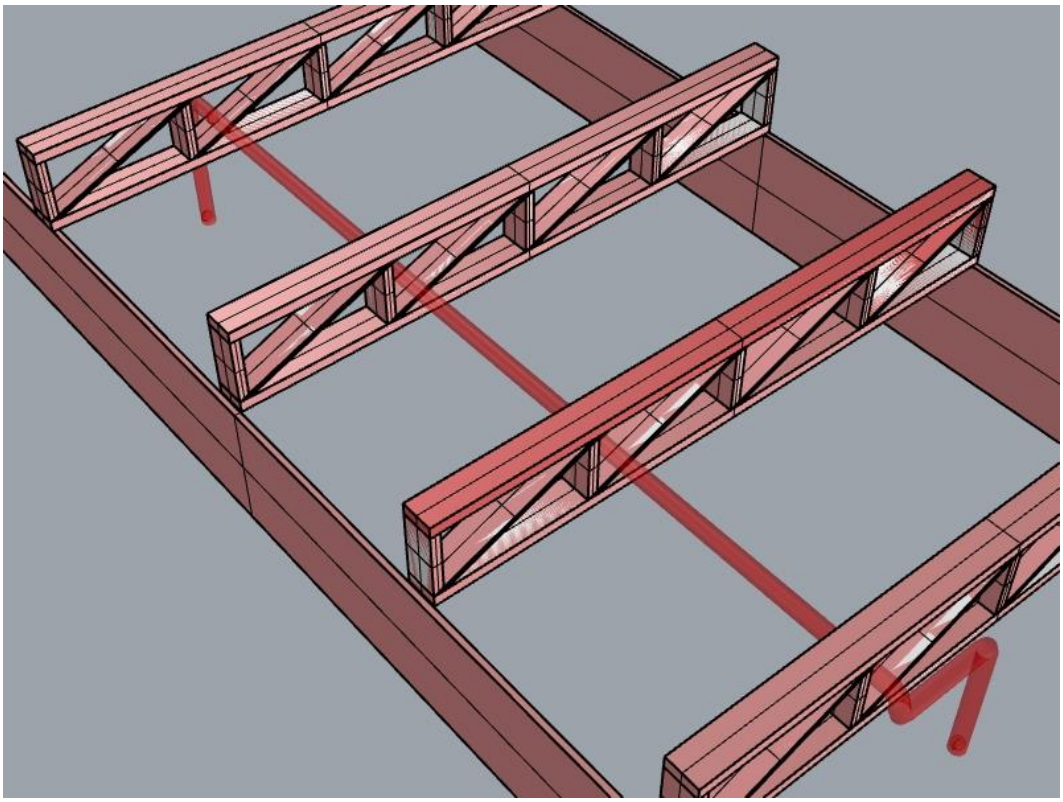
Master Thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Engineering
Department of Structural Engineering

Mathias Lien

A* in automation of modelling and clash avoidance in construction design

Trondheim, June 2020



NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Structural Engineering
Faculty of Engineering
NTNU – Norwegian University of Science and Technology

ACCESSIBILITY:

OPEN

MASTER THESIS 2020

SUBJECT AREA: ICT & Construction	DATE: 08.07.2020	NO. OF PAGES: XI + 40

TITLE:

A* in automation of modelling and clash avoidance in construction design

A* i automasjon av modellering og kollisjonsunngåelse i konstruksjon design.

BY:

Mathias Lien

SUMMARY

Context

Today one of the lowest scoring sectors in digitalization is the construction sector. Still much has changed the last decade in the digitalization in structural design. The construction industry is leaving the era of 2D drawings in favor of BIM and eventually parametric modeling. With new design software that facilitates programming, it has never been easier to develop new tools for digital design of buildings. Therefore, it can be interesting to explore whether the construction industry can learn from the industries that have come further in digitization and borrow techniques from there.

Objective

The aim of this thesis is to explore whether simple design problems can be reduced to a path finding problem and thus use a path finding algorithm to solve the problem. Such an algorithm will be able to update itself every time something changes and thus automatically avoid collision. This will then save time and money, and will integrate smoothly with parametric modeling.

Method

To test this in practice, a plugin was created for the Grasshopper visual programming interface which is part of the Rhinoceros 6 CAD program. This plugin contains custom grasshopper components. These components allow the user to divide the space around existing geometry into a grid and then run the A* path finding algorithm to find a path between two selected points. The user har the ability to influence which path the algorithm chooses by changing different values in the algorithms cost function. Finally, the result is shown in the form of a pipe modelled along the path between the two points.

Result

The algorithm was run on different cases and with different input values to test how the algorithm behaves. It is shown how the user can influence the algorithm to have different properties by changing the input values. The algorithm showed promising results in some cases, while in other it did not. A run time test was also done.

Conclusion and further work.

The algorithm shows promising results, but a lot of work remains. There are a number of things that need to be improved with the current algorithm to make it work more optimally. Among these is to reduce the number of turns the algorithm chooses to do as well as reduce run time. There are also several elements of the algorithm that may be interesting to explore further such as the heuristic function, the cost function but also other path finding algorithms.

RESPONSIBLE TEACHER: Professor Nils Erik Anders Rønnquist

SUPERVISOR(S): Professor Nils Erik Anders Rønnquist

CARRIED OUT AT: Department of Structural Engineering, Norwegian University of Science and Technology

Summary

Context
Today one of the lowest scoring sectors in digitalization is the construction sector. Still much has changed the last decade in the digitalization in structural design. The construction industry is leaving the era of 2D drawings in favor of BIM and eventually parametric modeling. With new design software that facilitates programming, it has never been easier to develop new tools for digital design of buildings. Therefore, it can be interesting to explore whether the construction industry can learn from the industries that have come further in digitization and borrow techniques from there.

Objective
The aim of this thesis is to explore whether simple design problems can be reduced to a path finding problem and thus use a path finding algorithm to solve the problem. Such an algorithm will be able to update itself every time something changes and thus automatically avoid collision. This will then save time and money, and will integrate smoothly with parametric modeling.

Method
To test this in practice, a plugin was created for the Grasshopper visual programming interface which is part of the Rhinoceros 6 CAD program. This plugin contains custom grasshopper components. These components allow the user to divide the space around existing geometry into a grid and then run the A* path finding algorithm to find a path between two selected points. The user har the ability to influence which path the algorithm chooses by changing different values in the algorithms cost function. Finally, the result is shown in the form of a pipe modelled along the path between the two points.

Result
The algorithm was run on different cases and with different input values to test how the algorithm behaves. It is shown how the user can influence the algorithm to have different properties by changing the input values. The algorithm showed promising results in some cases, while in other it did not. A run time test was also done.

Conclusion and further work.
The algorithm shows promising results, but a lot of work remains. There are a number of things that need to be improved with the current algorithm to make it work more optimally. Among these is to reduce the number of turns the algorithm chooses to do as well as reduce run time. There are also several elements of the algorithm that may be interesting to explore further such as the heuristic function, the cost function but also other path finding algorithms.

Sammendrag

Kontekst
Byggbransjen er i dag de en av bransjene som scorer dårligst på digitalisering sammenliknet med andre bransjer. Samtidig har mye skjedd på digitaliseringsfronten innen bygningsdesign. Byggbransjen er på tur ut av en æra med 2D tegninger til fordel for BIM og etter hvert parametrisk modellering. Med framveksten nye dataprogrammer som tilrettelegger for videre utvikling har det aldri vært så enkelt som nå å utvikle nye verktøy for digital design av bygninger. Derfor kan det være interessant å se om man i byggbransjen kan lære av de bransjene som har kommet lenger i digitaliseringen og låne teknikker derfra.

Målsetting
Målsetningen med denne oppgaven er å utforske hvorvidt enkle problemer innen konstruksjonsdesign kan reduseres til stifinnerproblemer og dermed benytte stifinneralgoritmer til å automatisk finner en vei. En slik algoritme vil kunne oppdaterer seg hver gang noe endres og dermed automatisk unngå kollisjon. Dette vil da kunne spare tid og penger, og vil gli naturlig inn med parametrisk modellering.

Metode
For å teste dette i praksis ble det laget en plugin til det visuelle programmeringsgrensesnittet Grasshopper som er en del av CAD programmet Rhinoceros 6. I denne pluginen inneholder skreddersydde grasshopperkomponenter. Disse komponentene gir brukeren mulighet til å inndele eksisterende geometri i et rutenett og deretter kjøre A* stifinneralgoritmen for å finne en vei mellom to valgte ruter. Brukeren har mulighet til å påvirke hvilke veier algoritmen velger ved å endre på ulike verdier i algoritmens kostfunksjon. Til slutt vises resultatet i form av et rør modellert langs ruta mellom de to punktene.

Resultat
I ulike tester vises det hvordan algoritmen oppfører seg med ulike inputverdier. Det vises hvordan brukeren kan påvirke algoritmen til å ha ulike egenskaper ved å endre på inputverdiene. Det diskuteres om algoritmen oppfører seg ønskelig, i hvilke tilfeller den ikke gjør det og det gjøres også en kjøretidstest.

Konklusjon og videre arbeid.
Konklusjonen er at algoritmen viser lovende resultater, men at mye arbeid gjenstår. Det er en rekke ting som må forbedres med nåværende algoritme for å få den til å fungere mer optimalt. Blant dette er å minske antall svinger algoritmen velger å gjøre samt redusere kjøretid. Det er også en rekke deler av algoritmen som kan være interessant å utforske videre som heuristikkfunksjonen, kostfunksjonen men også andre stifinneralgoritmer.

Acknowledgment

I would like to thank Professor Nils Erik Anders Rønnquist for wanting to supervise on this relatively unexplored topic and for great advise during this thesis. I am also grateful that Phd candidate Marcin Łuczkowski inspired me to chose this thesis instead of the one I originally had planned. Lastly I would like to thank Phd candidate Steinar Hillersøy Dyvik for the talk we had on Rhino and Grasshopper plugins.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) as part of the course TKT4920 Structural Design, Master's Thesis. The work has been performed at the Department of Structural Engineering, NTNU, Trondheim, under the supervision of Professor Nils Erik Anders Rønnquist.

Contents

Abbreviations **xi**

1 Theory **1**
1.1 The eras of Computer Aided Design(CAD) 1
1.2 The problem 2
1.2.1 How is this task handled today? 3
1.2.2 How will my solution help? 3
1.3 Software 4
1.3.1 Rhino Grasshopper 4
1.3.2 C#, Visual studio and Grasshopper plugins 5
1.4 A* Algorithm 7
1.4.1 Use in gaming 9
1.4.2 Development 9
1.4.3 Other uses in construction 9

2 Method **11**
2.1 Reducing the problem 12
2.1.1 The maze problem 12
2.1.2 Preparing the maze problem 12
2.1.3 How to influence the chosen path 14
2.2 The Rhino Grasshopper workflow 18
2.2.1 The Rhino Grasshopper cycle 18
2.2.2 Components 20
2.2.3 Grasshopper setup 21

3 Results **23**
3.1 Case studies 23
3.1.1 Around or through wall 23
3.1.2 Under truss 26
3.1.3 Through truss 27
3.1.4 Misc 28

4 Discussion **29**
4.1 Why the A* algorithm 29
4.2 A point about run time 30
4.2.1 Improving the runtime of the cost function calculations 32
4.3 A point about grid size, bends and other problems 33

5 Conclusion and further work **35**
5.1 Was it a good implementation? 35
5.2 Further work 35

Appendix **37**

List of Figures

1.1 Example on how to create a NURBS curve in Grasshopper . 5
1.2 Curve and points made in fig. 1.1 5
1.3 justification=raggedright 5
1.4 Gif of the A* algorithm. This opens in Adobe pdf viewers only. A sheet of the frames are available in the appendix. . . 7

2.1 Maze problem 12
2.2 The grid is not perpendicular to the walls 13
2.3 Cost of all tiles are 1. Walls are infinite 14
2.4 Example constricting geometry 16
2.5 justification=raggedright 16
2.6 Distance between cell and wall. 16
2.7 Rhino Grasshopper workflow 18
2.8 World Mesher 3D 20
2.9 PointSetter 20
2.10 CostSetter 20
2.11 A* Pipe 20
2.12 CostChanger 20
2.13 Grasshopper components in action 21

3.1 As the cost for going through the wall is high A* goes around. 23
3.2 Settings for above example 23
3.3 Here the cost for going through the wall is lowered. 24
3.4 Settings for fig 3.3 24
3.5 A "hole" or a preferred spot for going through the wall was created. 25
3.6 Same example as Fig 3.5 but with the start point moved one node to the left. 25
3.7 Example 26
3.8 Example 27
3.9 Here the cost along the walls are negative 28

4.1 Runtime during test 30
4.2 Dummy objects 30
4.3 Runtime with decreasing grid size 31
4.4 These two lines are costly functions 31
4.5 This figures shows some of the problems with the current algorithm 34

5.1 This shows the frame of the gif shown in fig: 1.4 38

Abbreviations

BIM	=	Building information modeling
NURBS	=	Non-uniform rational basis spline
CAD	=	Computer-aided design
CPU	=	Central processing unit
GPU	=	graphical processing unit
ICT	=	Information Communications and Telecommunications
2D	=	two dimensional
3D	=	three dimensional
GC	=	GenerativeComponents
WM3D	=	World Mesher 3D (custom grasshopper component)

Chapter 1

Theory

1.1 The eras of Computer Aided Design(CAD)

There are many ways to describe the history of Computer Aided Design (CAD). One way presented by Robert Aish is that the way we design with and think about CAD can be presented as three eras[1]. These in more or less chronological order these are *the 2D drafting era, the BIM era and the design computational era.* They are heavily overlapping, but the logic and individual adaption uusally follows this order.

The 2D Drafting era
This era started in the 1980s when the first computer tools were made to aid in the already existing practice of drawing 2D-drawings. 2D drawing was at the time the main way of designing. When computers started to become a common tool for engineering firms, programs like Sketchpad and early versions AutoCAD emerged. The way of designing still consisted of drawing a model line by line. All the information is written in clear text on the drawing. This was a natural progression as the workflow stayed the same, but the ability to digitally correct, redraw, store and reuse drawings made computer programs attractive. This era and way of designing, however, did not capitalize on the vast functionality a computer can provide.

The Building Information Modelling Era
As computers are able to store large amounts of data, the natural progression from the 2D drafting era is the building information model (BIM) era. Instead of designing through a series of separate drawings the idea behind BIM is to create a single 3D model instead. This has several advantages over 2D drawings. With a single 3D model it is a lot easier to force overall consistency especially when there are a lot of subcontractors. A lot of metadata can also be stored about the 3D model apart from just the geometry. Last but not least it is a lot easier to get an overview of the whole project. This also makes it possible to do clash detections tests where it is tested whether geometry which should not not intersect does that nonetheless. The model is still manually modeled like in the drawing era which means the work flow is not changed that much. This suites a conservative indus-

try well and is therefore a logical next step.

The Design Computational Era
The third and last era is the computational era. In the BIM era the engineer designed the model directly. In the computational era the idea is that the engineer designs a script or graph (e.g in Grasshopper, explained in section 1.3.1) which generates the model for the engineer. The motivation behind this is that even though the initial time spent on creating a script or graph is often longer than creating a 3D model in a designing software, the time it takes to change the model is significantly less. This era started with the appearance of parametric modeling software like Grasshopper, Autodesk Dynamo and Bentleys' GC around 2007 (commercial launch of Grasshopper and GC). In these programs the user creates a directed graph of functions which create geometry. The parameters of buildings are often easily changed by number sliders or similar and then the whole building is remodeled with the updated parameters. Although graphical scripting is intuitive it is rarely seen in the ICT industry. There is therefore quite probable that the architectural, engineering and construction industry will move towards imperative programming (i.e. text scripts in programming languages) which is the norm in high tech industries.

Where are we today?
This question is not as straight forward as it seems. The reason for this is that studies have shown that different firms are in different eras. For example in a study done about clash detection and avoidance [2] companies reported that a clashes occurred because subcontractors delivered 2D drawings instead of BIM models. This shows that there are both firms in the BIM and 2D drafting era. Grasshopper, GC, and Dynamo have growing functionality and which might indicate that there are firms entering the computational design era. As there are firms at all sides of the spectrum, the majority usually lay somewhere in the middle which in this case might be the BIM era, but it's all rather speculative. The important part is that imperative programming in the design computational era is a probable future.

1.2 The problem

Even though more and more firms and people in the construction industry are embracing BIM, parametric modelling and other more technological solutions, the construction industry is still far behind in terms of digitalization. McKinsey did a study in 2015 on how different sectors in the US compared in the use of digital tools and automation to boost productivity[3]. Of all the 21 sectors in the study construction came second to last, only better than the agriculture and hunting sector. Even though this was a US study it is not entirely unlikely that the results would be similar in Europe and Norway. As the construction sector is working its way through the CAD eras it is also working its way towards a more digital way of de-

signing. It can therefore be interesting to look at what technologies have been developed in the more digitalized sectors like the ICT sector and the media sector.

One of the big cost savers of having a common BIM model is the ability to do clash detection to avoid clashes between different engineering disciplines[2][4][5][6]. Clash detection and collision is not a subject specific to the construction sector. In the media sector for example this problem has been solved in a variety of ways. More specifically in computer games. Collision detection has been a subject in video games since the very first video game, a tennis game similar to Pong, was created in 1958. But video games today have taken it a lot further. Not only are there a vast number of methods for collision detection in use, there are also a lot of algorithms and methods for collision avoidance. Collision avoidance in construction is a subject which is currently lacking research[6]. Therefore the question of this thesis is the following:

Can we use methods for collision avoidance implemented in other fields to solve collision avoidance in construction?

1.2.1 How is this task handled today?

A study was done on clash avoidance and detection by Akponeware et al. in 2017[6]. It concluded that collision/clash avoidance during the design phase usually was solved indirectly. It was solved indirectly by putting all BIM sub assemblies on top of each other, then run the model through a clash detection program and then edit the parts which do clash. Although dealing with clashes during the design phase saves costs, it still demands resources. It is also not error proof as the clash detection programs might have problems when there is a combination of BIM models and 2D drawings. It is probably not the optimal solution, but it is the most widely used solution today according to Akponeware et al.

1.2.2 How will my solution help?

The idea behind this thesis is that since clash detection and clash avoidance has been studied thoroughly in other more digitalized sectors there must surely be lessons to be learnt. With the emergence of different parametric tools which also have support for imperative programming it is easier than ever to implement those solutions which have already been developed in other fields. In this thesis I will therefore experiment with the automation of creating a pipe with the help of path finding algorithms. This is beneficial as it solves these problems:

- It reduces the time needed to model, as the pipe will model itself.
- Most importantly it will avoid collision unless the user specifically wants it. Therefore no resources are needed to deal with clashes.

- Whenever some of the other geometry is changed, the piping algorithm will be rerun and therefore ensures a forever collision free pipe.

1.3 Software

1.3.1 Rhino Grasshopper

To be able to implement a programmed solution for the above described problem applicable software is needed. Rhinoceros 6 (Rhino) is a Computer-aided design software. Like other CAD programs its purpose is to let the user create, edit, analyze, document, render, animate, and translate 2D and 3D geometry. What differentiates Rhino from other similar CAD programs is its focus on Non-Uniform Rational B-Spline (NURBS) geometry[7]. NURBS is a mathematical approach to model curves and surface geometry. NURBS curves and surfaces are made with the help of control points and a mathematical function based on these control points. This is important because it makes it easier to change geometry by changing the control points used to create them. Because NURBS are mathematically defined, a lot less information is required to represent the geometry than with common faceted approximations [7]. The combination that most of the geometry can be changed with control points, that each geometry uses little memory and that all geometry is based on math makes Rhino a perfect combination with Grasshopper.

Grasshopper started as a plugin for Rhino and is now fully integrated into Rhino. Grasshopper functions as a visual scripting environment. This means that the user creates a script visually by making a directed graph. In the graph each node is a called a *component* and each edge is called a *connection*. A component gets data via the connections from upstream components, does some computation and sends data downstream to downstream components. A component can receive data, send data or do both.

An example of a grasshopper script is shown in fig. 1.1. In this script three number parameters are set by the number sliders. Then three points are created which are then used as control points for the NURBS curve shown in fig. 1.2. With this set up, the parameters of the points are easily changed and then the geometry will adjust accordingly as seen in fig. 1.3. As we can see, the Rhino Grasshopper combination is a excellent tool for parametric modeling. The visual way of scripting is a way of programming which is very easily understood by humans even without any earlier coding experience. This makes parametric modeling in Grasshopper available to many and is Rhino-Grasshoppers biggest selling point. Geometry can also be modeled "by hand" in Rhino, then captured by Grasshopper as a component and then further geometry may be made in the Grasshopper environment. Whenever the captured geometry is changed in Rhino the Grasshopper script wil be updated to fit with the new geometry. The ge-

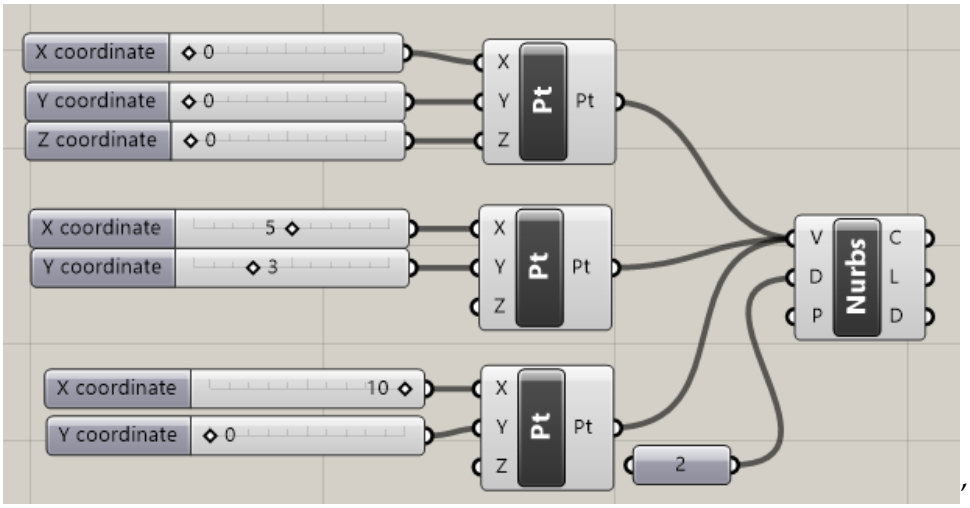


Fig 1.1
Example on how to create a NURBS curve in Grasshopper

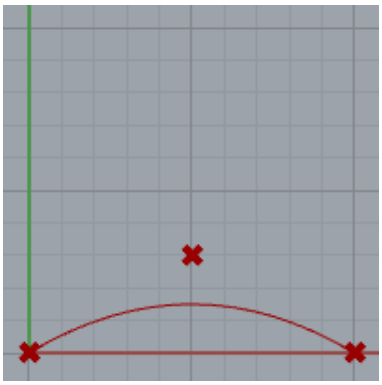


Fig 1.2
Curve and points made in fig. 1.1

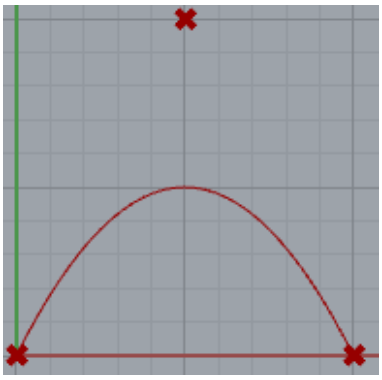


Fig 1.3
Y coordinate in the second point increased to 10

ometry created in grasshopper however are only visualized in Rhino. It can not be changed in Rhino before the user are satisfied with the results and exports the result from Grasshopper to Rhino. This might seem tedious but it is easily done and lessens the work needed for visualization and thereby speeds up the scripts.

1.3.2 C#, Visual studio and Grasshopper plugins

As mentioned earlier Grasshopper started as a plugin. It is written in the programming language C#. C# is a general-purpose object oriented programming language created by Microsoft around 2000 as a part of their .NET Framework initiative[8]. The syntax is C like and the standard library is covers a lot. As it is supported by one of the biggest companies in the tech industries, it has grown to be a modern, and quite popular language among programmers. Microsoft has made its own intelligent design environment (IDE) called Visual Studio where most of the set up for

a C# program is already done. If one needs further functionality than what Visual Studio offers on installation a vast amounts of plugins for Visual Studio exists. Grasshopper has created one of these plugins. With this plugin a developer can create, package and install plugins with custom components and functionality for Grasshopper.

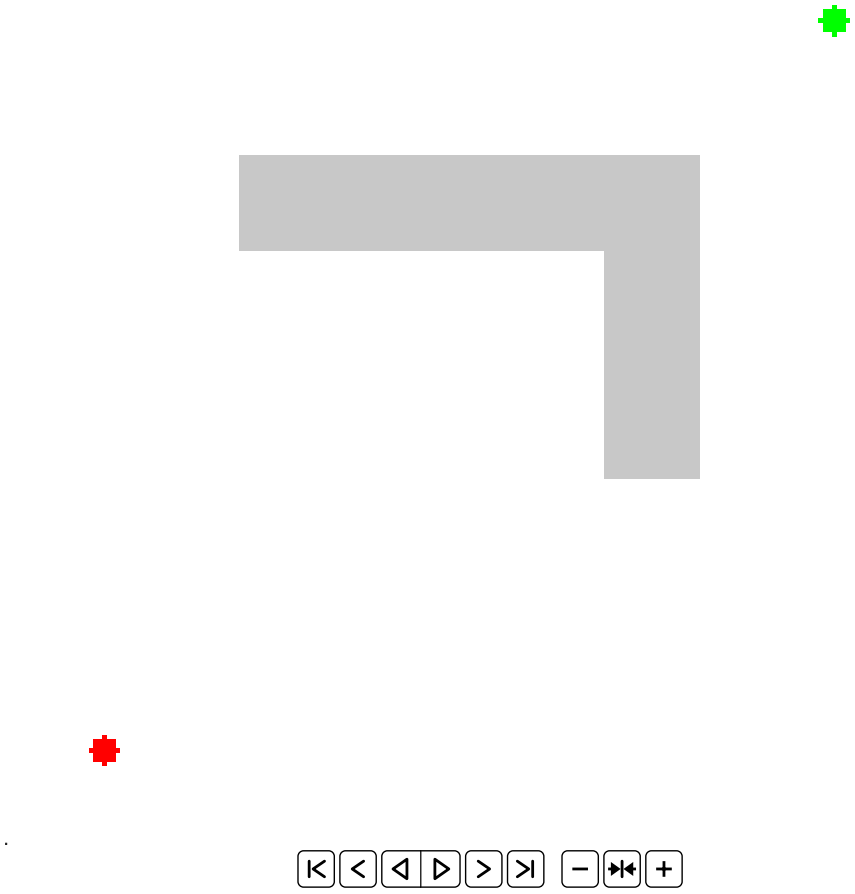


Fig 1.4
Gif of the A* algorithm. This opens in Adobe pdf viewers only. A sheet of the frames are available in the appendix.

1.4 A* Algorithm

The A* (pronounced A-star) algorithm is a greedy breadth first graph search algorithm[9]. It is built on Dijkstra's shortest path algorithm, and the algorithm was first published in 1968 as a part of the Shakey robot project[9]. The algorithm operates in a graph of nodes where every node is given a cost. The algorithm is given a start node and a goal node and then searches the graph for the path with the lowest cost. Compared to Dijkstra, the goal of the A* algorithm is the same but the A* algorithm is designed to find the optimal path while visiting the fewest nodes possible. An visualization of the algorithm is shown in fig 1.4. The algorithm works like this:

1. Add starting node to the OPEN list.
2. As long as there are elements in the OPEN list do the following:
   (a) Pop the highest priority (lowest values) item from the OPEN list. Set this to be the agent node.
   (b) Add this node to the CLOSED list.
   (c) Check if agent node is the goal node. If it is then stop and jump to step 3.
   (d) For each neighbor of the agent node do the following:
      i. Check to see if the neighbor is in the Closed List. If it is, ignore it and jump to next neighbor.
      ii. Check to see if it's in the OPEN list.
         A. If it's not, add it to the OPEN list. Set the agent node as its parent and record the cost so far. Set the priority to the cost so far plus the heuristic function.
         B. If it is in the OPEN list check if the path through the agent node is cheaper than through it's current parent node. If it is set agent as the new parent node and update with the new lower cost.
3. Backtrack from the goal node to the starting node by walking through each node's parent. Return this path.

As described above, the A* algorithm uses a priority to determine what node to visit next. The A* algorithm will prioritize to visit the node by the following function:

$$f(n) = g(n) + h(n) \tag{1.1}$$

Where $g(n)$ is the cost of reaching the node n from the start node and $h(n)$ is what's called a heuristic function. The heuristic function is an estimate on the reminding cost to reach the goal node from node n. For the A* algorithm to be optimal the heuristic function must be *admissable* and *consistent*. A heuristic function is *admissable* if it fulfills the following criteria: *The estimated remaining cost must not exceed the actual cost to reach the goal.* i.e. it must never overestimate the cost of reaching the goal from the current point. As an equation, that is:

$$h(n) <= G(n, g) \tag{1.2}$$

where $G(n, g)$ is the cost of the optimal path from node n to goal node g. A heuristic function is *consistent* if the following is true: *The estimated cost to reach the goal node is always less than or equal the estimated cost of reaching the goal from any neighbor plus the cost of reaching that neighbor*. This can be written as:

$$h(n) <= h(p) + G(n, p) \tag{1.3}$$

where p is a neighboring node of n. Say the A* algorithm operates on a 2D grid where each node is a square cell on the grid. The only allowed moves are up, down, left or right. Diagonal moves are not allowed. In this

scenario a widely used heuristic function is the Manhattan distance[10]. This function looks like this:

$$h_n = |x_n - x_{goal}| + |y_n - y_{goal}| \tag{1.4}$$

In other words the sum of the horizontal and the vertical distance to the goal. This heuristic function is admissable if the cost of each cell is at least 1. With the move set available and with the constraint that each cell cost at least one, the cheapest scenario possible is where the cost from start to goal is equal to the Manhattan distance. If costs are more than one or the cheapest path includes a detour the cost from start to goal will be larger than the Manhattan distance.

1.4.1 Use in gaming

The A* algorithm has been used for NPCs in many video games e.g[11]:

- Age of Empire series
- Sid Meyer's civilization series
- World of Warcraft
- Warcraft series

1.4.2 Development

The A* algorithm is known for it's use in video games[12]. In many video games a lot of movable characters need to find paths across a map or a world as efficient as possible. A* has shown that it is able to find an optimal path with a lower space requirement than many other methods. As long as the heuristic is both admissable and consistent the A* algorithm will find the optimal path, as long as it exists, while visiting equal or fewer nodes as any A* like algorithm[9]. Because of this, it has become very popular. A lot of work has been done to make it faster. Of the things that have been done, exploration of data structures are one of them. For the OPEN list a min-heap has been found to be one of the more optimal data structures. A min heap sorts the items added on insertion. Finding the lowest possible element will therefore take $O(1)$ time. For the Visited list a hashmap might be the optimal list as long as the required space is available. A hashmap is optimal because it uses $O(1)$ time to search for an element which is done quite often during the algorithm (see step ii in the algorithm). The result is an algorithm which runs much faster than one which is implemented with generic lists.

1.4.3 Other uses in construction

There exists at least one article [11] researching whether the A* algorithm can be used to automate generation of construction geometry. In this article A* has been used to calculate the optimal cutting lengths for concrete

reinforcing at the time of construction. The resulting geometry was a BIM model of the necessary reinforcement. The motivation was to lower the cost of excessive reinforcing steel usage during construction and with that also lower transportation costs and the time spent on construction.

Chapter 2

Method

In this chapter we will look at how the A algorithm in its original form can be used in a parametric environment to automatically generate desired geometry, more specifically pipes. The parametric environment chosen will be Rhino Grasshopper. The broad idea is that a grid is made around some existing geometry, a start and end point is chosen, a path is found with the A* algorithm and then a pipe is modeled along this path. Each step in the process will be as a separate Grasshopper component. Before the A* algorithm can be implemented a series of preparations needs to be done. Firstly the area containing constraining geometry will need to be divided into equally sized cells in a 3D grid. The resulting 3D grid will be the "World" where the A* algorithm will operate.The next thing to be done will be to set a cost for each cell in the 3D grid. A cost for each cell can be set based on the constraining geometry and parameters set by the user. Lastly the user will need to pick out a starting cell and a goal cell for the algorithm. When all this is done the A* algorithm can be used to find a path from cell A to cell B like in the maze problem (section 2.1.1). Exactly how this is implemented is described in the chapter. The heuristic function chosen in this master thesis will be the Manhattan distance and the cost function will also be described in this chapter.*

2.1 Reducing the problem

2.1.1 The maze problem

Fig 2.1
Maze problem

In this thesis the A* algorithm will be used in a way which simulates the maze problem. The maze problem is a common, well defined and well explored problem in computer science. The setup of the maze problem might look something like in fig 2.1. In a $N \times M$ grid maze the goal is to find the optimal path between a source cell and a destination cell. This problem can be solved by a lot of algorithms and the A* algorithm is one of them. For a discussion on why the A* was chosen among all path finding algorithms see section 4.1. In short, it is used because it is a generic algorithm, which is quite fast the first time it runs, no matter what variables are changed. The idea is therefore to take imported geometry from a BIM model, reduce the piping problem to the maze problem and then run the A* algorithm. To be able to do this there are a number of steps that needs to be done.

2.1.2 Preparing the maze problem

Firstly the geometry which the algorithm will work with needs to be imported. Let's say a company is designing a building and have created a virtual BIM model of the building. The important factors for the algorithm is not all the details of the facade, but the walls, columns, beams, floors and roofs. There might be a point to only include load bearing elements as dividing walls are often thin and cheap to penetrate. They are also not designed to carry the load of the pipes. Why this is a point will be explained later.

When the relevant geometry is imported it needs to be divided into a 3D grid of equally sized cubes. These cubes will be the cells which the algorithm move between. Each cell will have 6 neighbors, one to the left, right, up and down, one in the front and in the back. While this makes the problem a lot simpler it is also a restriction. By making cells in the form of cubes the pipe will only be able change direction in 90 degree turns. Here the assumption was made that most rooms and buildings are rectangular. Even in non-rectangular rooms it can often be observed that piping and ventilation ducts are still using only 90 degree bends. Therefore the reduction to only operate in 90 degree angles might be acceptable. Although there are a lot of non rectangular buildings, and algorithms which handles non rectangular rooms exists, the reduction to rectangular rooms was chosen to reduce the scope of this thesis.

Because the algorithm will be restricted to 90 degree turns it is important that the grid is perpendicular to the most defining geometries (i.e. walls). If this is not done one might have a case similar to fig 2.2. The pipe will follow the red path from point E to point J instead of the desired path shown in blue. Clearly this is not optimal, however if the grid cells are created perpendicular to the walls then the desired outcome is possible.

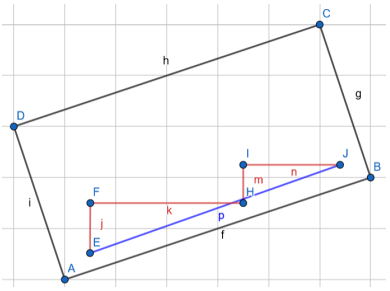


Fig 2.2
The grid is not perpendicular to the walls

When a grid is created the next thing to do is to assign a cost to each cell based on a chosen cost function. Lastly a source cell and a destination cell is chosen and the A* algorithm is run. If it exists, the A* algorithm will return a path between the source and destination cells. A pipe can then easily and automatically be modelled along this path. This is the idea of what the program is supposed to do. In short:

- Import relevant geometry.
- Create a grid around the geometry perpendicular to the most defining geometry.
- Apply a cost to each grid cell.
- Set a source and goal cell for the pipe and thereby defining it as the maze problem.
- Run the A* algorithm to find a path.
- Return a pipe modelled along the path.

2.1.3 How to influence the chosen path

As mentioned in equation 1.4 the algorithm is dependent on both a heuristic function and a cumulative cost function. Because the only movements allowed are up, down, left, right, forward and backward movements, the manhattan distance can be used as the heuristic as long as the cost of each cell is at least 1. If the cost is at least 1 the the manhattan distance will be both admissable and consistent. There is not much to influence here apart from completely changing the heuristic function, but that is beyond the scope of this thesis.

Until now it has only been mentioned that each cell has a cost. For the original maze problem a normal solution is to set the cost of each white cell as 1, and each black cell (walls) as infinite. In fact the A* algorithm often includes an additional step between step i and ii where it it checks if the tile is a wall. If this is the case, it is added to the CLOSED list and ignored. With a cost of 1 the manhattan distance is consistent and admissable. The algorithm wil find a way and it's usually to go straight at the goal, then dodge walls if any and then continue. An example of this can be seen in fig 2.3. This, however, is not necessarily the desired solution. Piping and

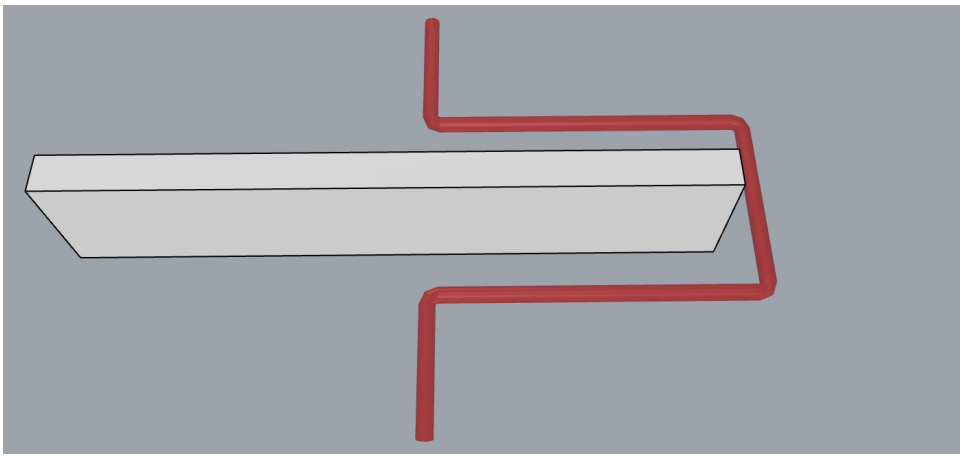


Fig 2.3
Cost of all tiles are 1. Walls are infinite

similar components like ventilation often follow walls and roofs. Piping straight through a room is often avoided as it occupies space which can be used for more useful things. Having components stick to walls and roofs also help lessen the structural complexity as piping moving straight through a room might need to be supported by very long cantilevers or columns, or have very large cross sections to be able to carry the self weight and additional load. Therefore it is often cheaper to have piping follow load bearing construction elements like load bearing walls instead of taking the shortest path from A to B. This is what creates the motivation to differentiate the costs of each tile. Therefore a lot of the work done during this thesis was to create a cost equation for the cells and to develop a system where the user is able to adjust the variables of the cost equation.

Cost function

Instead of having a constant cost or areas of constant cost for each cell in the grid, a cost will be calculated for each individual cell. The user will be able to set som parameters and then the cost of each cell is set by the following formula:

$$C_i = \begin{cases} \left[\dfrac{\left(\sum_{j=0}^{n} -\dfrac{1}{D_{ij}} \right) - c_{min}}{c_{max} - c_{min}} \right] \times MC + 1 & \text{if } D_{ij} > T \\ C_{wall} & \text{otherwise} \end{cases} \tag{2.1}$$

where:
D_{ij} = distance from tile i to geometry j.
T = wall tolerance, a distance set by the user.
c_{min} & c_{max} = min and max cost of all grid tiles before min-max normalization.
MC = Max Cost, a variable set by the user.
C_{wall} = cost of passing through a wall set by the user

It is important that the cost is no less than 1. This is because of the heuristic function used by the A* algorithm. For the A* algorithm to behave optimally it is required that the heuristic function never over estimates the cost to reach the goal. As the the heuristic function chosen is the Manhattan distance (see 1.4) then each move must cost at least 1. The idea is that the closer a node is to constricting geometry the cheaper it is. The constricting geometry is set by the user. The counter intuitive part of this equation is that the geometry given creates both the most expensive and the cheapest nodes. The cheapest nodes are those right next to the constricting geometry, while the nodes inside the geometries are often the most expensive. If the right geometry is chosen this makes sense from a construction perspective. From a construction perspective you want pipes, ventilation, and similar to follow load bearing construction elements. These elements can handle the extra load, while for example a light drywall cannot. At the same time it should be really costly to penetrate load bearing construction elements. The penetration point becomes a weak spot in the construction and therefore the element might have to be reinforced or the cross section might need to be enlarged. Either way this is expensive, increases the complexity of the construction and its often reasonable to avoid penetrating load bearing elements.

The path the algorithm chooses will have a varying degree of a property hence called stickiness. If the path is sticky it will cling to any wall or object possible, while unsticky it will tend to go straighter at the goal. An example on this is the left and right part of Fig 3.5. On the left side the pipe's path goes straight and therefore is unsticky, while on the right the pipe's path clings to the wall and is therefore sticky. The stickiness of the path is

controlled by the cost function. In the chosen cost function this is mainly done by the MaxCost (MC) variable. By increasing or decreasing this variable the user is able to control the stickiness of the algorithm. A large MC will give a large span in costs, and therefore more stickiness. The nodes in the middle of the room will become very expensive compared to the nodes near the walls.

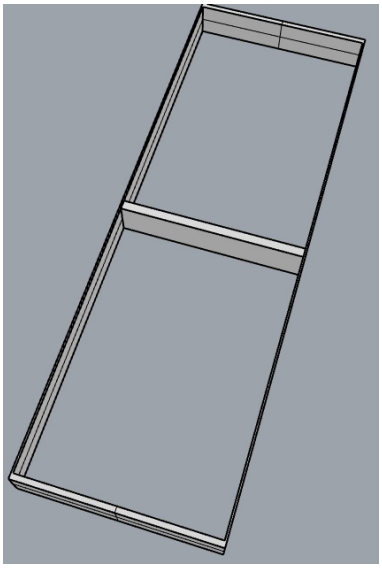


Fig 2.4
Example constricting geometry

1,81	1,81	1,81	1,81	1,81	1,81	1,81	1,81
1,80	1,93	1,93	1,93	1,93	1,93	1,93	1,80
1,80	1,92	1,98	1,98	1,98	1,98	1,92	1,80
1,80	1,92	1,97	2,00	2,00	1,97	1,92	1,80
1,79	1,92	1,97	2,00	2,00	1,97	1,92	1,79
1,79	1,91	1,96	1,99	1,99	1,96	1,91	1,79
1,78	1,90	1,96	1,98	1,98	1,96	1,90	1,78
1,77	1,90	1,95	1,98	1,98	1,95	1,90	1,77
1,76	1,88	1,94	1,96	1,96	1,94	1,88	1,76
1,74	1,87	1,92	1,95	1,95	1,92	1,87	1,74
1,72	1,84	1,89	1,92	1,92	1,89	1,84	1,72
1,67	1,79	1,84	1,87	1,87	1,84	1,79	1,67
1,56	1,68	1,73	1,76	1,76	1,73	1,68	1,56
1,00	1,12	1,18	1,20	1,20	1,18	1,12	1,00
999	999	999	999	999	999	999	999
1,45	1,58	1,63	1,66	1,66	1,63	1,58	1,45
1,64	1,76	1,81	1,84	1,84	1,81	1,76	1,64
1,70	1,83	1,88	1,90	1,90	1,88	1,83	1,70
1,73	1,86	1,91	1,94	1,94	1,91	1,86	1,73
1,75	1,88	1,93	1,96	1,96	1,93	1,88	1,75
1,77	1,89	1,94	1,97	1,97	1,94	1,89	1,77
1,78	1,90	1,95	1,98	1,98	1,95	1,90	1,78
1,78	1,91	1,96	1,99	1,99	1,96	1,91	1,78
1,79	1,91	1,97	1,99	1,99	1,97	1,91	1,79
1,79	1,92	1,97	2,00	2,00	1,97	1,92	1,79
1,80	1,92	1,97	1,97	1,97	1,97	1,92	1,80
1,80	1,93	1,93	1,93	1,93	1,93	1,93	1,80
1,80	1,80	1,80	1,80	1,80	1,80	1,80	1,80

Fig 2.5
Example of costs for each cell in 2D

For example lets say we have the case of fig 2.4. Then an example of the costs calculated could look something like in fig 2.5. In this example $C_{wall} = 999$ and $MC = 1$. The observant reader would notice that a MC of 1 gives the cell with the highest cost a value of 2. The cells have a cost in the range $[1, 2]$. The cheapest ones are in the corners adjacent to the side wall and the middle wall. The wall's exact location goes from the middle and a little down, but width of the wall is smaller than the width of a cell. Therefore the cells above the wall get a lower cost than the cells under it. This is because the cells above are right next to the wall, while

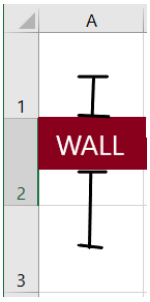


Fig 2.6
Distance between cell and wall.

the cells under the wall are half the cell which contains the wall away from the wall(see fig 2.6). The cells containing the middle wall gets assigned the wall cost. The most expensive cells are those that are far away from the middle wall, and not too close to the side wall. One thing to note is that the surrounding walls are one single geometry. Therefore the distance to this geometry is most often the distance from the cell and to the long side. The most expensive cells are therefore those which are about as far away from the short side as the long side of the surrounding wall. In this example it would have made sense to divide the surrounding wall into individual walls. In a real world scenario with hundreds of different geometries this might be too costly and not necessarily straight forward to do. In the case of other geometries this might also make sense. A truss contains lots of parts, but should be handled as a single geometry.

An important part of the use of an equation instead of a constant cost is that the algorithm will have a different behavior with the varying costs. An equally important part is that the user is in control of this behavior. This is done through varying the MC, T and C_{wall} variables. Varying MC creates a more or less sticky behavior. Varying C_{wall} varies the algorithm's tendency to go around or through obstacles. Varying T makes a path closer or further away from obstacles the optimal. In the grasshopper environment, which will be introduced in the next section, this is done quite easily.

To summarize, a lot has to be prepared before the A* algorithm can be used to find a path for piping. In short what needs to be done is the following:

1. Import relevant geometry.
2. Create a grid around the geometry perpendicular to the most defining geometry.
3. Set the MC, T and C_{wall} variables.
4. Calculate the cost of each cell according to the cost equation.
5. Set a source and goal cell for the pipe and thereby defining it as the maze problem.
6. Run the A* algorithm to find a path.
7. Return a pipe modelled along the path.

2.2 The Rhino Grasshopper workflow

2.2.1 The Rhino Grasshopper cycle

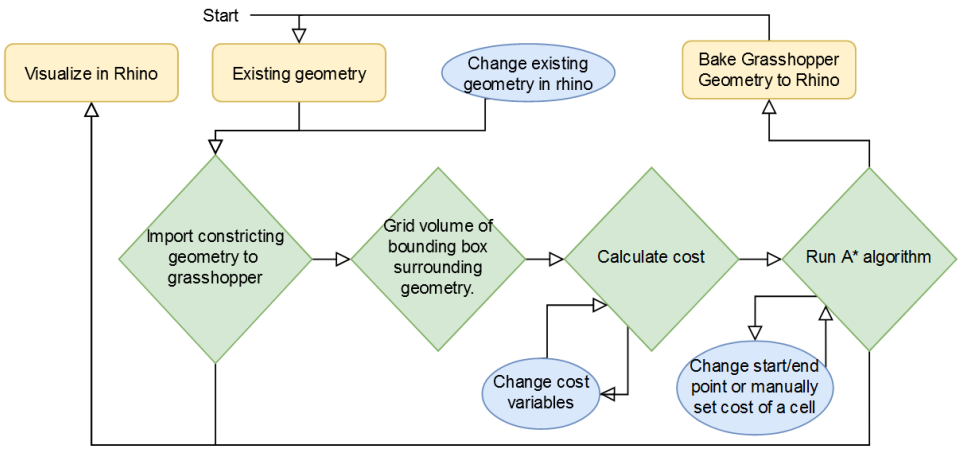


Fig 2.7
Rhino Grasshopper workflow

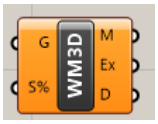
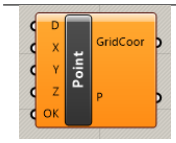
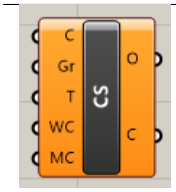
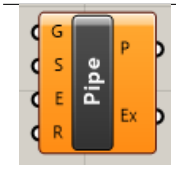
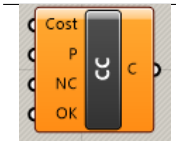
For the task of implementing the functionality described in the last section Rhino and Grasshopper was chosen. As mentioned in section 1.3.1 Rhino is a CAD program and Grasshopper is a plugin for Rhino which adds visual programming (and therefore parametric modelling) capabilities. The workflow of the resulting grasshopper program implemented in this thesis is shown very roughly in fig 2.7. Here the yellow boxes are things happening in Rhino, the green diamonds are things happening in Grasshopper and the blue ellipses are changes done by the user. The constricting geometry will need to be either modelled in Rhino or imported to Rhino. This geometry is then imported into Grasshopper. A bounding box is then created around the constricting geometry and divided into a 3D grid. The grid is oriented perpendicular to the largest vertical surfaces of the constricting geometry. The next thing which will be done in Grasshopper is that each cell will be assigned a cost based on the cost equation (eq 2.1). Then the user sets a start and end point and runs the A* algorithm. After the algorithm is run the resulting geometry will be visualized in Rhino. If the user is satisfied with the result and wants to be able to do changes to it in Rhino the user is able to export it from Grasshopper to Rhino by a function called *baking*. This geometry may then be processed further in Rhino or might be included in the constricting geometry for e.g. a second run of the algorithm. Say the user wants a second pipe, then the algorithm can be run with the first pipe as part of the constricting geometry to avoid collision.

All changes done by the user will initialize a rerun of the affected step in Grasshopper and all following steps. For example if the constricting geometry is changed the whole Grasshopper chain is rerun. If cost variables are changed i.e. C_{wall} then only the cost calculation and the A* algorithm is rerun. The grid will stay untouched.

The way this workflow is implemented in Grasshopper is through a plugin for grasshopper. During this thesis a plugin for Grasshopper was made. This plugin gives Grasshopper additional custom made *components* (the graph nodes which takes input and give an output) to excecute the desired workflow. The custom Grasshopper components created are presented in the table in section 2.2.2.

2.2.2 Components

Table 2.1
Custom components made with C#

 Fig 2.8 World Mesher 3D	G: Geometry defining the necessary space to grid. 5%: Size of each grid cube by percent of shortest side. (default 5%) M: The generated grid/Mesh Ex: Error messages D: Data for PointSetter component
 Fig 2.9 PointSetter	D: Data from WM3D x: x coordinate in the interval [0,1] y: y coordinate in the interval [0,1] z: z coordinate in the interval [0,1] OK: If true then all child components are updated
 Fig 2.10 CostSetter	C: Constricting and load bearing geometry Gr: Grid from WorldMesher3D T: Wall tolerance WC: Wall Cost, cost of going through walls MC: Max Cost. Cost lies in the interval [1,MC+1] O: Output grid of costs C: Debugging text. Not in use
 Fig 2.11 A* Pipe	G: Cost grid from CostSetter S: Start coordinates from PointSetter E: End coordinates from PointSetter R: Radius of pipe P: Pipe Mesh Ex: error messages
 Fig 2.12 CostChanger	Cost: Cost grid from CostSetter P: Coordinates from PointSetter NC: New cost of tile OK: If true the tile's cost is updated. C: The cost of the tile

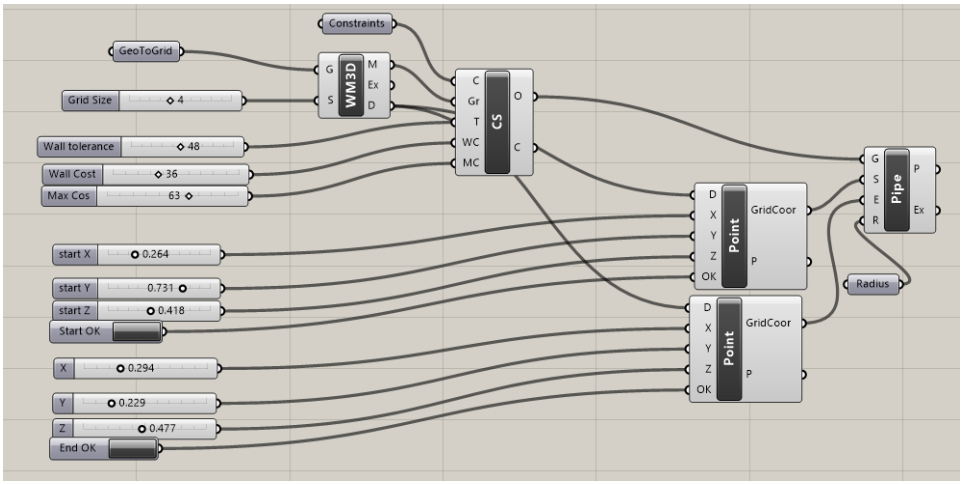


Fig 2.13
Grasshopper components in action

2.2.3 Grasshopper setup

The components in grasshopper are connected as seen in fig 2.13. The function of the individual components are as follows:

World Mesher 3D
The A* algorithm as described in section 2.1.1 requires a grid to operate in. The goal of the World Mesher 3D (WM3D) component is to create that grid. Often the A* is explained in 2 dimensions but there is no difference between 2D and 3D in terms of functionality. The algorithm behaves the same except with the ability to move vertically in addition to horizontally. This component takes geometry as an input to grid the world of the A* algorithm. These geometries are only used only for the grid task. This is to give the user the the ability to choose the outer walls of a building or dummy geometry inside a room to grid only part of the room. The geometry is then inscribed in a bounding box oriented according to the largest vertical surfaces of the input geometry. This may mean that the x and y axes are perpendicular to the outer walls for example. Volume around the bounding box is then divided into cubes of equal size. The size of the cubes are set to a percentage of the shortest side of the length or depth. For example if a room is $10m \times 20m \times 3.5m$ and the percentage is set to 10, the resulting grid will be $10 \times 20 \times 4$ cells large, and the cell size will be $1 \times 1 \times 1m$

CostSetter
The next requirement is for the grid cells to have a cost. This is done by the CostSetter. The CostSetter requires 5 different inputs. The first input is the geometry which will be used in the cost equation. The idea is that this is all the load bearing geometry as described before. This must then be all the relevant geometry and might include a different set of geometries than used for the WM3D component. Each cell gets a cost equal to equation 2.1. The distances are measured from the cell center and to the closest surface

point of each geometry.

A*Pipe
This component is where the pathfinding algorithm is executed based on the previous preparations. The user sets a start and end point with the PointSetter components (see below) in addition to the radius. The component uses the cost grid from CostSetter and finds the optimal route from start to end and creates a pipe mesh with the desired radius.

PointSetter
This components function is to let the user select a grid cell. These cells are used for the A* pipe component and the Cost Changer component. It takes three sliders from 0 to 1 as input for each axis. A ball is created to show which grid cell is currently chosen. When the OK button is pressed, all downstream components are updated.

CostChanger
This component is used if the user wishes to manually change the cost of some of the cells to tweak the path chosen. If for example the user wants the algorithm to be able to penetrate a wall at a specific location or does not want the algorithm to take a path e.g. right in front of a doorway, then the user can change the cost of these tiles to appropriate values.

Installation and use

A guide on this is found in the appendix.

Chapter 3

Results

3.1 Case studies

3.1.1 Around or through wall

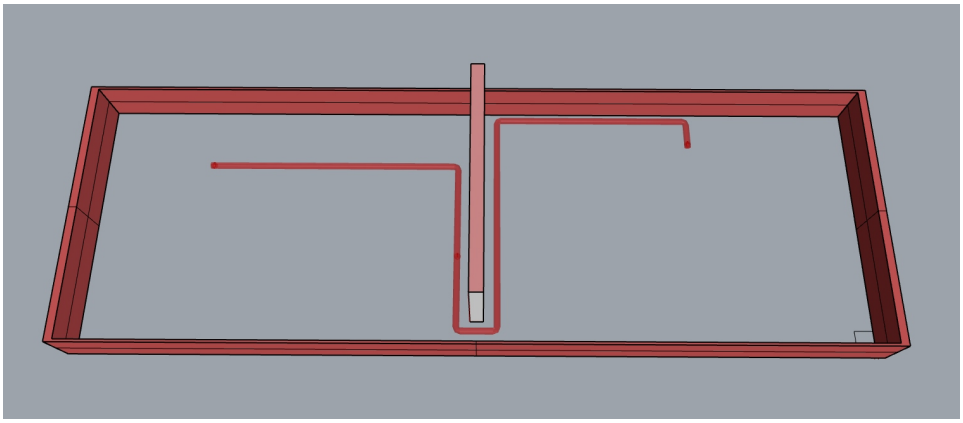


Fig 3.1
As the cost for going through the wall is high A* goes around.

There are several points that are interesting about Fig 3.1. The most obvious is that the cost of going through the wall is high so the preferred way is to go around through the gap. The pipe sticks to the dividing wall which is expected. On the right side it also sticks to the wall but on the left side the path goes straight away from the dividing wall.

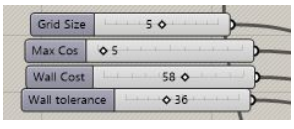


Fig 3.2
Settings for above example

As seen from the settings the wall cost is 58 which is comparatively quite high. As the nodes costs are closer to 1 near the walls the detour would need to be near 50 tiles before going through would be cheaper. Therefore it is natural that the path chosen utilizes the gap and goes around the wall. The stickiness is medium. A max cost of 5 does give cases as seen on the left side, where following the wall as close as possible is not the cheapest. On the left side there is a

possible other path where the path goes straight down and then follows the lower wall to the gap. The reason this is not really viable is that the nodes near the dividing wall is closer to the wall and therefore cheaper. This could be fixed with finer meshing. However the path straight down is also further away from the left wall than the path to the dividing wall is from the upper wall. Therefore the upper path is probably cheaper even though it is a bit longer before the path reaches a wall. If we increase the MaxCost to increase the stickiness the will probably just stick to the top. Therefore it is unlikely that the bottom path will ever be chosen unless given extra initiative by the user.

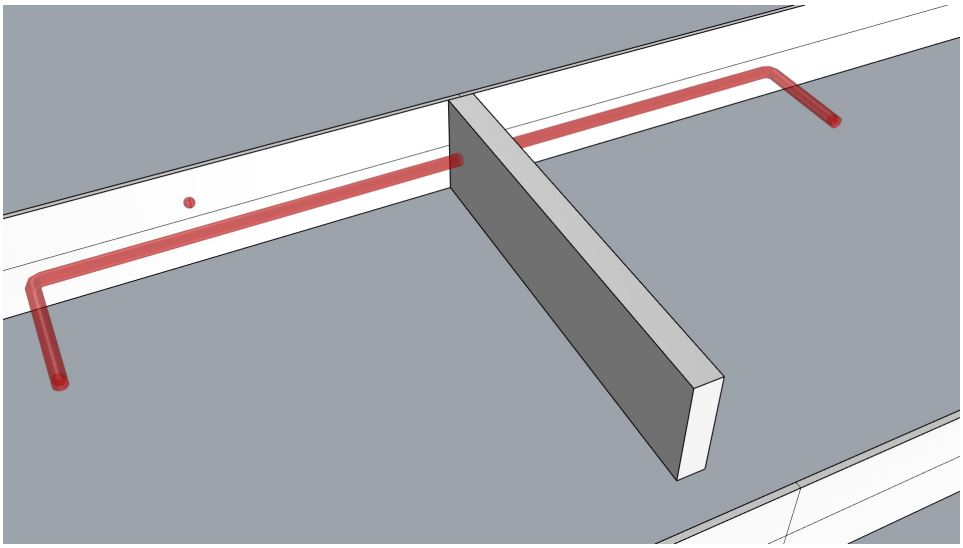


Fig 3.3
Here the cost for going through the wall is lowered.

As the cost of going through a wall is lowered in Fig 3.3 the preferred path is now through the wall. It does also stick to the wall all the way between the points.

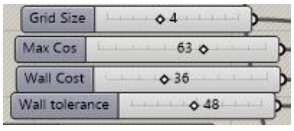


Fig 3.4
Settings for fig 3.3

As the cost of going through is lowered and the MaxCost is greatly increased the general cost of each node is increased. Therefore it is cheaper to take the shorter path through the wall, while because of the high MaxCost, the path will have a sticky behavior and therefore sticks to the wall.

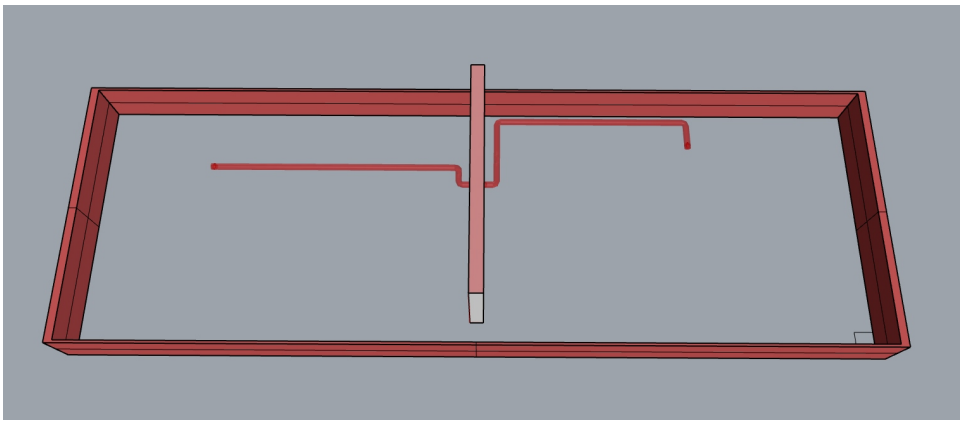


Fig 3.5
A "hole" or a preferred spot for going through the wall was created.

In this example a "hole" was created by manually setting the cost of the wall nodes where it penetrates to 0 with CostChanger. Otherwise this is the exact same setup as in Fig 2.3. It follows the same path except it's cheaper to go through the hole than all the way around.

As this has the same same settings as Fig 3.2 going through the hole is way cheaper than anything else. In this scenario the wall cost would need too be near 1 for the algorithm to choose anything else. We see the same tendency of a sticky right side and sticky left side. This is expected as the path is the same, only that the detour to get past the wall is shorter.

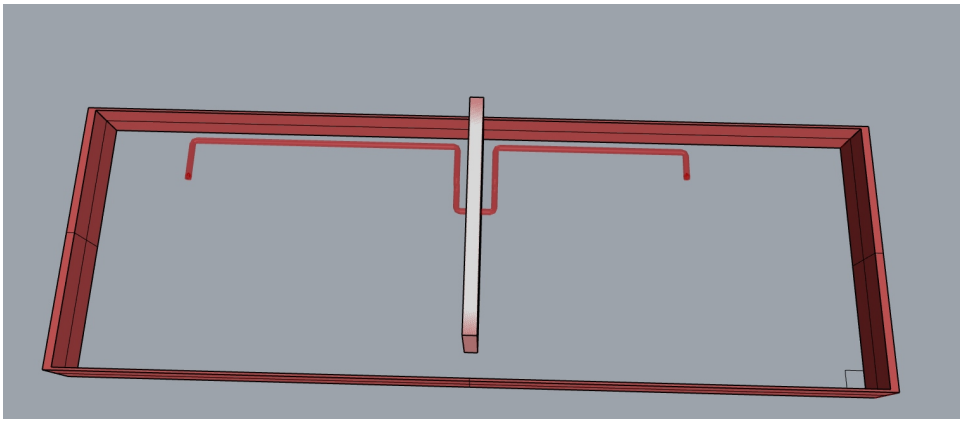


Fig 3.6
Same example as Fig 3.5 but with the start point moved one node to the left.

Fig 3.6 is the same setup as Fig 3.5 but with the left end node of the pipe moved one node to the left. This makes is worth taking the detour along the upper wall instead of going straight. The same was true if the left point was moved one node closer to the upper wall. This tells us that the left node is an edge case of when it's worth it or not to follow the upper wall with the current settings.

3.1.2 Under truss

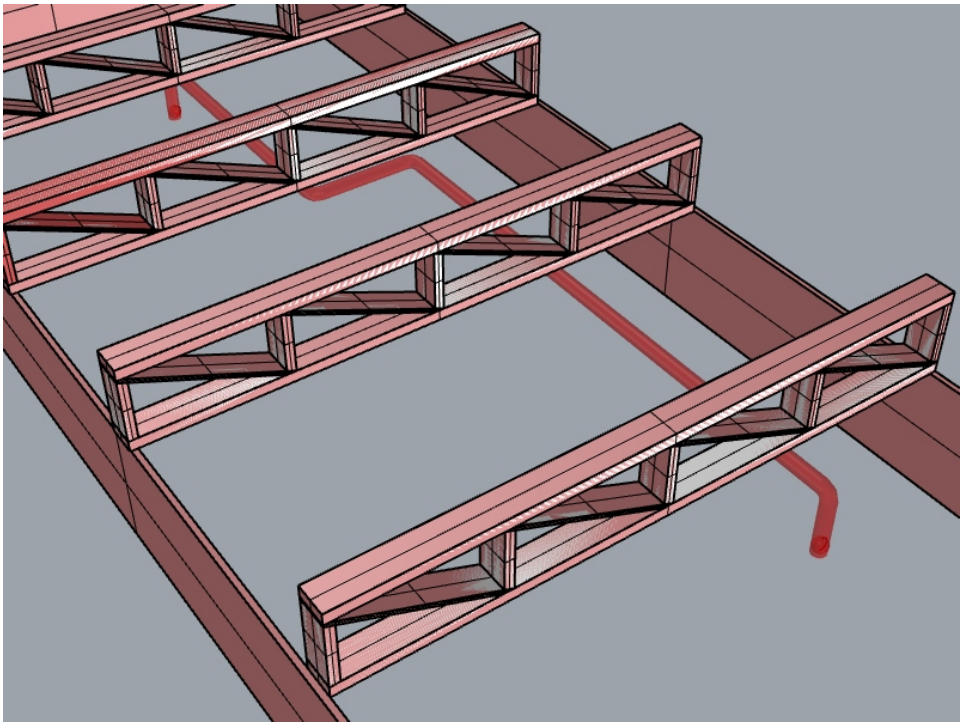


Fig 3.7
Example

This situation is interesting. The grid is coarser but most importantly the wall tolerance is set quite high. This makes A* unable to find a clear path (a path which does not include walls) through the truss even though the pipe in itself is small enough. It does however find that the cheapest solution is to go up and follow the relatively straight path under the trusses instead of going along the walls as previously seen. This includes the move to the left which is done straight under a truss. This is probably the path you want a pipe to take as the beams can carry the load. The detour to the wall is a lot longer so going up might be the better choice.

3.1.3 Through truss

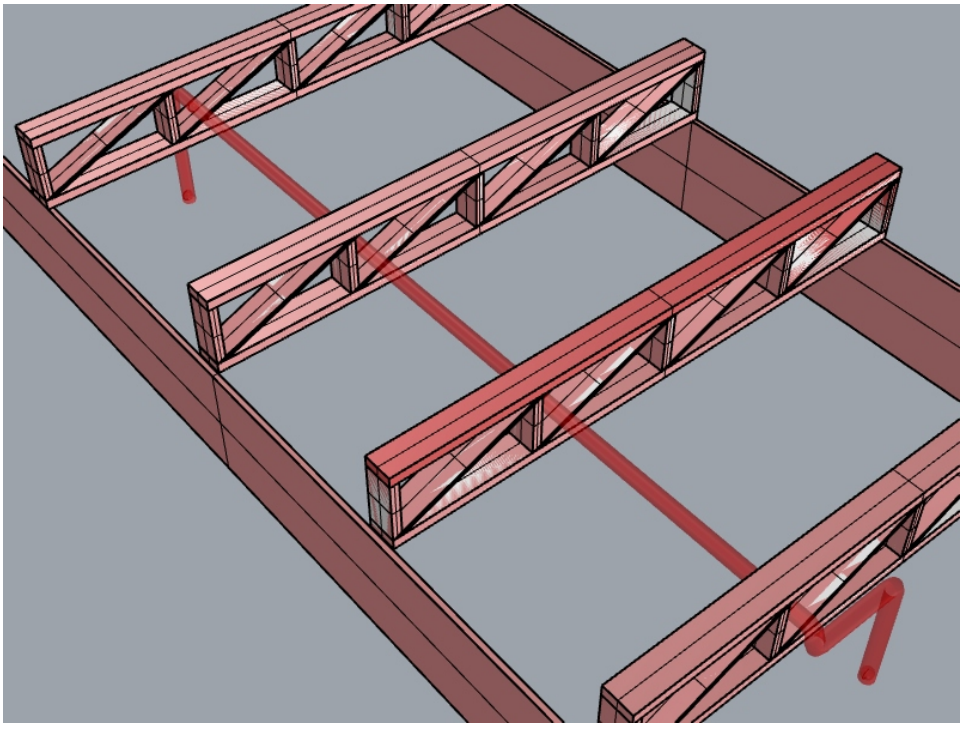


Fig 3.8
Example

One could also have a case where you'd want to go through a beam of e.g. concrete. In this case a hole could be made similar to in Fig 3.5. This is not very practical if operating on a large model with many beams. However that is not necessary a bad thing. Where exactly one penetrates a concrete beam have a lot to say for its model bearing capabilities. A better solution would be for the construction engineer to create beams with holes in them for all relevant beams, run the algorithm with a small enough mesh that it can pass through those holes like in Fig 3.8 and then exchange all non penetrated beams with regular beams afterwards.

3.1.4 Misc

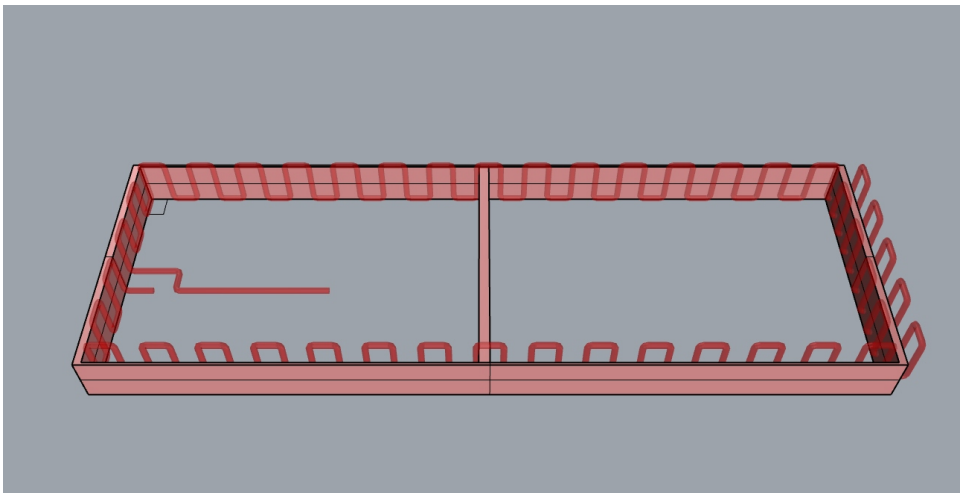


Fig 3.9
Here the cost along the walls are negative

This is why min-max normalization in the cost function is necessary. While experimenting with the cost function this situation was created where the cost along the wall became negative as min-max normalization was not implemented in the code and the scaling of the cost function too large. The cheapest path therefore is to maximize the length of the pipe along the wall to "collect" the negative cost before going to the goal. The result is the pretty pattern seen in Fig 3.9.

Chapter 4

Discussion

4.1 Why the A* algorithm

As mentioned in section 2.1.1 the objective of this thesis is to reduce the problem of modelling a pipe to the maze problem. When reduced to the maze problem a lot of different algorithms can be used. In the maze problem both the grid, source and destination do not change. If the source and destination cells change but the world remains static the A* is normally outperformed by algorithms which uses a pre-processed graph[13]. On the other hand if the source and destination stays the same but the world is partly unkown or dynamic then algorithms like D* Lite are better[14]. D* Lite is an algorithm which has evolved from the A* algorithm to tackle dynamic cases, but it is not used in this thesis. This is because the cases are not dynamic i.e. objects does not suddenly appear in real time, blocking the path. The advantage of using the A* algorithm is that it visits less cells than say the Dijkstra shortest path algorithm. If the world is static but the source and destination cells change then other algorithms which uses pre-processing might outperform A* in terms of speed. In other words there are a lot of algorithm which are more efficient than the A* algorithm in specific cases. The reason why the A* algorithm is chosen is because of its generality. An algorithm specifically made for an dynamic environment will be slow if the world never changes but the start and end points change. Likewise an algorithm which pre-processes the entire world will be slow if the world changes frequently. The A* algorithm also tackles some special cases of the maze problem. Normally going through walls is not allowed in the maze problem. This will be allowed in this master thesis as the A* algorithm allows this.

As described in section 1.4 the A* algorithm uses a cost function and a heuristic function. The algorithm in itself is not that interesting as. This is because it is proven that it is guaranteed to find the optimal path as long as the heuristic function is admissible and consistent[15]. It is also proved that the A* algorithm visits less or equal the amount of node as any other A* like algorithm. Therefore there is little point in changing the algorithm unless changing to a non-A* like algorithm. What can be changed however

is the cost and the heuristic function.

4.2 A point about run time

For the this kind of algorithm to be of any use it needs to be fast. Calculations which finish in near realtime will be much more versatile than calculations which needs hours or days to finish. In parametric modelling most of the effort is used in building the graph or script. After this is done the idea is that changes should be easy and quick to make to optimize the model. As this is to be used in a parametric modelling environment a quick runtime is much preferred. A slow algorithm slows down each incrementation step in the optimization phase when constructing with a parametric model. The alternative is to not run the algorithm whenever you change the parameters, but this goes against the parametric way of designing. Therefore a runtime analasys was done and the results are presented below.

Grasshopper has a tool for measuring the runtime of each component. An example from one of the tests is seen in fig. 4.1. Runtime under 1 second is marked in grey and above 1 second in red. The runtime during the case studies in section 3.1 were quite fast. The runtime of each component were in milliseconds, with other words near instant. The values where relatively similar so the time was mostly used on Grasshopper specific code which is running for every component regardless of function. In the case studies however a very small amount of geometries were used (two at the lowest). This is not realistic in a live use case. More geometry needed to be added. The 10 different objects seen in 4.2 where chosen and copied into the testing field a total of 33 times. This gave the algorithm a total number of 332 different geometries to grid, use for costs and navigate around/through. 332 objects is not a lot, but the thought behind the use case is that this kind of algorithm would only be used together with the relevant geometries in a BIM/parametric model. All the different angled parts of the facade is then irrelevant. The most relevant parts for this algorithm would be walls and beams and for this 332 objects is more adequate. With the new geometry added tests were run with a grid size of 4%, 3% and 1%. The results were as follows:

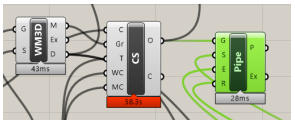


Fig 4.1
Runtime during test

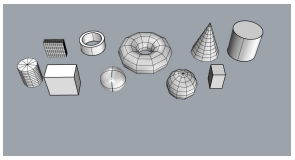


Fig 4.2
Dummy objects

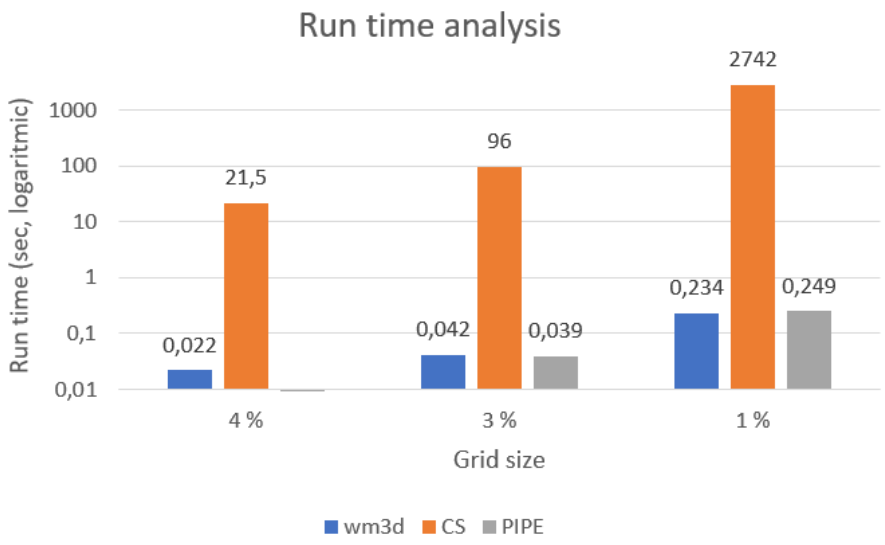


Fig 4.3
Runtime with decreasing grid size

Griding worked surprisingly well. The amount of time used is still in the milliseconds. CostSetters function takes the most of the time. That is 99,9% of the total time used. This was a expected result. In computer science runtime calculations are often done using what's called big O notation.
In big O notation the function $O(n)$ means that there will always exist a function $C \times n > n$ where C is a constant. If a software function loops through n objects and uses a x amount of seconds per object, where x is constant, then the runtime function of the software function can be written as $f(n) = x \times n = O(n)$ because there exists the function $g(n) = C \times n$ where $g(n) > f(n)$ if for example $C = x + 1$.
In the World Mesher 3D component the geometries (g) are looped through twice and the cells (n) once. The runtime of WM3D is therefore $O(g + n)$. The CostSetter loops through all the geometries for each cell so the runtime of CostSetter is $O(n \times g)$. The latter function is expected to be a lot larger. In a worst case scenario the A*Pipe component will run through all the cells once so the runtime of this function is $O(n)$.
With these calculations in mind the result that the CostSetter uses most of the time comes as no surprise. Also there is a very costly method in the middle of the inner most loop seen in fig. 4.4, which further increases the problem.

```
info = t.GetMethod("IsPointInside", new Type[] { Point3d.Unset.GetType(), typeof(double), typeof(bool) });
if (info != null && (bool)info.Invoke(g, new Object[] { grid[i][j][k].Center,tolerance , false })) //----------
```

Fig 4.4
These two lines are costly functions

These lines are costly because the computer is told that even though it does not know which object "t" is, it most likely got the method IsPointInside. The computer then has to find the method and use it. A

real world equivalent scenario of this would be if you and a friend were in a residential area and you need a dictionary. You point at a house and tell your friend that "This is a house. Houses are likely to contain a dictionary. Go into the random house, find the dictionary and look up a word for me." It takes time for a stranger to enter someones home orient themselves find a dictionary and then look up a word.
All this leads to CostSetter being the bottleneck of the system. This means several things. Whenever the geometry is changed the CostSetter needs to run again. This is bad if there are done lots of small changes to the defining geometry. With a grid size of 4% the runtime of CostSetter is around 20 seconds. Research on interactive websites has show that users lose their train of thought after 1s and starts doing other tasks after 10 seconds[16]. 20 seconds is therefore too long. There might be other rendering processes which take more time in a parametric model, but with 20s added on top of that we are at the edge of coffee break territory. Coffee break territory is when the wait time of a program is so long that you have time to get coffee or a snack. With a grid size of 3% the runtime is 1.6m. This is well inside coffee break territory. The users train of thought is longe gone when this finishes. If the user likes coffee brakes this might be acceptable but many might not find it acceptable. At 1% the runtime is 46 minutes. This is not workable so the user would have to disconnect the geometry components when not working with piping. However if the defining geometry is not changed that much, the rest of the system is near instant. In Fig 4.3 the A*Pipe does still have a runtime in milliseconds. Even for a grid size of 1% the runtime is under 1 second which means the user notices the delay, but their focus is not broken[16]. This means the user can do everything except changing the geometry in real time. This includes changing individual costs changing start and end points. The user can experiment with creating "holes" as seen in 3.5 and when satisfied change the geometry accordingly.
To summarize the algorithm in it's current state might be too slow for complex models with lots of geometry objects. On the other hand if the model is less complex and a courser grid is sufficient then the system might be just fast enough to for interactive use.

4.2.1 Improving the runtime of the cost function calculations

The cost function can be greatly improved by the use of multi-threading, and GPU calculations. As seen in equation 2.1 the cost function is linear and can be done in parallel. In the current implementation all the geometries are looped over for each element in the cost matrix. One could do the same in parallel. The problem with this is that each calculation would have to keep the whole set of geometries in memory. Geometries, at least complex ones, take up lot of space and when most processor cores only got a small amount of MB worth of memory this is not ideal. A lot of reading back and forth between the RAM would be necessary. Another way of doing it is not to do the calculations of each element in parallel but to create a cost matrix for each geometry. This might be counter intuitive if we think about the runtime calculations done in section 4.2. If done in parallel calculating each

node separately would have a runtime of $O(g)$ while calculating for each geometry separately would have a runtime of $O(n)$. If $n > g$ then surely calculating each node in parallel must be faster. The problem is memory as mentioned earlier. Each geometry alone does not take that much space and can often be represented mathematically. The result would be a cost matrix for each geometry. With only one geometry to worry about the calculation for each element in each can also be done in parallel. After all the matrices for each geometry is created all that is left is matrix addition and multiplication. All the earlier tasks are what GPUs are for. As long as the memory constraint of each task is low GPU calculations are superior to CPU calculations. If parallelism is implemented this would most likely improve CostSetters runtime significantly.

4.3 A point about grid size, bends and other problems

The runtime of the system is greatly affected by the resolution of the grid as described in section 4.2. A change in resoluton also affects the path as described in section 3.1.2 and 3.1.3. A finer resoluton is important if small openings or the pipe needs to be as close to walls and other geometry as possible. This is often the case and therefore it would be natural to think that one should set the resolution as fine as possible. Unfortunately, besides the increasing runtime, there are currently a lot of other problems which occur if the grid size is set to a very small value. Most of them can be seen in fig. 4.5. Currently the algorithm got no additional cost for making turns. This is fine if you are a character in a computer game, but in construction bends means more welds, which means higher costs. Because a lot of small geometries were added the problem with no cost of bends really shows itself. As the lowest cost is right beside or above geometry the path chosen includes a lot of small bends to pass over each and every geometry along the path. In fig 4.5 there are as much as 7 bends over a very small stretch of pipe. This is not cost efficient. The second problem which is also seen in fig 4.5 is that the grid is smaller than the radius of bends (and also the radius of the pipe). As seen in the figure (at point 2-5 and 7) the current system can not create a proper mesh because the stretches are too short for proper bends. This should not be allowed and should be restricted somehow.

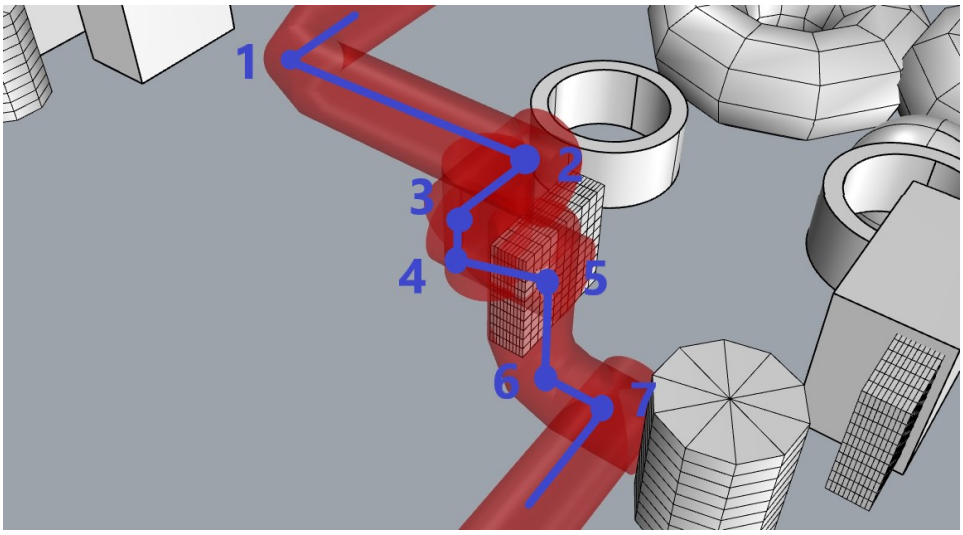


Fig 4.5
This figures shows some of the problems with the current algorithm

Chapter 5

Conclusion and further work

5.1 Was it a good implementation?

This thesis has shown that a implementation of graph searching algorithms, more specifically the A* algorithm, in parametric modelling is possible. It has shown as a promising technology which might some day in the future prove to automate mundane modelling tasks and prevent collisions in complex structures. It is also clear from this thesis that there is a long way to go before we reach that future.

5.2 Further work

The grasshopper plugin created in this thesis was never ment to become a finished product. If one were to develope the plugin further with the goal to produce a product for the industry a lot of changes would need to be made. One of the first thing that need to be fixed is the problem with grid resolution and bends. When the grid resolution is high the path chosen should not include stretches which are smaller than what is needed for proper bends. This would be the best solution. Alternatively a less ideal solution would be to restrict the grid resolution so that no cell is smaller than what is needed for a proper bend.

Secondly there needs to be a cost for adding bends. Going in zigzag might be the shortest route but when construction starts this is costly. Therefore adding bends should be costly so that if two paths are equivalent but one got less bends then the one with fewer bends are chosen. On the other hand on constructions with thermal expansions some bends are necessary to absorb the expansion. In this case long stretches should also be penalized.

Thirdly the CostSetter component needs a speed improvement to handle larger files. The first thing to do is to get rid of the two costly lines seen in fig 4.4. This is probably the easiest thing to do. The reason it was not done in the span of this master thesis was that the CostSetter component was more than fast enough for the experiments done in this thesis. In a real use case on the other hand it is not. If the cost function is kept as it is presented

in this thesis then the calculations should be parallelized as described in section 4.2.1. This will take more time than the first improvement but most likely yield larger rewards, especially when working on larger models.

For further research both the cost function and the heuristic function could be interesting to look at. In this thesis very little research was done on the heuristic function. The Manhattan distance was chosen as this is a quite common heuristic function for the A* algorithm. The cost function was then adjusted so that the Manhattan distance is an admissable heuristic. To research further on the topic of heuristic functions might therefore be quite interesting and possible rewarding. The A* algorithm depends both on the cost and the heuristic and each are equally important. As with the heuristic function only one cost function was implemented. This as well might be a interesting topic of research. As shown in this thesis even small changes to the parameters of the same cost function can make quite different results. Another cost function might therefore generate other solutions. It is likely that the cost function in this thesis might be far from an optimal one.

Appendix

Installation guide

To install the grasshopper plugin one must have Rhino 6 installed. The plugin is tested to work with Rhino Version 6 SR27 (6.27.20176.5001, 06/24/2020).
Extract all files under /plugin in the zip file, that is:

- CSP_for_piping.gha
- Priority Queue.dll
- Priority Queue.pdb
- Priority Queue.xml

Copy the uncompressed files to:
%appdata%/Grasshopper/Libraries/

Run Rhino and Grasshopper and check that you have a new tab called CSPiping in Grasshopper.

Alternatively if the zipped files does not work one can build the project from the source code files. To do this follow this https://developer.rhino3d.com/guides/grassho tools-windows/ and this https://developer.rhino3d.com/guides/grasshopper/your-first-component-windows/ guide on how to compile grasshopper plugins.

For use of the plugin, open the TestField3.3dm file and the CSPiping.gh file. It is advised to do this together with someone familiar with Grasshopper.

A* GIF frames

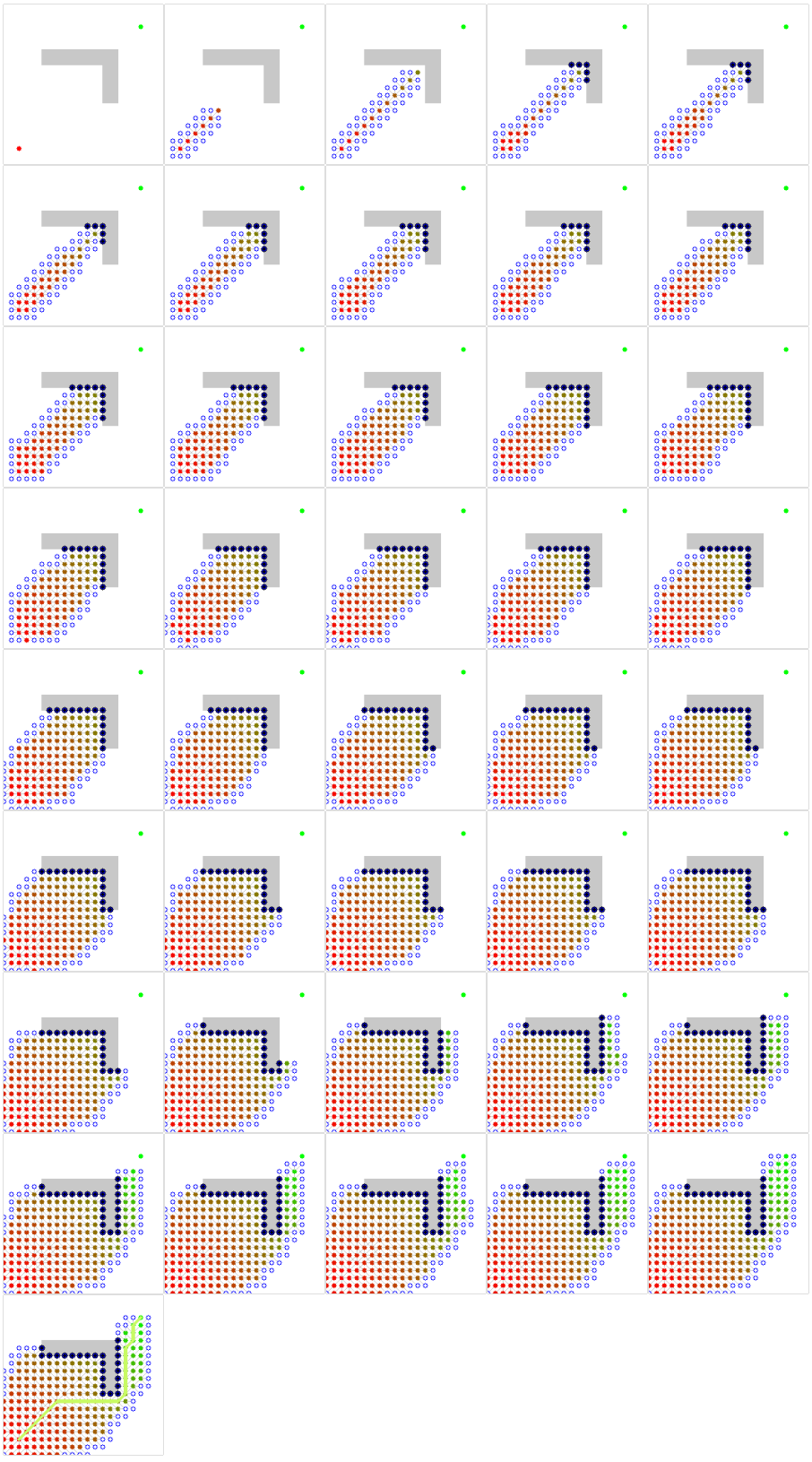


Fig 5.1: This shows the frame of the gif shown in fig: 1.4

Bibliography

[1] R. Aish. "First Build Your Tools." In: *Inside Smartgeometry: Expanding the Architectural Possibilities of Computational Design* (Jan. 2013), pp. 36–49. DOI: 10.1002/9781118653074.ch2.

[2] S. Azhar. "Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry." In: *Leadership and management in engineering* 11.3 (2011), pp. 241–252.

[3] J. Manyika et al. "Digital America: A tale of the haves and have-mores." In: *McKinsey Global Institute* (2015), pp. 1–120.

[4] C. Sun et al. "A literature review of the factors limiting the application of BIM in the construction industry." In: *Technological and Economic Development of Economy* 23.5 (2017), pp. 764–779.

[5] J. Zhang and Z. Hu. "BIM-and 4D-based integrated solution of analysis and management for conflicts and structural safety problems during construction: 1. Principles and methodologies." In: *Automation in construction* 20.2 (2011), pp. 155–166.

[6] A. O. Akponeware and Z. A. Adamu. "Clash detection or clash avoidance? An investigation into coordination problems in 3D BIM." In: *Buildings* 7.3 (2017), p. 75.

[7] R. M.
bibinitperiod Associates. *What are NURBS?* 2020. URL: https://www.rhino3d.com/nurbs (visited on 06/19/2020).

[8] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] N. J. Nilsson. *The quest for artificial intelligence*. Cambridge University Press, 2009.

[10] dummy. *dummy title*. 1980.

[11] S. Chang, R.-S. Shiu, and I.-C. Wu. *Applying an A-Star Search Algorithm for Generating the Minimized Material Scheme for the Rebar Quantity Takeoff*. English. Copyright - Copyright IAARC Publications 2019; Last updated - 2019-12-13. 2019. URL: https://search.proquest.com/docview/2268537017?accountid=12870.

[12] X. Cui and H. Shi. "A*-based pathfinding in modern computer games." In: *International Journal of Computer Science and Network Security* 11.1 (2011), pp. 125–130.

[13] D. Delling et al. “Engineering Route Planning Algorithms.” In: *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Ed. by J. Lerner, D. Wagner, and K. A. Zweig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 117–139. ISBN: 978-3-642-02094-0. DOI: 10.1007/978-3-642-02094-0_7. URL: https://doi.org/10.1007/978-3-642-02094-0_7.

[14] S. Koenig and M. Likhachev. “Fast replanning for navigation in unknown terrain.” In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363.

[15] R. Dechter and J. Pearl. “Generalized Best-First Search Strategies and the Optimality of A*.” In: *J. ACM* 32.3 (July 1985), pp. 505–536. ISSN: 0004-5411. DOI: 10.1145/3828.3830. URL: https://doi.org/10.1145/3828.3830.

[16] F. F.-H. Nah. “A study on tolerable waiting time: how long are Web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163. DOI: 10.1080/01449290410001669914. eprint: https://doi.org/10.1080/01449290410001669914. URL: https://doi.org/10.1080/01449290410001669914.

NTNU – Trondheim
Norwegian University of
Science and Technology