Pass_Through_Junctions:**
  Returns True if all pass-through junctions are proper. I.e. if they are of type (a) right angle, (b) obtuse angle, (c) straight line or (d) T-junction.
- **CEM__Proper_Inlet_Outlet_Junctions:**
  Returns True if all Inlet / Outlet junctions are of type (a) straight line or (b) leaf-node.
- **CEM__No_Intersections_Between_Edges:**
  Returns True if there are no intersections between edges.

4.3.1.5. Feasibility Score

The Feasibility Score attempts to capture the degree of feasibility of a solution in a gradient metric that can help the constraint solving algorithm to find feasible solutions. It is calculated as the average of the following metrics:

- **EM__CEM__Single_Island:**
  Returns a score in the range of [0…1], based on equation $S = 1/I$ where I is the total number of islands in the connectivity graph.
- **EM__CEM__No_Cycles:**
  Returns 1 if there are no cycles, or 0 otherwise.
- **EM__CEM__Proper_Pass_Through_Junctions:**
  Returns a score in the range of [0…1], based on equation $S = n/N$ where N is the total number of pass-through junctions and n is the number of proper pass-through junctions.
- **EM__CEM__Proper_Inlet_Outlet_Junctions:**
  Returns a score in the range of [0…1], based on equation $S = n/N$ where N is the total number of inlet / outlet junctions and n is the number of proper inlet / outlet junctions.
- **EM__CEM__No_Intersections_Between_Edges:**
  Returns a score in the range of [0…1], based on equation $S = 1 - (i \,/\, I)$, where i

is the number of edge-intersections found in the solution and I is the maximum number of edge-intersections (among all possible edges).

4.3.1.6. Fitness

There are four options for optimization, presented in the following list:

- **"Total length score"**
  - Implementation name: EM__Total_Length_Score
  - Description: Is calculated as $S = 1 - (L / L_{max})$, where L is the total length of active edges and L_{max} is the total length of all possible edges included in the problem specification.
- **"Adjusted total length score"**
  - Implementation name: EM__Total_Length_Score__Adjusted
  - Description: Is calculated as $S = 1 - \frac{L + P}{L_{max} + P_{max}}$, where where L is the total length of active edges, P is a penalty to length, calculated based on the types of active junctions, L_{max} is the total length of all possible edges included in the problem specification and P_{max} is the maximum penalty to length, calculated based on the types of junctions in the problem specification. The penalty to length is calculated as follows: Every square angle increases the total length by 1.2. Every 45 degrees angle increases the total length by 0.7. Every T-Junction increases the total length by 0.25.
- **"Farthest outlet distance score"**
  - Implementation name: EM__Farthest_Outlet_Distance_Score
  - Description: Is calculated as $S = \frac{L_{best-max-short}}{L}$ where $L_{max-short}$ is the best-case maximum shortest path between the inlet and all outlets, while L is the maximum shortest path between the inlet and all outlets for a specific MVS.

4.3.1.7. Diversity

The following list enumerates and explains the Behavioural Characterizations that have been implemented. All of them return a value in the range of [0…1] and can be used in any of the developed algorithms.

- **"Percent obtuse angle junctions"**
  - Implementation name: EM__Percent_Junctions__Obtuse_Angle
  - Description: Returns a value in the range of $[0 \ldots 1]$. It is calculated as $S = \frac{n}{N}$ where n is the number of obtuse-angle junctions and N is the total number of active junctions in the current MVS.
- **"Percent right angle junctions"**
  - Implementation name: EM__Percent_Junctions__Right_Angle
  - Description: Returns a value in the range of $[0 \ldots 1]$. It is calculated as $S = \frac{n}{N}$ where n is the number of right-angle junctions and N is the total number of active junctions in the current MVS.
- **"Percent straight or leaf junctions"**
  - Implementation name: EM__Percent_Junctions__Straight_Or_Leaf

- Description: Returns a value in the range of $[0 \dots 1]$. It is calculated as $S = \frac{n}{N}$ where n is the number of straight or leaf junctions and N is the total number of active junctions in the current MVS.

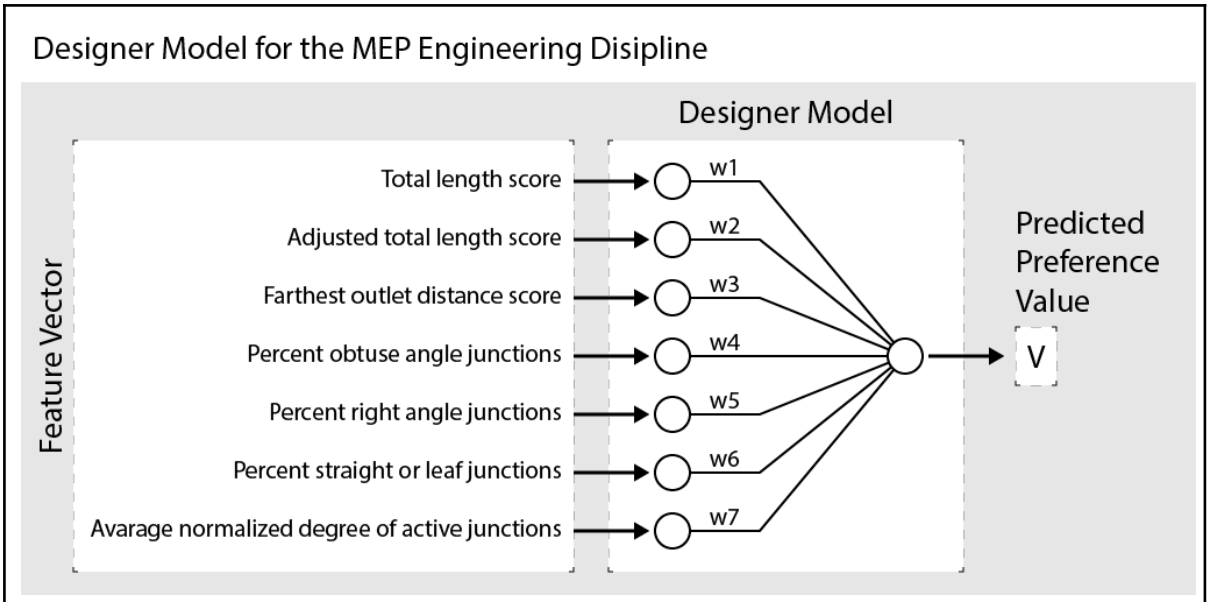
- **"Average normalised degree of active junctions"**
  - Implementation name: EM__Active_Junctions__Average_Degree__Normalized:
  - Description: Returns the average normalised degree of all active junctions. It is calculated as $S = \frac{1}{N} * \sum_{n=1}^{N} D(n)$, where N is the total number of active junctions, n is an active junction and $D(n)$ is the normalised degree of junction n. The normalised degree of a junction is calculated as $D(n) = \frac{e}{e_{activre}}$ where e is the number of edges connected to that junction and e_{active} is the number of active edges connected to that junction.

4.3.1.8. Designer Model

For the MEP engineering discipline, the Designer Model's input is a 7-dimensional feature-vector, that characterises an MEP design implementation, while its output is a single value that approximates the designer's preference for that design. The model's architecture is illustrated in the following figure. Importantly, the model's feature space consists of the following 7 Features, including Fitness Functions and Behavioural Characterizations (the detailed description of which can be found in Sections 4.3.1.6 and 4.3.1.7):

1. Total length score
2. Adjusted total length score
3. Farthest outlet distance score
4. Percent obtuse angle junctions
5. Percent right angle junctions
6. Percent straight or leaf junctions
7. Average normalized degree of active junctions

For the MEP Engineering discipline, the Designer Model is to be used as a Behavioural Characterization, which is updated in-between sessions, based on the Designer's recorded preferences at the end of each session. More details can be found in Section 4.3.2.5.



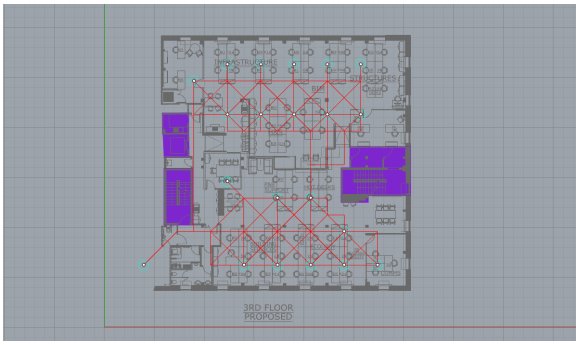
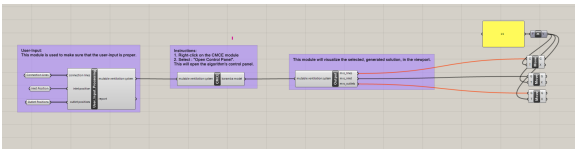
A functional diagram of the Designer Model for the MEP Engineering Discipline.

```
{
  "weights": [
    0.10252656838974294,
    -0.5571194065535066,
    0.9893482709253898,
    0.4951349941525305,
    0.038948392979311075,
    0.37489226151951227,
    -0.6305013315894181,
    0.038948392979311075
  ]
}
```

An example of a serialised Designer Model for MEP engineering. The model is represented as a list of 7 numbers, each of which corresponds to one of the Model's weights.

4.3.2. Interactive, DM-based QD for MEP Engineering

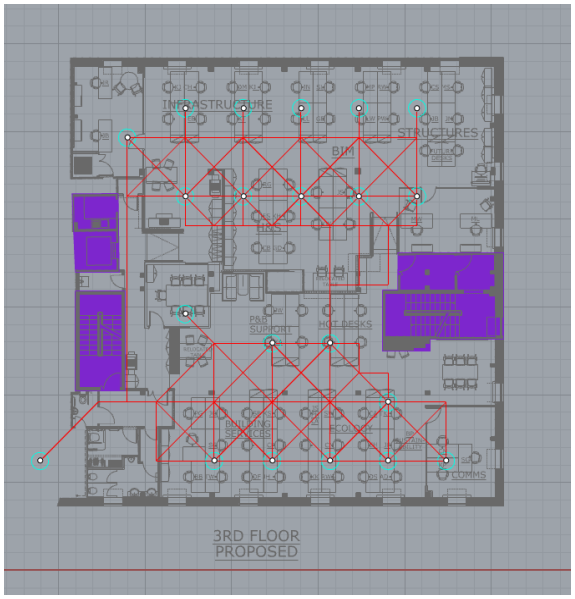
This section describes how the AI-Tool for MEP Engineering has been implemented within the Rhino / Grasshopper environment, thus allowing the user to explore the solution space of their design problems. The section is based on a case study that has been prepared in Rhino/Grasshopper, in cooperation with the MEP partners (Sweco).

Case study file screenshots	
	Screenshot of the relevant Rhino file, which includes the architectural design drawing as a background, as well as the definition of potential routes of the Ventilation System.
	Screenshot of the relevant grasshopper script, which processes the MEP engineer's input and initiates the AI-Tool for MEP Engineering.

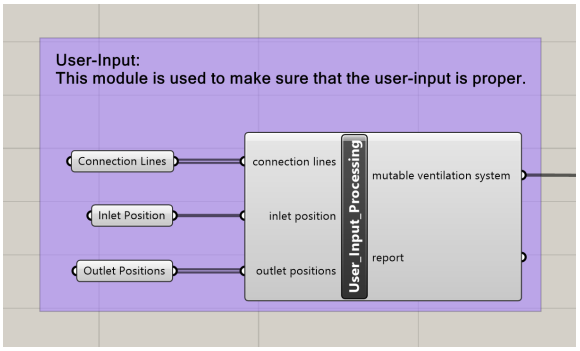
4.3.2.1. Specifying a Design Problem

The specification of a Design Problem (i.e. a Mutable Ventilation System / MVS), is executed within the Rhino / Grasshopper design software. The user must define an input-sketch, by using the native tools available in the Rhino design software. The input sketch will then be converted into an MVS so that diverse solutions can be explored. In more detail, the user must use line-segments to represent potential connection lines, and points to represent the inlet and outlets, as illustrated below. The input-sketch must include a relatively large number of connection-lines, so that the QD algorithm can explore a large number of alternative solutions. Importantly, the input-sketch must satisfy the following criteria, in order to be usable by the system:

- It must include at least one connection line.
- It must contain exactly one inlet.
- It must contain at least one outlet.
- It must not contain any duplicate connection lines.
- It must not contain any disconnected lines.
- Inlet and outlet positions must be unique.
- The orientation of all connection-lines must be a multiple of 45 degrees. I.e. all connection-lines must be either horizontal, vertical, or diagonal.
- The connection lines must form a connected graph. I.e. there must be no disconnected "islands" in the initial connectivity diagram.

	An example definition of a Mutable Ventilation System in Rhino's design environment.

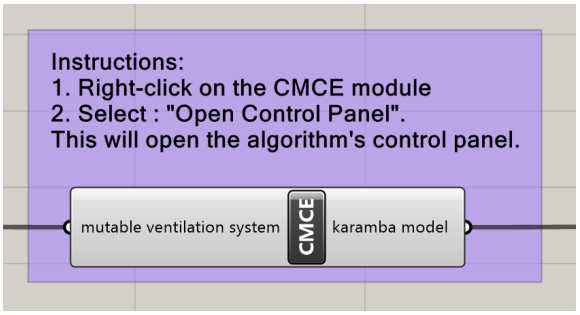
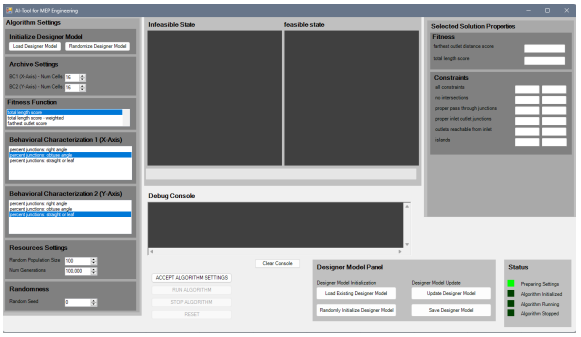
As soon as the Designer has prepared the input-sketch, taking into account the aforementioned requirements, they "feed" it to the "User_Input_Processing" node of the grasshopper script. This node will make sure that the requirements are satisfied and will inform the user if something has gone wrong. If all criteria are indeed satisfied, the node will output an MVS, ready to be processed by the QD algorithm.

	The "user-input-processing" component that pre-processes the user's input-sketch.

4.3.2.2. Algorithm Initialization

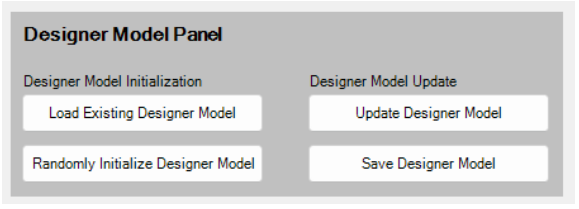
As soon as the input-sketch has been properly specified and processed, the designer can initialize the interactive QD algorithm to start exploring the solution space. All algorithm settings are accessible through a custom-designed user interface that can be accessed through the "CMCE" grasshopper node. The user can right-click the CMCE node and select "Open Control Panel" to bring the algorithm's control panel into focus. The following subsections describe all algorithmic options and parameters that are accessible to the designer, through the algorithm's control panel.

Screenshots of the main Grasshopper component and the AI-Tool's control-panel.

	
Screenshot of the main component of the AI-Tool that hosts the algorithm's control-panel.	Initial State of the algorithm's control panel

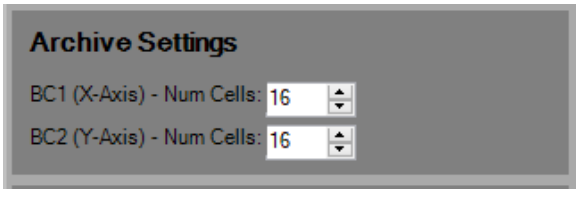
Loading an existing Designer Model

If a Designer Model already exists, based on previous interactions of the designer with the system, the designer may load this model and use it as one of the two Behavioural Characterizations. Loading the model is quite straightforward: The user clicks the "Load Designer Model" button and an open-file dialog appears that enables them to navigate to the serialised model's location on disk.

	Screenshot of the panel that allows the user to load an existing designer model.

Defining Archive Resolution

The archive's resolution is an important aspect of the *FI-MAP-Elites* algorithm. The number of subdivisions for each dimension of the Behavioural Space are accessible to the designer. A predefined value of 16 cells is initially set to both dimensions, but the optimal values are to be found by the designer themselves, through experimentation.

	Screenshot showcasing the input-fields for defining the archive's resolution along two Behavioural Characterizations.

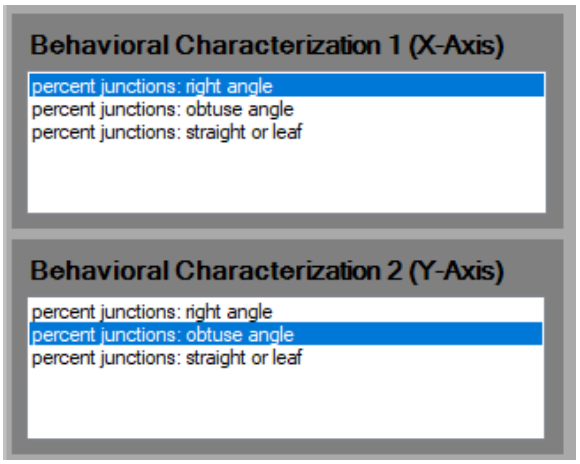
Selecting a Fitness Function

All three implemented fitness functions are available for the designer to select from: 1) Total length score, 2) Weighted Total Length Score and 3) Farthest Outlet Score. The user can select any one of them, and the selected one will be used as the fitness function of the feasible population.

	Screenshot showcasing the drop-down list for selecting a Fitness Function.

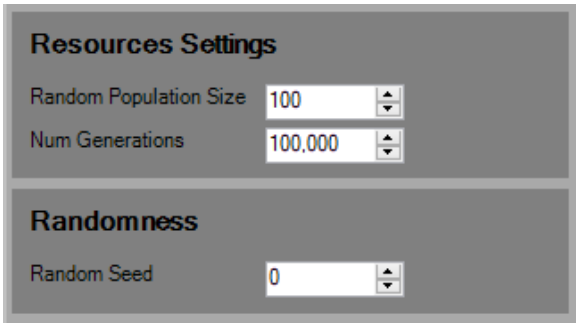
Selecting Behavioural Characterizations

All available Behavioural Characterizations are available for the Designer to choose from, as illustrated in the following figure. The designer can select the existing (loaded or randomised) Designer Model to be used as one of the two Behavioural Characterizations. Should they choose to do so, the one of the two axes of the archive will diversify solutions based on the designer model's evaluation criteria.

	Screenshot of the Behavioral Characterizations panels, where the designer may select the two design-dimensions to explore.

Managing Algorithm Resources

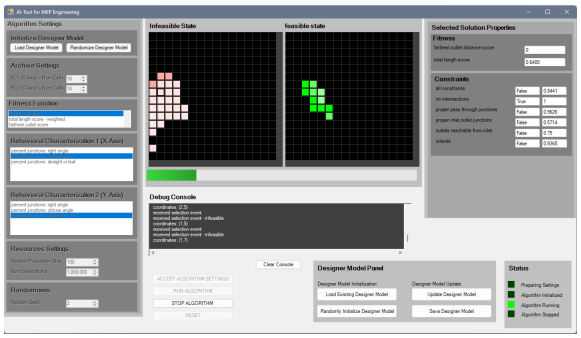
The designer can manage the algorithm's computational resources (as shown in the following figure): 1) the size of the initially generated population and 2) the total number of generations that the algorithm will execute. Furthermore, they can select a random seed, which guarantees repeatability of an experiment, should all other settings be the same.

	Screenshot of the algorithm-resources panel and the randomness panel.

4.3.2.3. Algorithm Execution

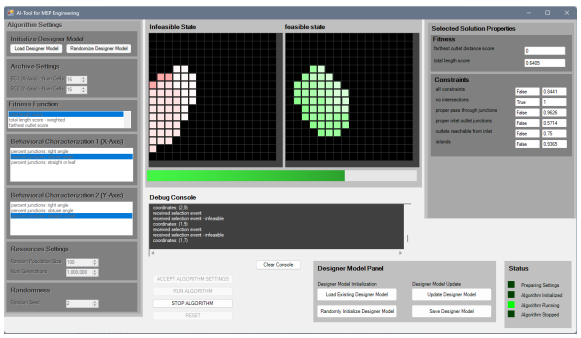
As soon as the designer has selected their preferred algorithm settings, they must hit the "Accept Algorithm Settings" button. This will initialise the back-end and prepare the feasible

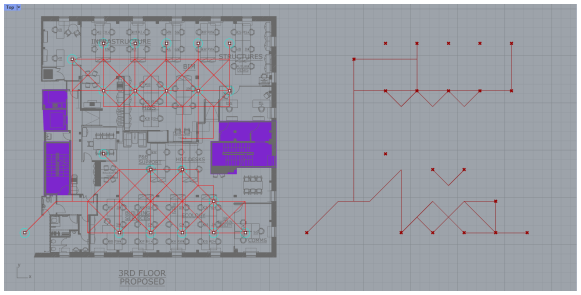
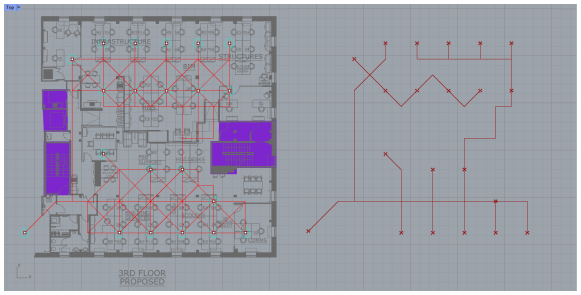
and infeasible archives of the algorithm in the control-panel. Then, the designer can hit the “Run Algorithm” button so that the algorithm’s actual operation can actually begin.
During the algorithm’s operation, the archives’ contents are visualised in real-time both in the control panel as well as in the viewport. The infeasible solutions are visualised in tones of red, ranging from pure red for the most infeasible solutions to almost pure white for the least infeasible ones. The feasible solutions are visualised in tones of green, ranging from pure green for the least fit solutions to pure white for the most fit ones. The archives’ visualisation in the viewport also visualises feasibility and fitness by using the 3rd dimension, for an easier overview of the archives’ contents.

	Screenshot of the algorithm’s control panel, during the initial stages of the algorithm’s execution, where few feasible solutions have been found.

4.3.2.4. Inspection of Results

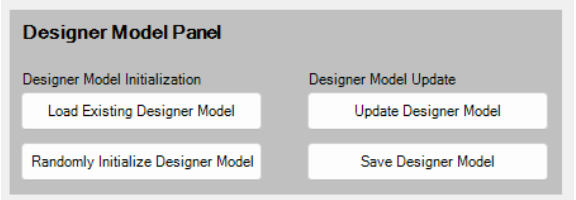
As soon as the algorithm’s operation is over, the designer can manually inspect the results in the following way: By clicking on any occupied cell of either the feasible or the infeasible archive, the system will visualise the corresponding solution in the viewport, as shown in the following figure. Furthermore, when a solution is selected, a number of measurable characteristics of that solution are also shown for inspection in the Selected Solution Properties panel. This can help the designer to make well-informed decisions about the solution at hand.

Inspection of results: examples of a feasible and an infeasible solution.	
	Screenshot of the algorithm’s control panel, during the final stages of the algorithm’s execution, where a fair amount of feasible solutions have been found.

	An example of an infeasible solution. The mutable ventilation system can be seen on the left, while the infeasible solution is shown on the right.
	An example of a feasible solution. The mutable ventilation system can be seen on the left, while the feasible solution is shown on the right.

4.3.2.5. Designer Model Update

After the algorithmic process is over, and the designer has taken some time to inspect the generated results, the designer may choose to actively participate in a process to either update an existing model or generate a new one, based on their preferences.
In order to initiate this process, the user must click the “Update Designer Model” button. Once they do so, a special session begins: The user is asked to select their preferred solutions, out of the currently available, feasible ones. The user’s explicitly stated preferences are converted into a set of preference pairs, which are used to update the existing model. Once the training process is complete, a dialog appears that lets the user save the new model, either overwriting the old one or not.

	Screenshot of the panel that allows the user to update and save the designer model.

5. Cross-Discipline Integration

The previous chapters showcased how the developed AI-Tools apply to each discipline individually. However, the design process is also integrated in a cross-disciplinary fashion, aligning with the vision of PrismArch to *“intersect the parallel words of the AEC industry into a common platform that will promote decision making and ultimately reshape the design process in architectural projects”*.

This integration and cross-disciplinary interaction paradigm, described below, has been devised to address practical concerns regarding feasibility of software development, end-user feedback in workshops, meetings and surveys. Moreover, this interaction paradigm is beneficial for the extendability of PrismArch algorithms beyond what is currently available in future years (see Section 6).

Central to the cross-discipline integration of the AI algorithms is the Speckle (https://speckle.systems/) data-base, the cloud-based repository where all speckle data within PrismArch are stored. The cloud-based repository contains both actual AEC content (as it is a general-purpose data format specifically tailored for 3D data) and metadata including designer models. The benefit of the speckle cloud is that it allows data to be stored centrally but only accessed by members with appropriate privileges. This strengthens the requirements of IP protection, which have been raised by AEC partners as a very important aspect in the context of cooperative projects, where partners from different companies / organisations may be involved. Essentially the system allows a user to share specific versions of content with specific people, disciplines, or organisations. Intuitively, a user would keep draft AEC content such as plans as private and share them with select colleagues (across disciplines) once the plans are completed in a satisfactory fashion.

Moreover, the speckle cloud allows for the persistent storage of designer models, which are marked private by default and can not be directly accessed by any user. Storing the designer model on the speckle cloud ensures that personalised AI feedback can be provided anywhere, since the system would always load this designer’s model every time AI interaction begins.

Connected to the above, it is important to point out that the metadata of interaction between the designer and the AI tool are not stored locally at any time. All the data generated by the designer’s interaction with the AI tool only exists temporarily on the end-user’s PC. Any type of persistent data (including designs, problem specifications, metadata and user models) is stored in the Speckle database as content of the user that generated them, and are only cached on the end-user’s PC during interaction (to speed up computational processes) and deleted after the interaction is completed. Thus the protection and / or sharing of AI - related information aligns to the general approach of PrismArch, in regard to IP, copyright, protection and controlled sharing of information during cooperation.

As a final note, the integration envisioned via the Speckle Cloud can work for asynchronous communication between disciplines, as different users may need to interact with the project at different timeframes. This asynchronous communication is vital since the AI interactions for some disciplines are in VR while for others they are within Rhino/ Grasshopper. However,

it should be noted that the Speckle Cloud is always available and the communications via this pipeline can easily work synchronously as well.

5.1. Cross-Discipline cooperation scenario:

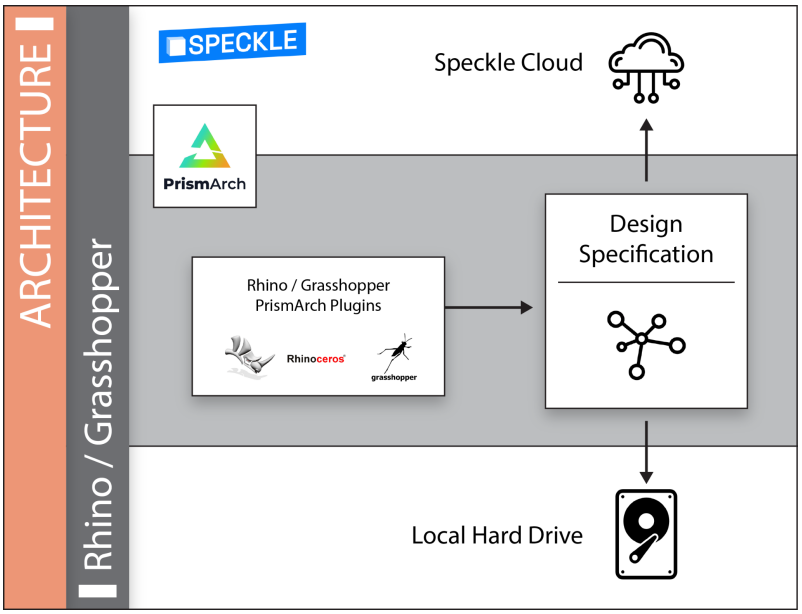
The following sections describe a scenario of cooperation between the three involved AEC disciplines: Architecture, Structural Engineering and MEP Engineering. We present a linear scenario, where the Architects propose an initial plan, then Structural Engineers develop a structural system to support it and, finally, MEP Engineers develop a ventilation system based on the constraints coming from both other disciplines. The focus of this storyline is to showcase how information can flow across disciplines, based on the shared database of the PrismArch platform, as well as how the AI-Tools are involved in this process. The linear flow of the scenario is selected for the sake of simplicity, but is not restrictive. In reality, any order of data exchange between partners is possible.

5.1.1. Architecture:

The following subsections present the process of an Architect working on an architectural layout, during the initial stages of design, starting from scratch and utilising the AI-Tool for architecture.

5.1.1.1. Step 1: Architectural Design Specification

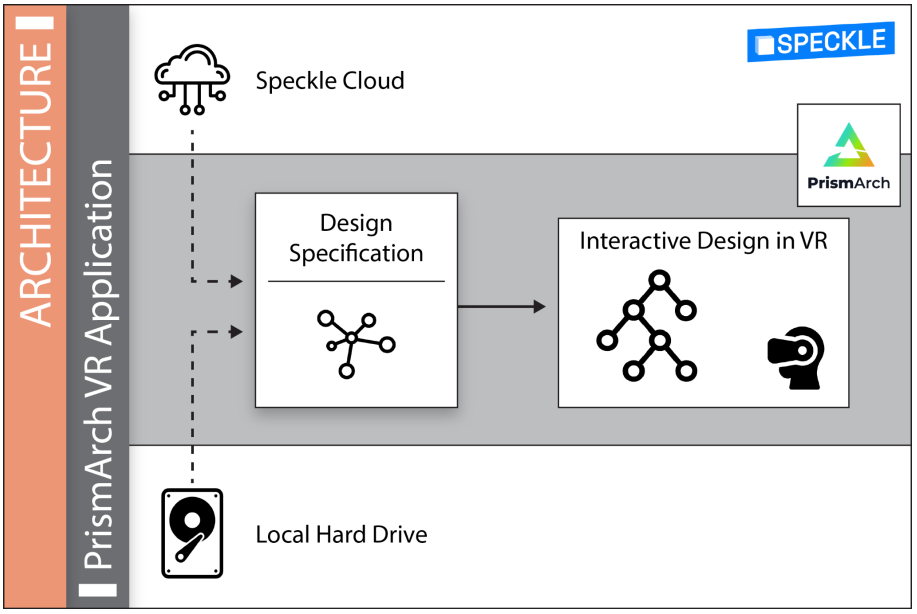
An architect can define a Design Specification via the connectivity graph tool, in the Rhino / Grasshopper environment (see section 4.1.2.1 for more details). As soon as they conclude its definition, they can either store it locally, or upload it on PrismArch's Speckle-based cloud.



5.1.1.2. Step 2: Interactive Architectural Design in VR

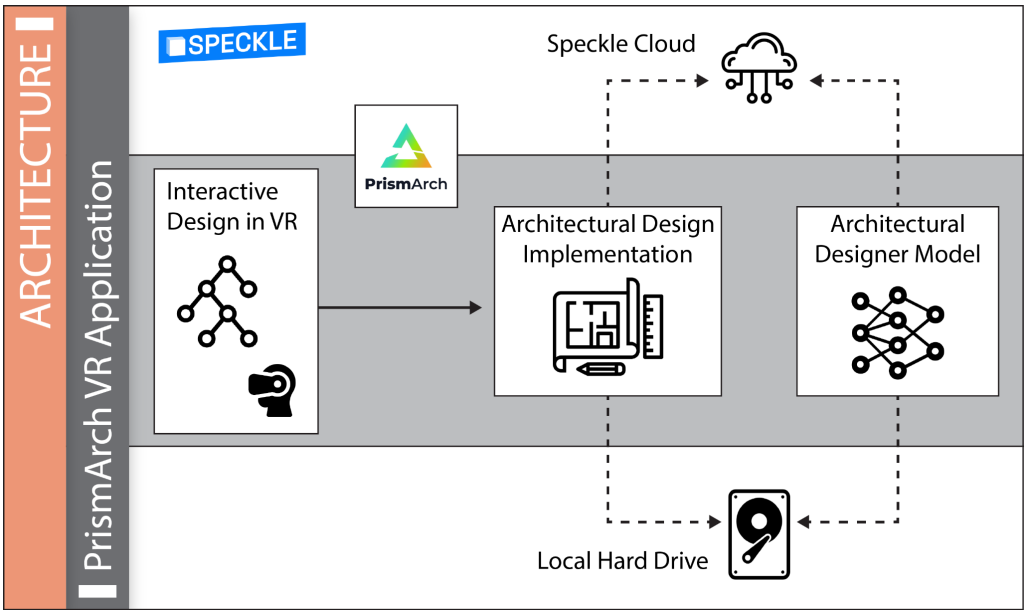
Once a Design Specification is available, the Architect can engage in a VR-based interactive exploration of the problem's design-space, using the AI-Tool for Architecture (see Section 4.1, as well as deliverable D4.3). During this process, the underlying algorithm will keep

offering diverse alternatives to the architect, while continuously updating a Designer Model that attempts to capture the architect's subjective preferences.



5.1.1.3. Step 3: Saving a Design Implementation and / or a Designer Model

At any time, the designer may choose to save a specific generated Design Implementation to Speckle, for later use, or for sharing with partners. Similarly, the designer may also choose to save the current snapshot of their personal Designer Model, so as to continue using it at a later session.

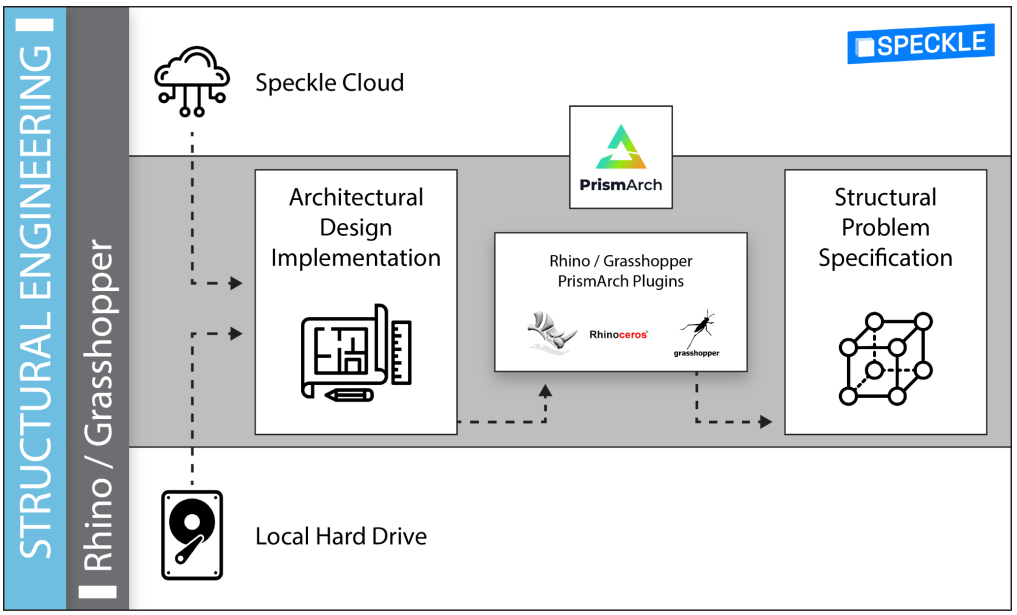


5.1.2. Structural Engineering

The following subsections present the process of a Structural Engineer approaching a structural design problem based on a preexisting architectural design.

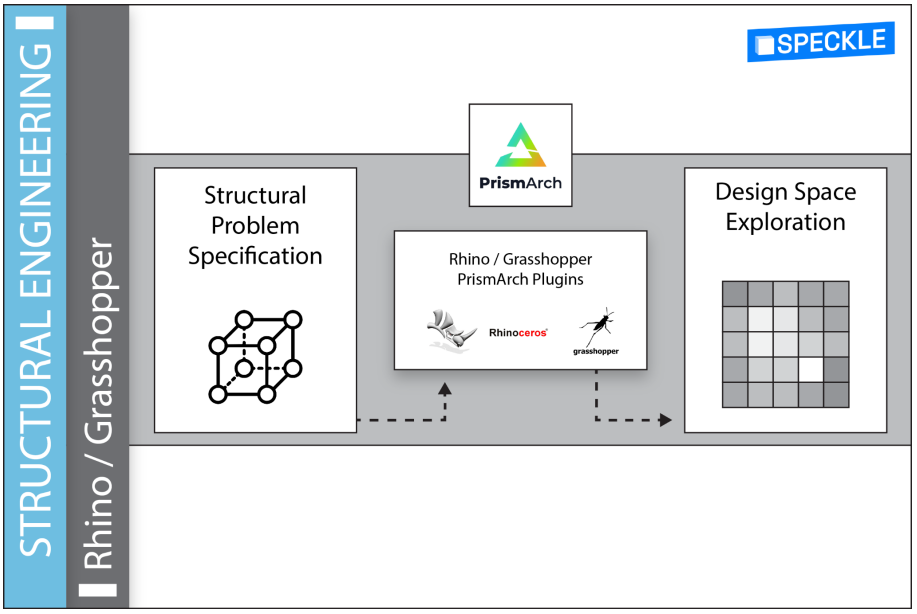
5.1.2.1. Step 1: Structural Problem Specification

Based on the architectural design that the Structural Engineer has access to, they may start working on a structural problem specification. Depending on the problem at hand, they may select to move forward with a stick-model approach, such as the one described in section 4.2.2, or a surface-based approach, such as the ones described in section 4.2.3.



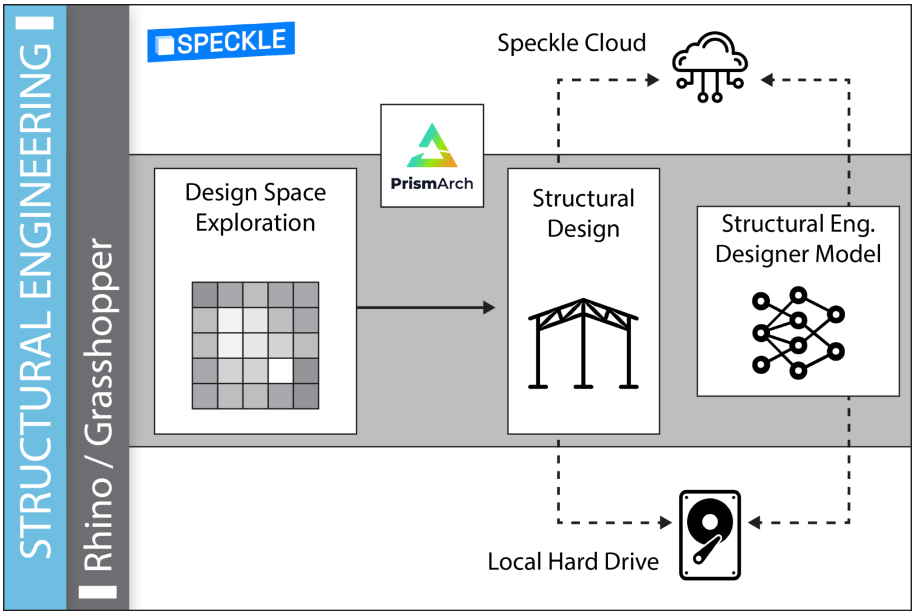
5.1.2.2. Step 2: Exploring the Design Space

As soon as the Structural Engineer has properly defined a problem definition, they may assign the exploration of the design-space to the AI-Tool for Structural Engineering. At this stage they may run a number of computational experiments, using variations of the QD algorithm's settings, as well as opt in for using a Designer Model to let the system capture and utilize their personal preferences.



5.1.2.3. Step 3: Saving a Structural Design and / or a Designer Model

After reviewing the design-space, as it has been illuminated by the QD algorithm, the Structural Engineer may come across one or more structural designs which they consider valuable and choose to save one or more of them for later use. They can save their solutions locally, or directly upload them to PrismArch's Speckle-based cloud. The same applies to the generated designer-model. It can also be stored either locally, or on the Speckle-based cloud.

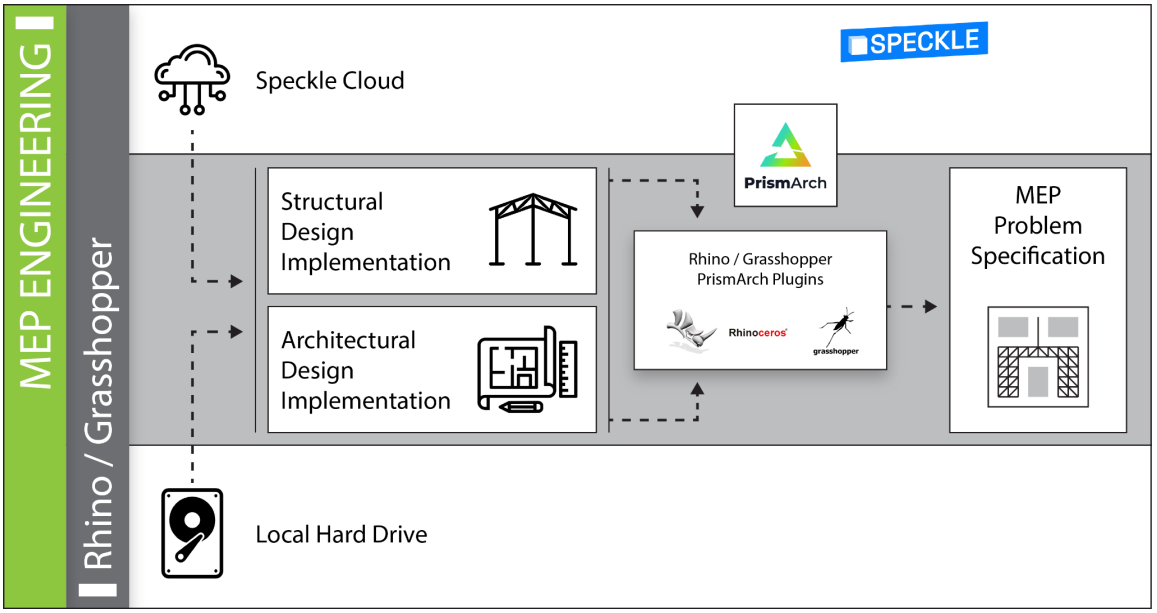


5.1.3. MEP Engineering

The following subsections present the process of an MEP Engineer approaching a structural design problem based on preexisting architectural and structural designs.

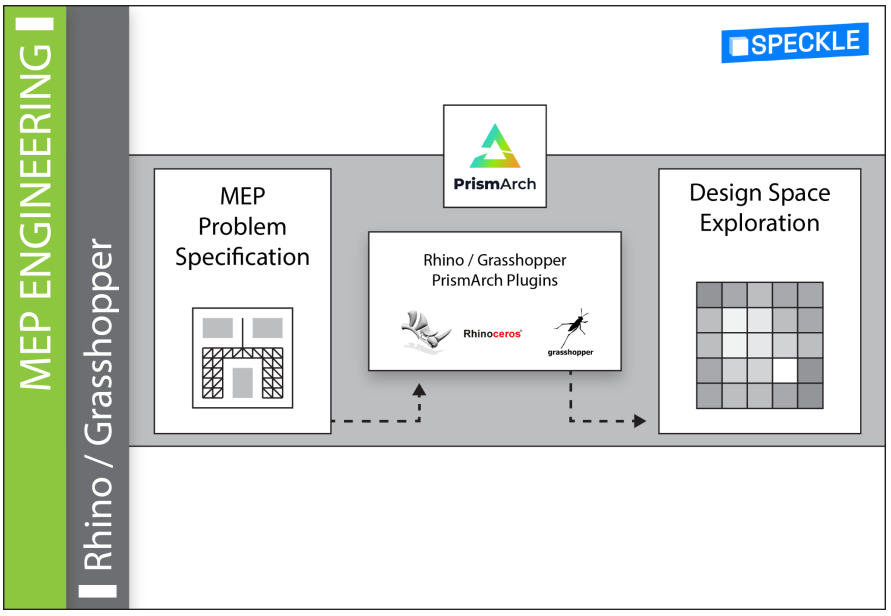
5.1.3.1. Step 1: MEP Problem Specification

Based on the architectural and structural design that the MEP Engineer has access to, they may start working on an MEP problem specification (for example a Mutable Ventilation System, as explained in section 4.3.1). The engineer can then use the available designs to form a problem specification.



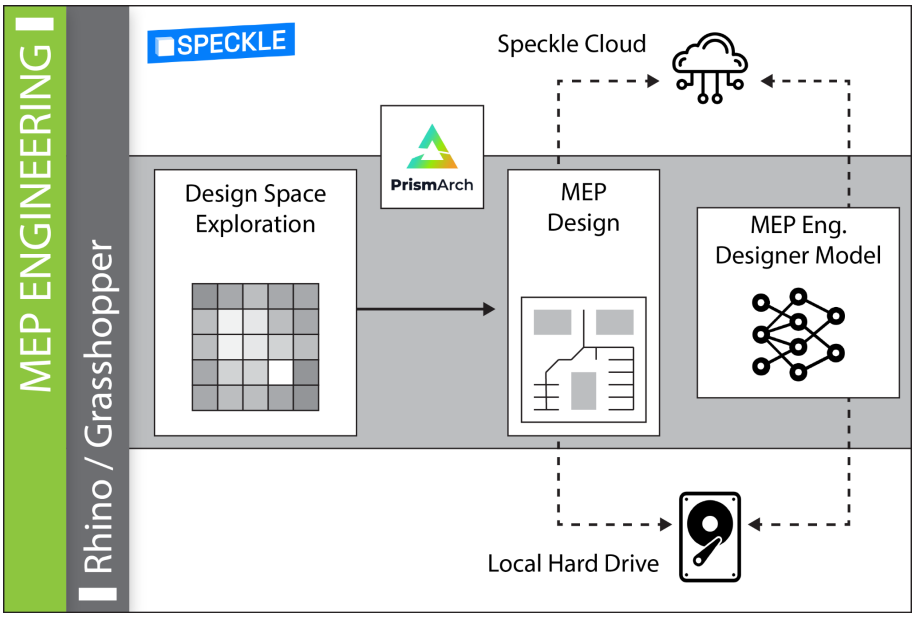
5.1.3.2. Step 2: Exploring the Design Space

As soon as the MEP Engineer has properly defined a Problem Specification, they may assign the search of the solution-space to the AI-Tool for MEP Engineering. During this process they may use variations of the QD algorithm's settings, as well as opt in or out of using a Designer Model to let the system capture and utilize their personal preferences.



5.1.3.3. Step 3: Saving an MEP Design and / or an MEP Designer Model

After reviewing the design-space, as it has been illuminated by the QD algorithm, the MEP Engineer may come across one or more structural designs which they consider valuable and choose to save one or more of them for later use. They can save their solutions locally, or directly upload them to PrismArch's Speckle-based cloud. The same applies to the generated designer-model. It can also be stored either locally, or on the Speckle-based cloud.


MEP ENGINEERING
Rhino / Grasshopper
SPECKLE
Speckle Cloud
PrismArch
Design Space Exploration
MEP Design
MEP Eng. Designer Model
Local Hard Drive

6. Conclusions

This final documentation captures the revised version of the parametric space of design across all disciplines, algorithms for AI assisted editing/design in VR, and algorithms for designer modelling.

The revised parametric space of design, originally presented in [D2.1], has been expanded based on close collaboration with AEC partners in numerous workshops and surveys. The algorithmic contributions on artificial intelligence (detailed in Section [3]) were revised from the first version documented in [D2.2] as the design and development of algorithms continued throughout the course of PrismArch, as new designer modelling methods were introduced, and as the user requirements in terms of interactivity coalesced through interactions with developers (CERTH), user experience researchers (ETH), and AEC end-users (ZH, AKT, SWECO).

In that regard, the protocol for cross-discipline integration, discussed in Section [5] was established based on the decisions taken during the project (including, for instance, the use of Speckle as the overarching format and the Speckle Cloud as the way of communication and synchronisation of the different components). Therefore, the final state of AI contributions and AI integration detailed in this deliverable was reached after multiple cycles of iterative design: in each cycle we collected or revised user needs, designed the interaction paradigm with user experience researchers and end-users, developed the individual AI algorithms or interfaces, and integrated them into the PrismArch software. Through this participatory design process, we ensure that novel AI algorithms are introduced into the field of computer-aided design but more importantly that these AI algorithms are accessible, understandable, and useful to the end-users that will be taking advantage of them through the PrismArch platform and beyond (e.g. through the Rhino plugins developed).

Due to the modular way in which the AI algorithms and interfaces are designed and implemented, there are many additional uses for the software described in this deliverable beyond (and after) PrismArch. First of all, the plugin for Rhino/ Grasshopper developed as part of [D2.3] can be used outside the design pipelines envisioned in PrismArch. The fact that such plugins are modular and do not rely on expensive VR hardware or an external database (although it is currently used in conjunction with the above) means that any designer can use them for their needs with minimal training and no additional costs. Moreover, since the Rhino/Grasshopper plugin uses a language that most AEC practitioners are familiar with, it can be easily extendable with more nuanced representations and behaviour characterizations. Already over the course of PrismArch, structural engineering partners have made their own modifications to the plugin, based on their needs and preferences, which have been integrated into the algorithm (see Section [4.2.3]). More additions and user-led edits are expected over the next few years, and there are plans to make the Rhino / Grasshopper plugins available in plugin repositories such as the official McNeel software repository (https://www.food4rhino.com).

Another indication of the extendability of the library is the way that different modules communicate through the Speckle Cloud (see Section [5]). This makes development of new tools or use of existing plugins beyond those developed during Prismarch much easier.

While the goal of PrismArch is to provide a common place where all disciplines can meet and collaborate, the ability to work asynchronously (or even independently) as needed and with the tools that are needed is also satisfied with the cloud-based interactions of the different components. Finally, we envision that the designer modelling and design space representations can be exploited beyond the needs of AEC: an obvious domain where such tools could be useful would be for content generation in digital games, which has already been explored in a preliminary fashion [14]. Moreover, designer modelling algorithms can be used in other creativity support tools, including game level editors [7], fashion design concept [15] or other AI art tools.

7. References

[1] Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. "Adapting models of visual aesthetics for personalised content creation." IEEE Transactions on Computational Intelligence and AI in Games 4.3 (2012): 213-228.

[2] Antonios Liapis, Hector P. Martinez, Julian Togelius and Georgios N. Yannakakis. "Adaptive game level creation through rank-based interactive evolution." 2013 IEEE Conference on Computational Intelligence in Games (CIG). IEEE, 2013.

[3] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[4] Rumelhart, David E., et al. "Backpropagation: The basic theory." Backpropagation: Theory, architectures and applications (1995): 1-34.

[5] Bai, Bing, et al. "Learning to rank with (a lot of) word features." Information retrieval 13.3 (2010): 291-314.

[6] Fürnkranz, Johannes, and Eyke Hüllermeier. "Preference learning and ranking by pairwise comparison." Preference learning. Springer, Berlin, Heidelberg, 2010. 65-82.

[7] Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. "Designer modeling for sentient sketchbook." 2014 IEEE Conference on Computational Intelligence and Games. IEEE, 2014.

[8] Liapis, Antonios, Georgios Yannakakis, and Julian Togelius. "Designer modeling for personalized game content creation tools." Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. Vol. 9. No. 2. 2013.

[9] Mouret, Jean-Baptiste, and Jeff Clune. "Illuminating search spaces by mapping elites." arXiv preprint arXiv:1504.04909 (2015).

[10] Steven Orla Kimbrough, Gary J. Koehler, Ming Lu, and David Harlan Wood. 2008. On a Feasible–Infeasible Two-Population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. European Journal of Operational Research 190, 2 (2008), 310–327.

[11] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. “Talakat: Bullet hell generation through constrained map-elites”. In Proc. of the Genetic and Evolutionary Computation Conf. 1047–1054.

[12] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. “Towards a Generic Method of Evaluating Game Levels”. In Proc. of the AIIDE Conf.

[13] Nathan Sorenson, Philippe Pasquier, and Steve R. DiPaola. 2011. “A generic approach to challenge modeling for the procedural creation of video game levels”. IEEE Trans. on Computational Intelligence and AI in Games 3, 3 (2011), 229–244.

[14] Konstantinos Sfikas, Antonios Liapis and Georgios N. Yannakakis. “A General-Purpose Expressive Algorithm for Room-based Environments”. in Proceedings of the FDG workshop on Procedural Content Generation, 2022.

[15] Grabe, M. González-Duque, S. Risi, and J. Zhu. “Towards a framework for human-AI interaction patterns in co-creative GAN applications”. CEUR Workshop Proceedings, 3124, 2022.

[16] Catmull, E. Clark. J “Recursively generated B-spline surfaces on arbitrary topological meshes". Computer-Aided Design. 10

[17] Breadth-First Search (BFS) algorithm https://en.wikipedia.org/wiki/Breadth-first_search

[18] Minimum Spanning Tree algorithm https://en.wikipedia.org/wiki/Minimum_spanning_tree

[19] A* (A-star) search algorithm https://en.wikipedia.org/wiki/A*_search_algorithm