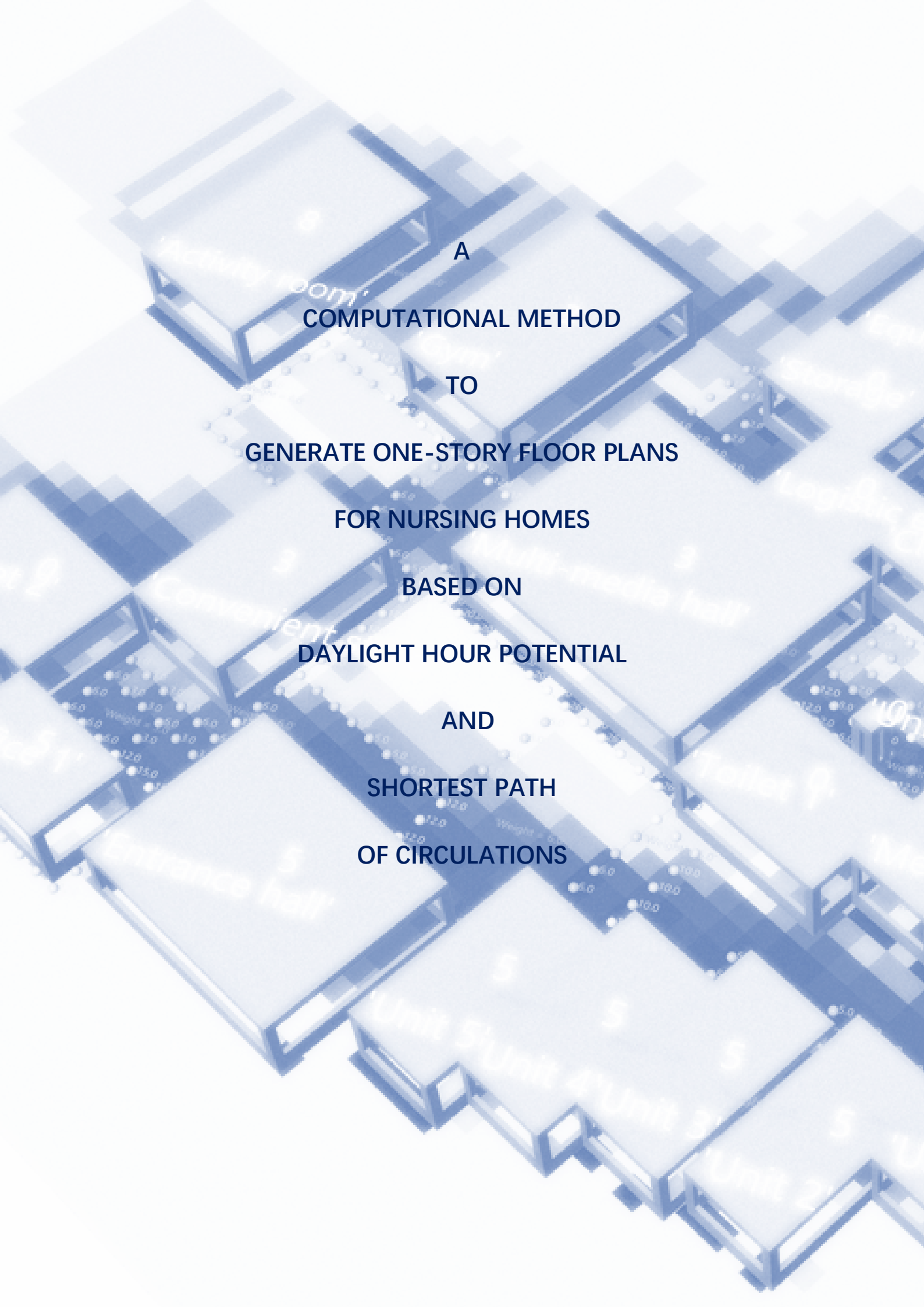


A COMPUTATIONAL METHOD TO GENERATE ONE-STORY FLOOR PLANS FOR NURSING HOMES BASED ON DAYLIGHT HOUR POTENTIAL AND SHORTEST PATH OF CIRCULATIONS

A COMPUTATIONAL METHOD TO GENERATE ONE-STORY FLOOR PLANS FOR NURSING HOMES BASED ON DAYLIGHT HOUR POTENTIAL AND SHORTEST PATH OF CIRCULATIONS

MASTER THESIS
TRACK OF BUILDING TECHNOLOGY

STUDENT:
LINCHENG JIANG
4801334

SUERVISOR:
PIROUZ NORIAN
REGINA BOKEL

BOARD OF EXAMINERS DELEGATE:
GEERT COUMANS

FACULTY OF ARCHITECTURE AND THE BUILT ENVIRONMENT
DELFT UNIVERSITY OF TECHNOLOGY 2019-2020

Preface

In the spring of 2015, as a 3rd year bachelor architecture student, I visited Renzo Piano's Piece by Piece exhibition in Shanghai. I was fascinated by the way that humanity and technology co-living and enhancing each other within his works. It became one of the initial motivations for me to choose building technology of TU Delft as my master track.

During the study in this track, I found the computer is a powerful tool to execute ideas that designer can think of but hard to implement due to both the advantage and disadvantage of our beautiful human brain.

"Maybe with its help, by designing the way to design the design, and realizing the design with the designed computational tool, I can empower myself to better achieve my goals of design" – This is why I choose computational layout optimization as my graduation topic. And the experience of designing a nursing home as my bachelor's graduation project brings out the title of this thesis.

In this thesis, I focus myself on finding a method to embed the optimization of the living environment of elderlies, daylight hour of rooms, and the efficiency of the layout, shortest path of circulations, into the early design workflow, realize the method with the computational tool, and implement it on a test design case. Hope you find this thesis interesting.

Lincheng Jiang
October 2020

Content

1. Introduction

1.1. Context

China is stepping into an aging society, needs for elderly care related infrastructures and facilities are rising. Nursing homes as important components of elderly welfare require complex decision making during the design process. With the interest in the topic of utilizing algorithms to promote design capability of architects over such complex design issues, I choose studio Computational Layout Optimization for my graduation thesis.

Nursing homes have been facilities where requirements for efficiency and residents' needs for quality of life collide, involving even more complex design and decision-making process when the call for sustainability is brought up.

Industries of architecture have developed experiences over such issues and depend on them over the design of such types of buildings during past years. Those experiences effectively helped architects to ensure the minimal requirements of both aspects being met, but also set up constraints on the design process.

With the help and developments from computer science, it is very possible to provide architects with more simple and powerful tools to better achieve their design vision over these complex design issues.

1.2. Social relevance

As mentioned above, Chinese society is aging fast. Needs for elderly welfare infrastructures and facilities are rising. Nursing homes as important components of elderly welfare require complex decision making during the design process. It serves as a humanity infrastructure which requires for liveability, in this thesis, daylight hour is discussed, yet also provides services that ask for circulation efficiency.

This graduation work would help architects better optimize spatial plans of buildings under these two requirements, which potentially improves the future built environment for elderlies. It can also work as a framework for other design cases that requires the satisfaction of multiple criteria.

1.3. Scientific relevance

This graduation thesis is within the scope of the studio topic Computational Layout Optimization, within the track of Building Technology. It is part of the research framework of Chair Design

Informatics. This graduation thesis uses computational method bridges between architecture design and computer science, it aims at providing a tool to generate practical floor plans regarding circulations and daylight hour potentials of the floor plan.

1.4. Scope

This thesis is aiming at developing a methodology to computationally generate a practical one-floor plan, which can be integrated into the architectural design process, it is about integrating daylight potential analysis and pathfinding to improve both liveability and efficiency of the building. This thesis consists of the research on methodology and design cases to which the method is implemented.

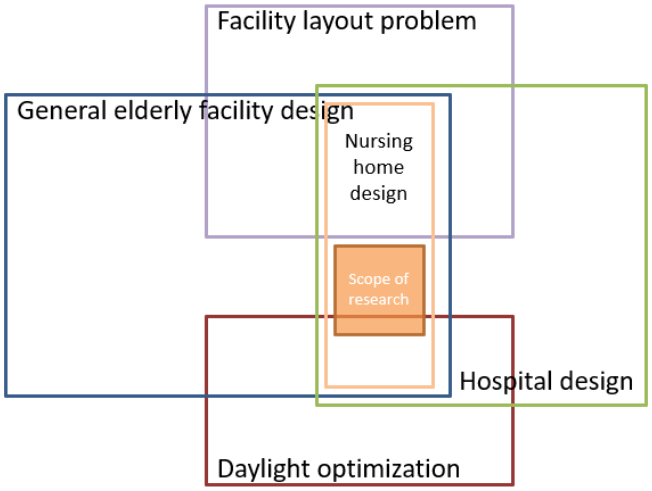


Figure 1 Scope of this thesis.

Methodology wise, this work is in the area of the "Science of the Artificial" (Nourian, 2019). While the thesis involves optimization theory, graph theory, force-directed methods, computational geometry and daylight hour analysis in some way, the detailed discussion of each subject falls out of the scope of the project.

1.5. Objective

The objective of this thesis is to develop a computational method that generates one-story-heigh floor plan, that meets both circulation and daylight hour requirements of the building, which can be integrated into the architectural design process.

1.6. Research questions

The main research topic can be break into several sub-questions:

- How should the program of requirements be modelled into spatial relations?
- How can spatial relations be projected into an actual space?

- How should the relations among evaluating circulations, walking distance and daylight hour be? In another word, what is the hierarchy?
- How should the designer interact with the computational process? What is his/her role?

1.7. Problem statement

Design of elderly care buildings like nursing homes usually looks for balances between the livability for residents and the circulation efficiency for staffs and workers. Losing the balance between these two has the potential of either dissatisfying residents, uncomfortable living experiences, or causing low efficiency in operations, which might lead to extra working loads, longer service hours and higher service costs. Architecture industries have been relying on experiences during the design process of it for a long time. While in the world of computer science and gaming industry, people have been dealing with lots of optimization problems and computer geometry generation problems. It is a good choice to learn from those areas and develop a practical computational methodology to provide an answer to floor configuration for these types of architecture design issues.

1.8. Approach and methodology

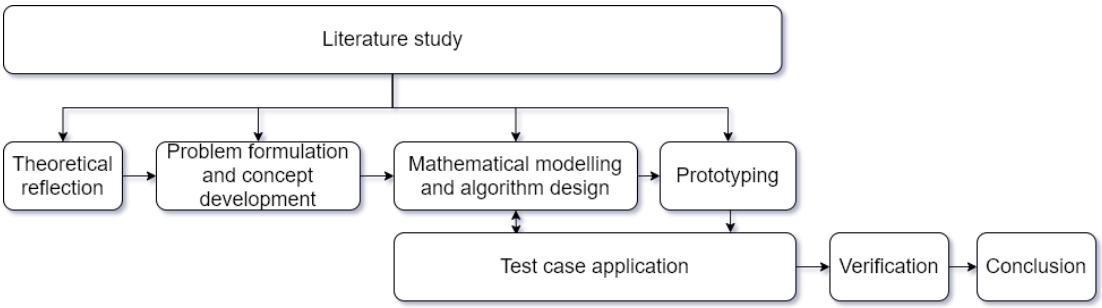


Figure 2 Research workflow

A. Literature and background knowledge study

First, a broad study of literature within the scope of interests is conducted. While narrowing down the scope of research, background knowledge about set theory and graph theory, topology and geometry, and algorithms are being learned through online open courses. At this stage, the basic knowledge that is required to complete the thesis is obtained. Also, the status quo of studies and practices of computational layout generation is being reviewed. To ensure the validity of the research, literature studies will be continually carried on through the research process.

B. Problem formulation and concept development

This step is about breaking down the topic into subproblems, formatting specific problems that need to be solved. Developing feasible concept solutions for each subproblem including concept workflows, mathematical modelling concept and algorithm concept.

C. Mathematical modelling and algorithm design

At this stage, the feasible concept solutions are developed into mathematical models and algorithms. Algorithms are broken into several parts, each part deals with specific subproblem.

D. Prototyping

At this stage, algorithms are implemented with Python, and trials and errors and done through a series of toy problems. Toy problems are specific problems needs to be solved or realised towards the final goal.

E. Test case application

The prototype algorithm is implemented to generate a one-story-heigh nursing home plan.

F. Verification

Using existing analysis theories and software to analyse the outcome of the test case. Compare it with a manually designed floor plan. Evaluate the effectiveness of the general methodology.

G. Conclusion and summary

Conclude and summery the result of the design and the developed methodology.

1.9. Research tools and techniques

Computational and mathematical modelling are the main tools of this research. Related tools used in this thesis are:

- Python 3.7 with Numpy, Networkx, Scipy, Pymunk and Pygame libraries.
- Use Rhino 6 Grasshopper as the main platform for prototyping and implementation developed methodologies.
- Ladybug Tools (a tool that supports the evaluation of initial design options through solar radiation studies, view analyses, sunlight-hours modelling, and more. Which can be installed on grasshopper platform.) is used for daylight hour analysis. (Roudsari & Pak, 2013)
- GH_CPython component is used for running CPython in Rhino 6 grasshopper. (Abdelrahman, 2017)

1.10. Assessment and related tools

Result of the test cases is assessed with Ladybug Tools for the analysis of daylight hour of rooms.

2. Literature study

2.1. Elderly care building design

Elderly care buildings are where the needs of liveability and the requirement for efficiency collide. Elderlies living within these buildings seek for comfort environment, convenient facilities. Workers and staffs within these buildings also ask for reasonable working circulations.

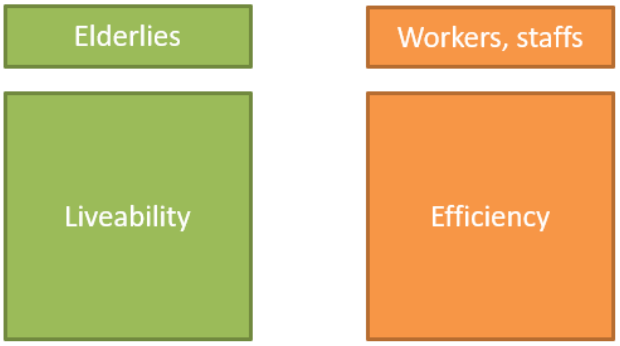


Figure 3 Needs of elderly care buildings

A. Elderly lives

As World Health Organization catalogues the physical environment and home environment as part of Quality of Life domains (World Health Organization, 1995). Lots of elderlies over the age of 60 spend most of their “indoor” time under low illuminance levels. Flores Villa et al. found a notable difference in the health and sleep condition between housebound and people able to go out, daylight availability is particularly important for housebound elderly with limited access to outdoors. (Flores Villa et al., 2017)

The research of Ryoji Suzukil et al (Suzuki, R. et al, 2006) conducted monitoring daily living activities of elderly people in a nursing home using an infrared motion-detection system recorded the daily motion of 3 elderlies for 28 days. The actual activities of the elderlies of the first 7 days were also recorded by questionnaires. The result is shown in figure 4 below.

Figure 4 Daily indoor activities for three elderly subjects in a nursing home（Suzuki, R., et al, 2006）.

The meal schedule in the nursing home was breakfast at 7:30, lunch at 12:00, and dinner at 17:00. These three periods framed the daily schedule of recorded elderlies.

B. **Elderly buildings**

To support elderly livings, lots of circulations crossing over each other within the building. Typical circulations within a Chinese flag-ship elderly apartment includes family visiting circulation, laundry circulation, dirty logistic circulation, meal delivery circulation, nursing circulation, visitor circulation, volunteer activity circulation, etc. (Zhou, 2018) Among which, usually the dirty logistic circulation should not intersect with clean circulation like meal delivery, for example, using the separate elevator for each. Visitor circulation should as less interrupting to elderly living circulations as possible, yet still, reveal the elderly's life within the facility. Circulations for staffs and workers should try the best to form a closed circle, so it improves the efficiency and no need to walk back after a dead end.

2.2. Generative design approaches

Exploration of generative design over spaces has been out in the gaming industry and architectural industry for a while. While inheriting knowledge from computer science, although both the gaming industry and architecture industry are working with spaces. The distinction between the requirements of them made the targets and approaches of these two industries differently.

A. Generative design in the gaming industry

Research of Niemann, M. (2015) and Linden, R. V. D., et al. (2014), reviewed multiple current procedural dungeon generation algorithms.

Category	Case	Approach	Control provided	Gameplay features	Output Types	Variety of results
Cellular Automata	1	Grid cells state modification	Initial state; # of generations		2D dungeon with floor, rock, wall cells	Chaotic maps: little variation
Generative Grammar	2	Graph grammars	Difficulty, size, fun	Difficulty, fun parameters	2D room graph	Dependent on hard-coded graph grammar
	3	Graph and shape grammars	Input missions	Missions	2D dungeon with rooms, locks, keys	Maps as versatile as missions
	4	Gameplay grammar and 3D geometry generator	Customizable grammars, parameter-based rule selection	Player actions as nodes, gameplay parameters for rule re-writing	3D dungeon	As versatile as grammars and parameters
Genetic Algorithms	5	Space tree mutation	Game story, fitness function, player model	Game story	2D dungeon-like game world	Combination of premade location types
	6	Tree mutation	Fitness function, genetic parameters	Special event rooms	2D dungeon	Tightly packed rooms connected by hallways
	7	Combining 4 genetic representations and 5 fitness functions	Fitness function, genetic parameters		2D maze	Larger between combinations
	8	Mixed initiative: map sketch and constrained optimization	User sketching and fitness function	Playability, pace, balance	2D dungeon	Large, even beyond dungeons
Constraint-based	9	Constrained graph generation	Topology, node placement		3D underground level	Small rooms connected by hallways
Hybrid Approach	10	Compositional PCG: ASP and evolutionary search	ASP constraints, fitness functions		2D dungeon	
Generative Grammar	11	Rhythm-to-actions-to geometry grammars	Path, rhythm, objects, critics	Player-based rhythm	2D platformer level	Combination of premade level segments
Occupancy Regulated Extension	12	Position-based combination of level chunks	Chunk library, mixed initiative	Chunks contain game-related items	2D platformer level	Combination of premade level chunks
Real-World Data	13	Bayesian network trained with real data	# of rooms, sizes, distances, style of the building		3D residential building layout	Varied residential building models

Table 1 OVERVIEW OF SURVEYED METHODS AND THEIR PROPERTIES (Linden, R. V. D. et al, 2014)

As the majority of automatic space generation cases either rely intensively on the random process like mutation or generates chaotic maps like cellular automata. The work of Joris Dormans and Sander Bakkes (Dormans, J., & Bakkes, S., 2011), using graph and shape grammars, drew my attention with the similarity towards architectural design.

Joris Dormans and Sander Bakkes developed a workflow to generate gaming map basing on the activity that the designer wants gamers to experience.

First, a directed mission graph is generated, then tasks are replaced with rooms of various sizes. Then the mission structure is translated to a spatial construction. Then space is grown with shape grammar and analysed for natural paths connecting the rooms.

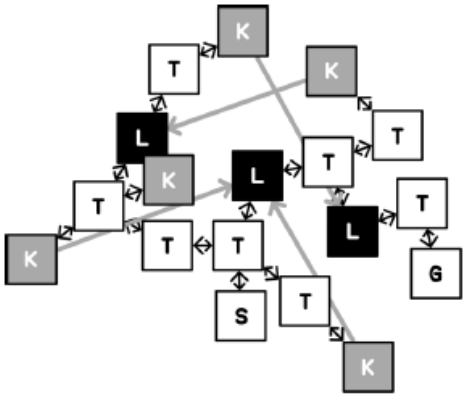


Figure 5 Directed mission graph in an organic layout (Dormans, J., & Bakkes, S., 2011)

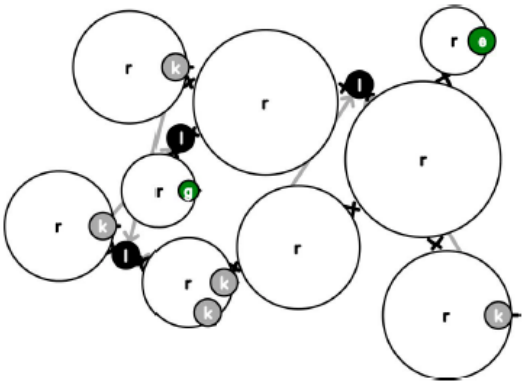


Figure 6 Room with sizes (Dormans, J., & Bakkes, S., 2011)

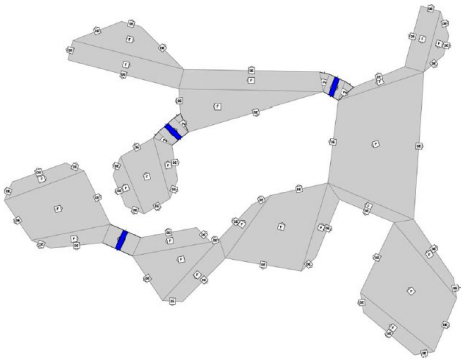


Figure 7 Spatial construction (Dormans, J., & Bakkes, S., 2011)

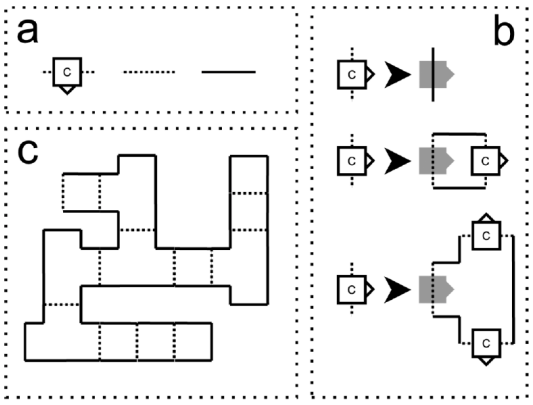


Figure 8 Shape grammar (Dormans, J., & Bakkes, S., 2011)

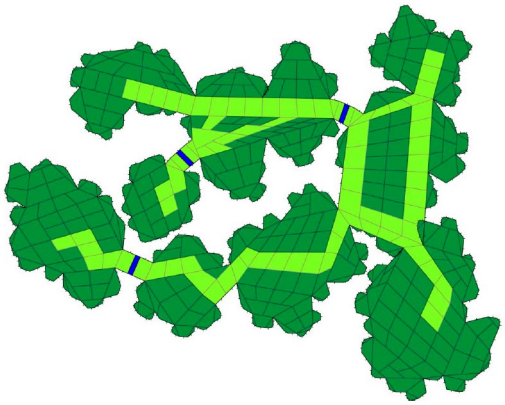


Figure 9 Final space (Dormans, J., & Bakkes, S., 2011)

The work of Joris Dormans and Sander Bakkes can be a showcase of how to translate activity sequence into spaces. However, it's still far from architecture design. The relatively low level of control required by the gaming environment over the generation process does not provide sufficient control for architecture design. Therefore, more direct application cases of computer science within architecture design are studied.

B. Generative design in the architecture industry

In the architecture industry, where more criteria are added to spaces, things are a bit different. Simon's Evolving Floorplans (Simon, J., 2019) took an approach solely from the perspective of optimization. Using genetic algorithm, physics simulation, graph-contraction, ant-colony pathing, produces one-story Voronoi floor plan with rooms, hallways with the requirement for daylight. He states that the limitation of it is that the shape regardless of convention and constructability.

Chaillou took a Generative Adversarial Nets approach with his AI Architecture Generative Design Housing (Chaillou, S., 2019). He uses pix2pix GAN-model and Google deep learning tools to generate one-story floor plan images with furniture. The limitation is that the generation remains on 2D and lacks the possibility of extending the pool of room types without the training of machine learning.

Nisztuk and Myszkowski's ELISi is a program that approaches from metaheuristics, using Hybrid Evolutionary Algorithm （NSGA-II algorithm + Greedy-based algorithm） to produce one-story floor plan. (Nisztuk, M., Myszkowski, P. B., 2019) The limitation is that all location preference of rooms needs to be preset. And there is no interaction with the environment.

The research of Egor et al. produced a grasshopper component, Magnetizing Floorplan Generator, brings some results that look very convincing. They started with a statement as a starting point: "Each of the rooms in a building is somehow accessible from any other room through a corridor. It means that the whole communication structure is interlinked and there is one corridor structure, connected to all rooms." (Egor et al., 2020)

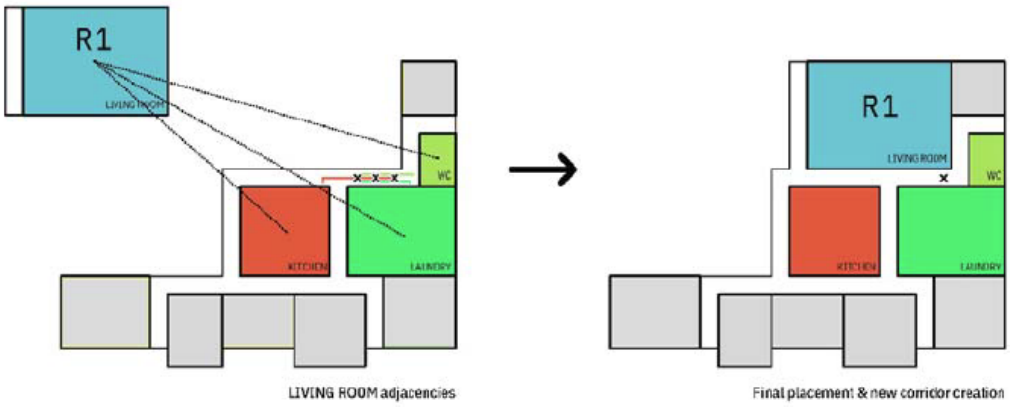


Figure 10 Example of an evacuation plan in comparison to generated plan. (Egor et al., 2020)

They analyse the graph of the floor plan, giving the order of room placement by the degree of each room. The highest room is the potentially most difficult one to place, so it is placed the first.

Then they optimize the placement by iteration and retrieving to previous iterations when things go wrong. The result of it looks convincing.

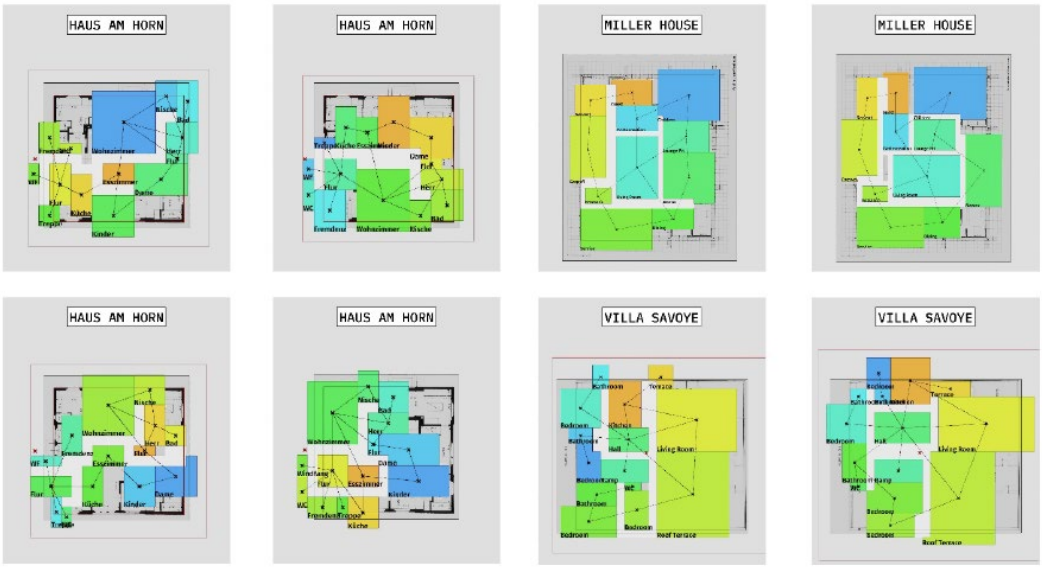


Figure 11 Generated variants, which were created on base of plans of iconic houses. (Egor et al., 2020)

3. Methodology

3.1. Methodology

To empower research and design, some methods are adopted from math and computer science. These methods allow the mathematical and computational modelling to be better achieved.

A. Graph theory

As Pirouz Norian stated in his work (Norian, 2016), bubble diagrams have been widely used through the concept design phase of architecture design. Architectural spaces are usually represented with bubbles and their connection are represented with lines which connect each bubble.

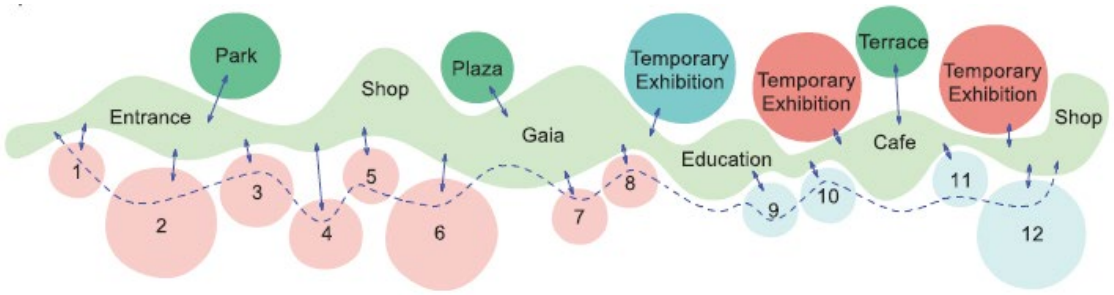


Figure 12 A bubble-like diagram from SANAA((New National Gallery and Ludwig Museum, 2015)

In graph theory, such spatial relations can be mathematically modelled with nodes and links (or vertex and edges):

Nodes represent actual spaces.
Links represent circulations.
Weights on links represent the importance of the adjacency or connectivity.

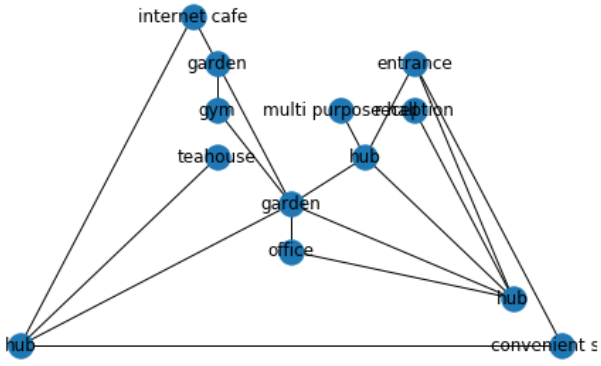


Figure 13 A graph representation of an architecture program

B. Force-directed graph drawing

To bring the graph into space. It is helpful to use the force-directed graph drawing method as it can intuitively form constraints among rooms and keep them tight. Force can be assigned to the links with the weight of the link, the higher the weight, the larger the force.

Given the graph $\Gamma = (V, E)$, $E = (V_i, V_j)$ if V_i is linked to V_j

Output a kissing-disk drawing of graph Γ

- **Do**
  - **For Each** vertex $u \in V$
    - $Resulting\_Forces = \sum \boldsymbol{Attraction\_Forces}(u) + \sum \boldsymbol{Repulsion\_Forces}(u)$
    - $u = u$ moved by the Resulting_Forces
  - **Next**
  - Recompute Continuance_Condition:
    $\forall (i,j) \in E, x_{ij} \neq (R_i + R_j) \mp ErrorTolerance$
  - Iteration_Count=Iteration_Count+1
- **Until** (Continuance_Condition=False Or Iteration_Count>MaximumIterations)

- **Attraction_Forces=** $AF_{ij} = k_a \Delta x_{ij}$, if $(I, j) \in E$,
  - k_a = attraction strength factor,
  - $\Delta x_{ij} = Distance\ V_i\ to\ V_j\text{-}RestLength(i,j)$
  - $RestLength(i,j) = R_i + R_j$
- **Repulsion_Forces=** $RF_{ij} = \frac{k_r}{x_{ii}}$, for all (I, j) $if\ x_{ij} < RestLength(i,j)$
  - k_r = repulsion strength factor
  - $x_{ij} = Distance\ V_i\ to\ V_j$
  - $RestLength(i,j) = R_i + R_j$

Figure 14 Abstract bubble diagrams drawn automatically by the force-directed graph-drawing algorithm. Changing the area values changes the diagrams. (Norian, 2016)

3.2. Design methodology

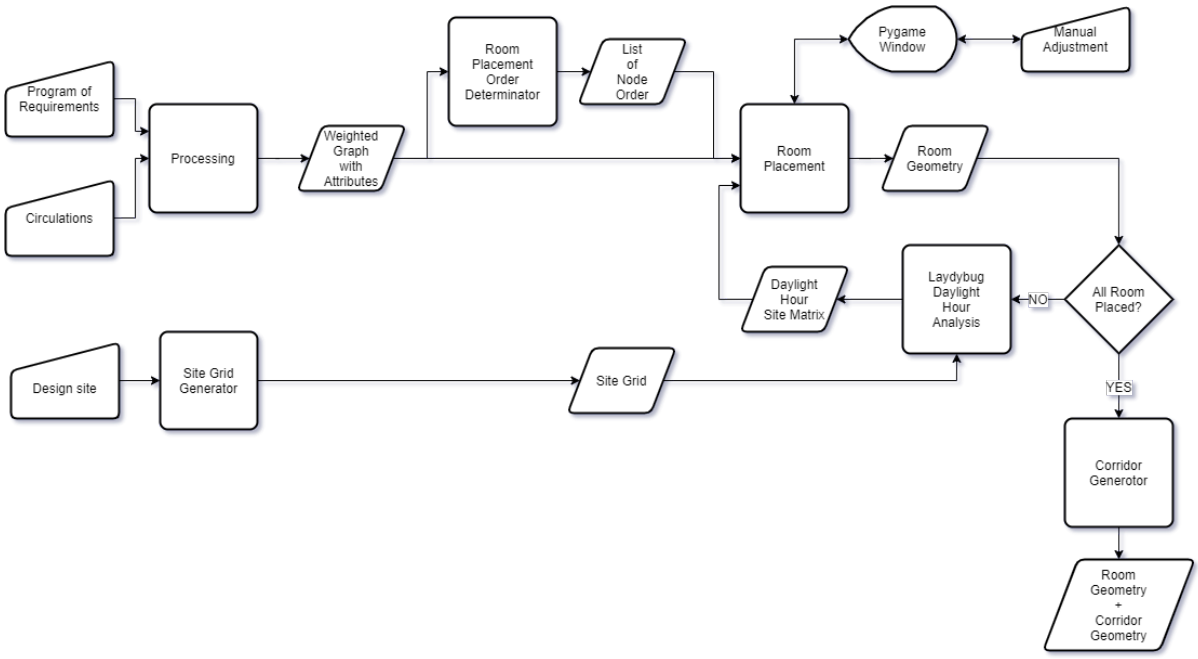


Figure 15 Design methodology

Starting with the characteristics of the design requirements. The daylight hour potential of rooms is strongly related to the relative position of the room, while the circulation efficiency is strongly related to the direction of the corridor and the position of the door opening in the room.

Therefore, I divided the research topic into two parts, using the force-directed method of the Pymunk physics engine to solve the problem of room location, and solving the problem of corridor direction and door opening location through pathfinding algorithm and genetic algorithm. Because daylight hour potential is strongly related to the location of the room, daily analysis is embedded in the process of generating the room location through the physics engine.

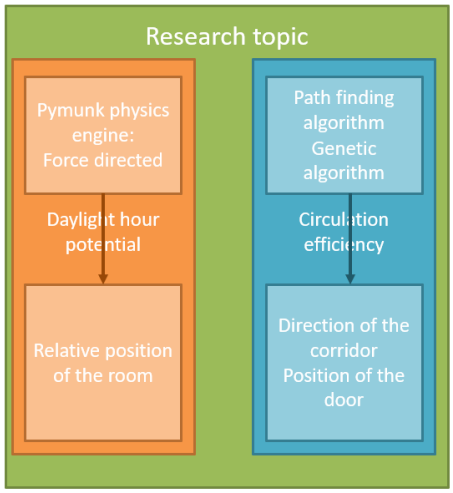


Figure 16 Relation between the technique and the problem within the research topic.

The methodology design is to first generate a weighted graph that includes all rooms and circulations by processing the circulation in the building, the attributes of each room and the requirements for daylight.

Then, through a series of judgments to generate the order of placing the rooms, first judge the summed weight of the edges connected to each node, then judge which node has the least degree (higher average weight), and then judge which room has the largest size.

Then place the rooms in order and use spring to connect the rooms with the physics engine.

When all the rooms are placed, analyse the daylight hour in the order of placement. When one room is analysed, the room is set as a static body, and the rest of the rooms are squeezed out of the range that does not meet the daylight hour requirements, and then the next room and all rooms that have become static bodies are analysed for daylight hour together until all rooms are placed.

In each process of placing the room, the designer can use the mouse to drag the position of the room to adjust the relative relationship between them.

Finally, when all the rooms have completed the daylight hour analysis and fixed their positions, according to the weights of different circulations, the pathfinding algorithm is used to generate corridors.

This set of methodology proposes a feasible way to realize the circulation based on daylight hour analysis and room relationship and the shortest path circulation optimization method at the same time in one consistent workflow, and the designer can intuitively intervene in the computer design process, intuitively affect the final result.

3.3. Processing of Program of Requirement

First, all the requirements and attributes of rooms and circulations within the building needs to be processed to a form that can be easily processed and acquired in the future process. The information can be either process into matrices or graph.

The advantage of matrices is that they can easily extract numeric information. Mathematical operations can be done on matrices. The disadvantage is that matrices can only store numeric information, no string can be saved to matrices, string information must be mapped to numbers, then store them in matrices.

The advantage of the graph is that it can store both numeric information and string information. Networkx library of Python stores graph information in the forms of a dictionary, therefore it is

easy to extract information with keys. Some mathematical operation with a graph can also be done. The graphical representation is intuitive.

I choose to use the graph to store all the information I need. But it needs to be awarded, that what does the edge of the graph stands for. In my case, I use the edge to store circulation information and the weight of each circulation.

To proceed:

A. Acquire the name and number of rooms within the building.
B. Acquire the sizes and desire daylight potential of each room.
C. Acquire the preferred side of the door for each room, 'none' means no preference.
D. Acquire circulations and weight(importance) of different circulations (in the scale of 1 – 5) within the building.
E. Calculate the total number of rooms connected to each room.
F. Calculate the sum of the weight of each connection to each room.
G. Store the above-mentioned information as Numpy array in Python.
H. Generate a graph with weights and attributes from the Numpy array.

Construction of the graph:

```
Input:
    rooms = ["RoomA", "RoomB", "RoomC", "RoomD", "RoomE"]
    sizes = [100, 50, 30, 40, 80]
    lights = [8, 7, 6, 5, 4]
    door_direction = ["N", "none", "none", "none", "W"]
    circulations = [["RoomA", "RoomB", "RoomE", "RoomA"],
                    ["RoomB", "RoomD", "RoomC", "RoomE", "RoomB"],
                    ["RoomA", "RoomB", "RoomC", "RoomE", "RoomA"]]
    cir_Weight = [3, 1, 2]
Graph generation:
    G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)
```

Figure 17 Processing of Program of Requirement

As shown in figure 17, to store all the information, the inputs below are used to construct graph G:

List 'rooms': stores the name of each room.
List 'sizes': stores the size of each room.
List 'lights': stores the desired daylight hour of each room.
List 'door direction': stores the desired door position in relation to the room.
List 'circulation': stores circulation information with sub-list, each sub-list is a circulation.
List 'cir_Weight': stores the given weight of each circulation.

```
ReL_Array = np.zeros([len(rooms), len(rooms)])

# creating ReL_Array
for i in range(len(rooms)):
    for a in range(len(circulations)):
        for b in range(len(circulations[a]) - 1):
            if circulations[a][b] == rooms[i]:
                ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]

# Flip the array left and right then rotate the array by 90 degrees
ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
```

Figure 18 Creating relation array

A N x N Numpy relation array is created first to store and calculated circulation information of rooms. It is created by examining each circulation, once a room is found following previous, the intersection coordination is added by the weight of the circulation.

Then this array is rotated by 90 degrees and flipped left and right, then add to its original array, then we have an array that can be transformed to a non-directional graph.

```
# get Sum of edge weights on one node
sum_weight = []
for i in range(len(rooms)):
    sum_weight.append(sum(ReL_Array[i]))

# Creating graph in networkx from ReL_Array
G = nx.from_numpy_array(ReL_Array)
for i in range(len(rooms)):
    G.nodes[i]['room'] = rooms[i]
    G.nodes[i]['sizes'] = sizes[i]
    G.nodes[i]['lights'] = lights[i]
    G.nodes[i]['sum_weight'] = sum_weight[i]
    G.nodes[i]['door_direction'] = door_direction[i]
```

Figure 19 Graph creation

To generate the graph, first, the summed weight of all edges that connect to each node is calculated by calculating the sum of the one row, this will be used for further proceeding. Then the graph is generated from the Numpy array using Networkx. Information of each are assigned as node attributes to the node that represents each room.

After having the graph, the order to place rooms into space can be determined.

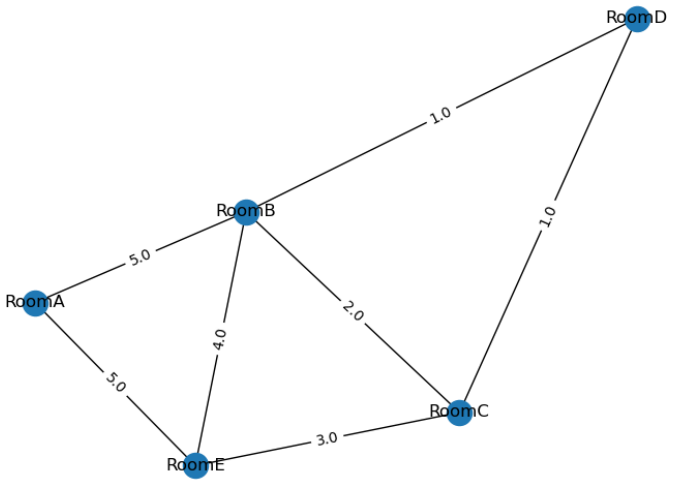


Figure 20 Graph G

3.4. Determining the order of room placement

Egor et al determined their order to place room by sorting the list of room in the order of the node degree, the higher the degree, the more connections the node has, thus the more difficult it is to be placed. So, they decided to place the one that has the most connection first. (Egor et al., 2020)

Inspired by their work, I decided to look at the importance of each node to the whole circulation network. The more important the node is, the higher the priority it has. The needs of the less important nodes can compromise to the more important one. After the highest weight, the number of degrees is compared. The one with less degree means it has edges with higher weights, which means the node is within a more important circulation. In the last, the size of each room is compared, the larger one is assumed to be harder to place at the end, so the one with a larger size is placed earlier:

To proceed:

A. First check which room has the highest total weight from connected edges, the highest gets placed first.
B. If the total weight is the same, then check which room has less degree (higher average edge weight), the least gets placed first.
C. If B is still the same, then check which room has a larger size, the larger gets placed first.

To generate the room placement order list, node index, sum_weight and sizes of each room are stored in the same sub-list of one list. Then the list is sorted using x : (-x[0], x[1], -x[2]) as key.

The order list is acquired by appending the node index from sorted list to the empty order list.

```python
def room_placement_order(graph, rooms):
    # Order of Placement
    lst_sum_weight = []
    lst_degree = []
    for i in range(len(rooms)):
        lst_sum_weight.append(graph.nodes[i]['sum_weight'])
        lst_degree.append(graph.degree[i])

    lst = [lst_sum_weight, lst_degree, sizes, rooms]
    lst_rev = [[] for _ in range(len(lst[0]))]

    for i in range(len(lst)):
        for a in range(len(lst[i])):
            lst_rev[a].append(lst[i][a])

    d = sorted(lst_rev, key=lambda x: (-x[0], x[1], -x[2]))
    # get the room placement order(in node index)
    order = []
    for i in range(len(d)):
        a = rooms.index(d[i][-1])
        order.append(a)

    return order
```

Figure 21 Determining the order of room placement

After we have the order list for order of placement. The actual room placement can be done.

3.5. Room placement

In the room placement phase, the main aim is to configure the position of each room. The logic is simple:

A. The site environment is analysed with Ladybug, and the generated daylight hour matrix of the site is used for room placement.
B. A rectangular room in the list with the given dimension or the dimension of the two closest factors of its size is placed in the site.
C. Then check if there are already neighbours of the room placed in the space. If there is, connect the two rooms with spring with the stiffness scaled by the weight of the edge between the two rooms. It there is not, then place the next room in the order.
D. Repeat B till all rooms in the order are placed. The designer now can use a mouse to drag

To realize this, Pymunk, a physical simulation library of Python is used for physics. And Pygame, a cross-platform game design module, is used for visualization and interacting with the process.

The space of Pymunk and the window size of Pygame is set with the dimension of the site.

In Pymunk, bodies represent points, they contain information like velocity, position, shapes are given to bodies, they contain information like boundaries, collision. Bodies have three types: dynamic, kinematic, and static:

“Dynamic bodies react to collisions, are affected by forces and gravity, and have a finite amount of mass··· Dynamic bodies interact with all types of bodies and can generate collision callbacks···

Kinematic bodies aren't affected by gravity and they have an infinite amount of mass so they don't react to collisions or forces with other bodies··· Static bodies are bodies that never (or rarely) move···" (API Reference — Pymunk 5.7.0 Documentation, n.d.)

At this phase, all bodies are generated as dynamic body and are connected by spring with the order of placement.

The position of each body is placed from bottom to up, while shifts from left to right. Every time a room from the room placement list is placed, it will search through its neighbouring nodes, if there are nodes that are already placed in the space, then it will be connected with the spring, the length of the spring is the sum of the square root of each room size, and the stiffness is given with the weight of the edge, with an index of *100.

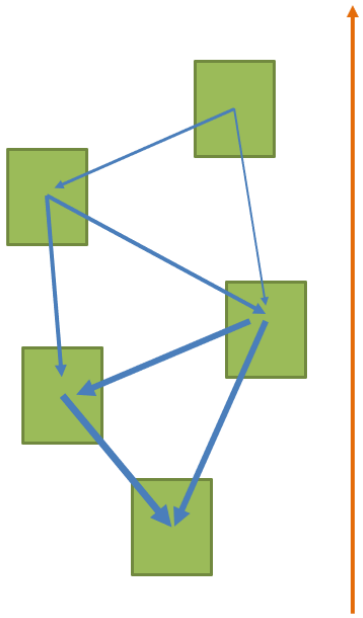


Figure 22 A representation of room placing order and spring connecting order.

In this way, when adding a new room to space, its constraints are also added.

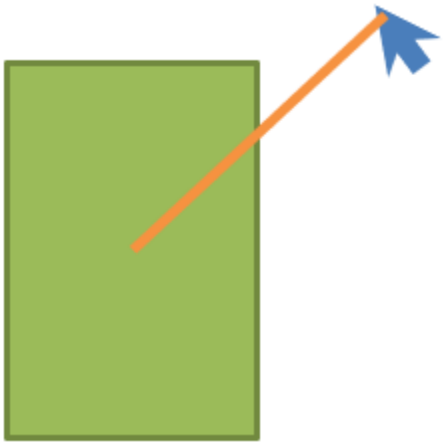


Figure 23 Joint created between the mouse and the shape.

Also, by giving a body to the mouse, it creates joints when the mouse is clicked at a shape of a body, and giving it a huge stiffness to the joint. In this way, shapes displayed on the Pygame window can be dragged by the mouse. By doing this, designers can adjust the configuration with the physics engine.

```
def room_placement(screen, space, graph, room_placement_order):
    rooms_shape_list = []
    spring_list = []
    for i in room_placement_order:  # order =  [4, 1, 0, 2, 3]
        s = G.nodes[i]['sizes']
        g = room_placement_order.index(i)
        lst_len = len(room_placement_order)
        exec(f'a{i}, b{i} = add_room_dynamic(space, s, (800+100*(-1)**g, 960 - g*(960/lst_len)), {i})')
        exec(f'space.add(a{i},b{i})')
        exec(f'rooms_shape_list.append(b{i})')
        exec(f'l{i}= (s*100)**0.05')

        f = [n for n in graph.neighbors(i)]  # getting neighbors of node i

        for i_f in range(len(f)):
            for k in room_placement_order[0:g]:
                if k == f[i_f]:
                    weight = G[i][f[i_f]]['weight']  # getting weight of edge connected to node i
                    # if room is inside space:
                    h = f[i_f]
                    sh = G.nodes[h]['sizes']
                    exec(f'l{h} = (sh*100)**0.05')
                    exec(f'length = (l{i} + l{h})')
                    exec(f'add_spring(space, a{h}, a{i}, weight, length)')
                    exec(f'spring_list.append((a{h}, a{i}))')
                else:
                    pass
    return rooms_shape_list, spring_list
```

Figure 24 Room placement

After the designer finds a satisfying initial configuration, the data is stored externally. The order of placement is updated with the order of y-axis form lower part of the screen to the upper part of the screen.

The reason to update the order of placement is that the previous order is used to solve the problem of which room to consider first, that can result in a good layout. After the initial placement, the layout is probably not from screen lower part to upper anymore, to proceed to daylight hour analysis, the order needs to be from bottom to up because in the northern sphere of earth, most time it's the object at the south side blocking the sun for the object at the north side.

In the external file, the following information is stored:

Int 'phase_index': an index to tell in which phase is this file generated.
List 'rooms_shape_list_pos_out': a list of coordinates of room shape positions.
List 'body_type_list_out': a list of body types of the room body.
List 'order_generate': the updated order of placement.
List 'rooms': stores the name of each room.
List 'sizes': stores the size of each room.
List 'lights': stores the desired daylight hour of each room.

List ‘door direction‘: stores the desired door position in relation to the room.
List ‘circulation’: stores circulation information with sub-list, each sub-list is a circulation.
List ‘cir_Weight’: stores the given weight of each circulation.

After all rooms are initially placed in the space, and data is stored in the external file, daylight hour potential can be analysed.

3.6. Assessment of daylight potentials and optimization of the layout

Reading all information from the external file, then the assessment of daylight potentials and the optimization can proceed.

From the external file, the graph of the room is again generated, rooms are generated on the position written on file, constraints are generated from the graph, and phase index is updated every time the file in opened.

Ladybug Tool is used as the daylight hour analysis components. Daylight hour on the date of the winter solstice on the northern sphere is analyzed in the order of updated placement, ‘phase_index’ is used as an indicator, it will place rooms no more than phase_index at each iteration. And after each iteration, information is stored to the same external file from ‘3.5 Room Placement’, with the same form.
During each process of placing the room, the designer can use the mouse to drag the position of the room to adjust the relative relationship between them.

To proceed:

A. Generate first room in Rhino.
B. Analyse the daylight hour of the room, map generated results to Pymunk and Pygame space.
C. Set the analysed room as a static body, and the rest of the rooms that do not meet the daylight hour requirements are squeezed out of the range. After all rooms are adjusted and relocated, generate the next room in Rhino.
D. Analyse all the rooms like step B and C in the order of placement.
E. Location of rooms can be dragged with the mouse after each analysis.

To be able to realize the collision between the Ladybug sunlight hour analysis result and the rooms. Ladybug sunlight hour analysis results are mapped into Pymunk and Pygame space.

Ladybug outputs daylight hour analysis result as a continues list, the result is written in the order of columns of the sub surfaces from bottom to up. So first a Numpy array ‘site_daylight_map’ is generated.

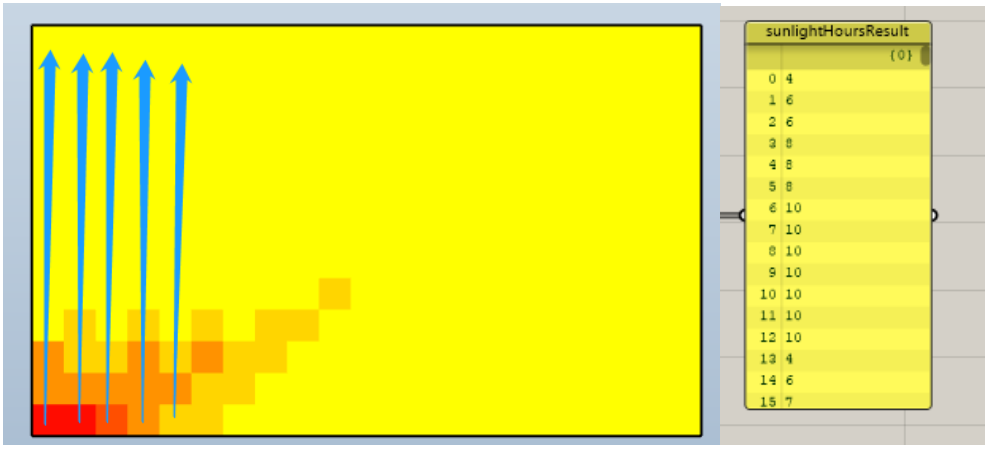


Figure 25 Order of sunlight hour analysis result list

```
def site_daylight_map(daylighthours, map_height):
    '''
    site_daylight_map = site_daylight_map(daylighthours, map_height, map_width)
    site_daylight_map = site_daylight_map.tolist()

    These two lines should be used at the end of the code in GH_CPython components,
    to pass arrays among GH_CPython components within grasshopper in the form of list.
    :param daylighthours: "sunlightHoursResult" list received from Ladybug Tools "sunlightHoursAnalysis" component,
    :param map_height:  Number of grid on y axis
    :return: A numpy array of "sunlightHoursResult" that corresponds to the site
    '''
    daylight_map = [daylighthours[i:i + map_height] for i in range(0, len(daylighthours), map_height)]
    np_daylight_map = np.array(daylight_map)
    np_daylight_map = np.rot90(np_daylight_map)  # match the numpy array with the site
    return np_daylight_map
# ↑ done, returns a numpy array, in the form of site grid.
```

Figure 26 Generation of Numpy array 'site_daylight_map'

Then binary Pymunk shape filters are assigned to all site_daylight_map pixels and rooms according to their daylight hour results and daylight hour requirements. Pymunk shape filter allows shapes in one category collie with shapes that are in a category that is not masked. After apply shape filter to all shapes, room shapes with lower daylight requirements can go through map pixel that has higher daylight hour value.

Object	Object Category	Category Mask
Player	0b00001 (1)	0b11000 (4, 5)
Enemy	0b00010 (2)	0b01110 (2, 3, 4)
Player Bullet	0b00100 (3)	0b10001 (1, 5)
Enemy Bullet	0b01000 (4)	0b10010 (2, 5)
Walls	0b10000 (5)	0b01111 (1, 2, 3, 4)

Figure 27 An example of shape filter. ((API Reference — Pymunk 5.7.0 Documentation, n.d.))

After all rooms are analysed and set to static, the corridors can be generated.

3.7. Evaluating the shortest walking distance to generate corridors

After the general layout is set after daylight hour optimization, A* pathfinding algorithm is used to find the best path among rooms.

A* pathfinding algorithm is used to find the direction of the corridor and the position of the door of each room. As it no longer involves physics simulation, it is done in Pygame space only.

The general process is:

A. Set the grid of the space.
B. Locations of the doors are set on the boundary of rooms according to the program of requirements.
C. Paths are generated between rooms connected with edges by pathfinding algorithm, record the list of nodes of paths.
D. After all paths are generated, generate corridor shapes with the node lists in rhino software.

The Pygame set up and A* pathfinding algorithm construction is configured from the work of Tim Ruscica(Ruscica, n.d.). The site is set to the grid of 96 x 160, which is the same dimension as the site. Class Spot is created to represent the grid unit, each spot represents a 1-meter by 1-meter space, on the grid. The coordinate of the spot is represented by grid[row][column].

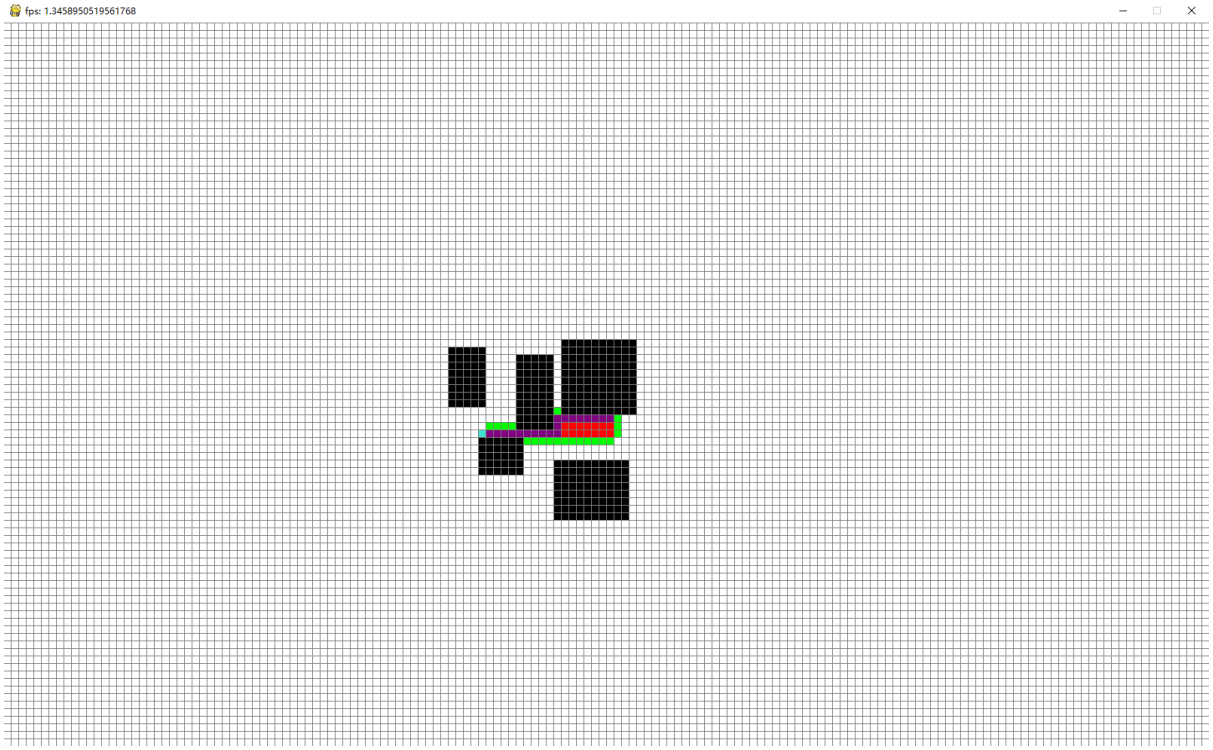


Figure 28 Pathfinding

To mapped rooms as barriers in the grid, first, need to transform the Pygame space coordinates into rows and columns. This is done by divide the XY coordinates by 10, which is the unit size of the grid on a 1600 x 960 pixel window.

Then the bounding box of the barrier is created with the coordinates of vertices of the room shape, first, the coordinates of four corners of the room are calculated into rows and columns of the pathfinding grid, then all spots within the boundary of rows and columns are set as barriers.

```
room_corner_A = (room_pos_x - room_width / 2, room_pos_y - room_length / 2)
room_corner_B = (room_pos_x - room_width / 2, room_pos_y + room_length / 2)
room_corner_C = (room_pos_x + room_width / 2, room_pos_y + room_length / 2)
room_corner_D = (room_pos_x + room_width / 2, room_pos_y - room_length / 2)
rowA, colA = get_barrier_pos(room_corner_A, ROWS, width)
rowB, colB = get_barrier_pos(room_corner_B, ROWS, width)
rowC, colC = get_barrier_pos(room_corner_C, ROWS, width)
rowD, colD = get_barrier_pos(room_corner_D, ROWS, width)

for j in range(int(colB - colA)):
    for i in range(int(rowD - rowA)):
        row = int(rowA + i)
        col = int(colA + j)
        spot = grid[row][col]
        spot.make_barrier()
```

Figure 29 Creating barrier

The coordinates for doors are pre-assigned to the boundary of each room. If the coordinates of for corners are A(row_A, col_A), B(row_B, col_B), C(row_C, col_C), D(row_D, col_D).
When the room is rectangular, row_A=row_D, row_B=row_C, col_A=col_B, col_C = col_D.

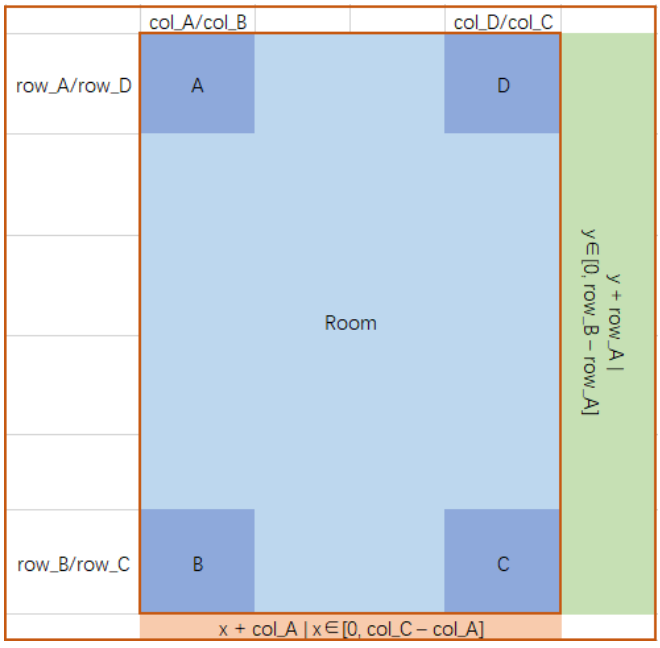


Figure 30 Coordinate of a room

Then the coordinates set of door position is:
Set(door_position) = {(row_A, x + col_A), (y + row_A, col_A), (row_B , x + col_A), (y + row_A, col_C) | x∈[0, col_C – col_A] , y∈[0, row_B – row_A] }

If the preferred door direction is north:
door_position ∈ {(row_A, x + col_A) | x∈[0, col_C – col_A] }

If the preferred door direction is south:
door_position ∈ { (row_B , x + col_A) | x∈[0, col_C – col_A] }

If the preferred door direction is west:
door_position ∈ { (y + row_A, col_A) | y∈[0, row_B – row_A] }

If the preferred door direction is east:
door_position ∈ { (y + row_A, col_C) | y∈[0, row_B – row_A] }

The start point and end point of pathfinding are the door position of the pair of rooms linked with edges within the graph. After each pathfinding, lists of path nodes are recorded to external files. After all paths are generated, rhino geometry of paths can be generated from the lists of node coordinates from the external file.

4. Progress

4.1. Toy problem 1: A simple floor layout consists of 5 rooms and 3 circulations

In this problem. I need to realize the algorithm to generate a simple floor layout consists of 5 rooms and 3 circulations.

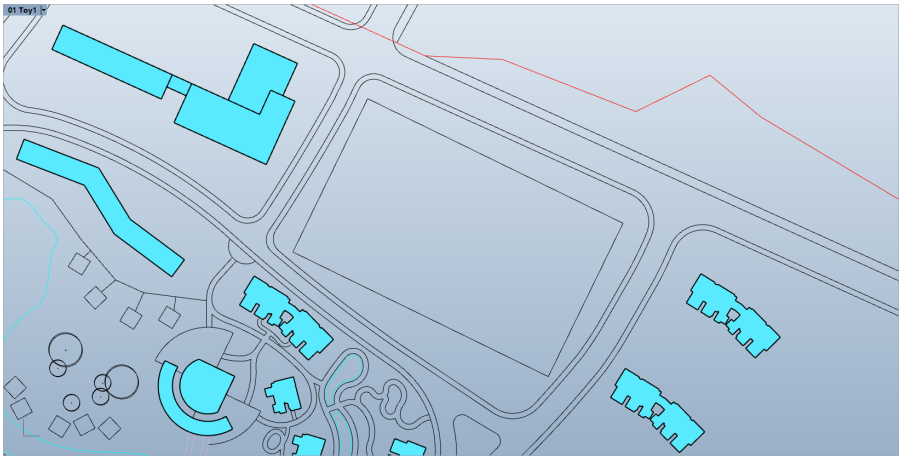


Figure 31 Site plan

The design site is shown as above, it's a rectangular space of 160-meter x 96-meter.

```
Input:
    rooms = ["RoomA", "RoomB", "RoomC", "RoomD", "RoomE"]
    sizes = [100, 50, 30, 40, 80]
    lights = [8, 7, 6, 5, 4]
    door_direction = ["N", "none", "none", "none", "W"]
    circulations = [["RoomA", "RoomB", "RoomE", "RoomA"],
                    ["RoomB", "RoomD", "RoomC", "RoomE", "RoomB"],
                    ["RoomA", "RoomB", "RoomC", "RoomE", "RoomA"]]
    cir_Weight = [3, 1, 2]
Graph generation:
    G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)
```

Figure 32 Input information

First, to process the program of requirements. Information of room names, room sizes/dimensions, desired daylight hour potentials, door orientations, circulations and weights of circulations are gathered. And graph G is generated from the inputs.

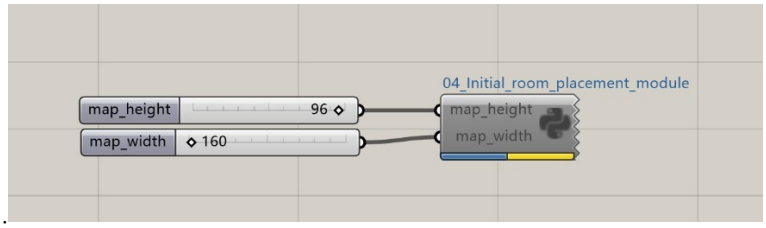


Figure 33 Initial room placement module

Then the initial room place module is enabled.

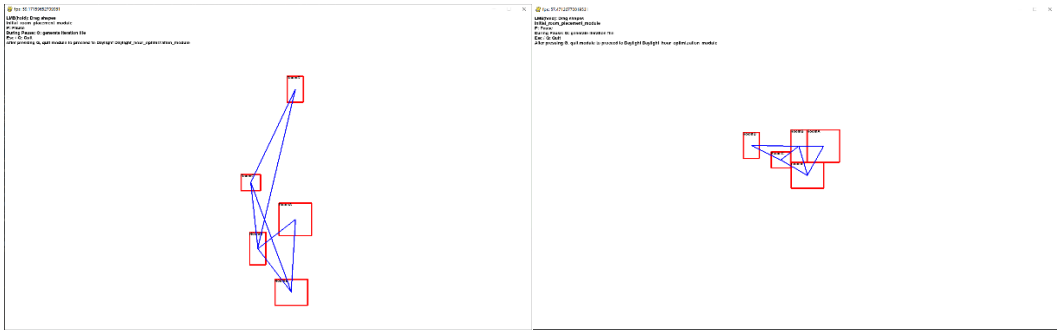


Figure 34 Enabled module(left) / drag mouse to adjust

The shapes of rooms are placed in Pymunk and Pygame space, and constrains are loaded.

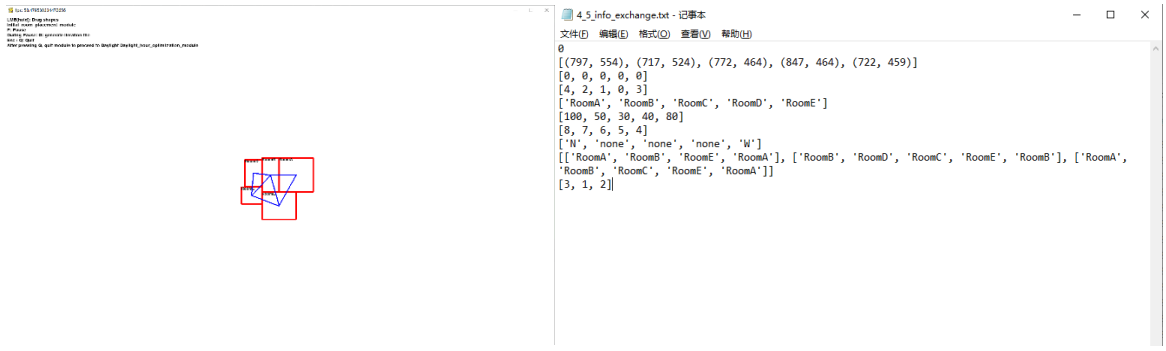


Figure 35 Press 'P' button to freeze the screen. (left)/ External information file generated by pressing 'G'.

After the user is satisfied with the initial layout, the user can press the 'P' button to freeze screen, then press the 'G' button to generate an external information file for passing data.

4.2. Toy problem 2: A simple floor layout considering daylight potentials.

In this problem. I need to continue from toy problem 1, realize an algorithm to generate a simple floor layout insists of 5 rooms and 3 circulations, while daylight potentials are considered.

First, ladybug components need to be set to proceed with the analysis. In Figure 36, part01 reads epw weather file and construct the sun condition on the date of the winter solstice on the northern sphere. Part02 receives the sun condition data and rhino geometry data to run the analysis and generate sunlight hour analysis result. Part03 is a module that consists of Python codes reads the external information file then output corner coordinates of room shapes, and grasshopper components that generate room shapes from the coordinates in rhino space. (Figure 36, lower)

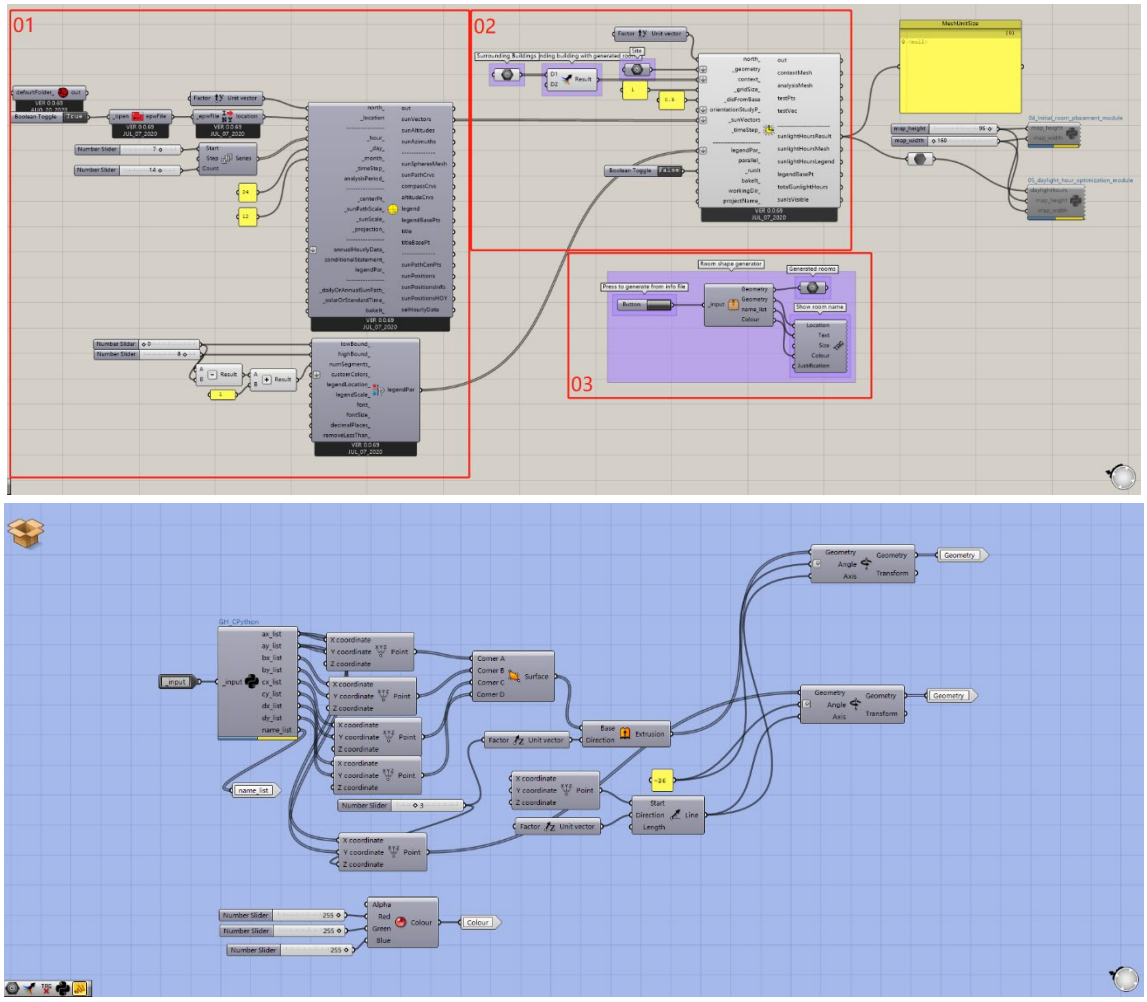


Figure 36 Grasshopper component set up.

First, initialize the analysis by enabling Ladybug component to analyse surrounding environments.

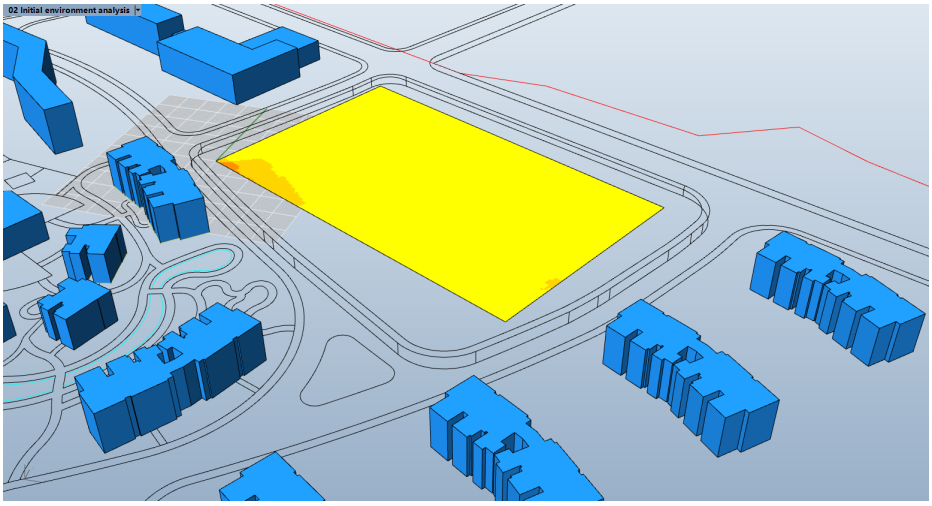


Figure 37 Initial analysis

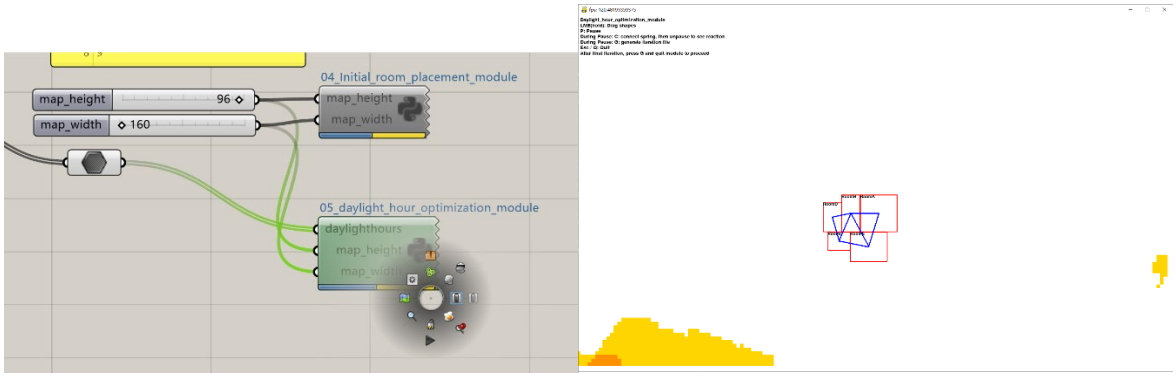


Figure 38 Enable daylight hour optimization module.

Then enable daylight hour optimization module, Pygame window will popup. Now all rooms are floating, and there's no spring, first press 'P' to freeze screen, then press 'C', springs will be added. Then press 'P' again to unfrozen the screen. After moving the group to a preferred position. Then press 'P' to freeze screen and press 'G' to output information.

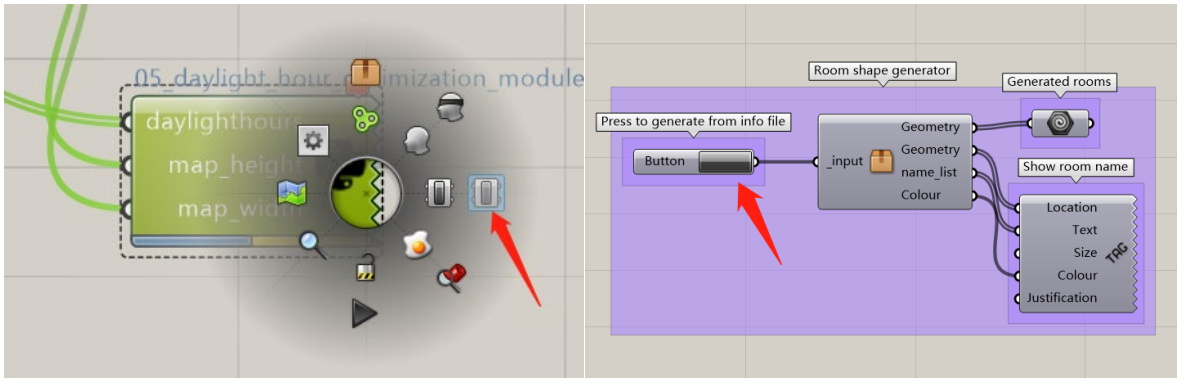


Figure 39 Disable module and generate rhino shapes

Then disable the module, and press the button to generate rhino shapes.

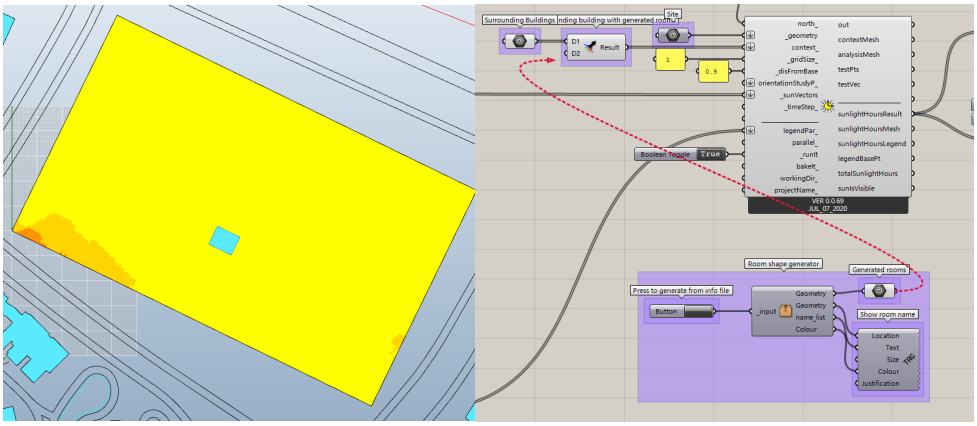


Figure 40 Pass the shape to Ladybug component

Pass the shape to Ladybug component, and run the analysis.

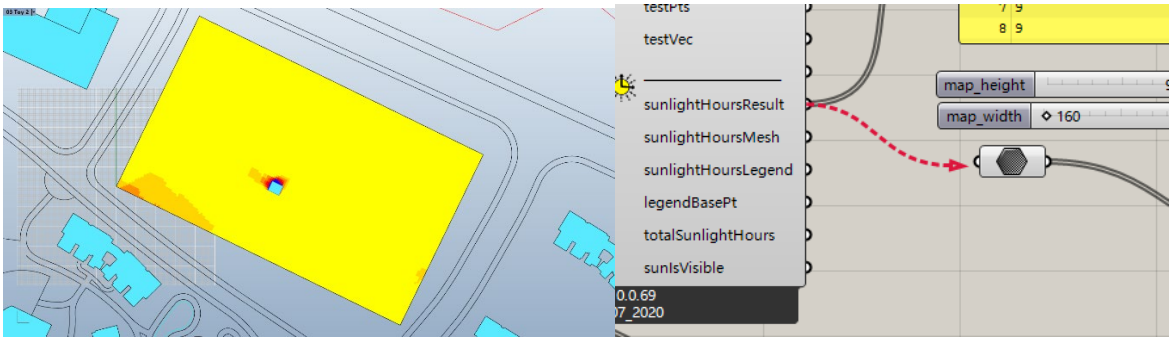


Figure 41 Result and internalize data.

After having the result, connect the result to the data component, then, internalize the data.

Figure 42 Second iteration

After A.internalizing data, B enabling daylight hour optimization module, the window will pop up again, with the result room analysis and first shape become static. First press ‘P’, ‘C’,’P’ to add spring, then after adjusting the layout, press ‘P’ and ‘G’ to export info file.

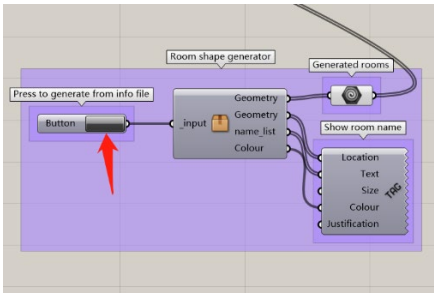


Figure 43 Update rhino shape

Then press the grasshopper button to update rhino shape.

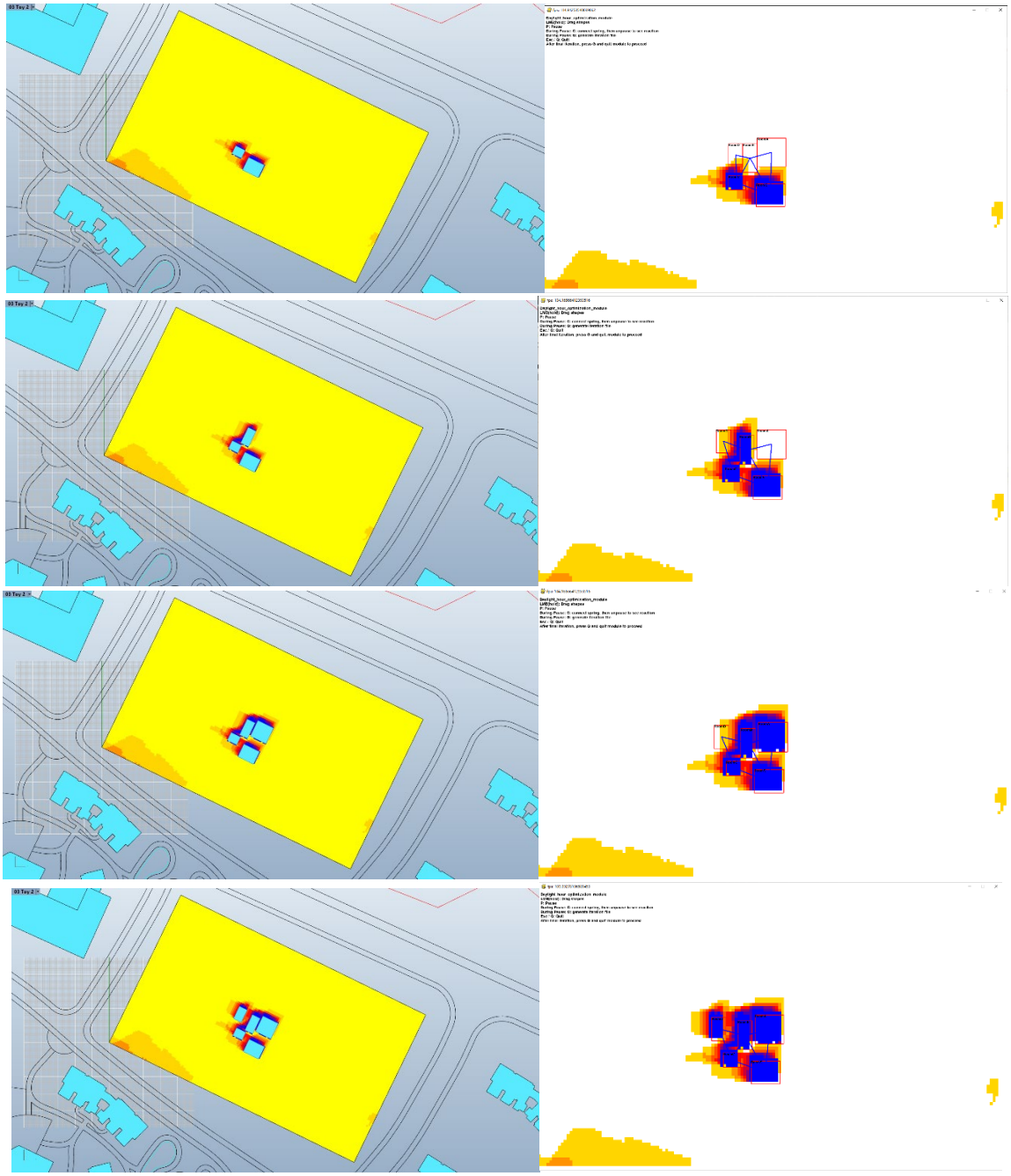


Figure 44 Finish iteration

Repeat previous steps until all rooms are static.Then go to the pathfinding module to generate corridors.

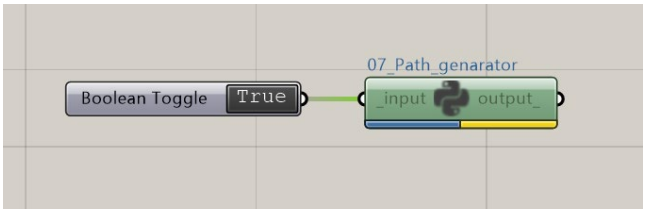


Figure 45 Pathfinding module

After enabling the pathfinding module, it will read from the external info file, and generate grid, obstacles and door positions in Pygame. Each time it generates one path, and store the path coordinates in another external file. In Pygame window, the purple grid represents the path. Green grids are the grids in the open list. Red grids are in the closed list.

After having the path coordinate, a GH_CPython read coordinate lists from the external file, then the corridor is generated with grasshopper componnets from the coordinates.

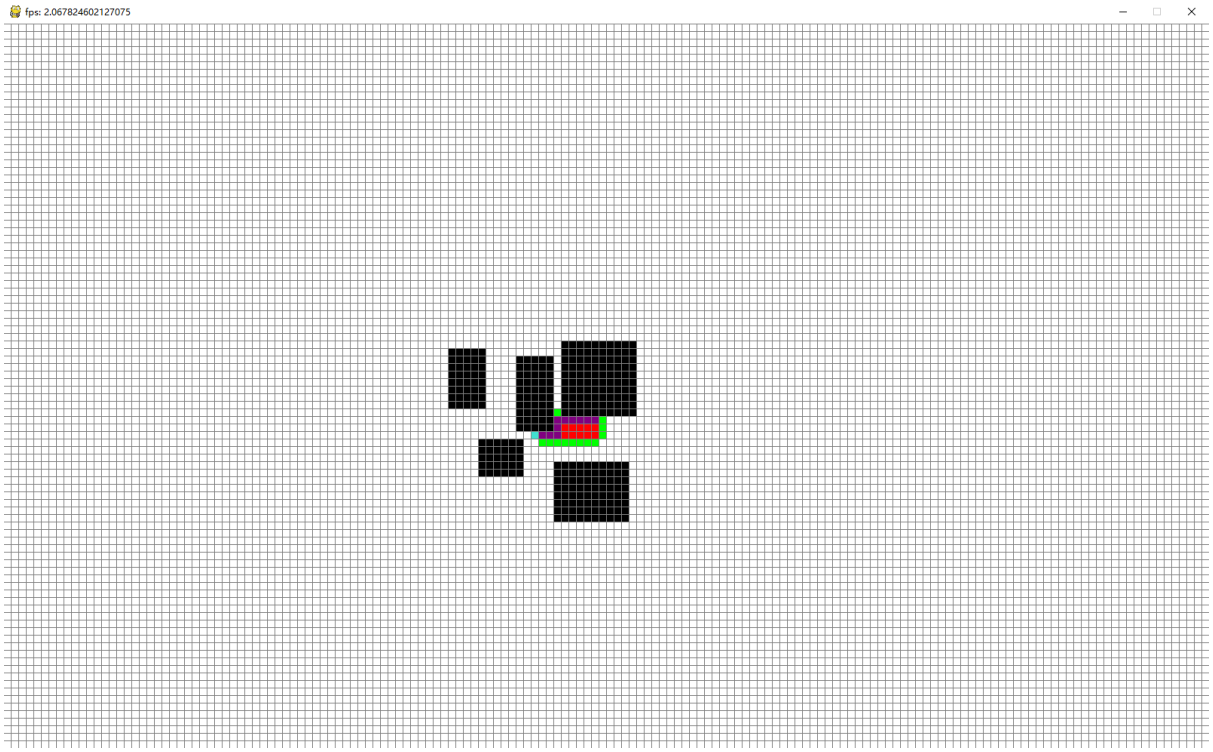


Figure 46 Pathfinding

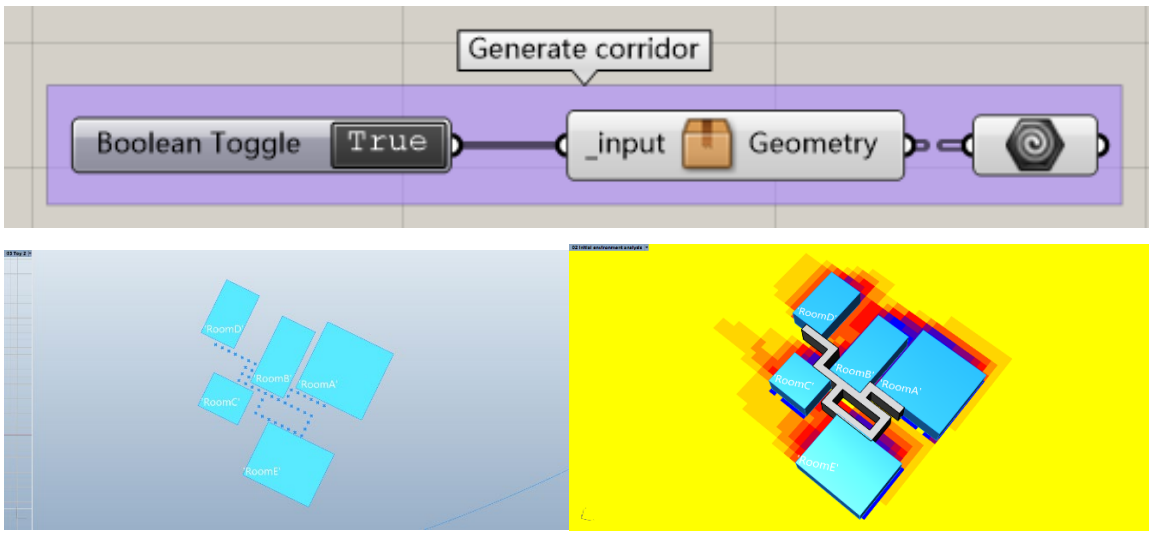


Figure 47 Generate corridor in rhino

4.3. Results

Figure 48 is the analysis result of the concrete room shapes from toy problem 2. Each number in the center of the room represents its desired sunlight hours we have inputted previously. The pure yellow grid means that the grid has a sunlight hour of 8 or more on the date of the winter solstice in the northern sphere. Each gradient of color represents the sunlight hour difference of 1 hour on that date.

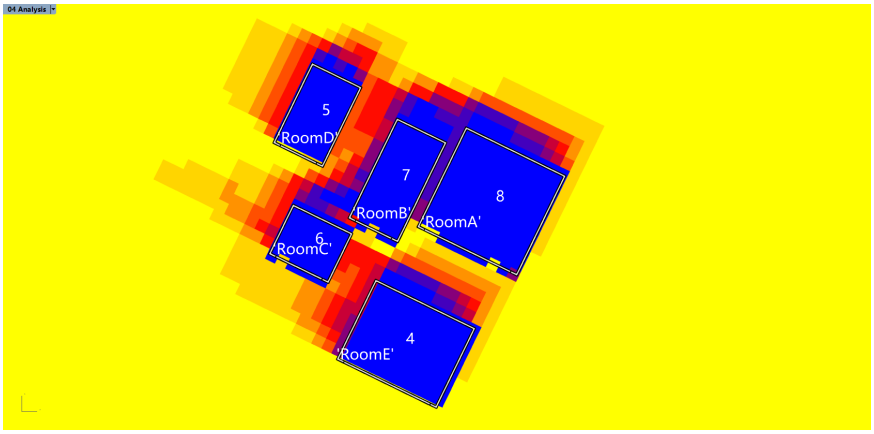


Figure 48 Ladybug daylight hour analysis result of toy problem 2.

After giving 1.8-meter height windows at a bottom height of 0.9 meters to one side of rooms. The indoor sunlight hour condition does meet some requirements. But still, the more barricades it has in front of the window, the less sunlight hour it gets. (Figure 49)

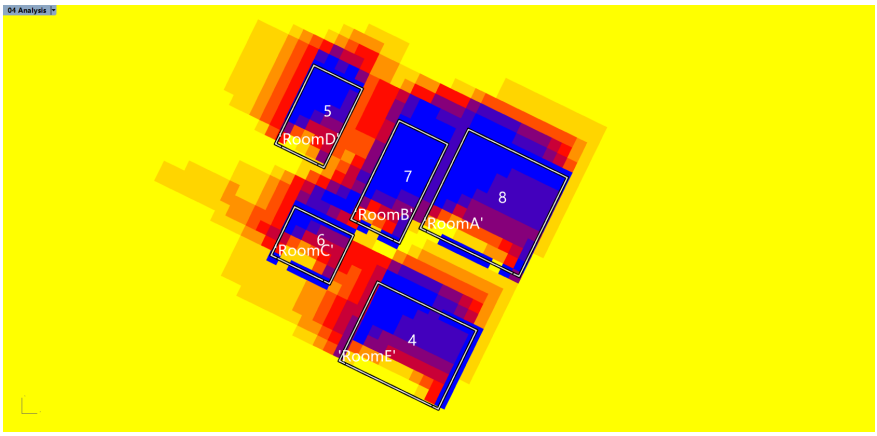


Figure 49 Ladybug daylight hour analysis result of toy problem 2 with window

After giving corridor a cap, here I assume it is an open corridor, the sunlight hour results get worse. RoomB and RoomA clearly do not meet their needs. (Figure 50)

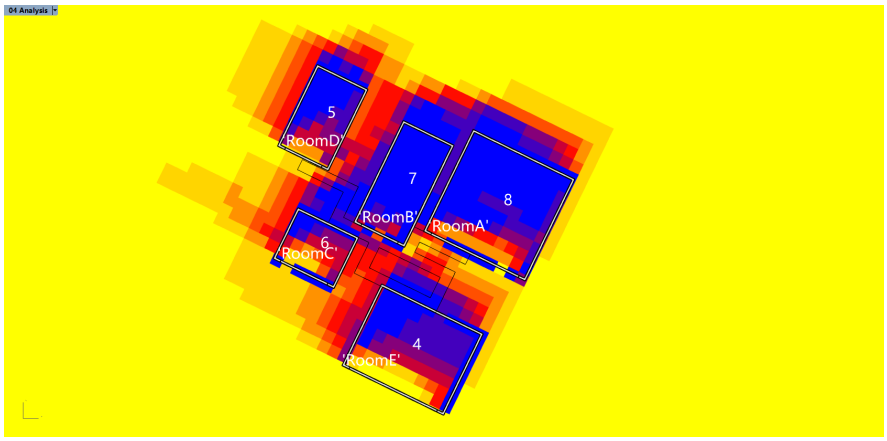


Figure 50 Ladybug daylight hour analysis result of toy problem 2 with window and corridor.

Adding more windows to the room does improve the daylight hour condition of the room, but it is not solving the problem. (Figure 51)

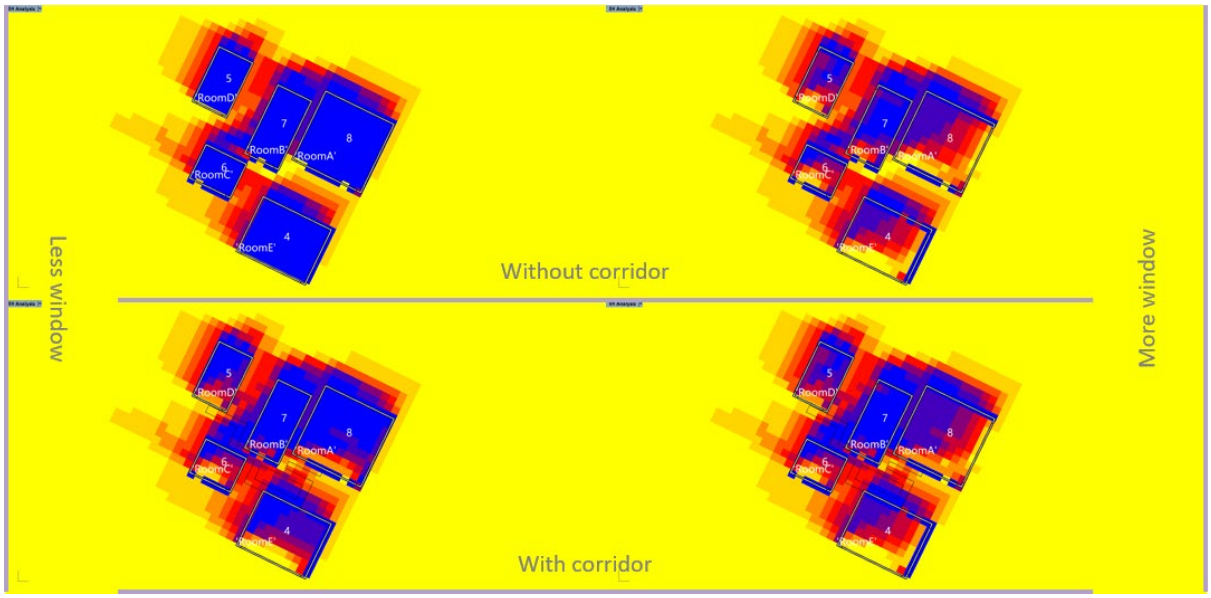


Figure 51 Comparison among results of less/no window, more windows, without corridor, with corridors.

Looking at the result, there are 2 questions that need to be thought:

1. Is the order of placement previously determined representative to the needs towards certain requirements?
The order of placement previously determined generally took the aspect of 'node importance' in a graph, then place the room from south to north. The direction of placement is purely from the consideration of putting down the shape without conflict. But if the designer is not aware of the process, the first impression might create more costs more for designers to reach a satisfying result.

2. What forms of information should be provided?

The placement method right now involves human intervention. So, there is a need of the information actively update to the designer, when he/she is dragging the mouse.

Next, viewing the result of the pathfinding algorithm (Figure 52), it produces redundant results. Path A is not the most optimal one among all. The logic of the pathfinding algorithm needs to be improved.

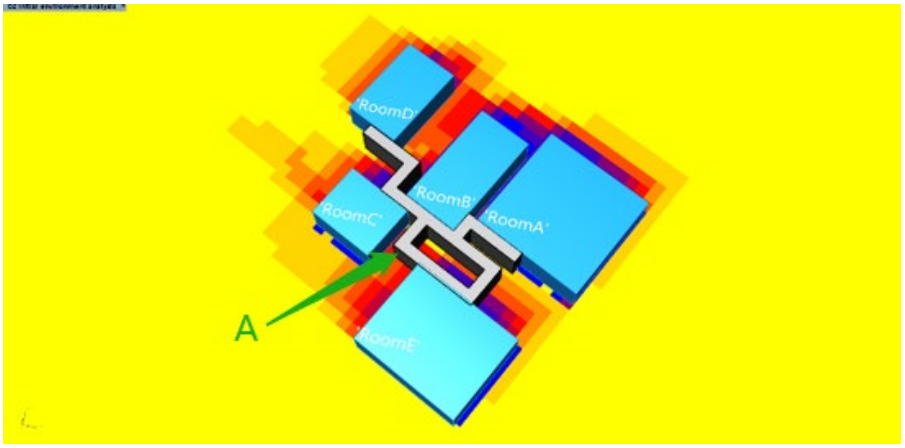


Figure 52 Example of non-optimal path direction.

5. Improvements

5.1. Optimization indicators

To provide designers with information through the process, indicators below are provided to designers:

Convex Hull to Room Ratio: It is the ratio between the size of the convex hull of the corner of all rooms and the total size of the rooms. The closer it is to 1, the more compact the layout is. The calculation of the convex hull is done by using Scipy Python library.

Potential length of each circulation: It is calculated by the Manhattan distance among all doors of rooms within the circulation. Therefore, it is only a very rough estimation, does not represent the final path length.

Satisfied unit to total unit ratio: It is the ratio of the number of grid unit that satisfied the desired daylight hour to the total grid unit of one room.

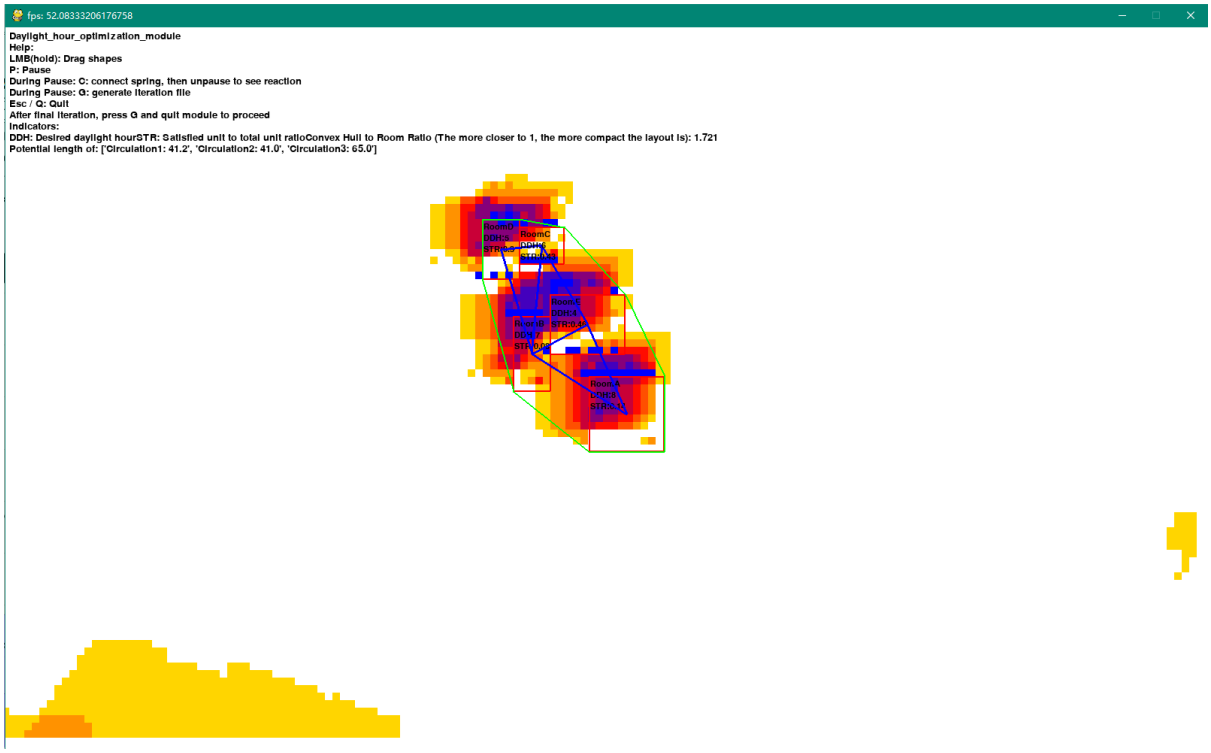


Figure 53 Improved interaction interface with indicators.

As shown in Figure 53, the green polyline represents the convex hull formed from vertices of all rooms, Convex Hull to Room Ratio and potential length of each circulation is shown on the up-left corner of interface together with help text. For easier acquiring of information during the interaction with room shapes, satisfied unit to total unit ratio (STR) and desired daylight hour (DDH) is shown together with the room name in the red room shape.

5.2. Object for daylight potential analysis

To provide more reasonable daylight hour potential of each room, the rhino geometry to be analysed is generated with windows on all sides of the walls.

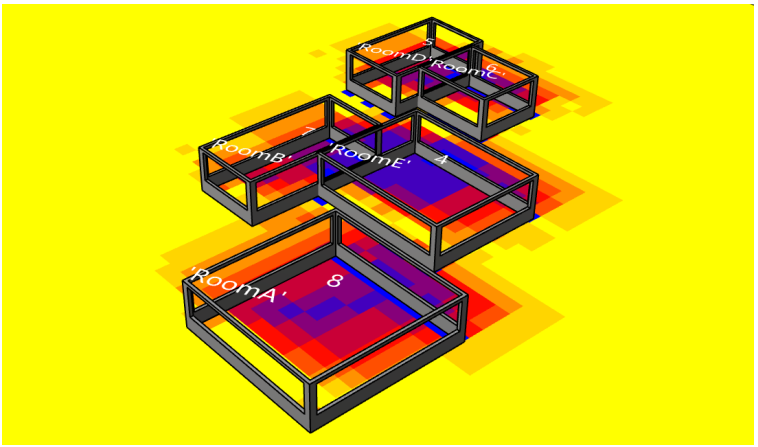


Figure 54 New geometry generated with windows.

5.3. Corridor selection

To find more optimal paths, when doing pathfinding, besides only from A to B, the path of B to A is also generated. Weights of circulations are given to each passed nodes. Nodes that are passed more than one times will accumulate the weight of passed circulations to the nodes. This is done by transforming the site matrix to Numpy matrix and adding node weight to corresponding elements of the matrix.

```
path = eval(infolist[0])
x_list = []
y_list = []
weight_list = []

path_array = np.zeros([96,160])

for i in range(len(path)):
    for j in range(len(path[i])):
        x = int(path[i][j][0])
        y = int(path[i][j][1])
        weight = path[i][j][2]
        path_array[y][x] += weight

x_list = []
y_list = []
weight_list = []
for i in range(96):
    for j in range(160):
        if path_array[i][j] != 0:
            x_list.append(j)
            y_list.append(96 - i)
            weight_list.append(path_array[i][j])
```

Figure 55 Accumulating weight to nodes by Numpy matrix.

In the end, after all paths are generated, nodes with accumulated weights are shown in Rhino space (Figure 56). Higher the weight, more important the node is within the overall circulation of the floor plan.

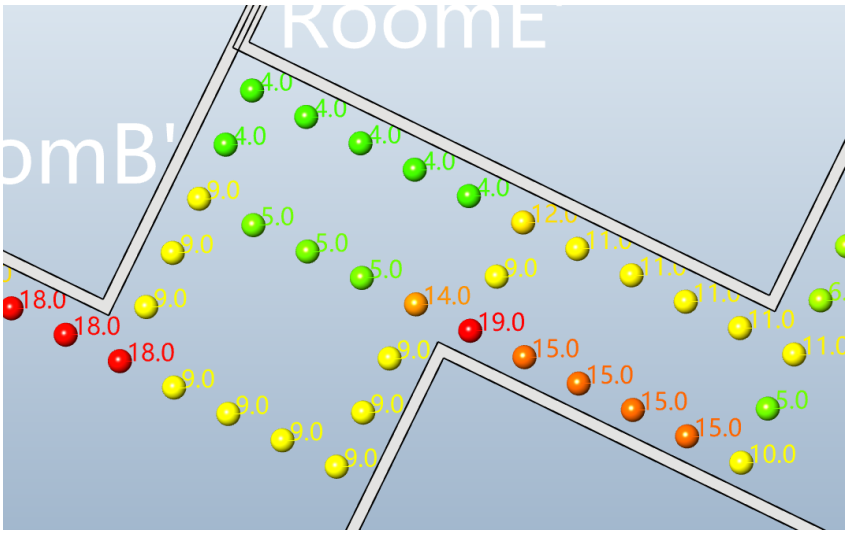


Figure 56 Nodes with accumulated weights

Then edges and edge weights are also shown in Rhino space, to help the designer with route selection. The designer can now decide which route to take to form corridors with not only the importance of circulation in mind but also the overall space structure it can form on the floor plan (Figure 57).

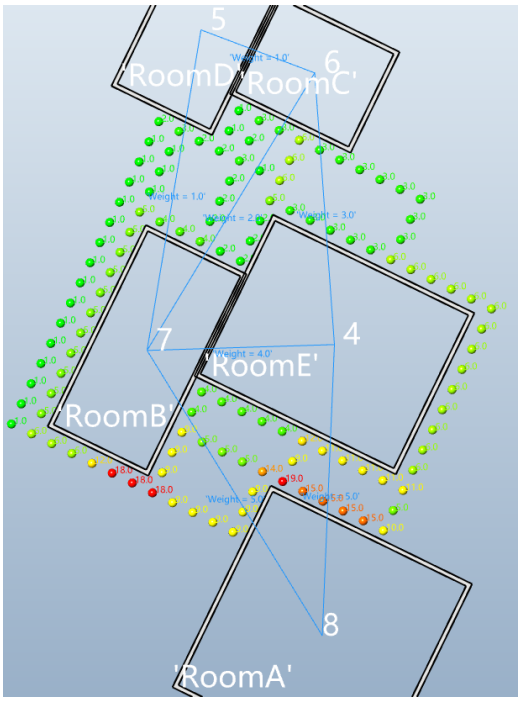


Figure 57 Corridor generation process

After above-mentioned improvements, the improved design methodology is as shown in figure 58. The workflow after the daylight hour analysis now requires more human interaction and decision making, while being provided with more information.

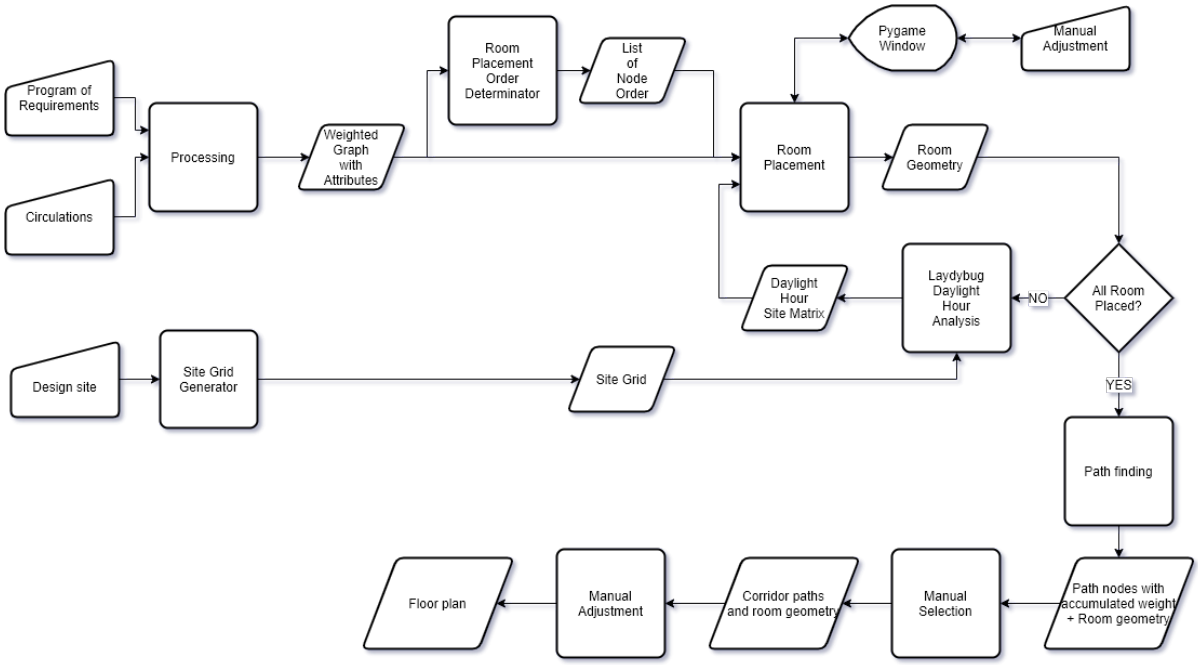


Figure 58 Improved design methodology

5.4. Test case: design of a single floor nursing home

To test the improved methodology, a simplified program is used to design a single floor nursing home. (Table 2)

Space	Size	Quantity	Space	Size	Quantity
Entrance hall	80	1	**Recreation:**		
Living:			Activity room	60	1
Living units	32	10	Gym	60	1
Medication:			Multi-media hall	120	1
Nursing station	20	1	**Administration:**		
Medical room	32	1	Offices	24	2
Public service:			**Others:**		
Cafeteria	120	1	Toilet	35	2
Kitchen	120	1	Equipment room	50	1
Laundry	40	1	Storage	24	1
Cleaning room	12	1	Logistic entrance	20	1
Convenient store	40	1			

Table 2 A simplified program

The simplified program is processed into excel files for easier input, then it is read with Python and an info file is generated.

Space	Size	Daylight hour	Door direction
Entrance hall	80	5	S
Unit 1	32	5	N
Unit 2	32	5	N
Unit 3	32	5	N
Unit 4	32	5	N
Unit 5	32	5	N
UnitNodeA	1	0	none
Unit 6	32	5	N
Unit 7	32	5	N
Unit 8	32	5	N
Unit 9	32	5	N
Unit 10	32	5	N
UnitNodeA	1	0	none
Nursing station	20	5	S
Medical room	32	5	S
Cafeteria	120	8	N
Kitchen	120	3	W
Laundry	40	3	E
Cleaning room	12	3	E
Convenient store	40	3	W
Activity room	60	8	N
Gym	60	3	E
Multi-media hall	120	3	W
Office 1	24	5	N
Office 2	24	5	N
Toilet 1	35	0	S
Toilet 2	35	0	S
Equipment room	50	0	S
Storage	24	0	S
Logistic entrance	20	0	S

PoR | Circulation | Sheet1 | Simplified

Circulation 1	Circulation 2	Circulation 3	Circulation 4	Circulation 5	Circulation 6	Circulation 7	Circulation 8	Circulation 9
UnitGroup1	UnitGroup2	UnitNodeA	UnitNodeB	Cleaner	Medical	Administrativ	Food	Logistic
5	5	3	3	4	4	3	2	1
Unit 1	Unit 6	UnitNodeA	UnitNodeA	Cleaning room	Nursing station	Entrance hall	Kitchen	Logistic entrance
Unit 2	Unit 7	Toilet 1	Toilet 1	UnitNodeA	Medical room	Office 1	Cafeteria	Cleaning room
Unit 3	Unit 8	Convenient store	Convenient store	UnitNodeB	UnitNodeA	Office 2	UnitNodeA	Equipment room
Unit 4	Unit 9	Activity room	Activity room	Laundry	UnitNodeB	Toilet 2	UnitNodeB	Storage
Unit 5	Unit 10	Gym	Gym	Cleaning room	Nursing station	Entrance hall	Kitchen	Logistic entrance
UnitNodeA	UnitNodeB	Multi-media hall	Multi-media hall					
Unit 1	Unit 6	UnitNodeA	UnitNodeA					

PoR | Circulation | Sheet1 | Simplified

Figure 59 Excel file with information

With the initial room placement module, the graph that contains all information is generated, and rooms are initially placed and adjusted by designer under the physics spring system (Figure 60 - 61).

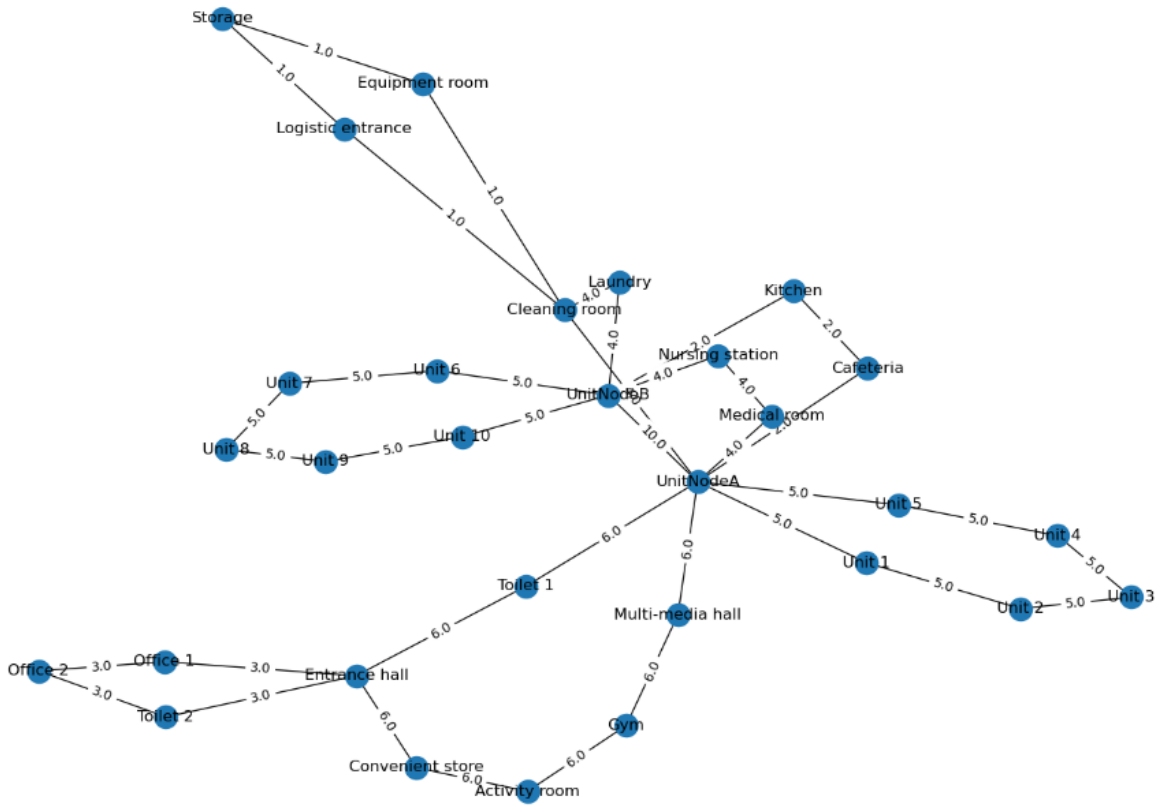


Figure 60 Graph generated

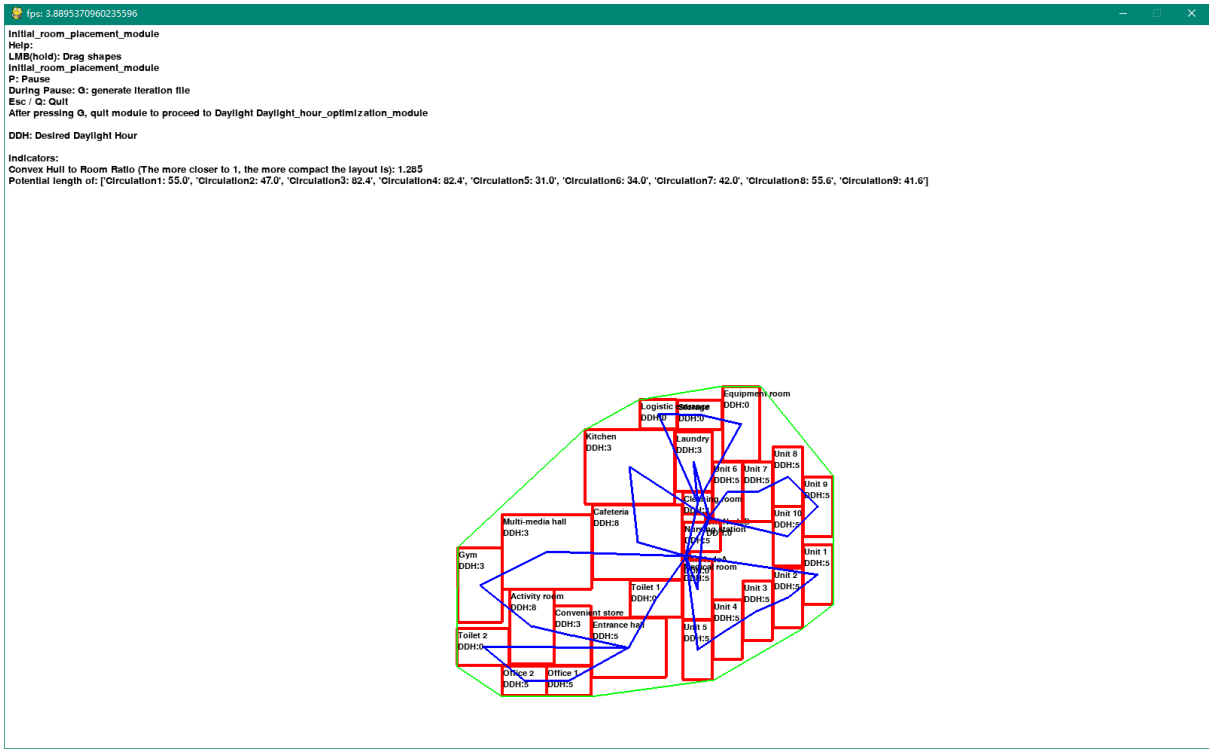


Figure 61 Initial room placement

After the designer is satisfied with the initial room placement, the information exchange file can be generated, then proceed to daylight hour optimization module. The same as in toy problem 2, after all rooms are placed and analysed step by step (Figure 62), all rooms are generated within the rhino space, and designer has a final room placement (Figure 63). Then proceed to the pathfinding module.

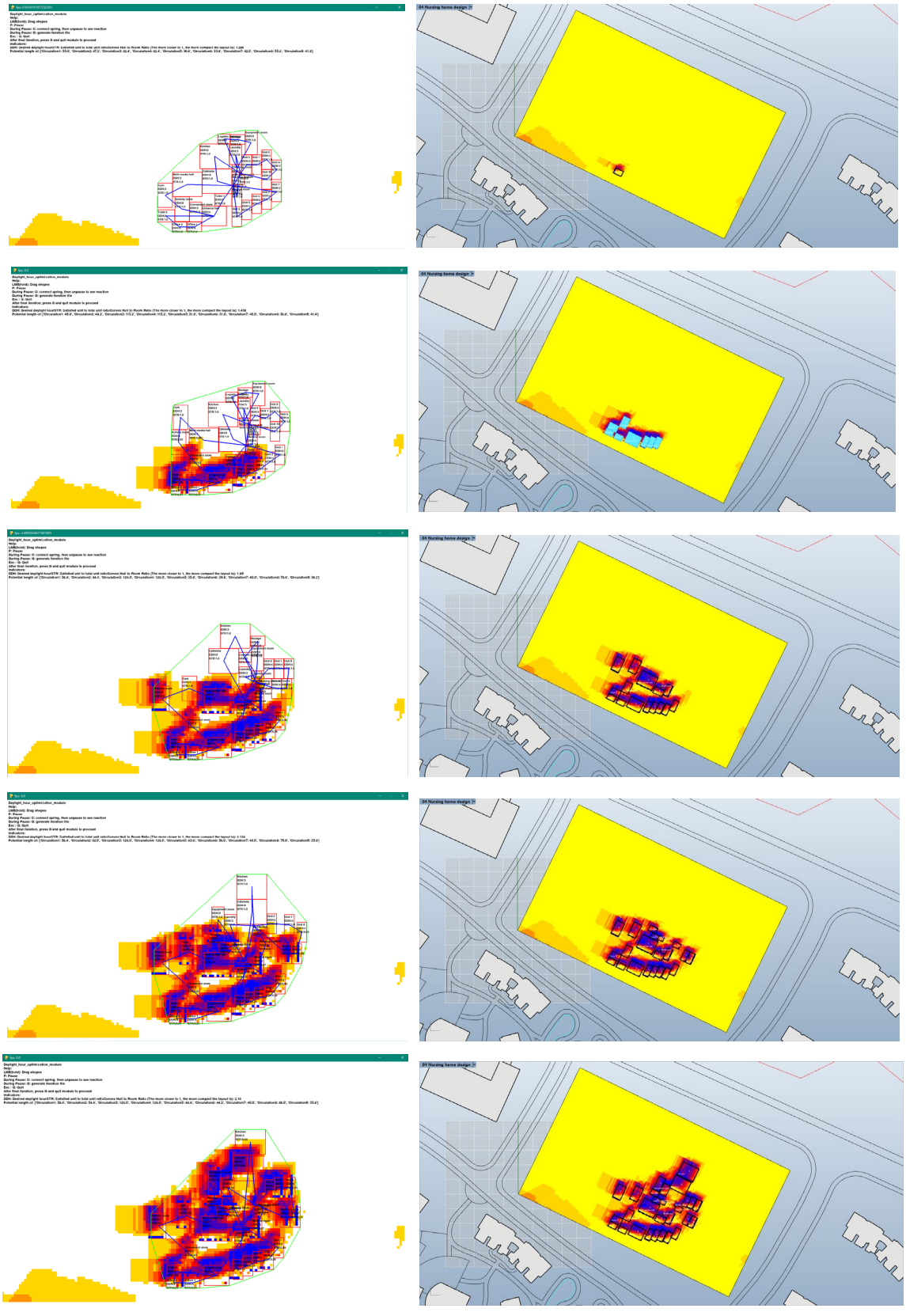


Figure 62 Start daylight hour optimization

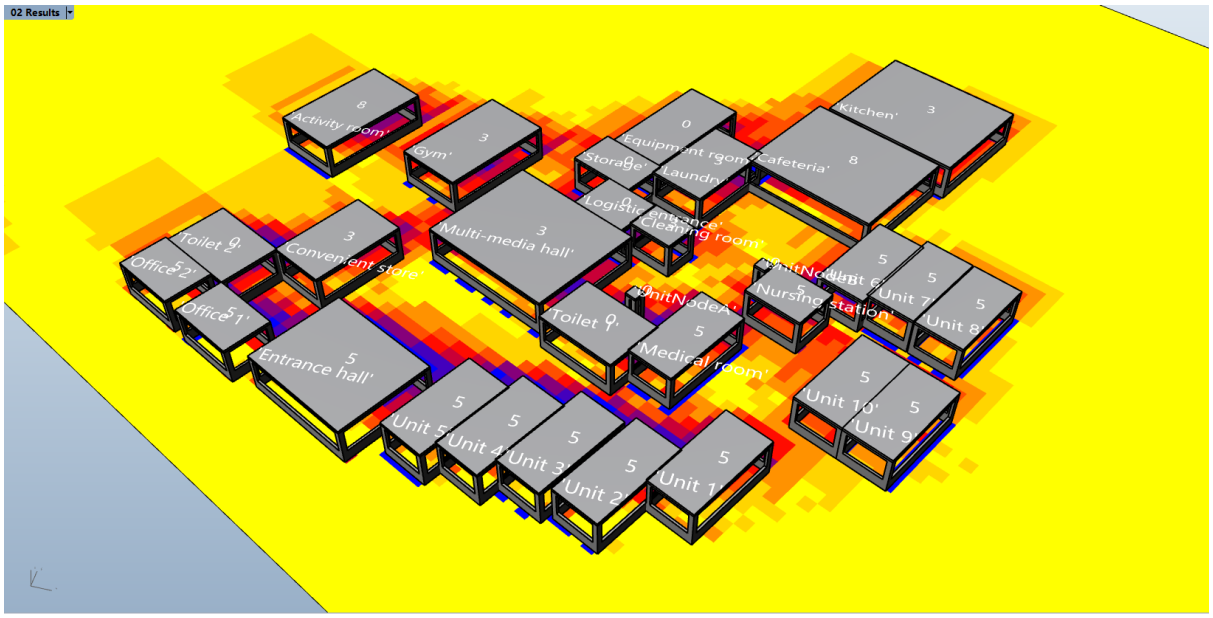


Figure 63 Final room placement in Rhino

In the pathfinding module, path node list generated with accumulated weight, after finished all pathfinding, proceed to corridor finder module.

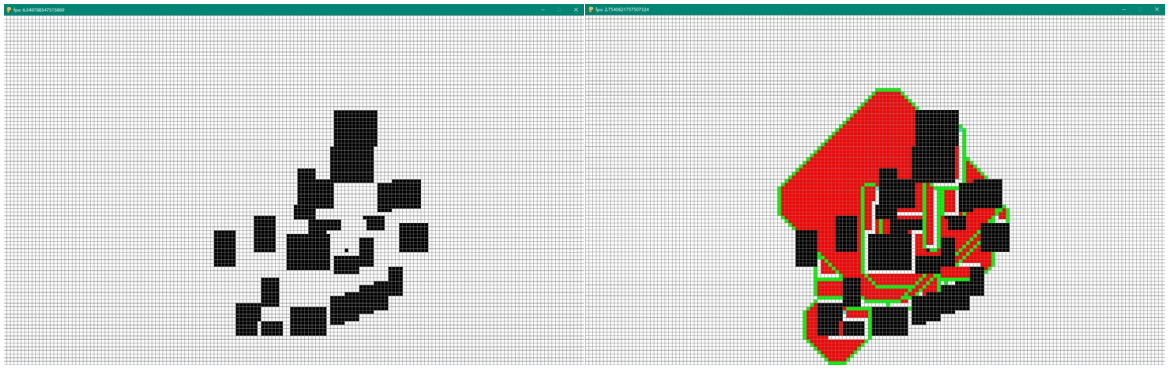


Figure 64 Pathfinding start (left) and finished (right)

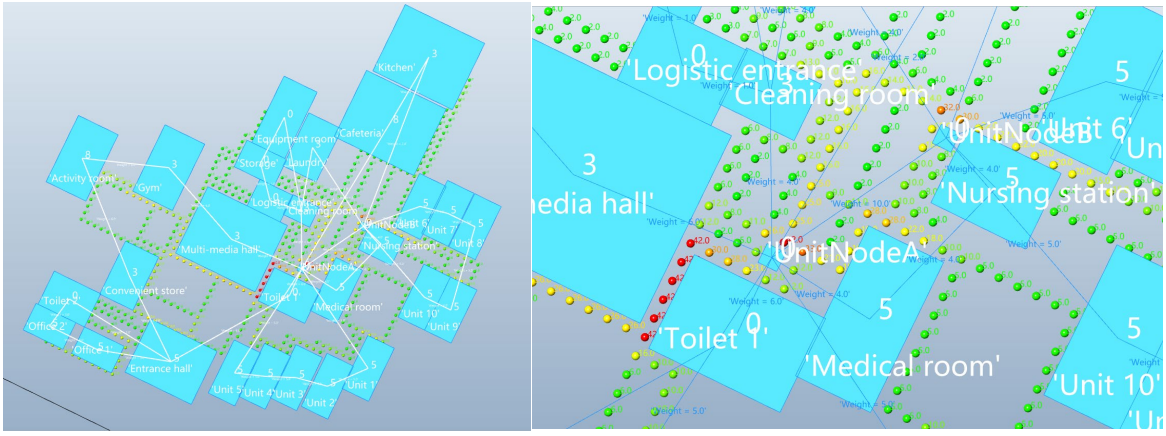


Figure 65 Path nodes with accumulated weight and edges shown with weight

In corridor finder module, it shows nodes in different colors together with their accumulated weight, the higher the weight, it means more shortest paths with high weight passed this node, make it potentially efficient to choose. In this step, it is the designer to combine his/her knowledge on the experiences of usability, form and structure of space, to decide, when all answers are suitable, which one suits his/her design vision more (Figure 66).

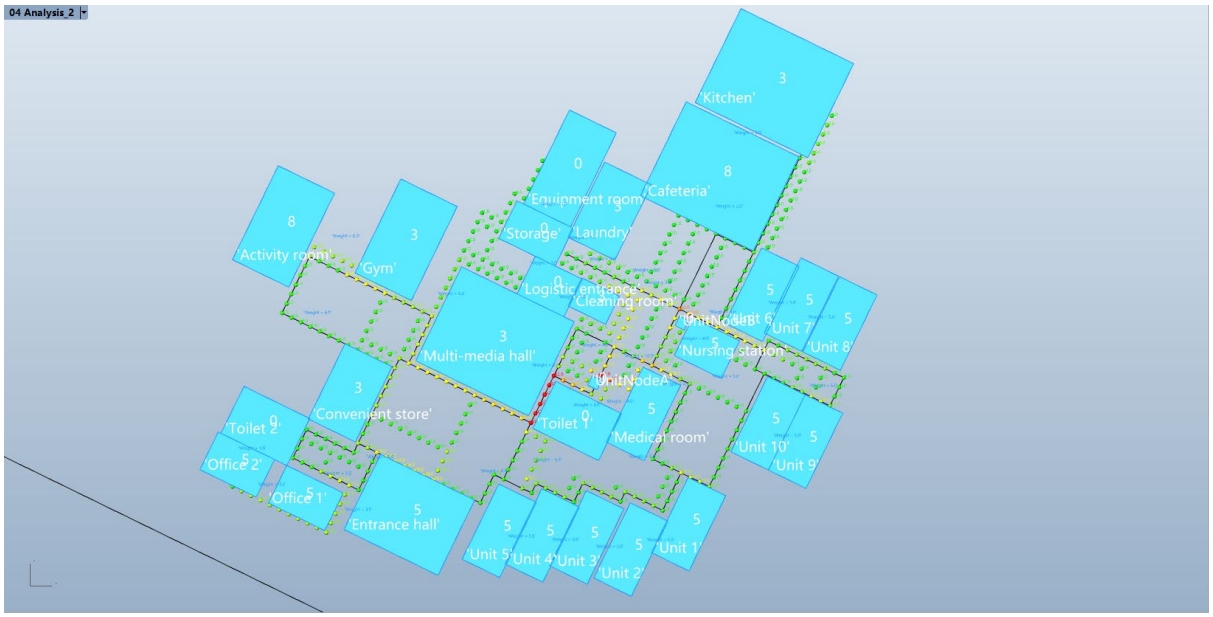


Figure 66 Manually selecting paths

The last is to manually adjust the layout if needed, then create corridors for the floor plan (Figure 67).

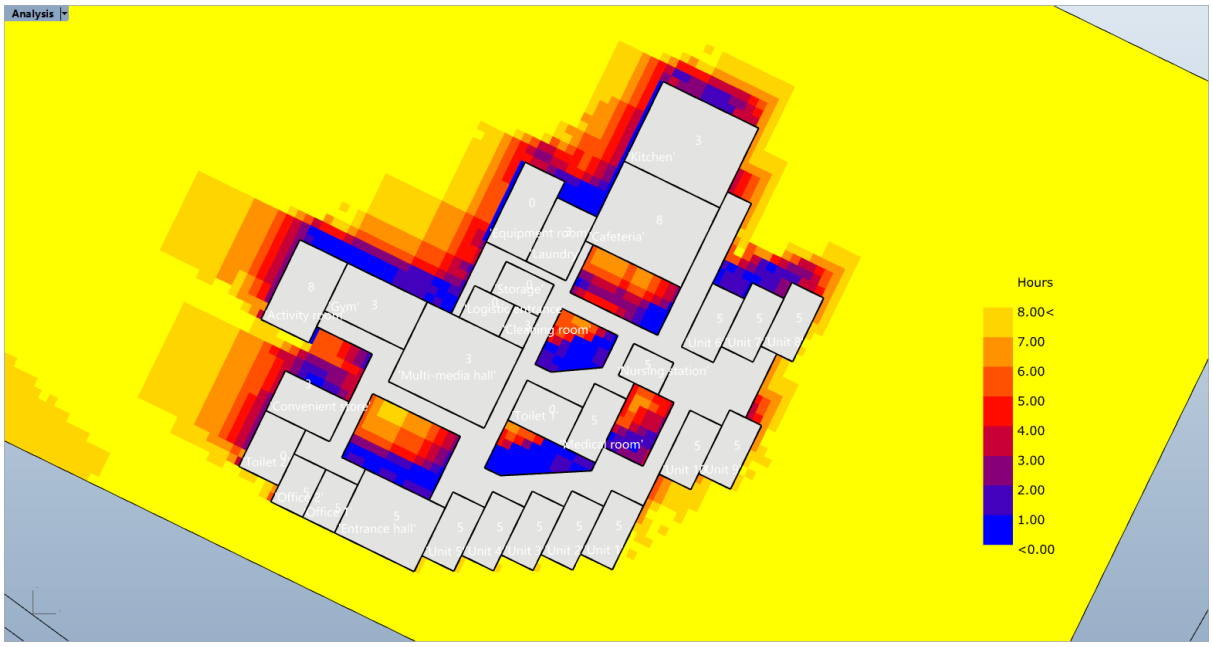


Figure 67 Manually adjusted results

6. Results

6.1. Evaluation and comparison with a manually designed result

Before applying this computational method, a single floor nursing home was designed manually under the same program of requirements.

The shape of the manually designed results is more regular, assumingly, much easier to construct, and looks more convincing (Figure 68). While the computationally designed result is irregular.

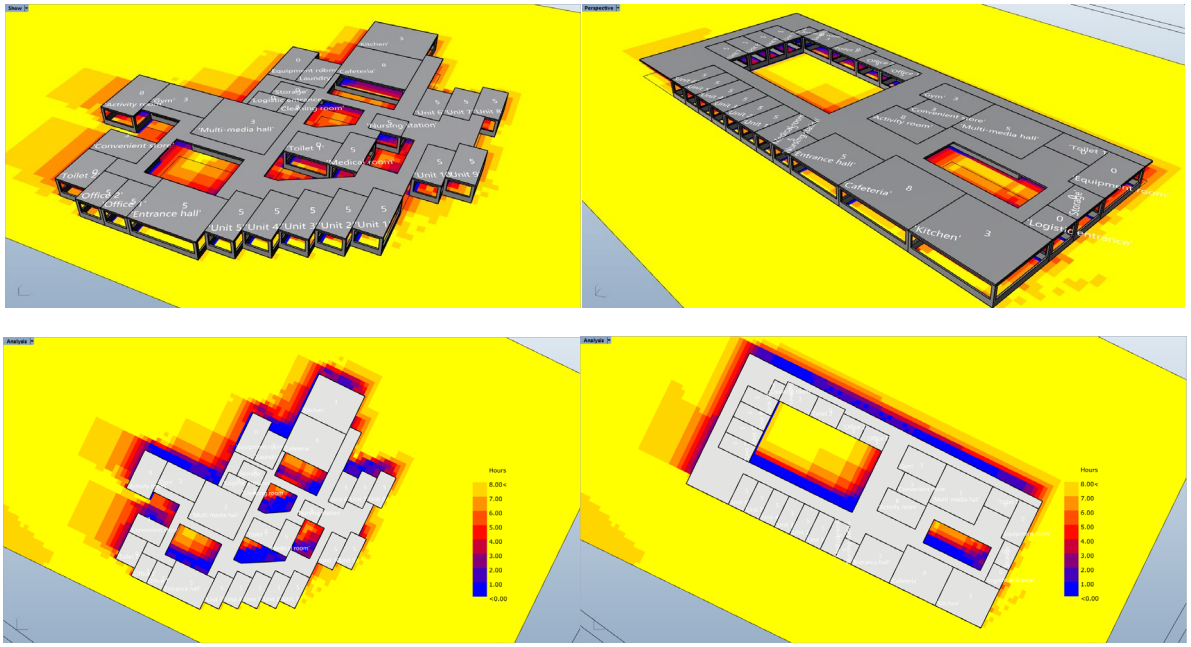


Figure 68 Comparisons between design done by the method (left) and design done by hand (right).

Using ladybug daylight hour analysis tool to analyse each result as a whole, when both fail to meet the desired daylight hour of some rooms, the daylight hour condition of computationally designed has more variety, which has the potential to provide interesting daylight environment. To be noticed, it might be the result of opening windows on all sides of the walls.

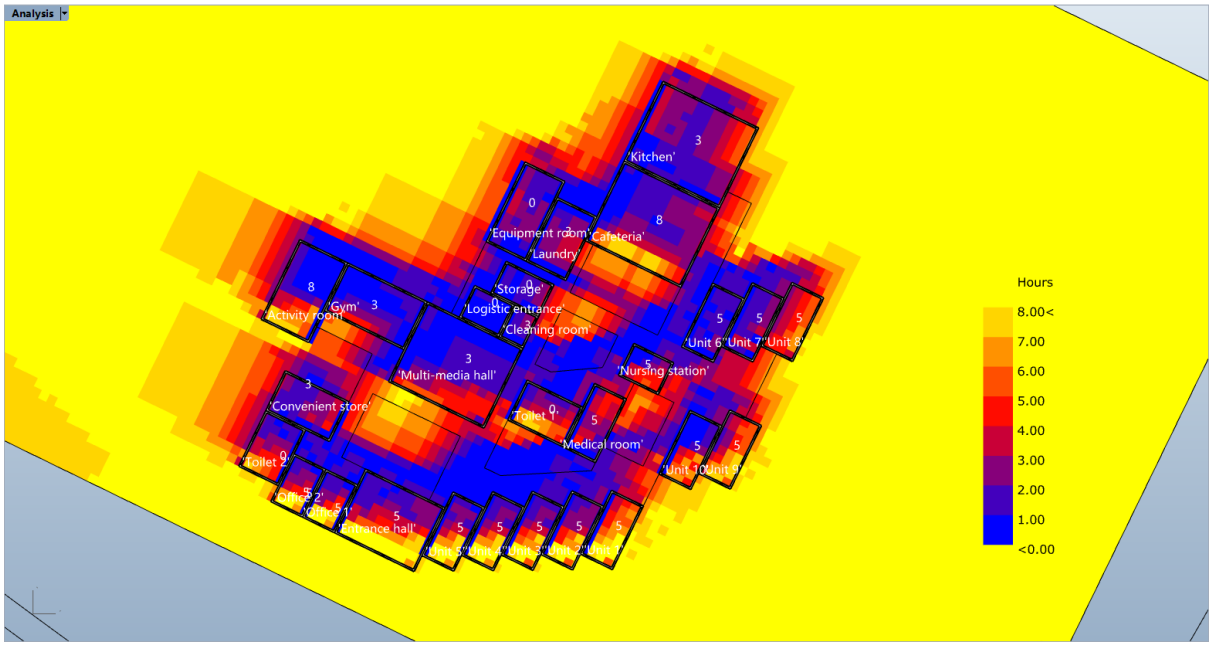


Figure 69 Ladybug sunlight hour analysis result on the winter solstice of design done by the method.

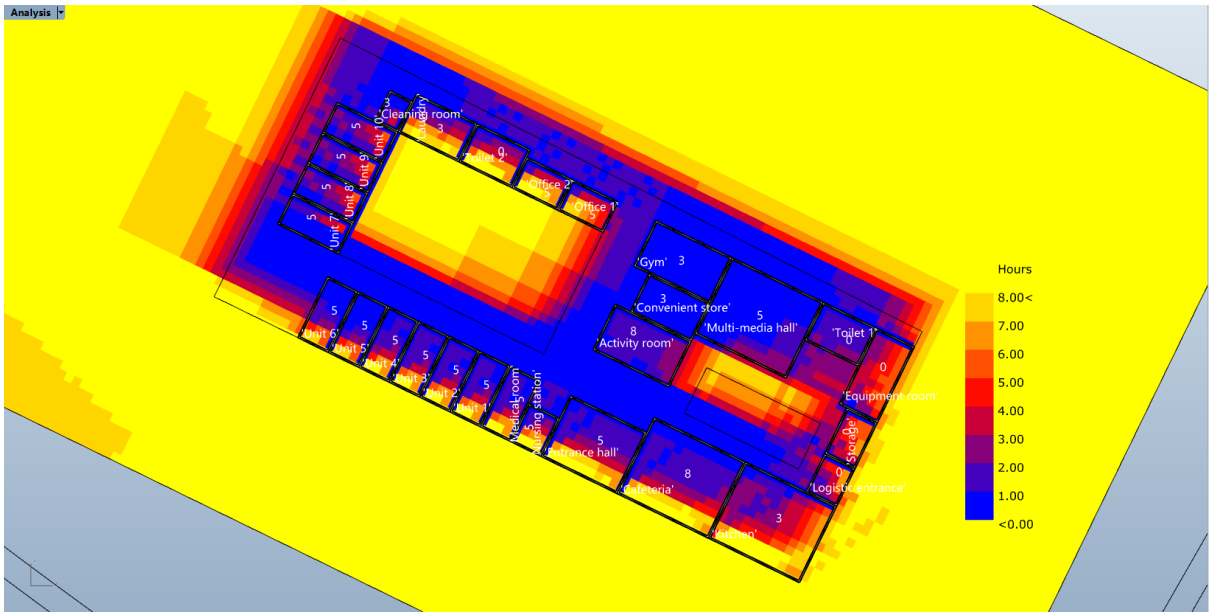


Figure 70 Ladybug sunlight hour analysis result on the winter solstice of design done by hand.

Geometry-wise, the area of convex hulls of both designs are very close, while the area of the corridor of the computationally designed plan is 52% of the manually designed plan. Comparing the results of the calculated length of each circulation, the computationally designed plan is more efficient than the manually designed plan (table 3).

	Designed by the method	Designed by hand
Area of convex hull	2583 m²	2580 m²
Area of corridor	568 m²	1084 m²
Length of complete elderly activity circulation	164.05 m	184.3 m
Length of cleaning staff circulation	103.70 m	137.73 m
Length of medical staff circulation	85.92 m	109.67 m
Length of administrative staff circulation	16.75 m	79.09 m
Length of logistic circulation	33.75 m	163.28 m
Length of food delivery circulation (to each unit)	126.37 m	155.64 m

Table 3 Comparison between two design results

But when looking at the program spatial distribution (Figure 71), computationally designed is interfering each other more. It reminds the designer to understand the mechanism of this method when using it as a tool.

Figure 71 Distribution of program on the floor plans

6.2. Summary

Comparing the two designs, both ways have their advantages. Manual design technique of designer is more trained, its result appears more regular and convincing. Program-distribution-wise, it is more reasonable, while this method brings more efficiency to the building, brings interesting daylight hour conditions. Some problems this computational method bring can be foreseeable solved. For example, by altering interacting behaviour with the module, the distribution of different program can be regulated to a certain extent, by adding new features to the module, like 'drawing courtyard' during the interaction, can create more space in the floor plan for desired daylight accessibility. There are still some problems left, which will be discussed in the next section.

7. Future work (limitation and open problems)

The first limitation of this methodology is that while this thesis involves optimization theory, graph theory, force-directed methods, computational geometry and daylight hour analysis in some way, the detailed discussion of each subject falls out of the scope of the project. So, the substitutional methods being used for specific purposes might be inaccurate.

For example, the reason that I define the daylight hour analysis as “daylight hour potential”, is because the actual daylight hour analysis requires much more, including room size, window size, room solid boundaries, etc. Placing the room with window opening on all sides above the Ladybug daylight hour analysis matrix just neglects lots of things mentioned above. Because of the appearance of the roof of rooms, placing and analysing rooms from south of the map to the north helps to regulate the result, but for rooms that are horizontally attached or close to each other, it is still inaccurate. Therefore, more detailed analysis needs to be done in the workflow of design after this. It cannot be seen as a final answer. In the future development, the smart window opening feature can be added to the module.

Second, the current workflow will cause a problem, that when generating corridors after the daylight hour analysis, the generated corridor is very possible to block the sun and change the daylight condition of the site. This requires a different workflow or method to maybe generate corridors while placing rooms for daylight analysis.

Third, this methodology for now only produces 2D plans, room shapes are set to rectangles, not flexible as real life, the room dimensions are not flexible. A more adaptive room shape generation module can be worked in future.

Fourth, in the pathfinding algorithm, the door direction and position are pre-assigned to each room and manually adjusted before the pathfinding process. Therefore, door positions in the results might not be the optimal ones, potentially making the layout and paths not the most optimal ones. The door-position finding module can be added in future for finding optimal door position in a room.

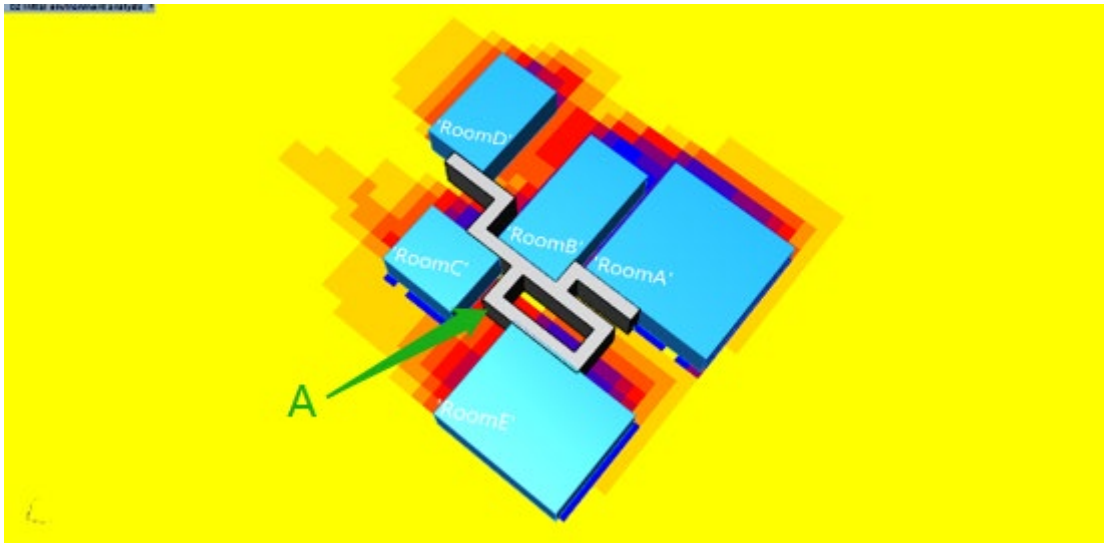


Figure 72 Example of non-optimal path direction.

Fifth, the iteration process of the daylight hour analysis is done manually, which is rather labour intensive. A better workflow might be possible by adding looping components to grasshopper. And, the current process requires human intervention, it can be good if a fully automatic option can be offered, in case of a large number of rooms. It requires the update of the algorithm on the room placement method, where more mathematics can kick in. And, current indicators are generally single-room there is a lack of an overall daylight hour evaluation of the whole floor plan

Sixth, following the fifth, the module is developed in Python 3, with 2d physic engine Pymunk, 2d game library Pygame. One thing it results is that, when the number of rooms and spring joints among rooms goes up, the general CPU occupation stays low while single-core occupation goes rocket high, and the frame rate drops significantly, for example, during the nursing home design, it is around 3 – 7 fps. So more efficient alternative platform or tools shall be sought in future developments.

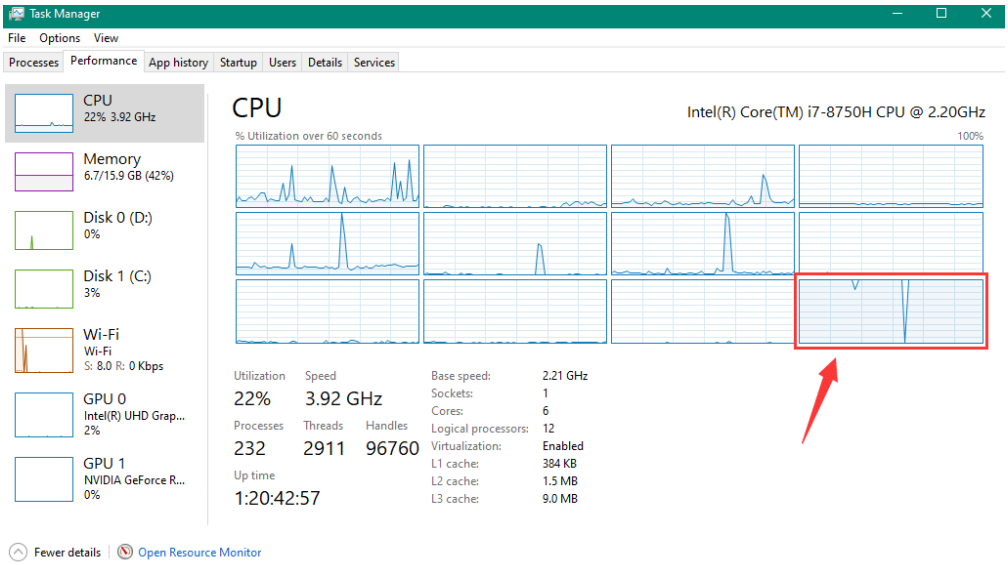


Figure 73 Hardware occupation

8. Discussion

In Yona Friedman's book "Towards A Scientific Architecture" (Friedman, 1975), Friedman visioned future architects to be professional consultants that clearly explain pros and cons of each factor from the pool of exhaustive design options that computer generates, then choose the one that bests suits clients' vision.

Meanwhile, computers generate computationally correct and perfect, but practically bad results, based on the objective functions and human-designed algorithms, it still cannot substitute the current role of designers.

While vision is far, designers need a practical method to embed multi-criteria evaluation into the early design process. The method I propose can be supplementary to the early design phase. It is intuitive, easy to apply. It can empower designers to sort out relations and reactions between circulation, walking path and daylight hour potential of rooms, by actively intervene in the process. And provides some suggestive solution.

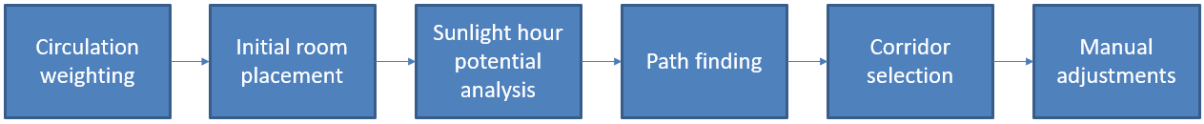


Figure 74 Proposed framework

I propose this framework of 'Circulation weighting – Initial room placement – sunlight hour potential analysis – pathfinding – corridor selection – manual adjustments' on 2-dimensional space. Future work is to make this tool more informative, efficient in performance, reliable in result, reduce the time of designers cost on trials and errors. And extend the framework to 3-dimensional space, where more complex cases can be helped.

9. Summary

Looking back at the sub-questions:

- How should the program of requirements be modelled into spatial relations?
  As illustrations like bubble diagrams have been widely used and accepted by designers, the graph can be a good and intuitive carrier of architectural information. In a design that considers circulation efficiency, circulation relations can be represented by edges of the graph.

- How can spatial relations be projected into an actual space?

Force-directed method can be an intuitive method to visualize spatial relations and project it into an actual space.

- How should the relations among evaluating circulations, walking distance and daylight hour be? In another word, what is the hierarchy?
  In my design, I put circulation evaluation in front of the daylight hour analysis, then after daylight hour analysis, I generated potential walking nodes with pathfinding algorithm. The result showed some disadvantages, but that is mainly due to the execution of room placement order, which needs to be improved. Daylight hour analysis should be more integrated into the previous process.

- How should designers interact with the computational process? What is his/her role?
  After having a full understanding of how the process work, designers can actively intervene the computation progress with his/her own thinking/consideration for a more humanity-wise ideal result, and like Friedman's vision, determining a convincing result at the end.

10. Conclusion

This methodology from a certain extent can be a supplementary method to the early stage of an architecture design workflow. It can help designers make fast decisions and evaluations at the early stage of design without investing too much into it.

While awaring of its limitations, it is applicable for designs with the need of daylight hour optimization, and it is sufficient for designs with the needs for circulation efficiency to a certain extent.

The ideology behind it can be a framework for the future development of such methodologies involving multi-criteria space configuration. Besides working with this methodology on the floor plan level, this method also has the potential of working on a larger scale, where the general layout of a set of buildings is required to consider the daylight.

Acknowledgement

I would like to thank my mentors, Pirouz Norian and Regina Bokel, and Geert Coumans, the delegate from Board of Examiners. Pirouz led me through the door of graph theory, computation and ideas of layout optimization, he and Regina brought me knowledge on every aspect from how to define research, how to plan and proceed them, to how to structure and write a thesis and how to present the result to the audience. Without the generous help and advice they provided, this thesis will not exist. Geert attended all my presentations, the advice from him has also been very helpful for going through each stage of this research.

A big thank you to all my friends, their existence enlightened my life during these two years of study, especially when going through such a difficult time.

In the end, I would like to thank my parents, without their care, help and support, I would not be able to pursue and finish my master study.

References

Abdelrahman, M. (2017). GH_CPython: CPython plugin for grasshopper. https://doi.org/10.5281/zenodo.888148

API Reference—Pymunk 5.7.0 documentation. (n.d.). Retrieved September 28, 2020, from http://www.pymunk.org/en/latest/pymunk.html

Chaillou, S. (2019, July 9). AI & Architecture. Retrieved from https://towardsdatascience.com/ai-architecture-f9d78c6958e0.

Dormans, J., & Bakkes, S. (2011). Generating Missions and Spaces for Adaptable Play Experiences. IEEE Transactions on Computational Intelligence and AI in Games, 3(3), 216–228. doi: 10.1109/tciaig.2011.2149523

Egor, G., Sven, S., Martin, D., & Reinhard, K. (2020). Computer-aided approach to public buildings floor plan generation. Magnetizing Floor Plan Generator. Procedia Manufacturing, 44, 132–139. https://doi.org/10.1016/j.promfg.2020.02.214

Flores Villa, L. M., Unwin, J., & Raynham, P. (2017, September 17). *The effect of daylight on the elderly population* [Proceedings paper]. Lux Europa 2017: European Lighting Conference - Lighting for modern society. Proceedings of the Lux Europa 2017 European Lighting Conference - Lighting for Modern Society; Lighting engineering society of Slovenia. http://www.sdr.si/sl/index.html

Friedman, Y. (1975). Toward a scientific architecture. MIT Press; WorldCat.org.

Linden, R. V. D., Lopes, R., & Bidarra, R. (2014). Procedural Generation of Dungeons. IEEE Transactions on Computational Intelligence and AI in Games, 6(1), 78–89. doi: 10.1109/tciaig.2013.2290371

New National Gallery and Ludwig Museum. (2015). El Croquis, 179-180, 160-165.

Niemann, M. (2015). Constructive Generation Methods for Dungeons (dissertation).

Nisztuk, M., & Myszkowski, P. B. (2019). Hybrid Evolutionary Algorithm applied to Automated Floor Plan Generation. International Journal of Architectural Computing, 17(3), 260–283. doi: 10.1177/1478077119832982

Nourian, P. (2016). Configraphics: Graph Theoretical Methods for Design and Analysis of Spatial Configurations [Delft University of Technology]. https://doi.org/10.7480/abe.2016.14

Nourian, P, (2019) How to write a thesis? A Generative Design Graduate Studio Guidebook. (2019). Delft University of Technology, Faculty of Architecture and Built Environment, Department of Architectural Engineering and Technology. DOI: 10.6084/m9.figshare.11987328

Roudsari, M. S., & Pak, M. (2013). Ladybug: A parametric environmental plugin for grasshopper to help designers create an environmentally-conscious design. Proceedings of the 13th International IBPSA Conference Held in Lyon, France Aug 25–30th, 3128-3135.

Ruscica, T. (n.d.). A* Pathfinding Visualization Tutorial—Python A* Path Finding Tutorial. Retrieved September 28, 2020, from https://www.youtube.com/watch?v=JtiK0DOeI4A&t=3818s

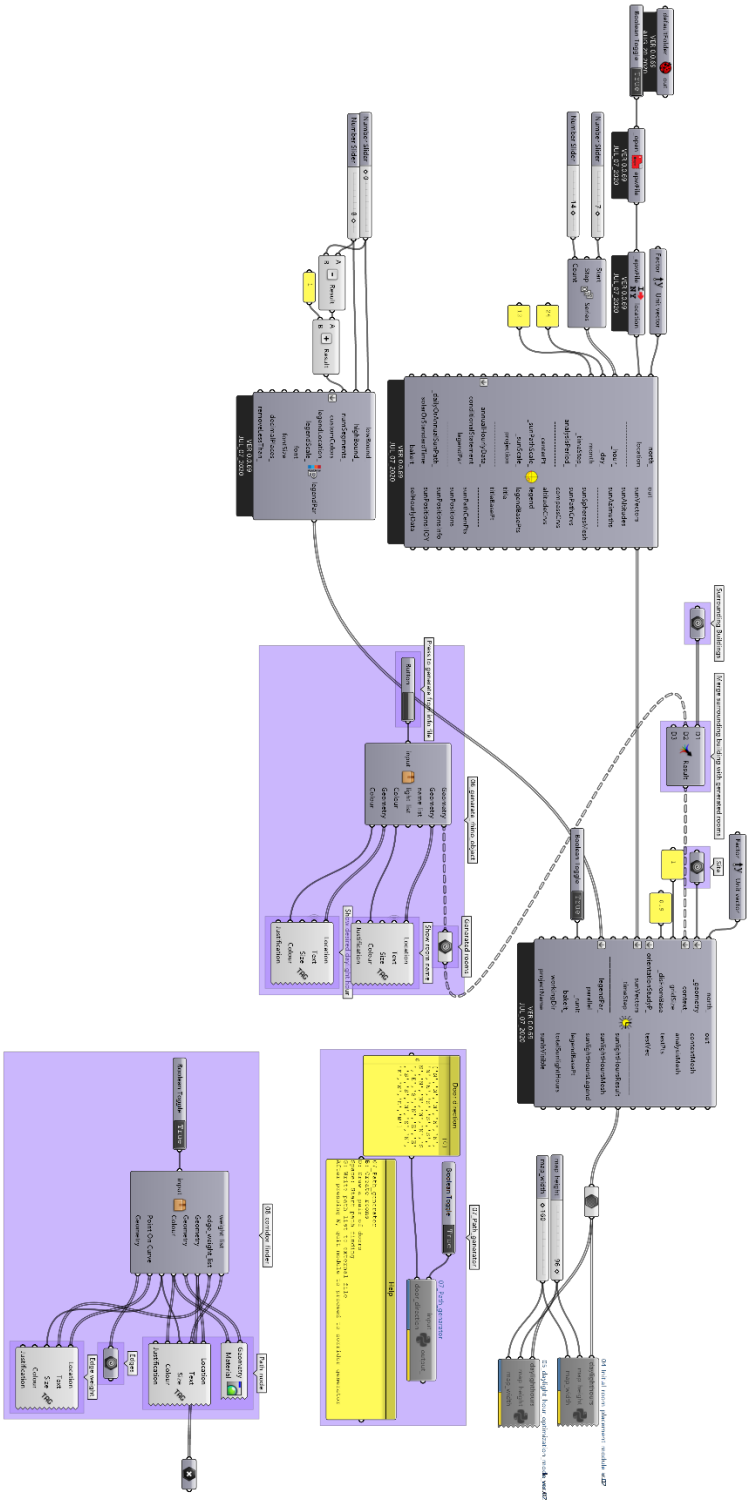
Simon, J. (2019). Evolving Floorplans. Retrieved January 8, 2020, from https://www.joelsimon.net/evo_floorplans.html.

World Health Organization. (1995). WHOQOL-100. Geneva, Switzerland: World Health Organization.

Zhou, Y. (2018). Yang lao she shi jian zhu she ji xiang jie = Design and interpretation of elderly care facility. Beijing: China Architecture & Building Press.

Appendix

Appendix A: Grasshopper layout



Appendix B: Python code of program of requirements excel reading

```
import openpyxl

rooms = []
sizes = []
lights = []
door_direction = []
circulations = []
cir_Weight = []

wb = openpyxl.load_workbook('D:\Delft Courses\Graduation\P5\PoR.xlsx')

# Getting sheet from the workbook

sheet_PoR = wb["PoR"]
sheet_Circulation = wb["Circulation"]


colSpace = sheet_PoR['A']
colSize = sheet_PoR['B']
colDL = sheet_PoR['C']
colDD = sheet_PoR['D']

for i in range(2,32):
    room = sheet_PoR.cell(row=i,column=1).value
    size = sheet_PoR.cell(row=i,column=2).value
    light = sheet_PoR.cell(row=i, column=3).value
    dd = sheet_PoR.cell(row=i, column=4).value
    rooms.append(room)
    sizes.append(size)
    lights.append(light)
    door_direction.append(dd)

col_range = sheet_Circulation["A:I"]
for col in col_range:
    circulations_temp = []
    for cell in col:
        if cell.value != None:
            circulations_temp.append(cell.value)
        else:
```

```
                pass
        circulations.append(circulations_temp[3:])
        cir_Weight.append(circulations_temp[2])

print(rooms)
print(sizes)
print(lights)
print(door_direction)
print(circulations)
print(cir_Weight)


f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/01_Initial_program.txt',
mode='w',
                                        encoding='utf-8')
f.write(f'{rooms}')
f.write('\n')
f.write(f'{sizes}')
f.write('\n')
f.write(f'{lights}')
f.write('\n')
f.write(f'{door_direction}')
f.write('\n')
f.write(f'{circulations}')
f.write('\n')
f.write(f'{cir_Weight}')
```

Appendix C: Python code of module 04_initial_room_placement

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull, convex_hull_plot_2d
import os

import sys

import inspect
import math

import pygame
from pygame.color import *
from pygame.locals import *

import pymunk
import pymunk as pm
from pymunk.vec2d import Vec2d
import pymunk.pygame_util
import pymunk.autogeometry


def draw_helptext(screen, hull_to_room_ratio, len_text):
    font = pygame.font.Font(None, 20)
    text = ["Initial_room_placement_module",
            "Help:",
            "LMB(hold): Drag shapes",
            "Initial_room_placement_module",
            "P: Pause",
            "During Pause: G: generate iteration file",
            "Esc / Q: Quit",
            "After pressing G, quit module to proceed to Daylight
Daylight_hour_optimization_module",
            "",
            "DDH: Desired Daylight Hour",
            "",
            "Indicators:",
            f"Convex Hull to Room Ratio (The more closer to 1, the more compact the layout is):
{hull_to_room_ratio}",
            f"Potential length of: {len_text}"
            ]
```

```
        y = 5
        for line in text:
            text = font.render(line, 1, THECOLORS["black"])
            screen.blit(text, (5, y))
            y += 15


def draw_graph(graph):
        # draw graph with weight
        pos = nx.spring_layout(graph)
        nx.draw(graph, pos)
        node_labels = nx.get_node_attributes(graph, 'room')
        nx.draw_networkx_labels(graph, pos, labels=node_labels)
        labels = nx.get_edge_attributes(graph, 'weight')
        nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels)
        plt.show()
        pass


# ↑ done, draws the graph


def por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight):
        i_sizes = 0
        i_lights = 1
        i_weight_sum = 2
        i_edge_sum = 3
        i_door_direction = 4

        ReL_Array = np.zeros([len(rooms), len(rooms)])
        Attri_Array = np.zeros([len(rooms), i_door_direction + 1])

        # creating ReL_Array
        for i in range(len(rooms)):
            for a in range(len(circulations)):
                for b in range(len(circulations[a]) - 1):
                    if circulations[a][b] == rooms[i]:
                        ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]
                    else:
                        pass

        # Flip the array left and right then rotate the array by 90 degrees
        ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
```

```
    # Creating attribute array
    for i in range(len(rooms)):
        Attri_Array[i, i_sizes] = sizes[i]
        Attri_Array[i, i_lights] = lights[i]

    # get Sum of edge weights on one node
    sum_weight = []
    for i in range(len(rooms)):
        sum_weight.append(sum(ReL_Array[i]))

    # Creating graph in networkx from ReL_Array
    G = nx.from_numpy_array(ReL_Array)
    for i in range(len(rooms)):
        G.nodes[i]['room'] = rooms[i]
        G.nodes[i]['sizes'] = sizes[i]
        G.nodes[i]['lights'] = lights[i]
        G.nodes[i]['sum_weight'] = sum_weight[i]
        G.nodes[i]['door_direction'] = door_direction[i]

    print("G.edges = ", G.edges(data=True))    # print edges
    print("G.nodes = ", G.nodes(data=True))    # print all nodes with attribute dictionary
    print("G.degree() = ", G.degree())
    print("sum_weight = ", sum_weight)
    return G


# ↑ done, returns a graph contains all information, weights on edges, attributes on nodes


def site_daylight_map(daylighthours, map_height):
    '''
    site_daylight_map = site_daylight_map(daylighthours, map_height, map_width)
    site_daylight_map = site_daylight_map.tolist()

    These two lines should be used at the end of the code in GH_CPython components,
    to pass arrays among GH_CPython components within grasshopper in the form of list.
    :param daylighthours: "sunlightHoursResult" list received from Ladybug Tools
"sunlightHoursAnalysis" component,
    :param map_height:    Number of grid on y axis
    :return: A numpy array of "sunlightHoursResult" that corresponds to the site
    '''
    daylight_map = [daylighthours[i:i + map_height] for i in range(0, len(daylighthours), map_height)]
    np_daylight_map = np.array(daylight_map)
    np_daylight_map = np.rot90(np_daylight_map)    # match the numpy array with the site
```

```
        return np_daylight_map
# ↑ done, returns a numpy array, in the form of site grid.



def room_placement_order(graph, rooms):
        # Order of Placement
        lst_sum_weight = []
        lst_degree = []
        for i in range(len(rooms)):
                lst_sum_weight.append(graph.nodes[i]['sum_weight'])
                lst_degree.append(graph.degree[i])

        lst = [lst_sum_weight, lst_degree, sizes, rooms]
        lst_rev = [[] for _ in range(len(lst[0]))]

        for i in range(len(lst)):
                for a in range(len(lst[i])):
                        lst_rev[a].append(lst[i][a])

        print(lst_rev)

        d = sorted(lst_rev, key=lambda x: (-x[0], x[1], -x[2]))
        # get the room placement order(in node index)
        order = []
        for i in range(len(d)):
                a = rooms.index(d[i][-1])
                order.append(a)

        return order


# ↑ done, returns a list of node in the order of placement


def add_room_dynamic(space, room_dimension, position):
        '''
        Add a dynamic room to space
        :param space:
        :param room:
        :param room_dimension: Dimension: (width, length); or size: integer.
        :param position: coordinate: (x, y)
        :return: body of the room
```

```
    '''
    inf = pm.inf

    def crack(integer):
        start = int(np.sqrt(integer))
        factor = integer / start
        while not is_integer(factor):
            start += 1
            factor = integer / start
        return int(factor), start

    def is_integer(number):
        if int(number) == number:
            return True
        else:
            return False

    if type(room_dimension) == 'tuple':
        room_width = room_dimension[0] * 10
        room_length = room_dimension[1] * 10

    else:
        ret = crack(room_dimension)    # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10

    mass = (room_width * room_length) * 0.01
    poly_shape = pm.Poly.create_box(None, size=(room_width, room_length))
    poly_body = pm.Body(mass, inf)
    poly_shape.body = poly_body
    poly_body.position = Vec2d(position)
    poly_shape.friction = 0.3
    poly_shape.collision_type = 9
    return poly_body, poly_shape


# ↑ done, add a dynamic room to space


def add_spring(space, a1, a2, weight, length):
    index = 50    # the index to scale the stiffness of the spring according to the weight
    spring = pm.DampedSpring(a1, a2, (0, 0), (0, 0), length, weight * index, 0.3)
    space.add(spring)
```

```
def room_placement(screen, space, graph, room_placement_order):
    rooms_shape_list = []
    spring_list = []
    for i in room_placement_order:    # order =    [4, 1, 0, 2, 3]
        s = G.nodes[i]['sizes']
        g = room_placement_order.index(i)
        lst_len = len(room_placement_order)
        exec(f'a{i}, b{i} = add_room_dynamic(space, s, (800+100*(-1)**g, 960 - g*(960/lst_len)))')
        exec(f'space.add(a{i},b{i})')
        exec(f'rooms_shape_list.append(b{i})')
        exec(f'l{i}= (s*100)**0.05')

        f = [n for n in graph.neighbors(i)]    # getting neighbors of node i

        for i_f in range(len(f)):
            for k in room_placement_order[0:g]:
                if k == f[i_f]:
                    weight = G[i][f[i_f]]['weight']    # getting weight of edge connected to node i
                    # if room is inside space:
                    h = f[i_f]
                    sh = G.nodes[h]['sizes']
                    exec(f'l{h} = (sh*100)**0.05')
                    exec(f'length = (l{i} + l{h})')
                    exec(f'add_spring(space, a{h}, a{i}, weight, length)')
                    exec(f'spring_list.append((a{h}, a{i}))')
                else:
                    pass
    return rooms_shape_list, spring_list


def draw_convex_hull(screen, rooms_shape_list):
    v_list = []
    total_size = 0

    for room in rooms_shape_list:
        i = rooms_shape_list.index(room)
        o = order[i]
        s = G.nodes[o]['sizes']
        total_size = total_size + s

        def crack(integer):
            start = int(np.sqrt(integer))
            factor = integer / start
```

```
            while not is_integer(factor):
                start += 1
                factor = integer / start
            return int(factor), start

        def is_integer(number):
            if int(number) == number:
                return True
            else:
                return False

        ret = crack(s)    # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10

        v = room.body.position
        v1x = int(v.x - room_width / 2)
        v1y = int(v.y - room_length / 2)
        v2x = int(v.x + room_width / 2)
        v2y = int(v.y - room_length / 2)
        v3x = int(v.x + room_width / 2)
        v3y = int(v.y + room_length / 2)
        v4x = int(v.x - room_width / 2)
        v4y = int(v.y + room_length / 2)
        v1 = [v1x, v1y]
        v2 = [v2x, v2y]
        v3 = [v3x, v3y]
        v4 = [v4x, v4y]
        v_list.append(v1)
        v_list.append(v2)
        v_list.append(v3)
        v_list.append(v4)

v_array = np.array(v_list)
hull = ConvexHull(v_array)

for simplex in hull.simplices:
        a = v_array[simplex[0]]
        b = v_array[simplex[1]]
        plt.plot(v_array[simplex, 0], v_array[simplex, 1], 'k-')

        pygame.draw.line(screen, (0, 255, 0), b, a, 2)

hull_area = hull.volume
```

```
    hull_to_room_ratio = hull_area / (total_size * 100)
    return hull_to_room_ratio


def potential_path_length(circulations, rooms_shape_list):
    door_list = []

    for room in rooms_shape_list:
        i = rooms_shape_list.index(room)
        o = order[i]
        d_k = G.nodes[o]['door_direction']
        s = G.nodes[o]['sizes']

        def crack(integer):
            start = int(np.sqrt(integer))
            factor = integer / start
            while not is_integer(factor):
                start += 1
                factor = integer / start
            return int(factor), start

        def is_integer(number):
            if int(number) == number:
                return True
            else:
                return False

        ret = crack(s)    # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10
        v = room.body.position
        v_n_x = int(v.x)
        v_n_y = int(v.y - room_length / 2)
        v_s_x = int(v.x)
        v_s_y = int(v.y + room_length / 2)
        v_w_x = int(v.x - room_width / 2)
        v_w_y = int(v.y)
        v_e_x = int(v.x + room_width / 2)
        v_e_y = int(v.y)
        v_n = [v_n_x, v_n_y]
        v_s = [v_s_x, v_s_y]
        v_w = [v_w_x, v_w_y]
        v_e = [v_e_x, v_e_y]
```

```
        if d_k == 'N':
            pos = v_n

        elif d_k == 'S':
            pos = v_s

        elif d_k == 'W':
            pos = v_w

        elif d_k == 'E':
            pos = v_e

        else:
            pos = v_n

        door_list.append(pos)

length_text = []

for i in range(len(circulations)):
    length = 0

    for k in range(len(circulations[i])):
        room = rooms.index(circulations[i][k])
        o = order.index(room)
        door_pos = door_list[o]

        if k == len(circulations[i])-1:
            pass
        else:
            room_b = rooms.index(circulations[i][k+1])
            o_b = order.index(room_b)
            door_pos_b = door_list[o_b]
            length =length + (abs(abs(door_pos[0]) - abs(door_pos_b[0])) + abs(
                abs(door_pos[1]) - abs(door_pos_b[1]))) /10


    length_round = round(length, 1)



    text = f"Circulation{i+1}: {length_round}"
```

```
            length_text.append(text)



    return length_text


def draw_room(screen, rooms_shape_list, spring_list):
    # draw pygame rectangles
    for room in rooms_shape_list:
        i = rooms_shape_list.index(room)
        o = order[i]
        s = G.nodes[o]['sizes']

        def crack(integer):
            start = int(np.sqrt(integer))
            factor = integer / start
            while not is_integer(factor):
                start += 1
                factor = integer / start
            return int(factor), start

        def is_integer(number):
            if int(number) == number:
                return True
            else:
                return False

        ret = crack(s)    # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10

        v = room.body.position
        vx = int(v.x - room_width / 2)
        vy = int(v.y - room_length / 2)
        pygame.draw.rect(screen, THECOLORS["red"], (vx, vy, room_width, room_length), 4)

        for joint in spring_list:
            for i in range(len(spring_list)):
                ax = int(joint[0].position.x)
                ay = int(joint[0].position.y)
                bx = int(joint[1].position.x)
                by = int(joint[1].position.y)
                pygame.draw.line(screen, (0, 0, 255), (ax, ay), (bx, by), 3)
```

```
            font = pygame.font.Font(None, 16)
            room_info = [f"{G.nodes[o]['room']}",
                         f"DDH:{G.nodes[o]['lights']}"]
            y = 5
            for line in room_info:
                text = font.render(line, 1, THECOLORS["black"])
                screen.blit(text, (vx + 2,vy + y))
                y += 15




# ↑ done, draw all rooms and connections with pygame


def main():
    # initialize the environment
    pm.pygame_util.positive_y_is_up = False

    pygame.init()
    screen = pygame.display.set_mode((1600, 960))
    clock = pygame.time.Clock()

    space = pm.Space()
    space.gravity = (0.0, 0.0)
    # draw_options = pm.pygame_util.DrawOptions(screen)

    fps = 60

    static = [
        pm.Segment(space.static_body, (0, 0), (0, 960), 5),
        pm.Segment(space.static_body, (0, 960), (1600, 960), 5),
        pm.Segment(space.static_body, (1600, 960), (1600, 0), 5),
        pm.Segment(space.static_body, (0, 0), (1600, 0), 5),
    ]

    for s in static:
        s.collision_type = 0
    space.add(static)

    mouse_joint = None
    mouse_body = pm.Body(body_type=pm.Body.KINEMATIC)
```

```
    rooms_shape_list, spring_list = room_placement(screen, space, G, order)

    while True:
        pause = False
        for event in pygame.event.get():
            if event.type == KEYUP:
                if event.key == K_p:
                    pause = True
            if event.type == QUIT:
                exit()
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                exit()
            elif event.type == MOUSEBUTTONDOWN:
                if mouse_joint != None:
                    space.remove(mouse_joint)
                    mouse_joint = None

                p = Vec2d(event.pos)
                hit = space.point_query_nearest(p, 5, pm.ShapeFilter())
                if hit != None and hit.shape.body.body_type == pm.Body.DYNAMIC:
                    shape = hit.shape
                    # Use the closest point on the surface if the click is outside
                    # of the shape.
                    if hit.distance > 0:
                        nearest = hit.point
                    else:
                        nearest = p
                    mouse_joint = pm.PivotJoint(mouse_body, shape.body,
                                                (0, 0), shape.body.world_to_local(nearest))
                    mouse_joint.max_force = 5000000
                    mouse_joint.error_bias = (1 - 0.15) ** 60
                    space.add(mouse_joint)

            elif event.type == MOUSEBUTTONUP:
                if mouse_joint != None:
                    space.remove(mouse_joint)
                    mouse_joint = None

        while pause == True:
            for event in pygame.event.get():
                if event.type == KEYUP:
                    if event.key == K_p:
                        pause = False
                if event.type == QUIT:
```

```
                    exit()
                elif event.type == KEYDOWN and event.key == K_ESCAPE:
                    exit()
                elif event.type == KEYDOWN and event.key == K_g:   # 判断按下 g

                    phase_index = 0
                    rooms_shape_list_pos_out = []
                    body_type_list_out = []
                    room_info_lst = []
                    order_generate = []

                    for shape in rooms_shape_list:
                        i = rooms_shape_list.index(shape)
                        room = order[i]
                        v = shape.body.position
                        vx = int(v.x)
                        vy = int(v.y)
                        body_type = shape.body.body_type
                        shape_pos = (vx, vy)

                        room_info = (room, shape_pos, body_type, i, vy)
                        room_info_lst.append(room_info)

                    order_info_lst = sorted(room_info_lst, key=lambda x: (-x[4], x[3]))

                    for i in range(len(order_info_lst)):
                        room_node = order_info_lst[i][0]
                        shape_pos_order = order_info_lst[i][1]
                        body_type_order = order_info_lst[i][2]
                        order_generate.append(room_node)
                        rooms_shape_list_pos_out.append(shape_pos_order)
                        body_type_list_out.append(body_type_order)

                    f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/4_5_info_exchange.txt',
mode='w',
                             encoding='utf-8')
                    f.write(f'{phase_index}')
                    f.write('\n')
                    f.write(f'{rooms_shape_list_pos_out}')
                    f.write('\n')
                    f.write(f'{body_type_list_out}')
                    f.write('\n')
                    f.write(f'{order_generate}')
```

```
                        f.write('\n')
                        f.write(f'{rooms}')
                        f.write('\n')
                        f.write(f'{sizes}')
                        f.write('\n')
                        f.write(f'{lights}')
                        f.write('\n')
                        f.write(f'{door_direction}')
                        f.write('\n')
                        f.write(f'{circulations}')
                        f.write('\n')
                        f.write(f'{cir_Weight}')
                        print(order_generate)

            screen.fill(pygame.color.THECOLORS["white"])

            draw_room(screen, rooms_shape_list, spring_list)
            hull_to_room_ratio = draw_convex_hull(screen, rooms_shape_list)
            ratio = round(hull_to_room_ratio, 3)
            len_text = potential_path_length(circulations, rooms_shape_list)
            draw_helptext(screen, ratio, len_text)

            mouse_pos = pygame.mouse.get_pos()

            mouse_body.position = mouse_pos

            space.step(1. / fps)

            # space.debug_draw(draw_options)
            pygame.display.flip()

            clock.tick(fps)
            pygame.display.set_caption("fps: " + str(clock.get_fps()))

### Read the initial program file
f = open('D:\Delft
Courses\Graduation\PythonProject\daylighthour_layout_pathfinding\\01_Initial_program.txt',
mode='r', encoding='utf-8')
infolist = []
for line in f:
    infolist.append(line.strip())

rooms = eval(infolist[0])
sizes = eval(infolist[1])
```

```
lights = eval(infolist[2])
door_direction = eval(infolist[3])
circulations = eval(infolist[4])
cir_Weight = eval(infolist[5])

#### PoR for testing code start
# rooms = ["RoomA", "RoomB", "RoomC", "RoomD", "RoomE"]
# sizes = [100, 50, 30, 40, 80]
# lights = [8, 7, 6, 5, 4]
# door_direction = ["N", "none", "none", "none", "W"]
# circulations = [["RoomA", "RoomB", "RoomE", "RoomA"],
#                 ["RoomB", "RoomD", "RoomC", "RoomE", "RoomB"],
#                 ["RoomA", "RoomB", "RoomC", "RoomE", "RoomA"]]
# cir_Weight = [3, 1, 2]
#### PoR for testing code end

G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)

order = room_placement_order(G, rooms)

global order_list_out
global rooms_shape_list_out
global spring_list_out

# print("order = ", order)

draw_graph(G)

if __name__ == '__main__':
    sys.exit(main())
```

Appendix D: Python code of module 05_daylight_hour_optimization

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull, convex_hull_plot_2d

import sys

import inspect
import math

import pygame
from pygame.color import *
from pygame.locals import *

import pymunk
import pymunk as pm
from pymunk.vec2d import Vec2d
import pymunk.pygame_util
import pymunk.autogeometry

def crack(integer):
    start = int(np.sqrt(integer))
    factor = integer / start
    while not is_integer(factor):
        start += 1
        factor = integer / start
    return int(factor), start


def is_integer(number):
    if int(number) == number:
        return True
    else:
        return False


def draw_helptext(screen, hull_to_room_ratio, len_text):
    font = pygame.font.Font(None, 20)
    text = ["Daylight_hour_optimization_module",
```

```
                "Help:",
                "LMB(hold): Drag shapes",
                "P: Pause",
                "During Pause: C: connect spring, then unpause to see reaction",
                "During Pause: G: generate iteration file",
                "Esc / Q: Quit",
                "After final iteration, press G and quit module to proceed"
                "",
                "Indicators:",
                "DDH: Desired daylight hour"
                "STR: Satisfied unit to total unit ratio"
                f"Convex Hull to Room Ratio (The more closer to 1, the more compact the layout is):
{hull_to_room_ratio}",
                f"Potential length of: {len_text}"
                ]
        y = 5
        for line in text:
            text = font.render(line, 1, THECOLORS["black"])
            screen.blit(text, (5, y))
            y += 15


def draw_graph(graph):
        # draw graph with weight
        pos = nx.spring_layout(graph)
        nx.draw(graph, pos)
        node_labels = nx.get_node_attributes(graph, 'room')
        nx.draw_networkx_labels(graph, pos, labels=node_labels)
        labels = nx.get_edge_attributes(graph, 'weight')
        nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels)
        plt.show()
        pass
# ↑ done, draws the graph


def por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight):
        i_sizes = 0
        i_lights = 1
        i_weight_sum = 2
        i_edge_sum = 3
        i_door_direction = 4

        ReL_Array = np.zeros([len(rooms), len(rooms)])
        Attri_Array = np.zeros([len(rooms), i_door_direction + 1])
```

```
    # creating ReL_Array
    for i in range(len(rooms)):
        for a in range(len(circulations)):
            for b in range(len(circulations[a]) - 1):
                if circulations[a][b] == rooms[i]:
                    ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]

    # Flip the array left and right then rotate the array by 90 degrees
    ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
    # print("ReL_Array = ", ReL_Array)

    # Creating attribute array
    for i in range(len(rooms)):
        Attri_Array[i, i_sizes] = sizes[i]
        Attri_Array[i, i_lights] = lights[i]
    # print("Attri_Array = ", Attri_Array)

    # get Sum of edge weights on one node
    sum_weight = []
    for i in range(len(rooms)):
        sum_weight.append(sum(ReL_Array[i]))

    # Creating graph in networkx from ReL_Array
    G = nx.from_numpy_array(ReL_Array)
    for i in range(len(rooms)):
        G.nodes[i]['room'] = rooms[i]
        G.nodes[i]['sizes'] = sizes[i]
        G.nodes[i]['lights'] = lights[i]
        G.nodes[i]['sum_weight'] = sum_weight[i]
        G.nodes[i]['door_direction'] = door_direction[i]

    print("G.edges = ", G.edges(data=True))    # print edges
    print("G.nodes = ", G.nodes(data=True))    # print all nodes with attribute dictionary
    print("G.degree() = ", G.degree())
    print("sum_weight = ", sum_weight)
    return G
# ↑ done, returns a graph contains all information, weights on edges, attributes on nodes


def site_daylight_map(daylighthours, map_height):
    '''
    site_daylight_map = site_daylight_map(daylighthours, map_height, map_width)
    site_daylight_map = site_daylight_map.tolist()
```

```
    These two lines should be used at the end of the code in GH_CPython components,
    to pass arrays among GH_CPython components within grasshopper in the form of list.
    :param daylighthours: "sunlightHoursResult" list received from Ladybug Tools
"sunlightHoursAnalysis" component,
    :param map_height:    Number of grid on y axis
    :return: A numpy array of "sunlightHoursResult" that corresponds to the site
    '''
    daylight_map = [daylighthours[i:i + map_height] for i in range(0, len(daylighthours), map_height)]
    np_daylight_map = np.array(daylight_map)
    np_daylight_map = np.rot90(np_daylight_map)   # match the numpy array with the site
    return np_daylight_map
# ↑ done, returns a numpy array, in the form of site grid.


def from_list_to_array(list, array_width):
    '''
    Arrays are passed in the form of one dimensional list, each time receiving the one dimensional
list,
    it needs to be transformed back to two dimensional array.
    :param list: Array received
    :param array_width: Original width of numpy array
    :return: The original numpy array
    '''
    array = [list[i:i + array_width] for i in range(0, len(list), array_width)]
    array = np.array(array)
    return array
# ↑ done, used for passing arrays among GH_CPython components within grasshopper.


def room_placement_order(graph, rooms):
    # Order of Placement
    lst_sum_weight = []
    lst_degree = []
    lst_size = []
    for i in range(len(rooms)):
        lst_sum_weight.append(graph.nodes[i]['sum_weight'])
        lst_degree.append(graph.degree[i])

    lst = [lst_sum_weight, lst_degree, sizes, rooms]

    lst_rev = [[] for _ in range(len(lst[0]))]
    # print('lst_rev = ', lst_rev)
    for i in range(len(lst)):
```

```
            for a in range(len(lst[i])):
                lst_rev[a].append(lst[i][a])

    d = sorted(lst_rev, key=lambda x: (-x[0], x[1], -x[2]))
    # print("d = ", d)

    # get the room placement order(in node index)
    order = []
    for i in range(len(d)):
        a = rooms.index(d[i][-1])
        order.append(a)

    return order
# ↑ done, returns a list of node in the order of placement


def add_room_static(room_dimension, position):
    '''
    Add a static room to space
    :param room:
    :param room_dimension: Dimension: (width, length); or size: integer.
    :param position: coordinate: (x, y)
    :return: body of the room
    '''
    inf = pm.inf

    if type(room_dimension) == 'tuple':
        room_width = room_dimension[0]*10
        room_length = room_dimension[1]*10

    else:
        ret = crack(room_dimension)    # crack room size into two closest factors
        room_width = ret[0]*10
        room_length = ret[1]*10


    poly_body = pm.Body(body_type=pm.Body.STATIC)
    poly_shape = pm.Poly.create_box(poly_body, size=(room_width, room_length))
    poly_shape.color = (255, 0, 0, 255)    # setting color of the shape
    poly_body.position = Vec2d(position)
    poly_shape.friction = 0.7
    poly_shape.collision_type = 2
    return poly_body, poly_shape
# ↑ done, add a static room to space.
```

```
def add_room_dynamic(room_dimension, position):
    '''
    Add a dynamic room to space
    :param space:
    :param room:
    :param room_dimension: Dimension: (width, length); or size: integer.
    :param position: coordinate: (x, y)
    :return: body of the room
    '''
    inf = pm.inf

    if type(room_dimension) == 'tuple':
        room_width = room_dimension[0]*10
        room_length = room_dimension[1]*10

    else:
        ret = crack(room_dimension)    # crack room size into two closest factors
        room_width = ret[0]*10
        room_length = ret[1]*10

    mass = (room_width * room_length) * 0.01
    poly_shape = pm.Poly.create_box(None, size=(room_width, room_length))
    poly_body = pm.Body(mass, inf)
    poly_shape.body = poly_body
    poly_shape.color = (0, 0, 255, 255)    # setting color of the shape
    poly_body.position = Vec2d(position)
    poly_shape.friction = 0.3
    poly_shape.collision_type = 9
    return poly_body, poly_shape
# ↑ done, add a dynamic room to space


def add_spring(space, a1, a2, weight,length):
    index = 50        # the index to scale the stiffness of the spring according to the weight
    spring = pm.DampedSpring(a1, a2, (0, 0), (0, 0), length, weight * index, 0.3)
    space.add(spring)
# ↑ done, add spring


def draw_map(space, site_daylight_map, map_width, map_height):
    def draw_pixel(space, position, lighthour):
        poly_body = pm.Body(body_type=pm.Body.STATIC)
```

```
            poly_shape = pm.Poly.create_box(poly_body, size=(10, 10))
            poly_body.position = Vec2d(position)
            poly_shape.friction = 0.7
            if lighthour < 1:
                poly_shape.color = (0, 0, 255, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000000001,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000100000000)
            elif lighthour < 2:
                poly_shape.color = (67, 0, 189, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000000010,
mask=pm.ShapeFilter.ALL_MASKS^0b00000001100000000)
            elif lighthour < 3:
                poly_shape.color = (134, 0, 122, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000000100,
mask=pm.ShapeFilter.ALL_MASKS^0b00000011100000000)
            elif lighthour < 4:
                poly_shape.color = (201, 0, 55, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000001000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000111100000000)
            elif lighthour < 5:
                poly_shape.color = (255, 12, 0, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000010000,
mask=pm.ShapeFilter.ALL_MASKS^0b00001111100000000)
            elif lighthour < 6:
                poly_shape.color = (255, 79, 0, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000000100000,
mask=pm.ShapeFilter.ALL_MASKS^0b00011111100000000)
            elif lighthour < 7:
                poly_shape.color = (255, 146, 0, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000001000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00111111100000000)
            elif lighthour < 8:
                poly_shape.color = (255, 213, 0, 255)
                poly_shape.filter = pm.ShapeFilter(categories=0b00000000010000000,
mask=pm.ShapeFilter.ALL_MASKS^0b01111111100000000)
            space.add(poly_body, poly_shape)
            return poly_body

        pixel_list = []

        for i in range(map_height - 1):
            for j in range(map_width - 1):
                if site_daylight_map[i][j] < 8:
                    draw_pixel(space, (j * 10, i * 10), site_daylight_map[i][j])
```

```
                    a = (j*10, i*10, site_daylight_map[i][j])
                    pixel_list.append(a)

    return pixel_list
# ↑ done, draw map into the space


def draw_map_pg(screen,pixel_list):
    for pixel in pixel_list:
        vx = int(pixel[0] - 5)
        vy = int(pixel[1] - 5)
        lighthour = int(pixel[2])
        if lighthour < 1:    # setting color of the shape
            pygame.draw.rect(screen, (0, 0, 255), (vx, vy, 10, 10))
        elif lighthour < 2:
            pygame.draw.rect(screen, (67, 0, 189), (vx, vy, 10, 10))
        elif lighthour < 3:
            pygame.draw.rect(screen, (134, 0, 122), (vx, vy, 10, 10))
        elif lighthour < 4:
            pygame.draw.rect(screen, (201, 0, 55), (vx, vy, 10, 10))
        elif lighthour < 5:
            pygame.draw.rect(screen, (255, 12, 0), (vx, vy, 10, 10))
        elif lighthour < 6:
            pygame.draw.rect(screen, (255, 79, 0), (vx, vy, 10, 10))
        elif lighthour < 7:
            pygame.draw.rect(screen, (255, 146, 0), (vx, vy, 10, 10))
        elif lighthour < 8:
            pygame.draw.rect(screen, (255, 213, 0), (vx, vy, 10, 10))
# # ↑ done, draw map into the space


def room_placement_dlh(screen, space, graph, phase_index, rooms_shape_list_pos, body_type_list,
room_placement_order):
    # phase_index = infolist[0] + 1     # 0+1 = 1
    # rooms_shape_list_pos = infolist[1]    # [(337, 367), (395, 457), (437, 357), (356, 302), (345, 447)]
    # body_type_list = infolist[2]       #      [0, 0, 0, 0, 0]
    rooms_shape_list = []
    rooms_body_list = []
    connection_list = []
    # body_type_list = [2,0,2,0,2]
    for i in room_placement_order:    # order =    [4, 1, 0, 2, 3]
        g = room_placement_order.index(i)
        pos_x = int(rooms_shape_list_pos[g][0])
        pos_y = int(rooms_shape_list_pos[g][1])
```

```
            # pos = (pos_x*1.2, pos_y*1.2)
            s = graph.nodes[i]['sizes']
            name = graph.nodes[i]['room']
            # if body_type_list[g] == 2:
            if g < phase_index-1:
                pos = (pos_x, pos_y)
                exec(f'a{i}, b{i} = add_room_static(s, pos)')
                exec(f'space.add(a{i},b{i})')
                exec(f'rooms_shape_list.append(b{i})')
                exec(f'rooms_body_list.append(a{i})')
            # elif body_type_list[g] == 0:
            else:
                pos = (pos_x, pos_y)
                exec(f'a{i}, b{i} = add_room_dynamic(s, pos)')
                exec(f'space.add(a{i},b{i})')
                exec(f'rooms_shape_list.append(b{i})')
                exec(f'rooms_body_list.append(a{i})')

            exec(f'l{i}= (s*100)**0.05')
            light = graph.nodes[i]['lights']
            if light == 0:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00000000100000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011111111)')
            elif light == 1:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00000001000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011111110)')
            elif light == 2:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00000010000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011111100)')
            elif light == 3:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00000100000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011111000)')
            elif light == 4:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00001000000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011110000)')
            elif light == 5:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00010000000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011100000)')
```

```
            elif light == 6:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b00100000000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000011000000)')
            elif light == 7:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b01000000000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000010000000)')
            elif light == 8:
                exec(
                    f'b{i}.filter = pm.ShapeFilter(categories=0b10000000000000000,
mask=pm.ShapeFilter.ALL_MASKS^0b00000000000000000 )')

            f = [n for n in graph.neighbors(i)]    # getting neighbors of node i

            for i_f in range(len(f)):
                for k in room_placement_order[0:g]:
                    if k == f[i_f]:
                        # if room is inside space:
                        h = f[i_f]
                        exec(f'connection_list.append((a{h}, a{i}))')
                    else:
                        pass
        return rooms_shape_list, rooms_body_list, connection_list


def connect_rooms(space, graph, rooms_body_list, room_placement_order, phase_index):
        spring_list = []
        for body in rooms_body_list:
            g = rooms_body_list.index(body)
            i = room_placement_order[g]
            s = graph.nodes[i]['sizes']
            exec(f'l{g}= (s*100)**0.05')
            f = [n for n in graph.neighbors(i)]    # getting neighbors of node i
            for i_f in range(len(f)):
                for k in room_placement_order[0:g]:
                    l = room_placement_order.index(k)
                    if k == f[i_f] and (g > phase_index-1 or l > phase_index-1):

                        weight = graph[i][f[i_f]]['weight']    # getting weight of edge connected to node
i
                        # if room is inside space:
                        sh = graph.nodes[l]['sizes']
                        exec(f'l{l} = (sh*100)**0.05')
```

```
                        exec(f'length = (l{g} + l{l})')
                        exec(f'add_spring(space, rooms_body_list[{l}], rooms_body_list[{g}], weight,
length)')
                        exec(f'spring_list.append((rooms_body_list[{l}], rooms_body_list[{g}]))')
                else:
                    pass
    return spring_list


def room_daylighthour(room_pos, size, sdmap, light):
    room_pos_x = room_pos[0]
    room_pos_y = room_pos[1]

    if type(size) == 'tuple':
        room_width = size[0] * 10
        room_length = size[1] * 10

    else:
        ret = crack(size)
        room_width = ret[0] * 10
        room_length = ret[1] * 10

    room_corner_a = (room_pos_x - room_width / 2, room_pos_y - room_length / 2)
    room_corner_b = (room_pos_x - room_width / 2, room_pos_y + room_length / 2)
    room_corner_c = (room_pos_x + room_width / 2, room_pos_y + room_length / 2)
    room_corner_d = (room_pos_x + room_width / 2, room_pos_y - room_length / 2)
    a_i, a_j = get_map_pos(room_corner_a, 10)
    b_i, b_j = get_map_pos(room_corner_b, 10)
    c_i, c_j = get_map_pos(room_corner_c, 10)
    d_i, d_j = get_map_pos(room_corner_d, 10)

    i_list = [a_i]
    j_list = [a_j]

    total_unit = 0
    satisfied_unit = 0

    for i in range(int(b_i - a_i)):
        i_list.append(a_i + i)

    for j in range(int(d_j - a_j)):
        j_list.append(a_j + j)

    for i in i_list:
```

```
            for j in j_list:
                total_unit += 1
                a = sdmap[int(i)][int(j)]
                if a >= light:
                    satisfied_unit += 1
                else:
                    pass

    satisfied_to_total_ratio = round(satisfied_unit / total_unit, 2)
    return satisfied_to_total_ratio


def draw_room(screen, graph, rooms_shape_list, spring_list, daylightmap):
    # draw pygame rectangles
    for room in rooms_shape_list:
        i = rooms_shape_list.index(room)
        o = order[i]
        s = graph.nodes[o]['sizes']
        light = graph.nodes[o]['lights']

        ret = crack(s)    # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10


        v = room.body.position
        # print(f'{name} pos = {v}')
        vx = int(v.x - room_width / 2)
        vy = int(v.y - room_length / 2)
        room_pos = (v.x, v.y)
        pygame.draw.rect(screen, THECOLORS["red"], (vx, vy, room_width, room_length), 2)

        satisfied_to_total_ratio = room_daylighthour(room_pos, s, daylightmap, light)

        for joint in spring_list:
            for i in range(len(spring_list)):
                ax = int(joint[0].position.x)
                ay = int(joint[0].position.y)
                bx = int(joint[1].position.x)
                by = int(joint[1].position.y)
                pygame.draw.line(screen, (0, 0, 255), (ax, ay), (bx, by), 3)

        font = pygame.font.Font(None, 16)
```

```
            room_info = [f"{G.nodes[o]['room']}",
                         f"DDH:{G.nodes[o]['lights']}",
                         f"STR:{satisfied_to_total_ratio}"]
            y = 5
            for line in room_info:
                text = font.render(line, 1, THECOLORS["black"])
                screen.blit(text, (vx + 2, vy + y))
                y += 15



# ↑ done, draw all rooms and connections with pygame


def draw_convex_hull(screen, rooms_shape_list):
    v_list = []
    total_size = 0

    for room in rooms_shape_list:
        i = rooms_shape_list.index(room)
        o = order[i]
        s = G.nodes[o]['sizes']
        total_size = total_size + s

        ret = crack(s)   # crack room size into two closest factors
        room_width = ret[0] * 10
        room_length = ret[1] * 10

        v = room.body.position
        v1x = int(v.x - room_width / 2)
        v1y = int(v.y - room_length / 2)
        v2x = int(v.x + room_width / 2)
        v2y = int(v.y - room_length / 2)
        v3x = int(v.x + room_width / 2)
        v3y = int(v.y + room_length / 2)
        v4x = int(v.x - room_width / 2)
        v4y = int(v.y + room_length / 2)
        v1 = [v1x, v1y]
        v2 = [v2x, v2y]
        v3 = [v3x, v3y]
        v4 = [v4x, v4y]
        v_list.append(v1)
        v_list.append(v2)
        v_list.append(v3)
```

```
            v_list.append(v4)

        v_array = np.array(v_list)
        hull = ConvexHull(v_array)

        for simplex in hull.simplices:
            a = v_array[simplex[0]]
            b = v_array[simplex[1]]
            pygame.draw.line(screen, (0, 255, 0), b, a, 2)

        hull_area = hull.volume
        hull_to_room_ratio = hull_area / (total_size * 100)
        return hull_to_room_ratio


def potential_path_length(circulations, rooms_shape_list):
        door_list = []

        for room in rooms_shape_list:
            i = rooms_shape_list.index(room)
            o = order[i]
            d_k = G.nodes[o]['door_direction']
            s = G.nodes[o]['sizes']

            ret = crack(s)    # crack room size into two closest factors
            room_width = ret[0] * 10
            room_length = ret[1] * 10
            v = room.body.position
            v_n_x = int(v.x)
            v_n_y = int(v.y - room_length / 2)
            v_s_x = int(v.x)
            v_s_y = int(v.y + room_length / 2)
            v_w_x = int(v.x - room_width / 2)
            v_w_y = int(v.y)
            v_e_x = int(v.x + room_width / 2)
            v_e_y = int(v.y)
            v_n = [v_n_x, v_n_y]
            v_s = [v_s_x, v_s_y]
            v_w = [v_w_x, v_w_y]
            v_e = [v_e_x, v_e_y]

            if d_k == 'N':
                pos = v_n
```

```
        elif d_k == 'S':
            pos = v_s

        elif d_k == 'W':
            pos = v_w

        elif d_k == 'E':
            pos = v_e

        else:
            pos = v_n

        door_list.append(pos)

    length_text = []

    for i in range(len(circulations)):
        length = 0

        for k in range(len(circulations[i])):
            room = rooms.index(circulations[i][k])
            o = order.index(room)
            door_pos = door_list[o]

            if k == len(circulations[i])-1:
                pass
            else:
                room_b = rooms.index(circulations[i][k+1])
                o_b = order.index(room_b)
                door_pos_b = door_list[o_b]
                length =length + (abs(abs(door_pos[0]) - abs(door_pos_b[0])) + abs(
                    abs(door_pos[1]) - abs(door_pos_b[1]))) /10


        length_round = round(length, 1)



        text = f"Circulation{i+1}: {length_round}"


        length_text.append(text)
```

```
    return length_text


def get_map_pos(pos, map_grid):
    x, y = pos
    i = y // map_grid
    j = x // map_grid
    return i, j









def main():
    # initialize the environment
    pm.pygame_util.positive_y_is_up = False

    pygame.init()
    screen = pygame.display.set_mode((1600, 960))
    clock = pygame.time.Clock()

    space = pm.Space()
    space.gravity = (0.0, 0.0)
    # draw_options = pm.pygame_util.DrawOptions(screen)

    fps = 144

    static = [
        pm.Segment(space.static_body, (0, 0), (0, 960), 5),
        pm.Segment(space.static_body, (0, 960), (1600, 960), 5),
        pm.Segment(space.static_body, (1600, 960), (1600, 0), 5),
        pm.Segment(space.static_body, (0, 0), (1600, 0), 5),
    ]

    for s in static:
        s.collision_type = 0
    space.add(static)

    mouse_joint = None
    mouse_body = pm.Body(body_type=pm.Body.KINEMATIC)
```

```
    # Draw daylight map in to space
    # Test code start##########Enable this to test in Pycharm################
    # f = open('daylighthours.txt', mode='r', encoding='utf-8')
    # daylighthours = []
    # for line in f:
    #     line = line.strip()
    #     daylighthours.append(int(line))
    #
    # map_height = 96
    # map_width = 160
    # Test code end############################################################

    sdmap = site_daylight_map(daylighthours, map_height)
    print(sdmap)
    pixel_list = draw_map(space, sdmap, map_width, map_height)


    # rooms_shape_list, spring_list = room_placement(screen, space, G, order)
    rooms_shape_list, rooms_body_list, connection_list= room_placement_dlh(screen, space, G,
phase_index, rooms_shape_list_pos, body_type_list, order)
    spring_list = []









    while True:
        pause = False
        for event in pygame.event.get():
            if event.type == KEYUP:
                if event.key == K_p:
                    pause = True
            if event.type == QUIT:
                exit()
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                exit()
            elif event.type == MOUSEBUTTONDOWN:
                if mouse_joint != None:
                    space.remove(mouse_joint)
                    mouse_joint = None
```

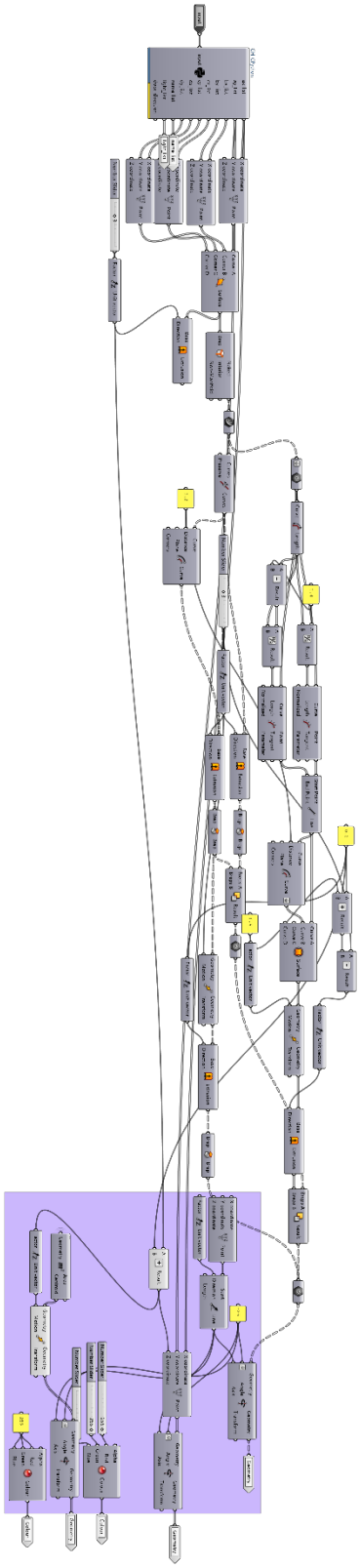
```
                p = Vec2d(event.pos)
                hit = space.point_query_nearest(p, 5, pm.ShapeFilter())
                if hit != None and hit.shape.body.body_type == pm.Body.DYNAMIC:
                    shape = hit.shape
                    # Use the closest point on the surface if the click is outside
                    # of the shape.
                    if hit.distance > 0:
                        nearest = hit.point
                    else:
                        nearest = p
                    mouse_joint = pm.PivotJoint(mouse_body, shape.body,
                                                (0, 0), shape.body.world_to_local(nearest))
                    mouse_joint.max_force = 50000000
                    mouse_joint.error_bias = (1 - 0.15) ** 60
                    space.add(mouse_joint)

            elif event.type == MOUSEBUTTONUP:
                if mouse_joint != None:
                    space.remove(mouse_joint)
                    mouse_joint = None

        while pause == True:
            for event in pygame.event.get():
                if event.type == KEYUP:
                    if event.key == K_p:
                        pause = False
                if event.type == QUIT:
                    exit()
                elif event.type == KEYDOWN and event.key == K_ESCAPE:
                    exit()
                elif event.type == KEYDOWN and event.key == K_c:
                    spring_list = connect_rooms(space, G, rooms_body_list, order,phase_index)


                elif event.type == KEYDOWN and event.key == K_g:   #  判断按下 g
                    rooms_shape_list_pos_out = []
                    body_type_list_out = []
                    room_info_lst = []
                    order_generate = []

                    for shape in rooms_shape_list:
                        i = rooms_shape_list.index(shape)
                        room = order[i]
                        v = shape.body.position
```

```
                    vx = int(v.x)
                    vy = int(v.y)
                    body_type = shape.body.body_type
                    shape_pos = (vx, vy)

                    room_info = (room, shape_pos, body_type, i, vy)
                    room_info_lst.append(room_info)

                order_info_lst = sorted(room_info_lst, key=lambda x: (-x[4], x[3]))

                for i in range(len(order_info_lst)):
                    room_node = order_info_lst[i][0]
                    shape_pos_order = order_info_lst[i][1]
                    body_type_order = order_info_lst[i][2]
                    order_generate.append(room_node)
                    rooms_shape_list_pos_out.append(shape_pos_order)
                    body_type_list_out.append(body_type_order)

                f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/4_5_info_exchange.txt',
mode='w',
                         encoding='utf-8')
                f.write(f'{phase_index}')
                f.write('\n')
                f.write(f'{rooms_shape_list_pos_out}')
                f.write('\n')
                f.write(f'{body_type_list_out}')
                f.write('\n')
                f.write(f'{order_generate}')
                f.write('\n')
                f.write(f'{rooms}')
                f.write('\n')
                f.write(f'{sizes}')
                f.write('\n')
                f.write(f'{lights}')
                f.write('\n')
                f.write(f'{door_direction}')
                f.write('\n')
                f.write(f'{circulations}')
                f.write('\n')
                f.write(f'{cir_Weight}')

        # for shape in rooms_shape_list:
        #     i = rooms_shape_list.index(shape)
```

```
            #         v = shape.body.position
            #         x = order[i]
            #         name = G.nodes[x]['room']
            #         print(f'{name} pos = ', v)

            screen.fill(pygame.color.THECOLORS["white"])

            draw_map_pg(screen,pixel_list)
            draw_room(screen, G, rooms_shape_list, connection_list, sdmap)
            hull_to_room_ratio = draw_convex_hull(screen, rooms_shape_list)
            ratio = round(hull_to_room_ratio, 3)
            len_text = potential_path_length(circulations, rooms_shape_list)
            draw_helptext(screen, ratio, len_text)

            mouse_pos = pygame.mouse.get_pos()

            mouse_body.position = mouse_pos

            space.step(1. / fps)

            # space.debug_draw(draw_options)
            pygame.display.flip()

            clock.tick(fps)
            pygame.display.set_caption("fps: " + str(clock.get_fps()))


f = open('D:\Delft
Courses\Graduation\PythonProject\daylighthour_layout_pathfinding\\4_5_info_exchange.txt',
mode='r', encoding='utf-8')
infolist = []
for line in f:
    infolist.append(line.strip())

phase_index = int(infolist[0]) + 1     # 0+1 = 1
rooms_shape_list_pos = eval(infolist[1])   # [(337, 367), (395, 457), (437, 357), (356, 302), (345, 447)]
body_type_list = eval(infolist[2])      #     [0, 0, 0, 0, 0]
order = eval(infolist[3])
rooms = eval(infolist[4])
sizes = eval(infolist[5])
lights = eval(infolist[6])
door_direction = eval(infolist[7])
circulations = eval(infolist[8])
cir_Weight = eval(infolist[9])
```

```
G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)

# print("order = ", order)

# draw_graph(G)

if __name__ == '__main__':
    sys.exit(main())
```

Appendix E: Grasshopper layout of module 06_generate_rhino_object



Appendix F: Python code of module 06_generate_rhino_object

```
import numpy as np
import networkx as nx

def por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight):
    i_sizes = 0
    i_lights = 1
    i_weight_sum = 2
    i_edge_sum = 3
    i_door_direction = 4

    ReL_Array = np.zeros([len(rooms), len(rooms)])
    Attri_Array = np.zeros([len(rooms), i_door_direction + 1])

    # creating ReL_Array
    for i in range(len(rooms)):
        for a in range(len(circulations)):
            for b in range(len(circulations[a]) - 1):
                if circulations[a][b] == rooms[i]:
                    ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]

    # Flip the array left and right then rotate the array by 90 degrees
    ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
    # print("ReL_Array = ", ReL_Array)

    # Creating attribute array
    for i in range(len(rooms)):
        Attri_Array[i, i_sizes] = sizes[i]
        Attri_Array[i, i_lights] = lights[i]
    # print("Attri_Array = ", Attri_Array)

    # get Sum of edge weights on one node
    sum_weight = []
    for i in range(len(rooms)):
        sum_weight.append(sum(ReL_Array[i]))

    # Creating graph in networkx from ReL_Array
    G = nx.from_numpy_array(ReL_Array)
    for i in range(len(rooms)):
        G.nodes[i]['room'] = rooms[i]
        G.nodes[i]['sizes'] = sizes[i]
        G.nodes[i]['lights'] = lights[i]
```

```
            G.nodes[i]['sum_weight'] = sum_weight[i]
            G.nodes[i]['door_direction'] = door_direction[i]

    print("G.edges = ", G.edges(data=True))    # print edges
    print("G.nodes = ", G.nodes(data=True))    # print all nodes with attribute dictionary
    print("G.degree() = ", G.degree())
    print("sum_weight = ", sum_weight)
    return G
# ↑ done, returns a graph contains all information, weights on edges, attributes on nodes

def crack(integer):
    start = int(np.sqrt(integer))
    factor = integer / start
    while not is_integer(factor):
        start += 1
        factor = integer / start
    return int(factor), start

def is_integer(number):
    if int(number) == number:
        return True
    else:
        return False




f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/4_5_info_exchange.txt', mode='r',
encoding='utf-8')
infolist = []
for line in f:
    infolist.append(line.strip())

phase_index = int(infolist[0]) + 1     # 0+1 = 1
rooms_shape_list_pos = eval(infolist[1])    # [(337, 367), (395, 457), (437, 357), (356, 302), (345, 447)]
body_type_list = eval(infolist[2])       #      [0, 0, 0, 0, 0]
order = eval(infolist[3])
rooms = eval(infolist[4])
sizes = eval(infolist[5])
lights = eval(infolist[6])
door_direction = eval(infolist[7])
```

```
circulations = eval(infolist[8])
cir_Weight = eval(infolist[9])

G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)



ax_list = []
ay_list = []
bx_list = []
by_list = []
cx_list = []
cy_list = []
dx_list = []
dy_list = []
name_list = []
light_list = []
door_direction_list = []

for node in order[0:phase_index-1]:
    i = order.index(node)
    name = G.nodes[node]['room']
    name_list.append(name)
    size = G.nodes[node]['sizes']
    light = G.nodes[node]['lights']
    light_list.append(light)
    if type(size) == 'tuple':
        room_width = size[0]
        room_length = size[1]

    else:
        ret = crack(size)   # crack room size into two closest factors
        room_width = ret[0]
        room_length = ret[1]
    pos_x = rooms_shape_list_pos[i][0]/10
    pos_y = (960-rooms_shape_list_pos[i][1])/ 10

    cornerA = (pos_x - room_width/2, pos_y - room_length/2, 0)
    cornerB = (pos_x - room_width / 2, pos_y + room_length / 2, 0)
    cornerC = (pos_x + room_width / 2, pos_y + room_length / 2, 0)
    cornerD = (pos_x + room_width / 2, pos_y - room_length / 2, 0)
    ax_list.append(cornerA[0])
    ay_list.append(cornerA[1])
    bx_list.append(cornerB[0])
    by_list.append(cornerB[1])
```

```
        cx_list.append(cornerC[0])
        cy_list.append(cornerC[1])
        dx_list.append(cornerD[0])
        dy_list.append(cornerD[1])
```

Appendix G: Python code of module 07_corridor_generation

```
"""
Acknowledgement to Youtube Channel Tech With Tim and its tutorial:
https://www.youtube.com/watch?v=JtiK0DOeI4A
"""

import pygame
import math
import numpy as np
import networkx as nx
from queue import PriorityQueue
import pymunk as pm
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import os

import sys

import inspect
import math

import pygame
from pygame.color import *
from pygame.locals import *

import pymunk
import pymunk as pm
from pymunk.vec2d import Vec2d
import pymunk.pygame_util
import pymunk.autogeometry

WIDTH = 1600
HEIGHT = 960
WIN = pygame.display.set_mode((WIDTH, HEIGHT))

RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 255, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
```

```
PURPLE = (128, 0, 128)
ORANGE = (255, 165, 0)
GREY = (128, 128, 128)
TURQUOISE = (64, 224, 208)


def draw_helptext(screen):
    font = pygame.font.Font(None, 20)
    text = ["LMB(hold): Drag shapes",
            "Initial_room_placement_module",
            "P: Pause",
            "During Pause: G: generate iteration file",
            "Esc / Q: Quit",
            "After pressing G, quit module to proceed to Daylight
Daylight_hour_optimization_module"
            ]
    y = 5
    for line in text:
        text = font.render(line, 1, THECOLORS["black"])
        screen.blit(text, (5, y))
        y += 15


class Spot:
    def __init__(self, row, col, width, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.width = width
        self.height = width
        self.total_rows = total_rows

    def get_pos(self):
        return self.row, self.col

    def is_closed(self):
        return self.color == RED

    def is_open(self):
        return self.color == GREEN

    def is_barrier(self):
```

```
        return self.color == BLACK

    def is_start(self):
        return self.color == ORANGE

    def is_end(self):
        return self.color == PURPLE

    def reset(self):
        self.color = WHITE

    def make_start(self):
        self.color = ORANGE

    def make_closed(self):
        self.color = RED

    def make_open(self):
        self.color = GREEN

    def make_barrier(self):
        self.color = BLACK

    def make_end(self):
        self.color = TURQUOISE

    def make_path(self):
        self.color = PURPLE

    def draw(self, win):
        pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.height))

    def update_neighbors(self, grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier():  # DOWN
            self.neighbors.append(grid[self.row + 1][self.col])

        if self.row > 0 and not grid[self.row - 1][self.col].is_barrier():  # UP
            self.neighbors.append(grid[self.row - 1][self.col])

        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier():  # RIGHT
            self.neighbors.append(grid[self.row][self.col + 1])

        if self.col > 0 and not grid[self.row][self.col - 1].is_barrier():  # LEFT
```

```
            self.neighbors.append(grid[self.row][self.col - 1])

    def __lt__(self, other):
        return False


def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)


def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()


def algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.make_end()
            return True
```

```
            for neighbor in current.neighbors:
                temp_g_score = g_score[current] + 1

                if temp_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = temp_g_score
                    f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                    if neighbor not in open_set_hash:
                        count += 1
                        open_set.put((f_score[neighbor], count, neighbor))
                        open_set_hash.add(neighbor)
                        neighbor.make_open()

            draw()

            if current != start:
                current.make_closed()

        return False


def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)
    return grid


def draw_grid(win, rows, width, height):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, height))


def draw(win, grid, rows, width, height):
    win.fill(WHITE)

    for row in grid:
```

```
            for spot in row:
                spot.draw(win)

    draw_grid(win, rows, width, height)
    pygame.display.update()


def get_barrier_pos(pos, rows, width):
    gap = width // rows
    y, x = pos

    row = y // gap
    col = x // gap

    return row, col


def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos

    row = y // gap
    col = x // gap

    return row, col


def por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight):
    i_sizes = 0
    i_lights = 1
    i_weight_sum = 2
    i_edge_sum = 3
    i_door_direction = 4

    ReL_Array = np.zeros([len(rooms), len(rooms)])
    Attri_Array = np.zeros([len(rooms), i_door_direction + 1])

    # creating ReL_Array
    for i in range(len(rooms)):
        for a in range(len(circulations)):
            for b in range(len(circulations[a]) - 1):
                if circulations[a][b] == rooms[i]:
                    ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]
```

```
    # Flip the array left and right then rotate the array by 90 degrees
    ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
    # print("ReL_Array = ", ReL_Array)

    # Creating attribute array
    for i in range(len(rooms)):
        Attri_Array[i, i_sizes] = sizes[i]
        Attri_Array[i, i_lights] = lights[i]
    # print("Attri_Array = ", Attri_Array)

    # get Sum of edge weights on one node
    sum_weight = []
    for i in range(len(rooms)):
        sum_weight.append(sum(ReL_Array[i]))

    # Creating graph in networkx from ReL_Array
    G = nx.from_numpy_array(ReL_Array)
    for i in range(len(rooms)):
        G.nodes[i]['room'] = rooms[i]
        G.nodes[i]['sizes'] = sizes[i]
        G.nodes[i]['lights'] = lights[i]
        G.nodes[i]['sum_weight'] = sum_weight[i]
        G.nodes[i]['door_direction'] = door_direction[i]

    print("G.edges = ", G.edges(data=True))    # print edges
    print("G.nodes = ", G.nodes(data=True))    # print all nodes with attribute dictionary
    print("G.degree() = ", G.degree())
    print("sum_weight = ", sum_weight)
    return G


def crack(integer):
    start = int(np.sqrt(integer))
    factor = integer / start
    while not is_integer(factor):
        start += 1
        factor = integer / start
    return int(factor), start


def is_integer(number):
    if int(number) == number:
        return True
    else:
```

```
            return False


def main(win, width, height):
    pygame.init()

    space = pm.Space()
    space.gravity = (0.0, 0.0)
    clock = pygame.time.Clock()
    fps = 120

    ROWS = 160
    grid = make_grid(ROWS, width)

    start = None
    end = None

    run = True
    f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/4_5_info_exchange.txt',
                mode='r', encoding='utf-8')
    infolist = []
    for line in f:
        infolist.append(line.strip())

    rooms_shape_list_pos = eval(infolist[1])
    order = eval(infolist[3])
    rooms = eval(infolist[4])
    sizes = eval(infolist[5])
    lights = eval(infolist[6])
    # door_direction = eval(infolist[7])
    # print(door_direction)
    # door_direction = ['N', 'S', 'S', 'S', 'S']
    circulations = eval(infolist[8])
    cir_Weight = eval(infolist[9])

    G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)

    print(order)

    start_end_list = []
    stepper = 0
    path_list = []
```

```
    for i in order:   # order =    [4, 1, 0, 2, 3]
        s = G.nodes[i]['sizes']
        g = order.index(i)
        room_pos_x = rooms_shape_list_pos[g][0]
        room_pos_y = rooms_shape_list_pos[g][1]
        size = G.nodes[i]['sizes']
        d_i = G.nodes[i]['door_direction']

        if type(s) == 'tuple':
            room_width = size[0] * 10
            room_length = size[1] * 10

        else:
            ret = crack(s)
            room_width = ret[0] * 10
            room_length = ret[1] * 10

        room_center = (room_pos_x, room_pos_y)
        room_corner_A = (room_pos_x - room_width / 2, room_pos_y - room_length / 2)
        room_corner_B = (room_pos_x - room_width / 2, room_pos_y + room_length / 2)
        room_corner_C = (room_pos_x + room_width / 2, room_pos_y + room_length / 2)
        room_corner_D = (room_pos_x + room_width / 2, room_pos_y - room_length / 2)
        rowA, colA = get_barrier_pos(room_corner_A, ROWS, width)
        rowB, colB = get_barrier_pos(room_corner_B, ROWS, width)
        rowC, colC = get_barrier_pos(room_corner_C, ROWS, width)
        rowD, colD = get_barrier_pos(room_corner_D, ROWS, width)

        # spot_i = grid[int(rowC+1)][int(colC)]
        # spot_k = grid[int(rowD)][int(colD)]
        # start_end_list.append([spot_i, spot_k])

        if d_i == 'N':
            row = int(rowA + int(round((rowD - rowA) / 2)))
            col = int(colA - 1)
            spot_i = grid[row][col]
        elif d_i == 'S':
            row = int(rowB + int(round((rowC - rowB) / 2)))
            col = int(colB)
            spot_i = grid[row][col]
        elif d_i == 'W':
            row = int(rowA - 1)
            col = int(colA + int(round((colB - colA)/2)))
            spot_i = grid[row][col]
        elif d_i == 'E':
```

```
                row = int(rowD)
                col = int(colD + int(round((colC - colD)/2)))
                spot_i = grid[row][col]
            else:
                row = int(rowA + int(round((rowD - rowA) / 2)))
                col = int(colA)
                spot_i = grid[row][col]

            f = [n for n in G.neighbors(i)]    # getting neighbors of node i

            for i_f in range(len(f)):
                for k in order[0:g]:
                    if k == f[i_f]:
                        weight = G[i][k]['weight']    # getting weight of edge connected to node i
                        sk = G.nodes[k]['sizes']
                        d_k = G.nodes[k]['door_direction']

                        ki = order.index(k)

                        if type(sk) == 'tuple':
                            roomk_width = sk[0] * 10
                            roomk_length = sk[1] * 10

                        else:
                            ret = crack(sk)
                            roomk_width = ret[0] * 10
                            roomk_length = ret[1] * 10

                        room_posk_x = rooms_shape_list_pos[ki][0]
                        room_posk_y = rooms_shape_list_pos[ki][1]

                        roomk_center = (room_posk_x, room_posk_y)
                        roomk_corner_A = (room_posk_x - roomk_width / 2, room_posk_y -
roomk_length / 2)
                        roomk_corner_B = (room_posk_x - roomk_width / 2, room_posk_y +
roomk_length / 2)
                        roomk_corner_C = (room_posk_x + roomk_width / 2, room_posk_y +
roomk_length / 2)
                        roomk_corner_D = (room_posk_x + roomk_width / 2, room_posk_y -
roomk_length / 2)
                        rowkA, colkA = get_barrier_pos(roomk_corner_A, ROWS, width)
                        rowkB, colkB = get_barrier_pos(roomk_corner_B, ROWS, width)
                        rowkC, colkC = get_barrier_pos(roomk_corner_C, ROWS, width)
                        rowkD, colkD = get_barrier_pos(roomk_corner_D, ROWS, width)
```

```
                    if d_k == 'N':
                        row = int(rowkA + int(round((rowkD - rowkA) / 2)))
                        col = int(colkA - 1)
                        spot_k = grid[row][col]
                    elif d_k == 'S':
                        row = int(rowkB + int(round((rowkC - rowkB) / 2)))
                        col = int(colkB)
                        spot_k = grid[row][col]
                    elif d_k == 'W':
                        row = int(rowkA - 1)
                        col = int(colkA + int(round((colkB - colkA) / 2)))
                        spot_k = grid[row][col]
                    elif d_k == 'E':
                        row = int(rowkD)
                        col = int(colkD + int(round((colkC - colkD) / 2)))
                        spot_k = grid[row][col]
                    else:
                        row = int(rowkA + int(round((rowkD - rowkA) / 2)))
                        col = int(colkA)
                        spot_k = grid[row][col]

                    start_end_list.append([spot_i, spot_k, weight])
                    start_end_list.append([spot_k, spot_i, weight])
                else:
                    pass

print(start_end_list)

while run:
    draw(win, grid, ROWS, width, height)

    for event in pygame.event.get():
        if event.type == KEYUP:
            if event.key == K_p:
                pause = True
        if event.type == QUIT:
            exit()
        elif event.type == KEYDOWN and event.key == K_ESCAPE:
            exit()

        if pygame.mouse.get_pressed()[0]:   # LEFT click
            pos = pygame.mouse.get_pos()
            row, col = get_clicked_pos(pos, ROWS, width)
```

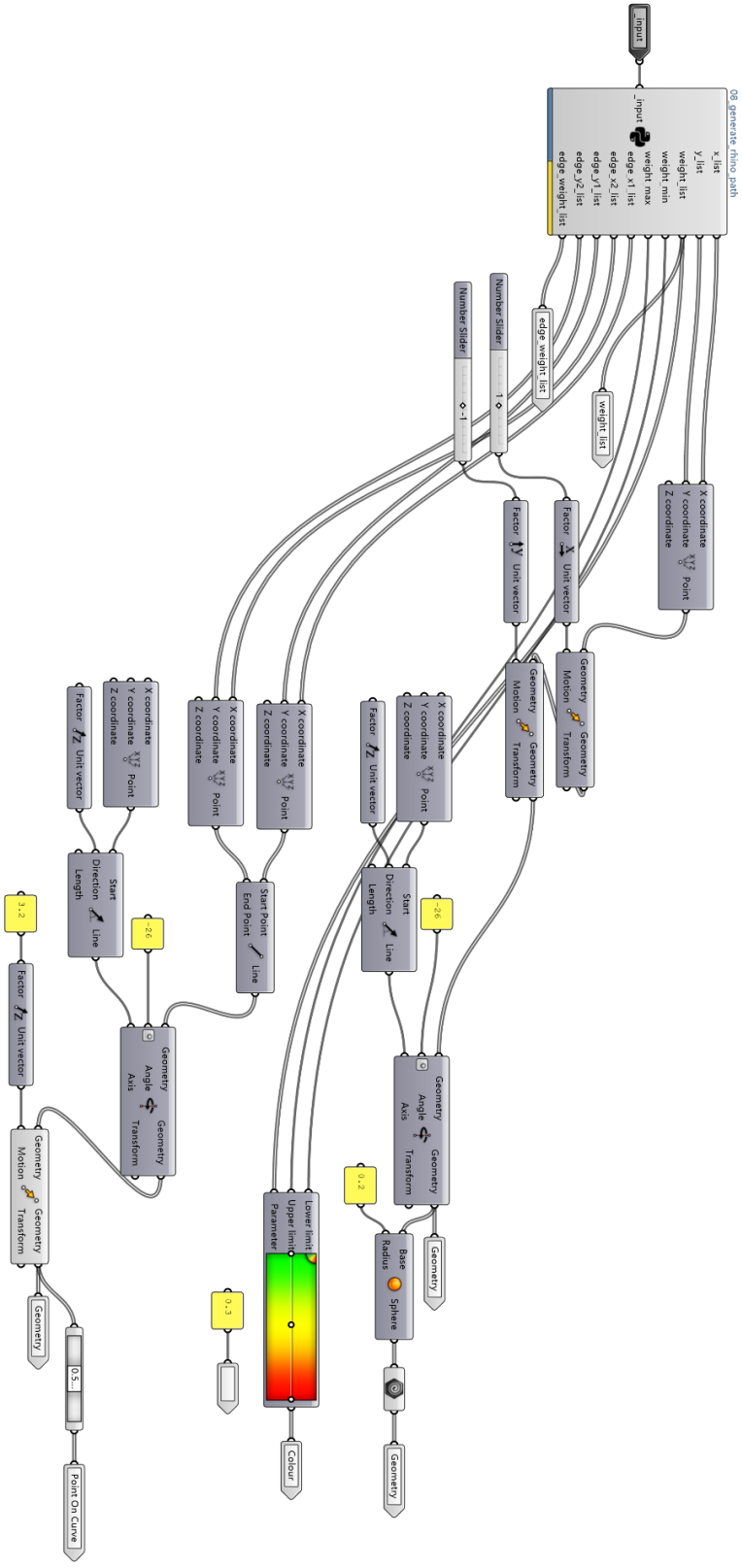
```
            spot = grid[row][col]
            if not start and spot != end:
                start = spot
                start.make_start()

            elif not end and spot != start:
                end = spot
                end.make_end()

            elif spot != end and spot != start:
                spot.make_barrier()

        elif pygame.mouse.get_pressed()[2]:   # RIGHT click
            pos = pygame.mouse.get_pos()
            row, col = get_clicked_pos(pos, ROWS, width)
            spot = grid[row][col]
            spot.reset()
            if spot == start:
                start = None
            elif spot == end:
                end = None

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_d:        # Press D
                # if stepper == len(start_end_list):
                #     pass
                #
                # else:
                #     spot_s = start_end_list[stepper][0]
                #     start = spot_s
                #     spot_s.make_start()
                #     spot_e = start_end_list[stepper][1]
                #     end = spot_e
                #     spot_e.make_end()
                #     stepper = stepper + 1

                while stepper < len(start_end_list):
                    spot_s = start_end_list[stepper][0]
                    start = spot_s
                    spot_s.make_start()
                    spot_e = start_end_list[stepper][1]
                    end = spot_e
                    spot_e.make_end()
```

```
                for row in grid:
                    for spot in row:
                        spot.update_neighbors(grid)

                algorithm(lambda: draw(win, grid, ROWS, width, height), grid, start, end)

                temp_list = []
                path_weight = start_end_list[stepper][2]
                for i in range(160):
                    for j in range(96):
                        if grid[i][j].color == PURPLE:
                            temp_list.append((i, j, path_weight))
                            spot = grid[i][j]
                            spot.reset()
                        else:
                            pass
                path_list.append(temp_list)

                stepper = stepper + 1

            else:
                pass


        if event.key == pygame.K_SPACE:        # Press SPACE
            for row in grid:
                for spot in row:
                    spot.update_neighbors(grid)

            algorithm(lambda: draw(win, grid, ROWS, width, height), grid, start, end)

            temp_list = []
            path_weight = start_end_list[stepper][2]
            for i in range(160):
                for j in range(96):
                    if grid[i][j].color == PURPLE:
                        temp_list.append((i, j, path_weight))
                        spot = grid[i][j]
                        spot.reset()
                    else:
                        pass
            path_list.append(temp_list)
```

```
                    if event.key == pygame.K_c:         # Press C
                        # start = None
                        # end = None
                        grid = make_grid(ROWS, width)

                    if event.key == pygame.K_g:         # Press G
                        f = open(
                            'D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/7_path_info_exchange.txt',
                            mode='w', encoding='utf-8')
                        f.write(f'{path_list}')


                    if event.key == pygame.K_b:         # Press B
                        for i in range(len(order)):
                            node = order[i]
                            room_pos = rooms_shape_list_pos[i]
                            room_pos_x = rooms_shape_list_pos[i][0]
                            room_pos_y = rooms_shape_list_pos[i][1]
                            size = G.nodes[node]['sizes']

                            if type(size) == 'tuple':
                                room_width = size[0] * 10
                                room_length = size[1] * 10

                            else:
                                ret = crack(size)
                                room_width = ret[0] * 10
                                room_length = ret[1] * 10

                            room_corner_A = (room_pos_x - room_width / 2, room_pos_y -
room_length / 2)
                            room_corner_B = (room_pos_x - room_width / 2, room_pos_y +
room_length / 2)
                            room_corner_C = (room_pos_x + room_width / 2, room_pos_y +
room_length / 2)
                            room_corner_D = (room_pos_x + room_width / 2, room_pos_y -
room_length / 2)
                            rowA, colA = get_barrier_pos(room_corner_A, ROWS, width)
                            rowB, colB = get_barrier_pos(room_corner_B, ROWS, width)
                            rowC, colC = get_barrier_pos(room_corner_C, ROWS, width)
                            rowD, colD = get_barrier_pos(room_corner_D, ROWS, width)

                            for j in range(int(colB - colA)):
```

```
                                    for i in range(int(rowD - rowA)):
                                        row = int(rowA + i)
                                        col = int(colA + j)
                                        spot = grid[row][col]
                                        spot.make_barrier()


            space.step(1. / fps)

            clock.tick(fps)
            pygame.display.flip()

            pygame.display.set_caption("fps: " + str(clock.get_fps()))

        pygame.quit()


main(WIN, WIDTH, HEIGHT)
```

Appendix H: Grasshopper layout of module 08_corridor_finder



Appendix I: Python code of module 08_generate_rhino_path

```
import numpy as np
import networkx as nx

def por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight):
    i_sizes = 0
    i_lights = 1
    i_weight_sum = 2
    i_edge_sum = 3
    i_door_direction = 4

    ReL_Array = np.zeros([len(rooms), len(rooms)])
    Attri_Array = np.zeros([len(rooms), i_door_direction + 1])

    # creating ReL_Array
    for i in range(len(rooms)):
        for a in range(len(circulations)):
            for b in range(len(circulations[a]) - 1):
                if circulations[a][b] == rooms[i]:
                    ReL_Array[i, rooms.index(circulations[a][b + 1])] += cir_Weight[a]

    # Flip the array left and right then rotate the array by 90 degrees
    ReL_Array = ReL_Array + np.rot90(np.fliplr(ReL_Array))
    # print("ReL_Array = ", ReL_Array)

    # Creating attribute array
    for i in range(len(rooms)):
        Attri_Array[i, i_sizes] = sizes[i]
        Attri_Array[i, i_lights] = lights[i]
    # print("Attri_Array = ", Attri_Array)

    # get Sum of edge weights on one node
    sum_weight = []
    for i in range(len(rooms)):
        sum_weight.append(sum(ReL_Array[i]))

    # Creating graph in networkx from ReL_Array
    G = nx.from_numpy_array(ReL_Array)
    for i in range(len(rooms)):
        G.nodes[i]['room'] = rooms[i]
        G.nodes[i]['sizes'] = sizes[i]
        G.nodes[i]['lights'] = lights[i]
```

```
            G.nodes[i]['sum_weight'] = sum_weight[i]
            G.nodes[i]['door_direction'] = door_direction[i]

    print("G.edges = ", G.edges(data=True))    # print edges
    print("G.nodes = ", G.nodes(data=True))    # print all nodes with attribute dictionary
    print("G.degree() = ", G.degree())
    print("sum_weight = ", sum_weight)
    return G
# ↑ done, returns a graph contains all information, weights on edges, attributes on nodes

def crack(integer):
    start = int(np.sqrt(integer))
    factor = integer / start
    while not is_integer(factor):
        start += 1
        factor = integer / start
    return int(factor), start

def is_integer(number):
    if int(number) == number:
        return True
    else:
        return False

# generate path points
f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/7_path_info_exchange.txt',
mode='r', encoding='utf-8')

infolist = []

for line in f:
    infolist.append(line.strip())

path = eval(infolist[0])
x_list = []
y_list = []
weight_list = []

path_array = np.zeros([96,160])

for i in range(len(path)):
    for j in range(len(path[i])):
        x = int(path[i][j][0])
```

```
            y = int(path[i][j][1])
            weight = path[i][j][2]
            path_array[y][x] += weight

x_list = []
y_list = []
weight_list = []
for i in range(96):
    for j in range(160):
        if path_array[i][j] != 0:
            x_list.append(j)
            y_list.append(96 - i)
            weight_list.append(path_array[i][j])

weight_sort = sorted(weight_list)
weight_min = weight_sort[0]
weight_max = weight_sort[-1]



# generate edges
f = open('D:/Delft
Courses/Graduation/PythonProject/daylighthour_layout_pathfinding/4_5_info_exchange.txt', mode='r',
encoding='utf-8')
infolist = []
for line in f:
    infolist.append(line.strip())

phase_index = int(infolist[0]) + 1    # 0+1 = 1
rooms_shape_list_pos = eval(infolist[1])   # [(337, 367), (395, 457), (437, 357), (356, 302), (345, 447)]
body_type_list = eval(infolist[2])     #     [0, 0, 0, 0, 0]
order = eval(infolist[3])
rooms = eval(infolist[4])
sizes = eval(infolist[5])
lights = eval(infolist[6])
door_direction = eval(infolist[7])
circulations = eval(infolist[8])
cir_Weight = eval(infolist[9])

G = por_to_graph(rooms, sizes, lights, door_direction, circulations, cir_Weight)

edge_x1_list = []
edge_x2_list = []
edge_y1_list = []
edge_y2_list = []
```

```
edge_weight_list = []

for i in order:    # order =    [4, 1, 0, 2, 3]
        g = order.index(i)
        room_pos_x = rooms_shape_list_pos[g][0]
        room_pos_y = 960 - rooms_shape_list_pos[g][1]



        f = [n for n in G.neighbors(i)]    # getting neighbors of node i

        for i_f in range(len(f)):
                for k in order[0:g]:
                        if k == f[i_f]:
                                weight = G[i][k]['weight']    # getting weight of edge connected to node i
                                ki = order.index(k)
                                room_posk_x = rooms_shape_list_pos[ki][0]
                                room_posk_y = 960 - rooms_shape_list_pos[ki][1]

                                edge_x1_list.append(room_pos_x/10)
                                edge_x2_list.append(room_posk_x/10)
                                edge_y1_list.append(room_pos_y/10)
                                edge_y2_list.append(room_posk_y/10)
                                edge_weight_list.append(f"Weight = {weight}")

                        else:
                                pass
```