A routing algorithm for building ventilation systems

S.A. Bijl
W.R. van Geest
B.A. van der Maarel
S. Ploegsma

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

A ROUTING ALGORITHM FOR BUILDING VENTILATION SYSTEMS

Simon Bijl
Wiebe van Geest
Bryan van der Maarel
Sander Ploegsma

Bachelor's Thesis
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
June, 2014

Presentation:	Thursday July 3, 2014	
Supervisor:	Dr. M.T.J. Spaan	
Thesis committee:	Dr. M.T.J. Spaan	TU Delft
	Dr. M.A. Larson	TU Delft
	Ir. R. Qualm	Stabiplan

TU Delft Delft University of Technology

Stabiplan

PREFACE

This is the bachelor's thesis created by Simon Bijl, Wiebe van Geest, Bryan van der Maarel and Sander Ploegsma as part of the Bachelor Computer Science program at Delft University of Technology. Over the course of three months we have examined the possibilities of automating air system creation in 3D modelling software. This project is commissioned by Stabiplan B.V.

We would like to thank everyone at the Stabiplan B.V. development team for their constant support over the course of our project. Another thank you goes out to René Qualm and Johan Schlingmann, Software Architect and Manager Product Management at Stabiplan B.V. for their excellent coordination.

Finally, a special thanks to our supervisor, assistant professor Dr. Matthijs Spaan from the Algorithmics Group at Delft University of Technology, for his guidance and enthousiasm during the entire project, as well as his clear expectations and feedback on our progress.

Simon Bijl
Wiebe van Geest
Bryan van der Maarel
Sander Ploegsma
Delft, June 2014

SUMMARY

The design process of buildings include the planning where mechanical, electrical and plumbing equipment should be placed, known as mechanical, electrical, and plumbing (MEP) design. This report describes the design and implementation of a software extension to automate the placement of air ducts in buildings, for a building modeling software package called Revit. The main goal is a system that, given a building with unconnected vents, will produce ducts to connect those vents to a certain source.

This report will discuss possible algorithms for solving the problem. One of these algorithms is an approach that creates a grid around each connector and uses *Kruskal's Algorithm* to create a minimum spanning tree (MST) through that grid, to create the shortest path through all connectors. This algorithm did not yield optimal solutions as it tended to make too many corners. Also, "greedily" selecting all shortest paths between connectors often resulted in strange paths through the grid.

Another approach, which is our final prototype, uses points created by the user which define the location where a *main trunk* of the duct system should be placed. The algorithm uses clustering techniques to divide the problem in smaller problems and will solve these clusters with structures that are regularly used in the building industry. It gives the user the opportunity to pick his or her desired solution for each cluster, enabling a lot of customization.

Features of the system include wall avoidance and the use of duct connectors using industry standards. Features which are desirable but not implemented within this project are an extended three-dimensional support, more and intuitive user interaction, more support for wall intersections and support for capacity calculations.

The testing of the prototype is based on unit tests, where all classes are tested individually and extensively during the entire course of the project to make sure that no unforeseen situations would occur during the development process. The performance of the algorithm was tested by hand using different Revit models. During these tests we observed whether the decisions made by the algorithms followed the concepts of MEP design used in real life.

CONTENTS

1

INTRODUCTION

A part of the design and construction process of buildings is to plan where ducts, cables and pipes have to be placed [1]. These tasks are often referred as mechanical, electrical, and plumbing (MEP) design, It depends on MEP design where certain facilities are available, how the documentation is done (for future maintenance), how the facilities perform and of course how much the building costs. Therefore MEP design has a crucial role in the whole design and construction process of buildings.

MEP design is often performed with the use of building information models (BIMs). These digital models contain detailed information about objects placed therein and the relationship between these objects [2].

In the process of designing buildings, many contractors and other actors play a role. BIMs have the advantage that those actors can all work with the same design system. This way conflicts are less likely to happen.

In this project, we will work with the BIM design software called Revit, made by AutoDesk.

The purpose of the project is to design an algorithm which creates a network of ducts to provide air to vents in a building, since this is still a manual process. In figure 1.1, an example has been given of the duct design within a building. During this project, we created an extension for Revit, in C#, implementing the algorithm.

Our first two prototypes were built parallel to each other, where one is based on a minimum spanning tree (MST) in combination with a grid, and the other one is based on a MST without a grid. After these two prototypes, we use the best method as basis to improve it to a final prototype.

This report is a combination of documentation and discovery of our progress and the choices we made during development, as the task was too large to complete in one quarter. We will describe the possible areas for other groups to look into to pick up and further improve the extension.

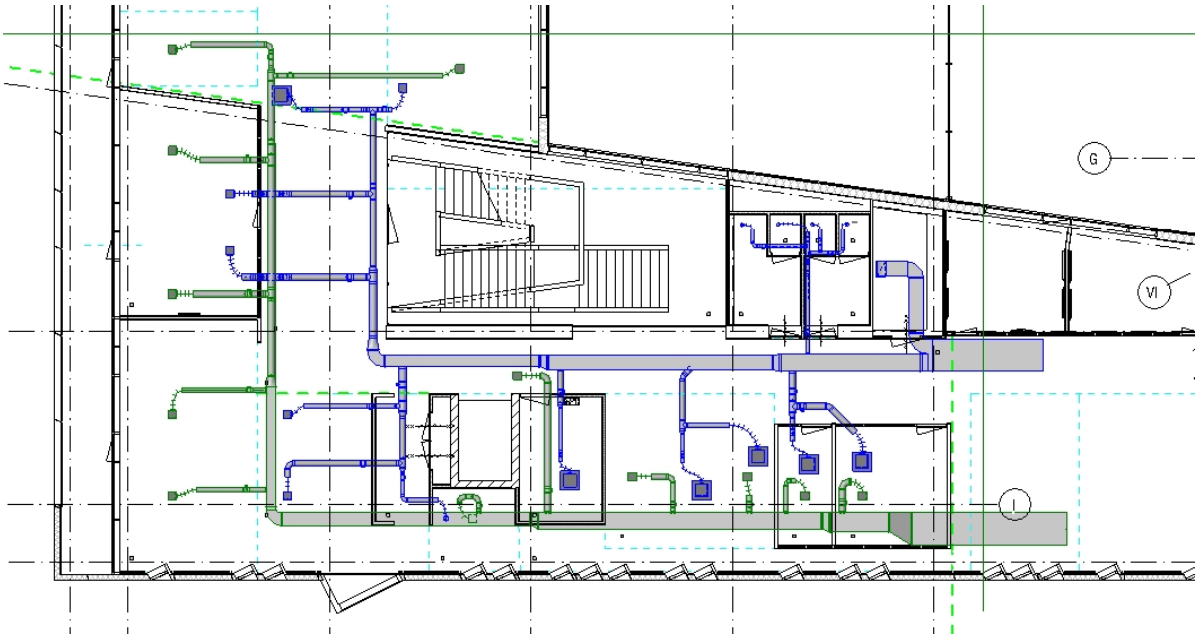


Figure 1.1: The design of the Enexis building in Maastricht. Two seperate networks are visible: the blue one to provide air and the green one to drain air.

2

PROBLEM DEFINITION

Stabiplan B.V. is a software development company focused on creating software for the MEP market. The primary product that they develop is called Stabicad.

Companies active in the MEP market design the layout of ducts, vents and service pipes. The development of a building starts with a client and an architect. The architect creates a design focusing mainly on aesthetics. Civil engineers then create a model suitable for construction. Revit is a program built to create this type of architectural and engineering work. Stabicad is designed on top of Revit and uses Revit models to create the layout of the ducts and pipes.

At the moment, this layout is manually designed and drawn. This makes it a costly and time consuming task, which may well lead to layouts that are suboptimal. Revit has built-in support for the automation of this task, but unfortunately it mainly produces solutions not feasible in real-life applications.

The goal of this project is to deliver a prototype that is able to automatically connect a given air system. After the project, Stabiplan B.V. will be able to decide if the automation of connecting MEP systems is something that is worth looking into.

2.1. REQUIREMENTS

Since this prototype is not yet meant as a final product that can be incorporated directly into Stabicad, some demands and constraints are defined:

- The prototype will only focus on air systems. The use of multiple kinds of systems only provides an increased difficulty during the prototype development, while the logic used in the algorithms may be easily adjusted to support multiple systems later on.
- The prototype will only focus on the underlying algorithm used to connect a system, as the knowledge needed to improve usability and Revit interaction is already present within Stabiplan B.V. and is therefore not our primary concern.
- The prototype will be able to connect a system that is moderate in size, meaning that the number of connectors is not too high. For this project, a maximum of 50 connectors is set. Scaling the algorithm to support incredibly large systems is not within the scope of this project.

The requirements for the prototype are as follows:

- The prototype is able to create a valid layout based on a given set of connectors. A layout is considered valid if it does not collide with bearing items such as some walls or pillars, and connects all given inputs and outputs.
- The prototype is integrated into Revit.

- The prototype is properly documented to maximize its maintainability and to encourage further development of the prototype by others.
- The prototype is able to handle user input, to enable users to influence the creation of layouts.

The prototype uses the following input:

- A designed building in Revit format. The location(s) of the air sources are already known.
- The locations of the air vents (the objects that should be connected).
- Points defined by the user indicating the location of the *main trunk*.

The output will be one or more feasible solutions if a feasible solution exists. A feasible solution should satisfy the following conditions:

- All chosen points of the input are connected to an air exit.
- No bearing walls are intersected by the solution.

3

THE ALGORITHM AND ITS IMPLEMENTATION

This section discusses the process of designing the final algorithm based on research and experiments. In section 3.1 the results of implementing the algorithms found in the research phase are discussed. Section 3.2 describes the different obstacles found when implementing the final solution and how the encountered problems were solved.

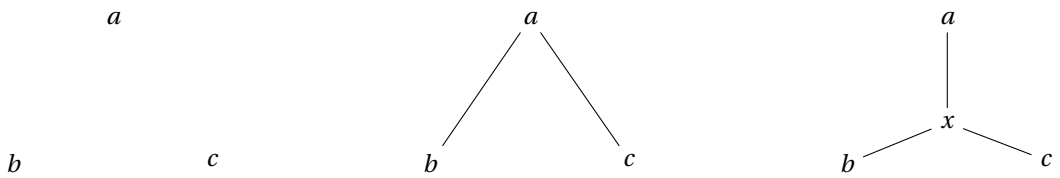
3.1. EXPLORING THE RESEARCH RESULTS

After the research phase, our initial goal for the algorithm was to connect all vents and sources without considering any constraints. The algorithm could then later be adjusted to include these.

The models used for these approaches are almost identical to the ones used in the final prototype. The only key difference is the use of collections of lines and points instead of the tree structure described in chapter 4.

3.1.1. RECTILINEAR STEINER TREE APPROACH

In the research report (found in appendix E) *minimum Steiner trees* [3] were discussed, which are MSTs with the addition of *Steiner points*; extra points created in order to further reduce the network length, as seen in figure 3.1. Rectilinear minimum Steiner trees (RMSTs), which consist of only straight angles, can be found using a grid consisting of lines and intersections formed by extending lines along, in this case, the horizontal and vertical axis from the points to be connected, as seen in figure 3.2. Such a grid is called a *Hanan grid* [4].



(a) Simple set of graphs (b) Example MST (c) Shortest network

Figure 3.1: Possible spanning solutions

One attempt at creating a solution path is a greedy algorithm for RMSTs using an adaption to *Kruskal's Algorithm*, an algorithm devised by Kruskal to find an MST in a set of nodes. First, a Hanan grid is created using the vents and air sources. This grid is then used as a graph through which a tree has to be found. In this grid a distinction between two types of points can be defined: *terminal points* and Steiner points. A valid solution is a tree that at least connects *all* terminal points, preferably using as little line length as possible. The steps to find such a solution are as follows:

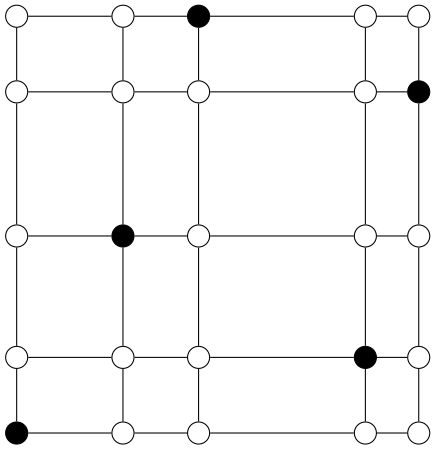
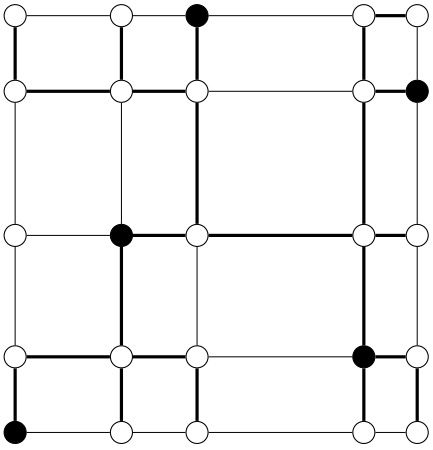


Figure 3.2: The Hanan grid for an RMST

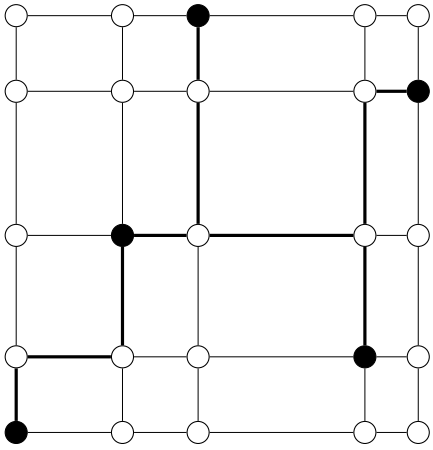
1. Sort all lines in the Hanan grid, first to ascending length, then descending on the amount of terminal points it connects.
2. While not every terminal point is connected, take the first line in the grid and, if it doesn't create a cycle, add it to the solution set.
3. When everything is connected, sort the lines in the solution descending on length.
4. For every line in the solution, remove it from the solution and check if all terminal points are still connected. If so, proceed, otherwise re-add it.

This follows the basics of Kruskal's Algorithm to create an MST through the graph. However, since the Hanan grid adds Steiner points on each line crossing, the graph contains many more points than necessary to create a solution. Therefore, once everything is connected, all lines that are not part of a branch that connects a terminal point are removed, starting with the longest (or most expensive) line.

A possible solution after step 2 is shown in figure 3.3a. While the path through all terminal points is present in this solution, it still contains unnecessary paths to other Steiner points. These paths are deleted in step 4 of the algorithm to obtain a valid and quite optimal solution, seen in figure 3.3b.



(a) A possible outcome after creating an MST through the Hanan grid.



(b) A possible outcome after removing the obsolete connections.

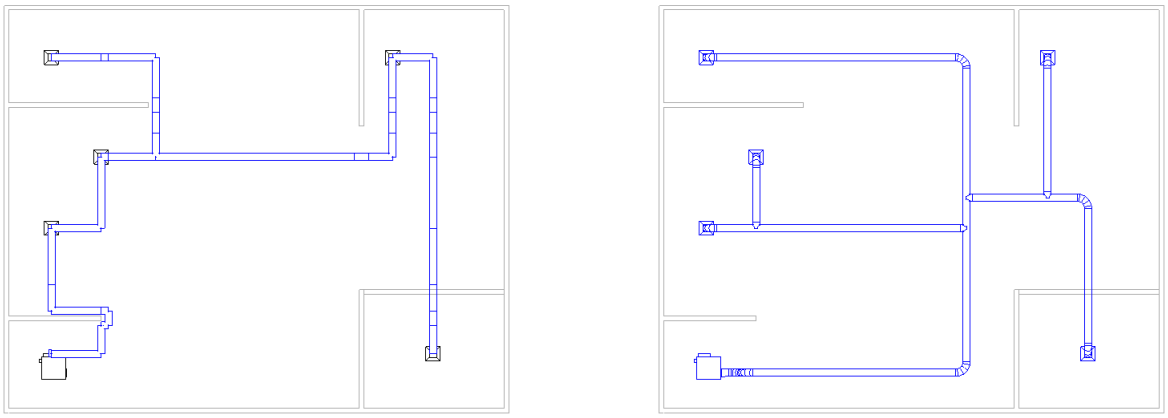
Figure 3.3: Possible outcomes of the RMST approach

To prevent collisions with bearing walls, Steiner points are set around the corners of each wall using a predefined offset, to encourage the use of paths along each wall. After the grid is created, each connection that intersects with any bearing wall is removed from the grid.

Conclusion

The use of Kruskal's Algorithm in combination with Hanan grids does not yield very elegant solutions in practice. It only accounts for the length of ducts, while the amount of turns, for instance, also greatly contributes to the optimality of a solution. One solution found by this algorithm can be seen in figure 3.4a. Note that the use of the smallest edges does not imply that the solution is in any way optimal, using longer edges may yield a far more feasible system, as seen in figure 3.4b. The main weakness in this approach is the fact that it sometimes makes an unfortunate decision, which may have great impacts on the final solution (an obvious example is the U-turn in the upper right corner of figure 3.4a).

Also, the length of ducts is just one of many indicators of cost. Air flow, material, and connectors all contribute to the cost of an entire system. These are all factors that can not easily be taken into account by Kruskal's Algorithm.



(a) A solution found by Kruskals Algorithm

(b) A more feasible alternative solution

Figure 3.4: Screenshots of solutions in a test environment

3.1.2. Regular MST algorithm

Another attempt to create a solution path is also loosely based on Kruskal's Algorithm, although it is far more versatile.

This implementation does not use a Hanan grid and therefore will not necessarily create a solution with only right angles. This may result in a more efficient solution with respect to material costs, but may also require more expensive custom made parts. In this approach, a user-defined main trunk line will be calculated using Kruskal's Algorithm, including the air source. This is done because users often prefer a solution which is not optimal with respect to line length or cost, mainly because it is more practical or it fits the building layout better. The customized algorithm works as follows:

1. Sort every edge between every node of the preferred main trunk line and call this set S.
2. For every element of S, add it to the solution iff this edge doesn't create a cycle in the solution.
3. For every vent, calculate a short path to a line in the solution and add this path to the solution. Please note that it is not obligated for this path to end directly on the main trunk: it is possible that the edge will end at a line of another vent to the main trunk.

Conclusion

Although this algorithm does not yet account for wall collisions and may create junctions for which no connectors exist, it does yield more viable solutions. The user-defined main trunk creates a very realistic situation that can be seen in many real-life solutions. A solution found by this algorithm is shown in figure 3.5.

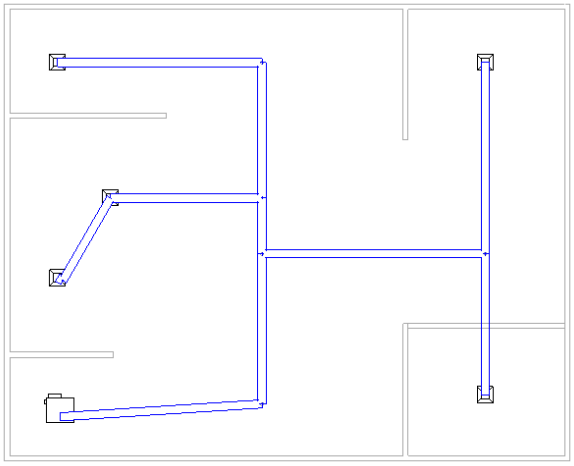


Figure 3.5: Solution found by the regular MST algorithm discussed in section 3.1.2.

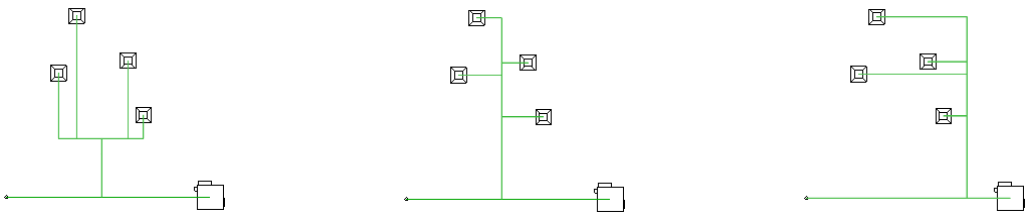
3.2. FINAL PRODUCT

We decided that the second of the two algorithms discussed in section 3.1 showed the most potential, and spent the rest of the project further developing that approach. This section gives a high level description of the algorithm. For a more in-depth description, please refer to appendix A.

3.2.1. CLUSTERING

We noticed a particular strategy while analysing the examples provided by the client, namely the clustering of air terminals. The idea behind this is that a set of air terminals are determined to be grouped together, connected to a side duct, and finally connecting this side duct to the main trunk. Such a group is called a *cluster*.

The most used cluster solutions observed in the examples are so-called forks, fish bone and knife connections for clusters. Fork cluster connections are in the shape of a fork: a main duct, called the *cluster trunk* is laid out, where the clustered air terminals all connect to. This is connected to the main trunk line in the middle of the cluster trunk, creating the characteristic fork shape. On the other hand, the cluster trunk of the fish bone and the knife, is perpendicular with the main trunk. These knife and the fish bone differ on where the cluster trunk is laid out: the knife cluster trunk lies outside the cluster while the fish bone cluster trunk lies inside the cluster. These three cluster types are illustrated in figure 3.6.



(a) A cluster connected by a fork (b) A cluster connected by a fish bone (c) A cluster connected by a knife

Figure 3.6: Screenshots of three approaches to solving a cluster

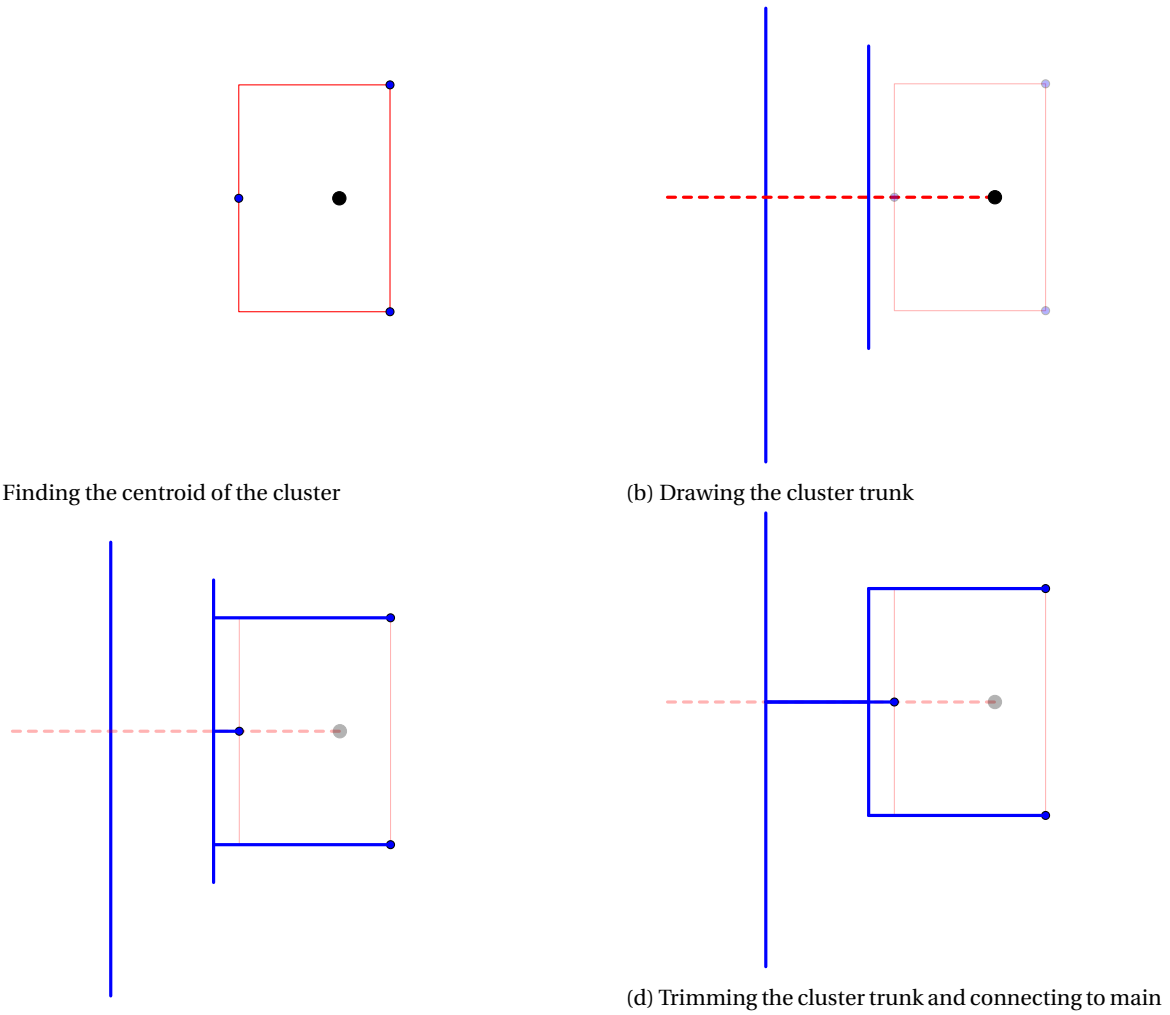
First, the algorithm will identify which points are part of a cluster. This process works as follows:

1. Generate an MST using Kruskal's Algorithm.
2. Remove the lines of the MST intersecting with the main trunk or with a wall. Furthermore, remove all lines which are too long (set by the user). All groups of points which are still connected are seen as clusters.

3. From this result, separate the clusters to single vents if the size of the cluster is smaller than the user defined minimum size of clusters.

Then, each cluster is solved individually, as illustrated in figure 3.7. This works as follows:

1. Calculate the centroid of this cluster.
2. Find the line segment of the existing solution where this cluster will connect.
3. Position a cluster trunk next to the cluster, perpendicular or parallel to the previously found line segment, depending on cluster type preference. This cluster trunk is stretched to make sure all vents can connect to it.
4. Connect the vents to the cluster trunk.
5. Trim the cluster trunk to the minimum size, so that all vents are still connected.
6. Add a small protrusion from the trunk to connect to, avoiding oddly shaped T sections. A fishbone or knife type cluster trunk will extend it in the direction of the cluster trunk, a fork type will extend it from the root location away from the cluster, perpendicular to the cluster trunk.
7. Return the cluster root.



(a) Finding the centroid of the cluster

(b) Drawing the cluster trunk

(c) Connecting the cluster

(d) Trimming the cluster trunk and connecting to main trunk

Figure 3.7: The cluster solving process

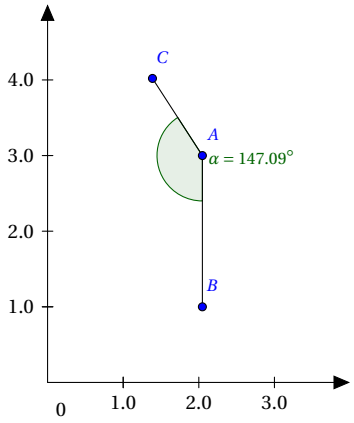
3.2.2. CONNECTING CLUSTERS TO THE MAIN TRUNK

After each cluster is connected, their root connectors are connected to the main trunk, as well as remaining connectors that do not belong to any cluster. This process is based on the algorithm defined in section 3.1.2, with some improvements:

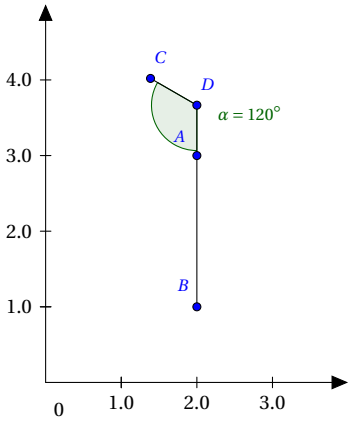
1. For each connector C in descending order of their distance to the main trunk, calculate the closest point C' on the main trunk.
2. Correct the angle between the main trunk and the line (C', C) in the way described in section 3.2.3. This may result in a new connector being added between C and C'.
3. Connect C' and C and check if that tree is feasible: if any connection in the tree intersects with any bearing wall it is considered infeasible. If the tree is feasible, connect it to the main trunk.
4. For each connector D that has not been connected in the previous steps, find a path around all walls to the main trunk in the way described in 3.2.4.

3.2.3. ANGLE ADJUSTMENTS

Since duct connectors only come in pre-defined sizes and angles, using arbitrary angles in a solution is not practical. Industry standard components usually have angles such as 30°, 45°, 60°, 90° and 120°. To make sure that only these angles are used, lines are adjusted in the solution when connecting parts of the system. In figure 3.8a a solution is shown that connects a point C to a line segment (A, B). In order to create a correct angle, the line (A, B) is extended to point D, as seen in 3.8b. The newly created line (A, D) forms an angle of 120° with the line (C, D).



(a) Original angle



(b) Angle after adjustment

Figure 3.8: The adjustment of angles

3.2.4. AVOIDING WALLS

In some situations, there is no direct path from a vent to the main trunk without a wall collision. Therefore, the algorithm contains a feature to navigate around walls, which is implemented with the following (brute force) algorithm.

First, it will calculate side points at a small distance from the corners of the wall that is the nearest wall between the vent and the main trunk. If none of these side points can be reached from the vent (because there is another wall between the side points and the vent), the procedure will be repeated with the wall between the vent and each side point until there are one or more feasible paths calculated.

With one or more feasible paths calculated, the algorithm will calculate the cost of each path and the path with the lowest costs will be part of the solution.

An illustration of the wall avoidance can be found in figure 3.9.

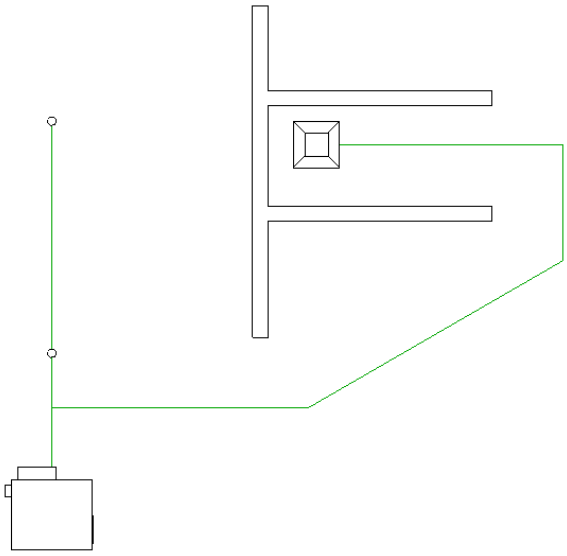


Figure 3.9: An example of wall avoidance. The source and the main trunk are on the left side of the image, while the vent is almost surrounded by walls.

TODO: do we want to show sub-steps as well? It could help in understanding the code.

3.2.5. Calculating costs

The research report states that multiple costs are essential to the optimality of a solution. Material costs, construction costs and energy costs are all factors that should be considered when solving a given system. Unfortunately, there is no way to deduct component costs, since the components used in Revit are generic and no price information is known to the system. Other cost factors such as flow capacity are also not available at runtime, since they are not known until the system is drawn in Revit, which is far too time consuming to do while calculating the cheapest path in segments.

We decided to link algorithmic cost estimates to our solution, where different connector types (such as T-junctions) have pre-defined costs. These costs are simple constants indicating which connection types are preferred over others. This way, our algoritm may prefer solutions with less corners over another one that includes many more.

3.2.6. Geometry

Since the algorithm uses a three-dimensional coordinate system, some of the calculations done while solving a system can be quite complex. This subsection covers the basics of most of the calculations done, as well as their reasoning.

Line-line intersections

The calculation of intersections between line segments in three dimensional space is based on an approach to find the shortest line between two lines [5]. After finding that shortest line, the endpoints of that line are checked for equality. If they are equal and either point lies on one of the line segments, then these segments are said to intersect. Due to the amount of operations done by this calculation, a threshold is defined to prevent both floating point and rounding errors.

Wall collisions

One of the major problems in the calculation of a feasible path between vents is the detection of walls. This problem relies on heavy calculations using vector algebra to check if a line intersects with any of the sides of a wall. If so, that line is said to collide with that wall and thus will be not be included in the path.

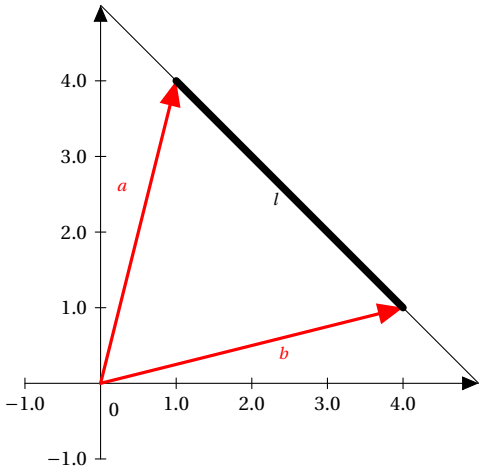


Figure 3.10: A (two-dimensional) representation of l

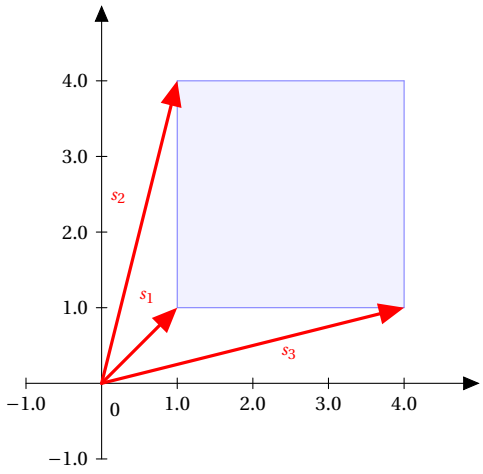


Figure 3.11: A (two-dimensional) representation of s

First, a line l is represented in vector format, where $\mathbf{a}$ and $\mathbf{b}$ are the start and end points of l and $\mathbf{l}(\mathbf{t})$ is any point on l with a distance t from $\mathbf{a}$, as seen in figure 3.10. Note that this formula does not respect the bounds of l and therefore can be seen as a ray, which has infinite length.

$$\mathbf{l}(\mathbf{t}) = \mathbf{a} + t(\mathbf{b} - \mathbf{a}) \tag{3.1}$$

The side of a wall s is also represented in vector format using one corner $\mathbf{s_1}$ as reference to origin and its two adjacent corners $\mathbf{s_2}$ and $\mathbf{s_3}$ to describe the object, as seen in figure 3.11. This does assume that all corners of s are coplanar.

The normal $\mathbf{n}$ to the plane P of s is given by $\mathbf{n} = (\mathbf{s_2} - \mathbf{s_1}) \times (\mathbf{s_3} - \mathbf{s_1})$ and any point $\mathbf{m}$ that belongs to P has to satisfy

$$\mathbf{n} \cdot (\mathbf{m} - \mathbf{s_1}) = 0. \tag{3.2}$$

Since the intersection between P and $\mathbf{l}(\mathbf{t})$ lies on both components, (3.1) and (3.2) can be combined to $\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{b} - \mathbf{a}) - \mathbf{s_1}) = 0$ and solved for t[1]:

$$t = \frac{-\mathbf{n} \cdot (\mathbf{a} - \mathbf{s_1})}{\mathbf{n} \cdot (\mathbf{b} - \mathbf{a})} \tag{3.3}$$

Substituting (3.3) in (3.1) gives the final equation for $\mathbf{m}$ defined as

$$\mathbf{m} = \mathbf{a} + \frac{-\mathbf{n} \cdot (\mathbf{a} - \mathbf{s_1})}{\mathbf{n} \cdot (\mathbf{b} - \mathbf{a})} (\mathbf{b} - \mathbf{a}) \tag{3.4}$$

Since $\mathbf{m}$ defines the intersection point between $\mathbf{l}(\mathbf{t})$ and P, it should be projected onto l and s to check if it lies within their bounds.

One small problem with this approach is that lines *touching* an edge of the wall are not considered as colliding with it. Also, lines completely *contained* by a wall will not intersect its sides at all, so those are also not considered collisions. Mathematically, this means our approach is not very solid, however, air vents or other mechanical equipment placed inside bearing walls are very unlikely, so lines like these will probably never exist. Since checking for above errors increases the difficulty of collision detection greatly, it was decided to omit these checks from the approach.

[1] Note that t essentially defines a distance from the ray's origin a to the point $\mathbf{l}(\mathbf{t})$ on that ray that is also on P. This means that when l is parallel to P, t will get infinitely high, since $\mathbf{n} \cdot (\mathbf{b} - \mathbf{a})$ will result in 0.

4

CODE DESIGN

Revit is a programme with a multitude of capabilities, which causes its objects models and functionalities to be complicated and sometimes hard to work with. Since the algorithm does not require all this functionality, it was decided to create a separate framework, enabling the algorithm to function without direct interaction with Revit. Section 4.1 describes this framework. Since input from and output to Revit is still necessary to interact with the algorithm, so a set of classes was created to collect and generate data from and in Revit, respectively, and translate the two different types of data between the algorithm and Revit. These classes are discussed in section 4.2.

4.1. ALGORITHM DESIGN

In order to be able to simulate the building environment, a data structure is needed. This section describes the classes used throughout the algorithm.

4.1.1. MODELS

The algorithm interprets each building as a set of connectors, such as air terminals or sources, and a set of walls. The following object classes are used to create such a system:

`APoint` A point in a three-dimensional space, given by X, Y and Z coordinates.

`AVector3` A three-dimensional vector representation. This class contains all the functionality that is needed for vector algebra, such as additions, dot-products, cross-products and normalization.

`APlane` A plane within a three-dimensional space. Uses three `AVector3` properties to describe the distance from the origin as well as two perpendicular edges of the plane to describe the direction of the plane.

`AConnector` A connector has a location, given by an `APoint` and a certain pre-defined `AConnector.Type`. This type is either one of the following: `Source`, `Vent`, `Trunk`, `Cluster` or `Other`. To simplify the connections between connectors, a tree-like structure is used, where all connectors contain references to their parent and children. This results in efficient path traversal as well as the support for the direction of air flow through the system. Every junction in a path is represented by a connector.

`AWall` A wall consists of six `APlanes`, defining the faces of the wall.

`ABuilding` A building contains a collection of `AConnectors` representing all the air vents in the building, a collection of `AWalls` representing the building's walls. It also contains a single `AConnector` representing the air source.

The class diagram seen in figure 4.1 depicts these models.

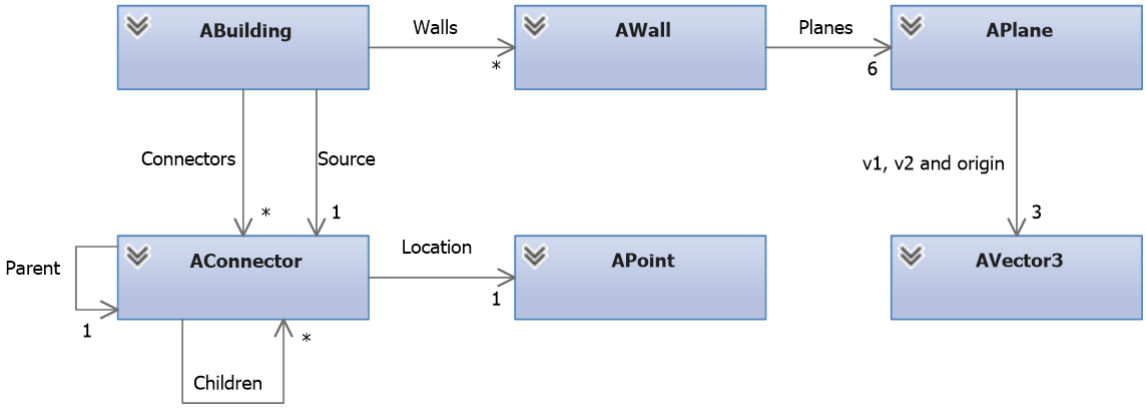


Figure 4.1: A simplified class diagram describing the models defined in 4.1.1

4.1.2. SOLVERS

To solve a system of connectors, multiple solvers are used:

`ASolver` The "master" solver. This class first calculates the main trunk using the source and the user points. It calculates which subsets of vents should be clusters (see section 3.2.1 how clusters are created) and the class uses `AClusterSolver` for creating all possible solutions within these clusters. When the user accepts a certain cluster solution combined with a path to the main trunk, the solver will add this cluster solution to the final solution. After the user has decided how all clusters should be solved, the final solution is finished.

`AClusterSolver` This class solves a given cluster. On the input of a subset of vents defined by `ASolver`, it will calculate possible cluster solutions such as a fork, a knife or a fish bone construction.

`AConnectionSolver` The class that is responsible for creating a path. This path may be a path between a vent or a cluster and the main trunk, but also within a cluster. This class is responsible for wall avoidance and for creating connectors with only certain angles (as described in sections 3.2.3 and 3.2.4).

`AMSTSolvers` A class for calculating MSTs with different algorithms using different output formats such as a tree structure or a collection of edges.

`ASolverUtil` A collection that contains multiple static utility methods used in the solvers.

The dependencies between classes are also shown in figure 4.2.

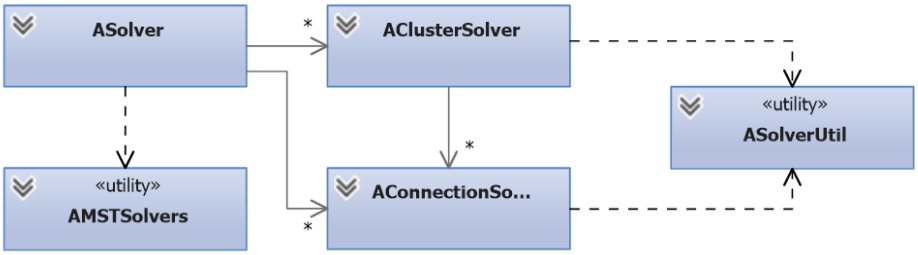


Figure 4.2: A class diagram of the solver classes defined in 4.1.2.

4.1.3. GEOMETRY

When connecting a system, complex calculations, such as line-line intersections in three-dimensional space, are performed extensively. All these calculations are centralized in the static `Geometry` class. It uses a combination of basic geometry and linear algebra to compute intersections, rotations and distances. It uses the

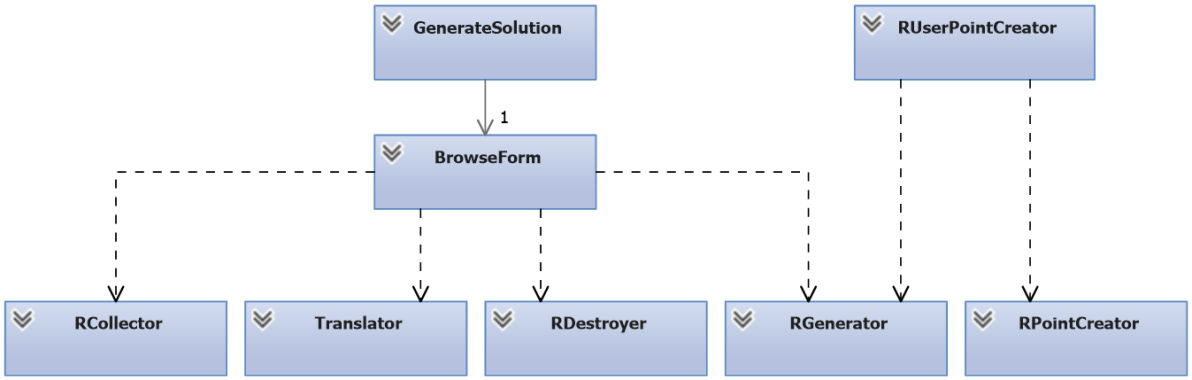


Figure 4.3: Layout of Revit data control

same `AVector3` and `APlane` models as defined in subsection 4.1.1.

4.2. Revit design

This section explains the interaction with Revit, which is all done using the built-in application programming interface (API), a library that specifies routines and data structures used in Revit.

4.2.1. Revit data control

As discussed before, since the Revit data structure is big and complex, the choice was made to limit interaction from the algorithm with Revit. However, there are four critical operations needed to make the algorithm function: the collection, translation, generation and destruction of data and objects, all of which will be described in the following subsections. The result is the class structure depicted in figure 4.3.

4.2.2. Collector

The algorithm requires a number of different objects in order to be able to create a solution:

Vents Vents are the locations where the air exits the system. Within Revit, they can be selected automatically by using the `Category` trait, which in this case equals `"Air Terminals"`.

Sources Air sources do not have a common shared trait like vents. Initially, a hard-coded specific element was used as source. After Prototype 1, the user was required to select an element which would function as source.

User Points User Points are custom elements, which makes selecting them easy, since they have a unique Family name.

Walls Walls are easy to select within a project, but difficult to define in a way which is sensible for the algorithm. As with Vents, these can be filtered using the `"BuiltInCategory.OST_Walls"` which returns all walls in the active document.

The data collection is handled by the RCollector class.

4.2.3. Translator

The algorithm uses its own custom data structure, which is not directly compatible with Revit. In order to enable the algorithm to use Revit data and creating the solution in Revit, a translation is required in both directions. The `Translator` class contains the functions required for this translation.

From Revit

With all elements collected from Revit, the translation into data structures used by the algorithm can start.

As described in section 4.1, the algorithm requires an `ABuilding` containing the elements specified in subsection 4.2.2.

Vents, sources and user points are all translated into `AConnector` objects. The location of the `FamilyInstance` is extracted and translated into an `APoint`, the Revit ID is used as `AConnector` ID and the type is set to `Vent`, `Source` and `Trunk` for vents, sources and user points, respectively.

Revit walls are translated to `AWall` objects, where the sides of each wall are calculated and stored in a list, and the Revit `Structural` property used to set the `AWall` boolean property `IsBearing`, where `IsBearing == Structural`.

The `GeometryData` of a wall is used to perform the actual translation from a `Wall` to an `AWall`. Most walls have a standard of 6 faces. Two specific edges of the face are chosen such that they are perpendicular to each other, which results in a rectangular face, regardless of the actual shape. Lastly, the normal to both those vectors is calculated and stored.

TO REVIT

To properly convert the algorithm data back into a suitable format for Revit, the tree structure has to be translated back into a set of lines. As the solution tree is traversed, starting at the root, the locations of each parent to child is converted into a `Line` element. No coordinate conversion is needed, since the solution space and the Revit environment share Revit's coordinate system. Using the `XYZ` coordinates as the endpoints of a `Line` gives the option to further use those to make either lines indicating the location of connections, called `ModelLines`, or `Ducts`.

4.2.4. GENERATOR

After the data is converted into a format Revit can comprehend, the parts that visualize the solution are generated.

DUCTS

The Ducts are first represented as `ModelLines` in Revit, to determine if the given solution is acceptable. The creation of a `ModelLine` involves calculating a `(Sketch)Plane` that spans the `Line`. This `Plane` is similar to the planes in linear algebra: they are found by calculating the normal of the line as a vector and creating a `Plane` with the same normal.

Once accepted these lines are converted to ducts in Revit. The Stabicad `GenericNodeSolver` is used to connect the ducts to the closest point in the solution (which is above the air terminal), which completes the solution of the system. This `GenericNodeSolver` calculates the type of junction the given ducts form and determines how to connect them in such as way that Revit accepts it as a valid connection.

4.2.5. DESTROYER

Even though our solution does not contain many elements we might want to destroy, being able to clean up and replace is a good thing to have. This class mainly used to remove the preview lines we use during the user input stage of the algorithm.

5

THE PROTOTYPE

Revit encorporates a way to extend its base functionality by adding external add-ins. These add-ins are basic programs that interact with the data within the current Revit document. Our prototype is developed as an add-in to enable complete Revit integration, providing its own submenu and interface, seen in figure 5.1.

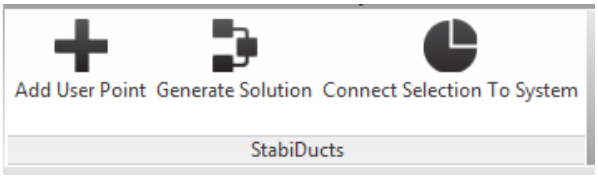


Figure 5.1: The StabiDucts menu in Revit

When a Revit design document contains the basics for a system, a user is able to automatically connect the air vents in that building to a source of their choosing. The algorithm will then try to connect all the air terminals in the document to the assigned source.

5.1. GENERATING A SOLUTION

First, the user is required to place *user points* indicating locations where the main trunk of the solution is laid out. After clicking the *Generate solution* button located in the *StabiDucts* menu, the user is presented with a window (seen in figure 5.2) through which he or she can modify the following settings:

Minimum cluster size Clusters may have a minimum size. All groups of connectors smaller than this minimum are seperated into single connectors, which are connected directly to the system.

Maximum cluster distance Specify the maximum distance between connectors within a cluster.

Avoid bearing walls The user may want to have the solver try to avoid bearing walls, both in the creation of clusters as well as while finding a solution.

When the *Generate Solution* button is clicked, the user is able to cycle through the clusters. For each cluster a number of solutions is available, using different cluster connection techniques, as discussed in 3.2.1. Since clusters can also be connected to each other, different solutions may appear for each ordering of clusters. When all clusters are accepted, the ducts are automatically placed and connected, resulting in a complete air system.

The user is also able to connect new vents to an existing system, by selecting the vents and clicking *Connect Selection To System*. The prototype will first ask the user to select a duct to connect it to, the rest of the process is equal to the normal generation process.

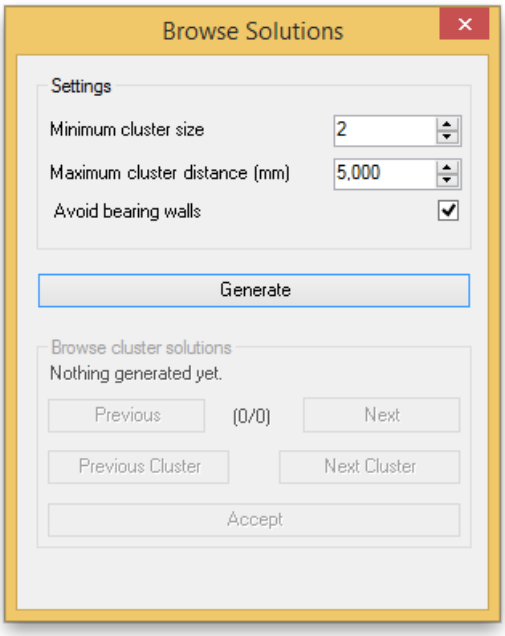


Figure 5.2: A user can modify these settings to influence the generation process.

5.2. PROTOTYPE RESULTS

In chapter 2 some demands, constraints and requirements were discussed. This section will reflect on these points and show how the prototype performs in different scenarios.

5.2.1. USER INPUT

The initial goal of the prototype was to enable the user to influence the generation process. In the final version a user should decide the placement of the main trunk. He or she is also able to change the order in which clusters are connected, creating different results. Finally, the user can choose the appropriate solution for each cluster.

Since the process of choosing the *"right"* solution for a cluster depends on the expertise of the user, the algorithm only eliminates those solutions that are not feasible, such as solutions that breach walls or use absurd connections. This leads to many customization possibilities, while still maintaining some basic rules and constraints.

5.2.2. BUILDING SIZE

Because the prototype divides a building into clusters, the algorithm does not drastically increase in diffulty when a building gets larger. However, it does have an effect on the performance, mainly because there are many more ducts that need to be connected in the final step of the generation process.

When the density of the connectors gets higher, the algorithm does seem to have a hard time finding out where to connect clusters, which sometimes results in strange situations, such as the one pictured in figure 5.3.

In general, buildings in real-life are quite consistent and somewhat symmetrical. In these cases, the prototype performs quite well. An example can be seen in figure 5.4.

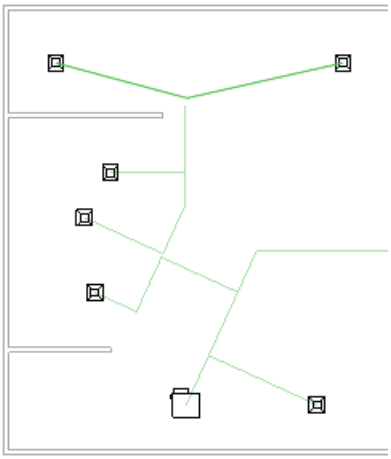


Figure 5.3: Solution with high connector density

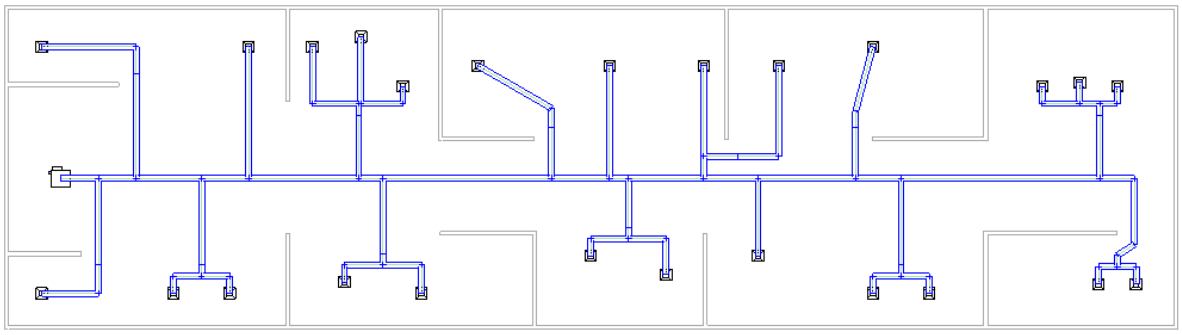


Figure 5.4: A solution of a building of moderate size

6

PROCESS

This chapter describes the process of the project. Development techniques and tools are discussed in sections 6.1, 6.2 and 6.3, as well as the feedback on our work by the Software Improvement Group which can be found in section 6.4.

6.1. SCRUM

From earlier projects as well as courses within the Bachelor Computer Science program we know that agile development methods are very efficient when working on small projects with a small team. By using small development cycles called *sprints* a team is able to focus on parts of the project without interference, resulting in a clean workflow and efficient use of time. This method is also used by Stabiplan B.V. in their development process.

A light variant of Scrum [6] has been used to manage the project, which we divided into three sprints, each one consisting of either two or three weeks. At the beginning of each sprint, we filled the backlog with everything that was needed for a new prototype which we would present to our client on the last day of that sprint.

REFLECTION

Although Scrum is a great way to manage small projects in an efficient way, it did not yield those results in our situation. This probably has to do a lot with the fact that the basis of our project is to provide a *proof-of-concept*, which implies much of our work is born *on-the-fly*, since we couldn't accurately predict the results of our approaches. This made time estimation and task assignments a lot harder. Although we tried to maintain a correct backlog and tracked all spent time accordingly, we believe that Scrum was not the way to go in our case.

6.2. TEST-DRIVEN DEVELOPMENT

Stabiplan B.V. immediately suggested that our code should be well-tested, so we decided to use test-driven development (TDD). This strategy requires writing automated tests prior to developing functional code in small, rapid iterations [7]. The biggest upside to TDD is of course a well-tested codebase, but it also leads to better thought-out code. Basically, writing tests before implementing anything forces a developer to analyze the expected behaviour of a program. This eliminates duplicate code and leads to "cleaner" programming.

REFLECTION

Our project can be divided into two main components: the interaction with Revit and the algorithm itself. Early research into the subject of testing the interaction with Revit showed that unit testing is not possible. Stabiplan B.V. confirmed this and recommended to continue manual testing for Revit interaction. It was

not a big problem that we were unable to test the Revit interaction as it mostly consisted of extracting data, altering and injecting (new) data back into the model. Debugging and visually confirming the model were good enough.

The algorithm side of the project is rigorously tested with Visual Studio's built-in Unit Testing framework. Almost all functionality of the algorithm side is covered by unit tests which leads to a unit test coverage over 95%. Unfortunately, not all of these tests are born out of TDD, simply because, again, the results of our approaches could not be easily predicted. A lot of finetuning was needed to deal with inaccuracies and other difficulties we encountered during the project.

6.3. DEVELOPMENT RESOURCES

6.3.1. YOUTRACK

Instead of a physical board to maintain the progress of the sprints, we used an online application called YouTrack. This application gave us the possibilities to divide issues in distinct types (bug, feature, task, research, enhancement, design, administrative types), in distinct subsystems (algorithm, Revit or documentation) and to determine which issue should be fixed in which sprint. Of course, it is also possible to indicate if a certain issue is still open, in progress or completed.

YouTrack has also been used for time tracking. A time estimate can be added to each task on the board, and we were able to track the time we spent on a task. This gives a good indication of the *burndown* [6] of our sprints.

Figure 6.1: An example screenshot of YouTrack at the algorithm board in sprint 3

6.3.2. C#

Choosing which language to use was easy. Because the API of Revit is in C# and .NET, it will be suitable for our project to also use these languages. The developers at Stabiplan B.V. also work with C# and .NET, so they were able to assist us whenever we encountered any difficulties concerning C# or Visual Studio.

6.3.3. GITHUB

Immediately after the project set off, we agreed to use version control for our code, and quickly settled for GitHub. GitHub is a hosting facility for the open source *Git* software. Git is a revision control system made for software development. One of the advantages of Git is the use of branching and merging. For large changes to the code, using a seperate branch instead of the "main" development branch results in less code breakage. After a change is complete and fully tested, it can be merged back to the main branch.

GitHub also supports communication with external applications through so-called “web hooks”. This meant we could link our backlog to GitHub, enabling us to have our commits linked to issues in YouTrack. This provided an easy insight to who did what at certain times.

6.4. Software Improvement Group

Part of the project requirements was the analysis of our code by Software Improvement Group B.V. (SIG). SIG performs static code review without actually executing anything. They assess the quality characteristic of maintainability by product properties such as volume, duplication, unit complexity, unit size, unit interfacing, module coupling, component balance and component independence [8].

6.4.1. Feedback

The first feedback from SIG was received June 20th and can be found in appendix B. They stated that our code received 3 stars on their 5-star scale. Our code could be improved on the following aspects:

- *Unit Size*: The size of the source code units in terms of the number of their source code lines [8]. At the time of the feedback, our solvers consisted of 600 to 1000 lines of source code each, which is too high for a single unit.
- *Unit Complexity*: The degree of complexity in the smallest executable parts of source code, such as methods or functions [8]. The complex operations performed by our solvers were all contained in big functions, which results in very high unit complexity.
- *Module Coupling*: The coupling between modules in terms of the number of incoming dependencies for the modules of the source code [8]. High coupling often results in unwanted behaviour since small changes may affect functionality in large parts of code.Both the `AClusterSolver` and the `AConnectionSolver` contained a lot of small utility methods that they both used heavily, which made them highly dependent on each other.

They also noted the presence of our test code, which was promising. The advice was to maintain the test code during the development process whenever new functionality were to be added.

6.4.2. Improvements

The main flaw in our code was the Unit Size and Complexity of the two solver classes `AConnectionSolver` and `AClusterSolver`. During the final weeks, we refactored these classes to reduce these flaws.

Basic utility methods such as `Trim` have been moved to a seperate centralized utility class `ASolverUtil`. This of course has an impact on Unit Size, but also decreases the Module Coupling. Together with small rewrites of the clustering process, this reduced the Unit size of the solvers to about 400 to 600 lines of source code each.

Complex methods such as `ConnectAroundWall` have been split up into parts to further reduce Unit Complexity and to seperate different functionalities within those methods.

7

Conclusion

In this quarter, we have explored the possibilities for an algorithm which creates a duct design for air ventilation. We have implemented such an algorithm with certain features. However, because of the limited amount of time, some features could not be implemented in this quarter.

The algorithm we have created supports the creation of clustering, a technique which is often used in the design of a duct system. The cluster types supported are a single-sided fish bone (knife), a two-sided fish bone and a fork construction. Furthermore, the ducts are adjusted to have angles which are regularly used in the industry. Another feature is the avoidance of bearing walls and the possibility to estimate the costs of a system. And, of course, the algorithm is able to communicate with Revit and to draw its solution into a Revit file.

We have experienced that many three dimensional operations are a lot harder than its two dimensional variant. We have implemented certain features in three dimensional space, but many calculations are still made in two dimensions. Furthermore, we were not able to implement capacity calculations onto our algorithm. We will discuss these features in more detail in chapter 8.

Considering the progress made in just over two months and the results generated by the algorithm, it seems worthwhile to continue researching this technology. Even if the current algorithm is not viable for commercial use yet, the sensible solutions it already creates imply the possibility of more advanced and complex designs. The challenges in reaching this solution are discussed in the next chapter, none of which are, in our opinion, impossible to overcome.

We will conclude this report by stating that automating the design of duct networks is hard but certainly not impossible.

8

RECOMMENDATIONS

Since our prototype is considered a proof-of-concept, there are a lot of ways to improve it. This chapter will recommend some things that we believe are essential for a viable product.

8.1. BETTER USE OF THREE-DIMENSIONAL SPACE

Although basic support for 3D space is available in the models and calculations, the prototype still creates layouts with a fixed height. Using the third dimension may not only create more viable solutions, but also enables the support for multiple systems.

8.2. USER-DEFINEABLE OPTIONS

Since MEP design relies heavily on user preference and expertise, a computer-generated solution will rarely be perfect. A user should therefore be able to influence the decisions the prototype makes, such as the parameters used in cluster creation, the cluster's connection preference or certain offsets used. The algorithmic cost of certain connections such as T-junctions should be override-able by the user, as well as the materials it uses.

Certain options are already available, such as the minimum cluster distance or whether or not to avoid bearing walls, but all the parameters that are currently hard-coded into the prototype could (and probably should) be set by the user through a preference pane.

8.3. MAIN TRUNK

The main trunk is laid out by a simple MST algorithm over the user points and the source. As it was desirable to give the user freedom for the creation of the main trunk, features like wall avoidance and angle adjustments are no part of the main trunk calculation.

The downside of this implementation is that the user might need many points for the desired main trunk as a MST. For example, if the desired path is from a source to a first user point to a second user point, but the third point is closer to the source than the second point, the created main trunk would be different from the desired main trunk and another user point is needed to enforce the right order.

Possible solutions could be to let the user define the main trunk entirely or to extend the algorithm that creates the main trunk.

8.4. WALL INTERSECTIONS

Currently, all bearing walls will never be crossed by ducts (except if this is forced by the user with user points) and non bearing walls will be ignored as they are not present and not counted in the cost calculations. However, this is far from ideal: with some restriction and costs, bearing walls may be intersected and of course the intersection of non bearing walls also adds to costs. It is therefore recommended to extend the system to process possible wall intersections more precisely.

8.5. CAPACITY

Factors like *flow capacity* and component cost are very important in the creation of a layout. However, this information is not available to the prototype using just the Revit API. Stabiplan has a database containing components produced by several manufacturers and all of the component's information. Since StabiBASE is able to provide such information, combining our code with StabiBASE is key to optimizing the solutions for capacity provided by the prototype.

8.6. REAL LIFE SITUATION TESTING

The approaches used by the algorithm are solely based on expertise of a handful of engineers at Stabiplan B.V. Since the concepts of MEP design rely so much on the user, the prototype should be tested by several independent engineers or architects. This way, different approaches may be incorporated into the algorithm, resulting in a more versatile product.

A

DETAILED DESCRIPTION OF GENERATING A SOLUTION

This document is provided to serve as a guide for the code involved in generating a solution using the StabiDucts algorithm. Not every step, especially steps involving selecting a parent or child, are intuitive and this document attempts to explain these non-intuitive steps, offering help for anyone trying to thoroughly understand the code.

It is assumed that the report associated with the program is read, so a basic understanding of the classes involved is expected.

The explanation of the `External Commands` is skipped, since this is considered relatively straight-forward and only requires basic Revit API knowledge to understand.

INITIALIZATION

The `BrowseForm` created in the `External Commands` is where the process is started. When the `Generate` button is pressed, the main trunk is generated and added to the `mainSolver`, which is the `AConnectionSolver` responsible for the complete solution. This main trunk is also added to `acceptedModelLines`, a collection of lines defining solutions accepted so far. Next, the clusters are calculated and the first cluster's solutions are generated by calling `NextClusterSolution` (`tempSolution` will be empty, so new solutions are generated).

CREATING SOLUTIONS FOR A CLUSTER

`BrowseForm.CreateTempSolutions()` is responsible for generating the solutions of the currently selected cluster. It first initializes an `AClusterSolver` with the main building source as main trunk root, the cluster `AConnectors` and the cluster type `Preference`.

The first thing to happen is that the algorithm creates an `AConnectionSolver` using the main building walls and the `weightedMiddle` of the current cluster. It then calculates a route to anywhere in the main solutions by either going around walls if necessary, or directly. Both can return multiple solutions: the former might find more than one path around walls and the latter might return solutions connecting directly to the nearest point in existing solution (including previously accepted clusters) or connecting directly to the main trunk, if possible.
There is an important thing to know about this: it is used to know at which side of the cluster the clustertrunk should be located. This should be near the *parent* of `weightedMiddle`, since this is most likely the place where the cluster root should be connected to.

The next part will generate the cluster solution. `solver.solver`'s root is set to the newly found main solution segment, which means it now takes that section as reference for the orientation of the cluster trunk. It will

then solve the cluster, using the path found by going around walls (or directly) as a reference for the cluster trunk. A more detailed explanation of how this works will be given later on in this document.

The clustersolver returns an `AConnector`, which is the root of the cluster solution. The lowest `AConnector` in the path to the main solution is then found and the cluster solution is added as a child, effectively linking the two solutions. After this, the parents of the vents are set back to `null`. The first solution is then generated in Revit.

BROWSING THROUGH SOLUTIONS

All solutions generated for a cluster are stored in `tempSolutions`, a `List` of `Tuple<>` objects, containing an `AConnector` and a line segment represented by `Tuple<AConnector, AConnector>`. Here, the first `AConnector` is the root of the solution, lying closest to the segment of the existing main solutions to connect to and the previously mentioned segment. Browsing through the solutions simply generates the lines from this root downward. Seeing how the root is the `AConnector` lying *closest* to the segment, but not *on* the segment, the solution is almost, but just not, connected to the main solution. Every time a next solution is generated, the vents in the cluster are connected to the solution.

ACCEPTING CLUSTER SOLUTION

When a cluster solution is accepted, everything has to be connected. The vents are connected to the selected solution using a new temporary `AConnectionSolver`. Then, the clusterroot is searched for. If the root is already on the main solution, a new nearest point is calculated to confirm it is still the closest point to the root, since the root might no longer be at the same location it was when first calculating the nearest point in solution. Afterwards, the whole clustertree is connected to the main trunk.

The solution is now added to the `acceptedModelLines` and is drawn.

EDGE CASES

Certain edge cases encountered are listed below.

- Single vents
  Single vents are simply solved by using the vent as root for a solution.
- No walls
  Surprisingly, the case where no walls were present to obstruct a path to the main trunk would fail. This was due to parent/child problems, since no path to the main solution was generated. This is solved in `ASolver` with the method `CreateDirectSolution`.

GENERATING CLUSTER SOLUTION

At the moment, there are three types of cluster solutions: `Fork`, `Knife` and `Fishbone`. Graphic examples of these solutions can be found in chapter 3 of the main report. The essence of solving each type is broadly the same: calculate a cluster trunk, connect the vents to it, trim the cluster trunk, find the correct root location and create a small extension outward from the solution (to avoid strange three-way connections). However, each step is slightly different.

FORK

The characteristics of a fork solution are that the cluster trunk is parallel to the reference line segment, the root is in or close to the center of the cluster trunk and all vents are on one side of the cluster trunk.

The cluster trunk is positioned at a set minimum distance from every vent in the cluster, parallel to the reference segment. At first, the cluster trunk is about two times the size of the cross-section of the cluster, just to

be sure it is large enough. The vents are connected to it using an `AConnectionSolver`, connecting perpendicular to it (since that is the shortest solution for the given vent). The cluster trunk is then trimmed to stop at the most outwardly vents to both sides and the root is set to the location of the weighted middle, since it is certain this will avoid a four-way connector. A small extension is then added at this location, facing away from the cluster, perpendicular to the cluster trunk. This is for the path to the main solution to connect to and avoids oddly angled three- or four-way connections on the trunk.

KNIFE

The characteristics of a knife solution are that the cluster trunk is perpendicular to the reference line segment, the root is at the end of the cluster trunk and all vents are on one side of the cluster trunk.

The cluster trunk is positioned at the same minimum distance from every vent as the fork solution, perpendicular to the reference segment. When looking at the cluster, if a possible fork trunk would be at the 'front' of the cluster, the knife trunk would be at the 'side'. The connecting of vents, size of the trunk and trimming it is the same as with the fork solution. The root of the knife trunk is located at the end of the trunk closest to the path to the main solution. An extension is created along the line of the cluster trunk for the same reason as with the fork solution.

FISHBONE

The characteristics of a fishbone solution are that the cluster trunk is perpendicular to the reference line segment, the root is at the end of the cluster trunk and it has at least two vents that are at different sides of the cluster trunk.

The cluster trunk is positioned straight across the weighted middle. Its connecting and trimming is the same as with the other solutions. It also checks if it has vents at both sides by picking a random vent (or first) and checking, for each other vent, if the line between it and the other vent crosses the cluster trunk. The extension is the same as with the knife solution.

Each solution type checks if a minimum percentage of connectors is connected to the solution (currently hard-coded to 0.5, could be implemented as an 'advanced option'). If fewer vents are connected, the method returns `null` for that solution.

B

Feedback Software Improvement Group

De code van het systeem scoort net 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size, Unit Complexity en Module Coupling.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de `connectAroundWall`-methode binnen `AConnectionSolver`, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld "*//Go through all points to check to see if possible points*" en "*//After having received solution, check distance*" zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methoden ook naar voren als de langste methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Voor Module Coupling wordt er gekeken naar het percentage van de code wat relatief vaak wordt aangeroepen. Normaal gesproken zorgt code die vaak aangeroepen wordt voor een minder stabiel systeem omdat veranderingen binnen dit type code kan leiden tot aanpassingen op veel verschillende plaatsen. Ook wijst deze meting over het algemeen naar classen waarin relatief veel functionaliteit in geimplementeerd is. Wat binnen dit project opvalt is dat de classen `AConnectionSolver` en `AClusterSolver` samen 1/3 van de code van het systeem bevatten, deze classen lijken dan ook teveel verantwoordelijkheid te hebben. Een indicatie hiervan zijn de statische methoden die in deze classen zijn geïmplementeerd, dit lijken voornamelijk utility-methoden te zijn. Het is aan te raden om deze classen kritisch te bekijken en waar mogelijk op te splitsen.

Over het algemeen scoort de code gemiddeld, hopelijk lukt het om dit niveau te behouden of te verbeteren tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

C

ORIGINAL PROJECT DESCRIPTION

The following description is the original project description as found on BepSys.

PROJECT DESCRIPTION

A major part of designing the technical installations of a building is the design and coordination of the different distribution systems. In a building project, a lot of work could be saved when the design of these systems can be done more automated. One of the first steps in the design is the routing of the pipes/ducts from the start point of the system (e.g. an air handling unit or a boiler) to the endpoints of the system (e.g. air terminals or radiators). For this, the architectural model of the building provides the most important input: where are the rooms, the corridors, what's the available space above the ceiling, etc. Apart from that, there are several practical rules that make one solution more optimal over others. The goal is to design intelligent routing algorithms giving options to the designer for the routing of the systems. The algorithm should be able to generate the routing with minimal user input. On the other hand, it must be possible for the designer to provide the algorithm with a set of (parametric) design rules and prioritize these rules, so he can compare different routing alternatives. It should be investigated which design rules are useful in building projects. The routing algorithm also includes the coordination of the different systems so collisions between systems are prevented. An extension of the algorithm would be to involve the schematical designs (the installation concept) in the routing algorithm. Then we do not only generate the piping layout, but also the devices and equipment of the distribution system is automatically generated, according to the schematic drawings. Questions about this project? Don't hesitate, for more information contact ir. Rene Qualm at r.qualm@stabiplan.nl or (0172) 65 02 65.

COMPANY DESCRIPTION

Since 1990 Stabiplan is leading in development and sales of Revit- and CAD-based design software for MEP engineering in Europe. Our multiplatform product Stabicad is adapted to the local needs and wishes of engineers: it is used by more than 3,500 clients and a daily user group of over 8,500 engineers. Stabiplan has offices in Belgium, France, The Netherlands and Romania. Starting the first day at Stabiplan you will be part of our international development team that is working with Scrums. Together with product owners and interaction designers the development team delivers high quality monthly releases. The results of today's sprint are put in practice in next month's software version. So you really make the difference in the life of our customers.

D

Action Plan

Project assignment

This chapter will describe the assignment of the project. First of all, the first section describes the current situation of the client that lead to the project. The second section explains the goal that the client wants to reach, after which the third section defines the assignment itself. The products that will be created are described in the fourth section. The restrictions and demands for the final product are stated in the fifth second and finally, the last section defines the conditions for the project.

Project Environment

Stabiplan B.V. is a software development company focused on creating tools for the so called MEP-market, where MEP stands for Mechanical, Engineering and Plumbing. The tool they develop is called Stabicad.

Companies active in the MEP-market design, among others, the layout of ducts, vents and service pipes. The development of a building starts with a client and an architect. The architect creates a design focussing mainly on aesthetics. Civil engineers then create a model suitable for construction. Revit is a program built to create this type of architectural and engineering work. Stabicad is designed on top of Revit and uses Revit models to create the layout of the ducts and pipes.

At the moment, this layout is manually designed and drawn. This makes it a costly and time consuming task, which may well lead to layouts that are suboptimal.

Project Goal

The goal for this project is to help automate the task of designing the layout of air vents, water and gas pipes, etcetera. The amount of user input involved in this automation can vary. If possible, optimisation will be implemented as well.

Assignment specification

The group will create a program fulfilling the aforementioned goal. This will start with researching both the abstract problem and the capabilities of Revit. The next phase will consist of the actual production of a program.

Final products

The final product will be a program that meets all the requirements and demands as specified below.

DEMANDS AND CONSTRAINTS

The main demands for the final product will be specified following the MOSCOW model [9].

- Must-have
  - Ability to create valid layout:
    - No illegal collisions
    - All given in- and outputs connected
    - Meets flow demands (especially air ducts)
  - Revit integration
  - Give multiple solutions for end-user to pick
  - Proper documentation
  - Pass tests based on actual designs
- Should-have
  - Use Stabicads MEPContent
  - Create (approximately) optimal layouts
  - Users can input constraints
- Could-have
  - Stabicad integration
- Would-have
  -

CONDITIONS

This section describes conditions that have to be met for the program to work. This is to create a limited input space so as not to require the program to work properly with any situation. It is not realistic to quantize these at the start of the program, since nothing is known about the increase in complexity that comes with these conditions.

- Amount of duct/pipe/etc. entries is not too large
- Building is not too complex
- All required user input is given
- A proper layout is possible
- A proper (type of) layout can be verified using Revit or Stabicad

APPROACH

This chapter describes how we will approach this project. In the first section, an explanation of the Scrum process is given. In the Techniques section, there is a summary of the software tools needed for the project. At the end of this chapter, there is a planning with the main dates of the project.

METHODS

We will make use of Scrum cycles [10] in this project. As the project time is relatively small, we will use a short Scrum sprint time of 2 weeks, except for the first sprint where we'll have a 3 week Scrum sprint. Our implementation of Scrum will be standard. That is, we will have a prioritized product backlog with tasks to be done. In the daily Scrum we will discuss shortly what everyone has done the previous day and what they

will do today. At the end of each sprint we will aim to have a working piece of software and review what we have achieved in the sprint. How time will be divided over the team members, will be discussed in the Scrum sessions (as well in the daily sessions as in the sprint sessions).

Techniques

Our project consists of building an extension for the software packet called Revit. To create this extension we will use C#, Visual Studio and the .NET Framework. This was advised by Stabiplan since existing software is also made with these languages. The Stabicad software will be provisionally disregarded.

We will apply several types of testing for our code. We have planned to at least implement unit tests, regression tests and integration tests. Also, we will validate our software, i.e. did we build the right software as stated in chapter D of this document?

Planning

The research phase will end at the end of week 18 (4.2). In the next phase we will have three Scrum sessions of 3, 2 and 2 weeks respectively. At the end of each sprint, it is our goal to have finished a working piece of software as well as an interim report. The last fase consists of final submissions of code, final report and of course the final presentation. The following list will summarize our planning.

- 2014, May 2 - End of research phase.
- 2014, May 5 - Start of main phase; first Scrum sprint session.
- 2014, May 26 - Second Scrum sprint session.
- 2014, June 9 - Third Scrum sprint session and submission of code to SIG.
- 2014, June 18 - Resubmission of the code to SIG if needed.
- 2014, June 26 - Deadline for the submission of the final report.
- 2014, July 3, 14:00 - Final presentation.

Project Setup

There are multiple aspects to consider when running a project, ranging from team-member allocation to what the office space looks like. This chapter will describe these aspects, starting with organisation in section D. Section D will explain what is expected from each team-member whereas D, D, D and D cover documentation, finances, communication toward the employer and the office resources respectively.

Organisation

The project can be divided into two main phases: two weeks of research followed by eight weeks of active development.

Research

The team is divided into two-man cells during research:

- Simon and Wiebe, who will focus on researching possible solution approaches and existing solutions while defining the problem.
- Bryan and Sander, who will be looking into the software environment used by Stabiplan and researching the posibilities in the API of Revit. The focus lies with techniques that can realise the proposed solution in the given developing environment. It is possible to look into Stabicad, developed by Stabiplan, in case we need specific functionality.

DEVELOPMENT

The team will operate in a Scrum setting after researching and finding a feasible solution to implement. One person will be assigned to represent the team in the Scrum meetings. This task will be rotated through the team-members as it is a valuable learning experience. The team will be on site full-time. The Scrum setting introduces no further roles so it not possible to determine further tasks, as this is done on a daily basis.

TEAM-MEMBERS

The requirements towards each member of the team are to possess a set of skills and proper quality of the delivered work, and a certain time invested. Each member spends 40 hours a week as the norm given by the Delft University of Technology. It is possible to cover these hours separate from the team as long as the delivered work reflects the time invested.

NECESSARY SKILLS

The skills needed to properly complete the project are the following:

- .NET The software-package Stabicad and Revit are both written with a .NET API. Each group member should be able to read and use the APIs.
- C# is the programming language used to both interact with .NET and develop Stabicad. This will be the language in which the extension is developed.
- LaTeX for documentation purposes we will use LaTeX as our text processor.
- Git to store both code and documentation under version control either on a private Github. The employer is given access to this repository.

ADMINISTRATIVE PROCEDURES

A collection of documenting tools will be used to keep track of the progress during the project:

- Version control comments will be submitted during development and writing documents and it is important to comment properly to make it easier to recover earlier version and keep track of progress.
- Scrum will generate documentation about cycle progress and updates multiple times a week.
- Prototypes, milestones and important decisions are accompanied by a short report to make it easier to compile the the final report.
- Meeting minutes written during each meeting. Someone from the team will be assigned to summarise meetings regarding the project. This will ensure no loss of critical information such as decisions made. These minutes are digitalised and uploaded to the Github Wiki for ease of access.

FINANCING

The finances are laid out in the contract for internship. Intelectual property is quite likely to go to the company. We can't be sure yet until the final contract is presented to us.

COVERAGE

The Scrum meetings between our team and the employer will take place on a weekly basis. The meeting will be joined by both our employer and our monitor within the company. It is also possible to inform, or ask questions to, our monitor during the week.

RESOURCES

The resources we need might change going along the project, but for now a rough estimate. The estimate includes:

- VPN for when we have the need to work off-site
- Desktop machines to run Revit and our software
- A student license for RevitMEP and Visual studio C#

Quality assurance

To ensure that our product meets every demand, certain measures should be taken into account during the course of the project to ensure the quality of the product. The following sections will define these measures.

Functionality

Since our product is an extension to an existing software product, it should function accordingly. This implies that we maintain the same quality measures used at Stabiplan to ensure the product's functionality and reliability. The use of proper error handling and testing will make sure that the product will not contain any "undocumented features".

Maintainability

Because our product will (probably) be incorporated in Stabicad, our code should be highly maintainable. This includes extensive documentation as well as the use of well-known and proven design patterns and principles. We will create class diagrams and documentation both in the code as well as an external document. The code will be thoroughly tested by both unit and regression tests and we will adopt the code style defined by Stabiplan to ensure that our work may be continued by the Stabiplan engineers after this project is finished.

Efficiency

Since our product will have to execute mathematically difficult computations, efficiency is key in the quality of our code. To ensure code efficiency, again the use of design patterns and principles is included, as well as ensuring overall code consistency. The algorithms used in the solution have to be picked and analyzed carefully to ensure proper results.

E

RESEARCH REPORT

INTRODUCTION

The goal of this project is to implement an algorithm that optimizes the design of ducts. This section will briefly introduce the main concepts for this purpose.

A part of the design and construction process of buildings is to plan where ducts, cables and pipes have to be placed [1]. These tasks are often referred as mechanical, electrical, and plumbing design, or MEP design. It depends on MEP design where certain facilities are available, how the documentation is done (for future maintenance), how the facilities perform and of course how much the building costs. Therefore MEP design has a crucial role in the whole design and construction process of buildings.

MEP design is often performed with the use of Building Information Modelling (BIM). An instance of a BIM is a design where all objects contain detailed information and relationships are defined. Computer aided design (CAD) models generally do not need to contain detailed information about objects and is often a normal 2D or 3D drawing without objects. So a BIM can be seen as an extended version of a CAD model.

In the process of designing buildings, many contractors and other actors play a role. BIMs have the advantage that those actors can all work with the same design system. This way conflicts are less likely to happen.

There are multiple software packages for designing BIMs. Well known BIM software packages are Revit and ArchiCAD, while known CAD tools are AutoCAD and VectorWorks. In this project, we will work with the BIM design software called Revit, made by AutoDesk.

We will build an extension for Revit with the use of C# and .NET to optimize the design of ducts for heating, ventilation, and air conditioning (HVAC). In this research report, we will discuss what an optimal solution means in this problem, we will discuss problems similar to our problem and find algorithms that aim to solve this problem. Furthermore, we will take a look at Revit and C#.

PROBLEM

This section describes the problem and will give a more formal, mathematical view on the problem.

INPUT AND OUTPUT

The system uses the following input:

- A designed building in Revit format. The location(s) of the air sources are already known.
- The locations of the air vents (the places that should be connected).
- The locations of points where the user prefers to have the main line.

The output will be one or more consistent solutions if a consistent solution exists. A consistent solution should satisfy the following conditions:

- All chosen points of the input are connected to an air exit.
- No collisions are created by the solution. In some situations, collisions may be fixed by making an opening, but this is not the situation for bearing walls.
- All vents are able to perform at their full capacity at the same time.

FORMALIZED

More formalized and slightly simplified, we have the following input, with points $p = \langle x, y, z \rangle$ and capacities c.

- A set of air sources: $S = \{s_1, \ldots, s_n\}$ where $s_i = \langle p_i, c_i \rangle$ and $n \geq 1$.
- A set of vents: $V = \{v_1, \ldots, v_m\}$ where $v_j = \langle p_j, c_j \rangle$ and $m \geq 1$.
- Two sets of walls (and floors) where a wall is defined as $w = \langle p_1, p_2, p_3, p_4 \rangle$:
  - A set of bearing walls: $B = \{w_1, \ldots, w_q\}$ where $q \geq 1$ which cannot be punctured.
  - A set of non-bearing walls $N = \{w_1, \ldots, w_r\}$ where $r \geq 1$, which may be punctured at a certain cost.
- A set of preffered main line points $P = \{p_1, \ldots, p_t\}$ which defines the preference of the user for the main line.

The output will be a set of lines l through the building: $L = \{l_1, \ldots, l_k\}$ where $l_i = \langle p_{i,1}, p_{i,2}, c_i \rangle$ with $p_{i,1}$ and $p_{i,2}$ as start and end point and d_i as diameter of l_i.

At least, the output has to satisfy:

- For all vents $v \in V$ there exists a connected path $\{l_1, \ldots, l_i\} \in L$ where $p_{1,1} = v_p$, $p_{i,2} = s_p$ and $s \in S$.
- For no line $l \in L$, there exists an intersection with the wall w for all $w \in B$.
- There exists a residual network of V, S and L, L_{res}, where all $c \in l \in L_{res} \geq 0$ (where direction changes in the residual network are denoted as a negative number).

OPTIMIZED SOLUTION(S)

Besides the fact that the system should be feasable, we also aim to have a optimized solution. It is hard to define an optimal solution because it depends on many variables (material costs, maintenance costs and much more). Therefore, we shall try to create multiple solutions, where the user is able to decide which one the user prefers. To create these solutions, we can take the following factors in consideration.

PREFERED MAIN LINES

In the problem of creating a network for HVAC, the most optimal algorithmic solution in sense of costs, is often not the solution which is the desired solution of the user. In many cases, users have an idea about how main lines should be build. Therefore, we will ask user input about the location of the main line.

MATERIAL COSTS

The price of the ducts are defined by Taecheol Kim [11] as $E_{duct} = S_d \pi DL$ for round ducts and $E_{duct} = 2S_d(H+W)L$ for rectangular ducts, where S_d is the price per m^2, D is the duct diameter, L is the duct length, H is the duct height and W is the duct width.

Also the cost of the fans will play a role. However, the choice of the right fan does not fall in the scope of this project.

Energy costs

The energy costs of a HVAC-system are an important part of the total costs. The energy costs of the system depends on the air flow and air pressure, the system operating time, the energy costs and efficiency of the fan.

A larger amount of ducts decreases the air pressure in ducts far away from the fan source [12]. To maintain the same performance, either an extra fan should be added or the power in the original fan should be increased. Thus, a larger amount of ducts will also increase the energy costs.

Construction and maintenance

When designing ducts, we also have to deal with the issue of construction and maintenance. We couldn't find much research on this issue, but it seems there is a preference to mount ducts as much as possible in aisles with simple divestitures to adjacent rooms for reasons of construction and maintanance. When ducts have to cross many rooms, construction and maintenance takes more time.

Diameter of ducts and sound control

The amount of air that passes through a duct in a certain time is defined by the diameter of the duct. Revit is able to determine an appropiate diameter for ducts themselves, given a network of ducts. However, the network can still be optimized to contain smaller, and therefore cheaper, ducts. For example, when a building contains more than one air source, it may be less expensive to divide the ducts between distinct air sources.

Due to sound control, different recommended maximum air flow speeds are applicable for different rooms [13]. Within an occupied office room, there is a relatively small maximum air flow in comparison with aisles and other spaces. The recommended maximum speed in an occupied room is about 30% lower. To achieve the same air flow within such an occupied room and apply the recommended maximum air flow speed, ducts have to be larger and thus more expensive. Therefore, it is wise to avoid these rooms for main duct lines.

Conclusion

To conclude, material costs, energy costs, construction/maintance costs and costs due to sound control are all factors which determine what an optimal solution may be. Material costs and energy costs have a similar influence: a higher amount of ducts leads to higher material and energy costs. However, for the purpose of construction and maintenance and sound control, it may be better to choose a solution where ducts are mounted in aisles and avoid rooms as much as possible. Therefore, we will aim to give users more than one solution to choose of.

Furthermore, users often have there own idea about the location of the main line, our implementation will ask user input for the location of the main line.

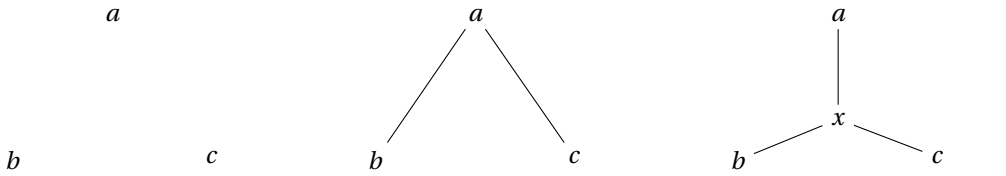
Known similar problems

This section will list a few known problems similar to the problem as stated in the second section, which might be used to create a solution to the main problem.

Steiner Trees

The Steiner Tree problem is an old problem that gained attention from the industry when large networks, like telephone networks, were sought to be made shorter [14]. A Steiner tree is a network connecting all points in a plane [3]. Two common planes in which Steiner trees are used are the Euclidean and rectilinear planes.

The difference between Steiner trees and the common minimum spanning tree is the addition of Steiner points: points created in order to get a smaller total network length. Figure E.1 shows how this follows.



(a) Simple set of graphs (b) Example MST (c) Shortest network

Figure E.1: Possible spanning solutions

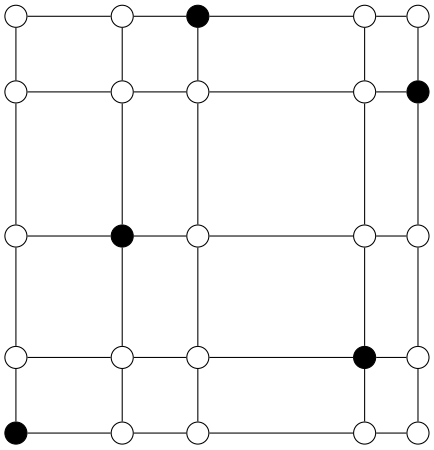


Figure E.2: Hanan grid for Rectilinear Steiner Tree

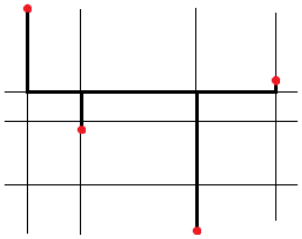


Figure E.3: Single-trunk Steiner Tree

Sadly, the Steiner Tree Problem is NP-Complete [15]. This means that probably no solution exists that solves the problem in less than exponential time. There are, however, approximation algorithms.

RECTILINEAR MINIMUM STEINER TREES

A more restricted type of Steiner trees are the rectilinear Steiner trees. These trees consist of only straight angles. Figure E.2 shows the Hanan grid that can be constructed around the nodes in a graph. Hanan [4] has proven that the search for a Rectilinear Minimum Steiner Tree can be limited to such a grid.

Finding an RMST is still an NP-Complete problem [16] and can thus only be approximated. A special case of the RMST problem is the single-trunk Steiner tree. An example of this is given in figure E.3. In this problem, only one main "trunk" exists from which the paths towards the nodes stem. This problem can be solved in linear time, but will not always yield an optimal solution.

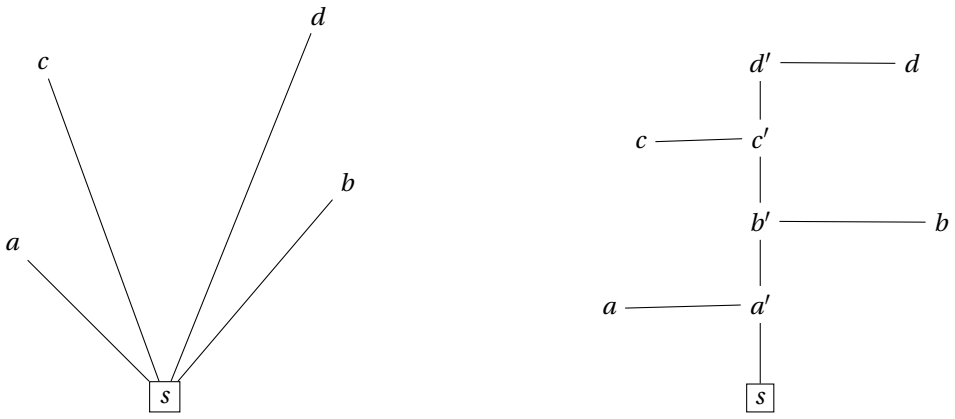
RMST WITH OBSTRUCTIONS IN PRACTICE

Alpert et al. [17] have used Rectangular Minimum Steiner Trees in an algorithm that creates a layout on a chip. Routes on this chip can use buffers, which increase performance, but only outside of reserved areas, where a route can be made, but no buffers placed. There are situations where it might thus be advantageous to create a route that circumvents this area in order to be able to place a buffer.

This research paper will not explain in detail how the algorithm works, since it is not clear as of yet if a similar method will be used. It is, however, a solution that seems to be easily adaptable to our problem.

STEINER TREE CONCLUSION

Depending on the shape and layout of a building, these rectilinear trees might be useful enough to construct an initial layout of the plumbing, etc. User input, as has been suggested by Stabiplan, can easily be implemented in the form of enabling users to add Steiner points, through which a path has to be found. However, curved buildings or corridors might render this solution useless.



(a) Direct connections

(b) Tree connection

Figure E.4: Examples of routings

SATISFIABILITY PROBLEM

A well known problem in the world of algorithms, the SATISFIABILITY problem (SAT) is the first problem to be proven as being NP-Complete, which means that it is probably impossible to solve in linear time. Countless other problems used the NP-Completeness of SAT to derive proofs of other NP-Complete problems. DUCT can also, with some assumptions, be reduced to SAT. To get an indication of what a reduction from SAT to DUCT would look like, a few examples are given. Since it is not likely SAT will be used in itself, no resources have been spent on creating a full reduction.

Figure E.4 gives two layout possibilities for a given graph. These situations can be described in SAT format. SATISFIABILITY has a number of literals l_i and a number of clauses c_j filled with the (negation of the) given literals. The question is whether a combination of literals exists that satisfies all given clauses. Note that the literals within clauses are disjunct, and the clauses are conjunct.

Both situations given in figure E.4 at least have the following set of literals:

$$L = \{a, b, c, d, \}$$

The situation given in figure E.4a has the following clauses:

$$C_1 = \{s, a\} \cap \{s, b\} \cap \{s, c\} \cap \{s, d\}$$

The situation given in figure E.4b has the following clauses:

$$C_2 = \{s, a', b', c', d'\} \cap \{a', a\} \cap \{b', b\} \cap \{c', c\} \cap \{d', c\}$$

It should be noted that, in order to avoid drawing direct lines from s to all other points, new points *have* to be added. This is very similar to the Steiner Tree problem, which is a problem directly reducible to SAT.

SAT CONCLUSION

In this context, SAT might be too abstract to use. Calculation of path cost is not inherently present in the SAT problem, which would result in a back and forth checking whether a path is feasible and not too costly. Moreover, when using a main pipe from which branches reach to the outputs, constructions using SAT start to resemble Steiner trees very closely, making Steiner trees a more advantageous starting point.

ALGORITHM BY BRAHME ET AL.

Brahme et al. [18] have designed, more as an example for their main point than an actual study, an algorithm that generates a duct layout to connect all air vents in a building. Their main argument in the paper is that

building designers should look at items such as heating and ventilation earlier in the design process, but this is not relevant.

The algorithm uses a couple of definitions:

- Zone: Group of architectural spaces in a building which are controlled by the same controller.
- Cluster: A region in a zone encompassing all the terminals located within a certain range.
- Branch: A branch is a duct or pipe section that allows fluid flow from one point to another.
- Secondary system: Secondary systems serve one or more sections of the building. They consist of prime movers (fan, pump), and heat and mass transfer components (coils, humidifiers).
- Start node: Building node at the location of the vertical shaft on floor.
- System node: Building node at the location of the secondary system.
- Plenum: Collection of building nodes through which ducts or pipes can pass.

Figure E.5 shows the step by step procedure used by Brahme et al. The main function of their algorithm is to divide the building into sub-parts such as zones and clusters. Then the algorithm creates a distribution network for each of the sub-parts by creating a path based on the cluster of end nodes (e.g. air vents at the lowest level) along the longest axis of the given cluster. It then creates a distribution node somewhere along this path. These distribution nodes are then used a level higher to connect the sub-parts. This continues until everything has been covered.

No specific methods have been given, other than that a shortest path algorithm is used to connect the nodes, presumably forcing a route along the calculated "mid-path" that goes through the cluster. Based on this paper, some more research into cluster theory is started.

CLUSTERING

When enticed by the previously explained paper to take a look at clustering, an idea occurred.

The vents in a floor/room/etc. could be clustered together, either by being designated to a room or other type of area layout (i.e. by the user) or by using an algorithm. Using a linear regression technique, a line with the minimal total distance to each node could then be found and used for the duct, placing a distribution node there. This could be done recursively to include all levels (rooms, zones, floors, etc.).

CLUSTERING CONCLUSION

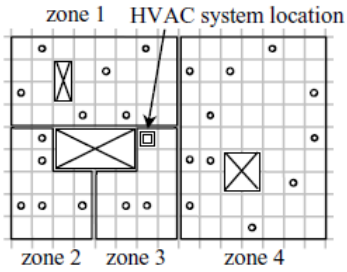
This idea seems feasible, but no documentation of this type of usage has been found and when considering the time constraints, it might be too ambitious to try a completely new approach to this type of problem.

CONCLUSION

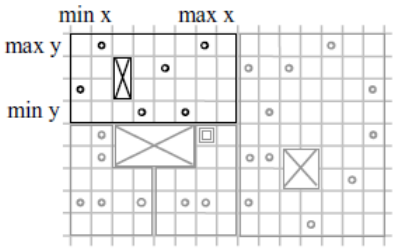
The Steiner Tree Problem was the first problem that appeared to be useful for use in the main problem. Since then, other solutions such as various genetic algorithms and clustering theories have been found, but none seemed to be directly applicable to the main problem. Therefore the Rectilinear Minimum Steiner Tree problem and its existing solvers are chosen for use in this paper's solution.

REVIT ANALYSIS

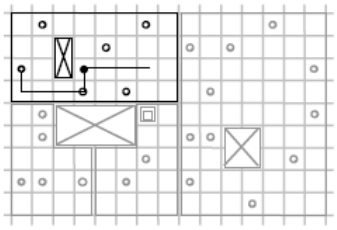
The other part of the research is focused on the realisation of our proposed solution. This chapter will document the analysis of the existing functionality in Revit to further define and refine our solution.



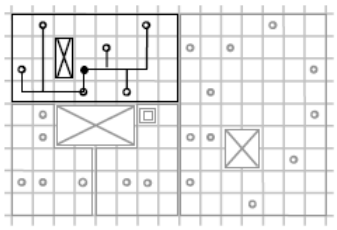
Input for distribution layout generation



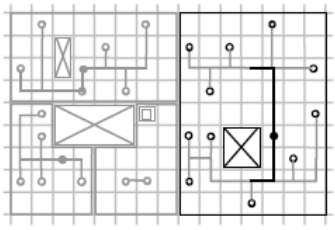
Step 1: Find min and max coordinates



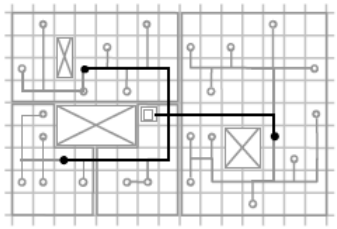
Step 2: Establish Clusters



Step 3: Connecting terminals to the cluster branch



Step 4: Generate zone branch



Step 5: Connecting zones points to start node

Figure E.5: Distribution network design sequence used in [18]

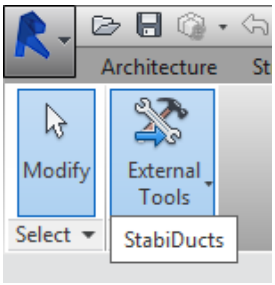


Figure E.6: An external command in the Revit UI

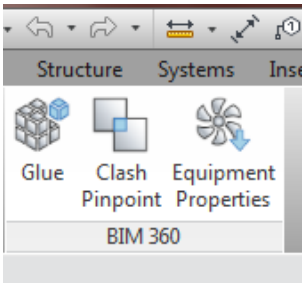


Figure E.7: An external application in the Revit UI

REVIT

Revit is software used for Building Information Modeling (BIM) released and maintained by Autodesk. It includes features for architectural design, construction, structural engineering and MEP. Stabiplan focuses on the latter.

Designing MEP systems is done by hand and takes quite the manpower and time to complete. It is not enough to simply connect the pipes. There are standards to consider, flow control, distance optimisation, material choice and the list goes on. An improvement would be to automatically make certain choices for the user, lowering workload and accelerate the process.

The built-in layout solver provided by Revit seems to use a few approaches to see how it would connect the ducts, without regards to most design principles. The results usually contain more than one layout suggested by the algorithm that is impossible to realise, whereas those possible are inefficient in the eyes of experienced designers.

The target is to develop an extension for Revit to deliver solutions based on more variables that are more likely to result in minimal adjustments by the designer. A possible option would be to mark points as point of preference to pass through when routing, while expanding the routing criteria to take cost into account.

It would be beneficial to make full use of the application designed by Stabiplan as that would give us access to a database of components to match with to avoid designing a layout that is not possible in reality.

REALISATION

Developing an extension for Revit is possible through the Revit and RevitUI APIs. These APIs are based on .NET, which enables users to use any .NET compliant programming language such as C#, VB.NET etc. to develop an add-in. Revit supports two ways of extending its functionality:

- *External command*, which is a single object that is destroyed after its `Execute()` method returns to Revit. As a result, data cannot persist in the object between command executions. An external command usually resides in the "External Tools" menu of the "Add-ins" tab in Revit, where it can be triggered manually. See Figure E.6.

- *External application*, which enables the use of custom "ribbon tabs" or "ribbon panels". An external application is loaded when Revit starts and destroyed when Revit closes. This implies that it may interact with Revit at any given moment. These applications may contain several buttons triggering external commands as seen above. See Figure E.7.

When an external command is triggered, the Revit API provides the command with an object named `commandData`, which contains the application data and settings for the current Revit session. This is the main entry point for accessing the data and elements within the active document. [1]

[1] For a more extensive explanation please refer to the official API documentation: `http://help.autodesk.com/cloudhelp/2014/ENU/Revit/files/GUID-F0A122E0-E556-4D0D-9D0F-7E72A9315A42.htm`

Development Tools

There are two main tools needed to develop an utility to solve the routing problem given in a 3d MEP model: Visual Studio and the Revit API.

Visual Studio acts as multiple tools combined into one: It compiles the code, handles debugging, manage version control, develop and run tests and as a bonus auto-complete and correct terms while displaying function explanations.

All versions of Revit are supplied with an API for development purposes. The API provides ways to automate tasks, extend the core functionality and manage the data. This last point is quite complicated. A 3d model contains vast amounts of data and in Revit anything in the design is stored as an Element. These elements hold information such as xyz coordinates, the type of element amongst other things. It will be crucial to design and implement an interface to translate between our algorithmic format and elements from Revit's environment.

There are multiple ways to extend the core functionality, but most functions are blackboxes not allowing us to call sub-sections of a function. Clicking the button is the furthest you get. Calling existing functionality within Revit is possible through PostCommand, which adds a command call (click the button) to the queue to execute once our function is done.

Stabicad API

There is a third tool to consider using, asides from the two forementioned tools, for this project namely Stabicad. Stabicad is a software package developed by our employer Stabiplan and acts as a wrapper around the revit API adding MEP specific strong features to Revit. Features such as cross-referencing against existing components, junction solver and simulating flow models for air ducts. It would be beneficial to use Stabicad, as we would have access to the source and could use their code base to accelerate our progres. Stabicad uses an internal format to converse with, and convert to, Revit.

It is hard to determine how far acces to Stabicad would carry us, as we can not look into the API at this point in time. What we can say, based on conversations with the developers, is that Stabicad has the necessary wrapper functions to communicate with Revit and uses an internal model format when planning complex operations.

Concepts that will be needed to implement the solution

Validity check

Revit has built-in checks for both pipe, duct and electrical systems. These checks show places where the system is not connected properly and check the system flow for errors. Revit is also capable of producing "pressure loss reports" for pipe and duct systems. These functions are all accessible through the API [19].

Collision detection

There are ways to detect collisions using the Revit API. Proximity of elements can be found using built-in methods, although this only works for elements within the host file, not for elements in linked files [19]. Since most MEP models have the architectural design linked to them, this means that wall collisions cannot be detected this way. There are, however, other ways to accomplish this, such as using an `ElementIntersects-ElementFilter` to get a collection of colliding elements in the active document. This seems to work with elements in linked files.

Mechanical systems

Revit is able to create systems from different connectors, thus seperating the entire problem into multiple sub-problems. Usually a system is made up of several terminals and one supplier, such as air vents and one fan. These systems might then be connected using the built-in solver, or using our own approach.

VISUALISING THE SOLUTION

The layout solver that Revit uses visualises their solution in line segments. It would be wise to use a similar displaying method, keeping it an option to allow user-input. Simple colorcoded line segments should be sufficient without obfuscating information.

USER INPUT

It would be beneficial to allow the user to alter the given solution before and after we compute it. An experienced designer can cover for flaws and preferences which might show up because certain details were ommited from our solution model. Preempting those misses would improve the solution over the whole, whereas slight corrections are significantly better than redoing half the solution.

CONVERTING THE SOLUTION TO COMPONENTS

The last step would be to convert the solution to existing components, choosing pipes and ducts compliant with the standards while satisfying the flow requirements. Using the flow modeling and Stabicads component conversion to existing components will make it possible to create a realisiting component list. This is also the step where material costs are of greatest importance, and can help decide which supplier to choose.

CONCLUSION

Analysis of the problem showed that there were multiple factors to account for when aiming for an optimal solution. Considering practical costs such as material cost which are easier to model than theoretical costs such as maintainability which are hard to capture in a formula. From an algorithmic approach the network of vent and ducts strongly resembled graph theory, finding a similar problem however, deemed dificult.

One of the problems that stuck out was the Steiner Tree Problem, specifically the more restricted Rectilinear Minimum Steiner Tree Problem. This problem showed similarities with how solutions are designed in practise now, as in general duct systems are deisgned with a preference for 90 degree turns. And as such, the RMST is the approach chosen to implement.

To implement it as an extention to Revit will need careful handling of Revit's data structure and certain concepts worked out to relay the solution to the user. Implementing directly into Stabicad would be beneficial as large portions of the functionality we would have to implement is already available in Stabicad.

GLOSSARY

Cluster	A cluster is a grouped set of air terminals.
Cluster trunk	A cluster trunk is a duct that serves as a main trunk for a subset of connectors.
Git	Git is a revision control system made for software development.
Hanan grid	A rectangular grid consisting of the lines and intersections formed by extending lines along two or more axis from a set of nodes.
Kruskal's Algorithm	A "greedy" algorithm devised by Kruskal. It looks at edges in order of increasing length and the first tree that can be formed with those edges is an MST.
Main trunk	A main trunk is a concept often used in MEP design where a system consists of one big, single duct that supplies air to the entire system.
Minimum Steiner tree	An MST with the addition of Steiner points.
Steiner point	A point that is added to an MST to further reduce the network length.

Acronyms

API	application programming interface
BIM	building information model
MEP	mechanical, electrical, and plumbing
MST	minimum spanning tree
RMST	rectilinear minimum Steiner tree
SIG	Software Improvement Group B.V.
TDD	test-driven development

BIBLIOGRAPHY

[1] V. Sirianni and et al., *Department of facilities building systems design handbook,* (Massachusetts Institute of Technology, 2001) 1.1 ed.

[2] C. Eastman, P. Teicholz, R. Sacks, and K. Liston, *BIM handbook: A guide to building information modeling for owners, managers, designers, engineers, and contractors,* (John Wiley & Sons, Inc., 2011) Chap. Foreword, 2nd ed.

[3] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner tree problem* (Elsevier, 1992).

[4] M. Hanan, *On steiner's problem with rectilinear distance,* SIAM Journal on Applied Mathematics **14**, 255 (1966).

[5] P. Burke, *The shortest line between two lines in 3d,* (1998).

[6] M. Cohn, *Succeeding with Agile: Software Development Using Scrum* (Pearson Education, 2009).

[7] D. S. Janzen and H. Saiedian, *Test-driven development: Concepts, taxonomy, and future direction,* Computer Science and Software Engineering , 33 (2005).

[8] J. Visser, *SIG/TÜViT evaluation criteria trusted product maintainability,* (2014).

[9] S. Hatton, *Choosing the right prioritisation method,* in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on* (IEEE, 2008) pp. 517–526.

[10] L. Rising and N. S. Janoff, *The scrum software development process for small teams,* IEEE software **17**, 26 (2000).

[11] T. K. et al., *Optimum duct design for variable air volume systems, part 1: Problem domain analysis of vav duct systems,* ASHRAE Transactions 2002 **108** (2002).

[12] T. Dwyer, *Airflow pressure drop in hvac ductwork,* Chartered Institution of Building Services Engineers Journal (2011).

[13] ASHRAE, ed., *Chapter 48. noise and vibration control,* (ASHRAE, 2011).

[14] X. Cheng and D.-Z. Du, *Steiner trees in industry*, Vol. 11 (Springer, 2001).

[15] R. Karp, *Reducibility among combinatorial problems,* in *Complexity of Computer Computations*, The IBM Research Symposia Series, edited by R. Miller, J. Thatcher, and J. Bohlinger (Springer US, 1972) pp. 85–103.

[16] M. R. Garey and D. S. Johnson, *The rectilinear steiner tree problem is np-complete,* SIAM Journal on Applied Mathematics **32**, 826 (1977).

[17] C. J. Alpert, G. Gandham, J. Hu, J. L. Neves, S. T. Quay, and S. S. Sapatnekar, *Steiner tree optimization for buffers, blockages, and bays,* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **20**, 556 (2001).

[18] R. Brahme, A. Mahdavi, K. Lam, and S. Gupta, *Complex building performance analysis in the early stages of design,* in *Seventh International IBPSA Conference* (2001) pp. 661–668.

[19] J. Tammik, *Revit MEP programming: All systems go,* (2012).